CHAPTER 2

The WebSocket API

This chapter introduces you to the WebSocket Application Programming Interface (API), which you can use to control the WebSocket Protocol and create WebSocket applications. In this chapter, we examine the building blocks of the WebSocket API, including its events, methods, and attributes. To learn how to use the API, we write a simple client application, connect to an existing, publicly available server (`http://websocket.org`), which allows us to send and receive messages over WebSocket. By using an existing server, we can focus on learning about the easy-to-use API that enables you to create WebSocket applications. We also explain step-by-step how to use the WebSocket API to power HTML5 media using binary data. Finally, we discuss browser support and connectivity.

This chapter focuses on the client application side of WebSocket, which enables you to extend the WebSocket Protocol to your web applications. The subsequent chapters will describe the WebSocket Protocol itself, as well as using WebSocket within your environment.

Overview of the WebSocket API

As we mentioned in Chapter 1, WebSocket consists of the network protocol and an API that enable you to establish a WebSocket connection between a client application and the server. We will discuss the protocol in greater detail in Chapter 3, but let's first take a look at the API.

The WebSocket API is an interface that enables applications to use the WebSocket Protocol. By using the API with your applications, you can control a full-duplex communication channel through which your application can send and receive messages. The WebSocket interface is very straightforward and easy to use. To connect to a remote host, you simply create a new WebSocket object instance and provide the new object with a URL that represents the endpoint to which you wish to connect.

A WebSocket connection is established by upgrading from the HTTP protocol to the WebSocket Protocol during the initial handshake between the client and the server, over the same underlying TCP connection. Once established, WebSocket messages can be sent back and forth between the methods defined by the WebSocket interface. In your application code, you then use asynchronous event listeners to handle each phase of the connection life cycle.

The WebSocket API is purely (and truly) event driven. Once the full-duplex connection is established, when the server has data to send to the client, or if resources that you care about change their state, it automatically sends the data or notifications.

V. Wang et al., *The Definitive Guide to HTML5 WebSocket*

With an event-driven API, you do not need to poll the server for the most updated status of the targeted resource; rather, the client simply listens for desired notifications and changes.

We will see different examples of using the WebSocket API in the subsequent chapters when we talk about higher-level protocols, such as STOMP and XMPP. For now, though, let's take a closer look at the API.

Getting Started with the WebSocket API

The WebSocket API enables you to establish full-duplex, bidirectional communication over the Web between your client application and server-side processes. The WebSocket interface specifies the methods that are available for the client and how the client interacts with the network.

To get started, you first create a WebSocket connection by calling the WebSocket constructor. The constructor returns a WebSocket object instance. You can listen for events on that object. These events tell you when the connection opens, when messages arrive, when the connection closes, and when errors occur. You can interact with the WebSocket instance to send messages or close the connection. The subsequent sections explore each of these aspects of the WebSocket API.

The WebSocket Constructor

To establish a WebSocket connection to a server, you use the WebSocket interface to instantiate a WebSocket object by pointing to a URL that represents the endpoint to which you want to connect. The WebSocket Protocol defines two URI schemes, `ws` and `wss` for unencrypted and encrypted traffic between the client and the server, respectively. The `ws` (WebSocket) scheme is analogous to an HTTP URI scheme. The `wss` (WebSocket Secure) URI scheme represents a WebSocket connection over Transport Layer Security (TLS, also known as SSL), and uses the same security mechanism that HTTPS uses to secure HTTP connections.

■ **Note** We'll discuss WebSocket security in depth in Chapter 7.

The WebSocket constructor takes one required argument, `URL` (the URL to which you want to connect) and one optional argument, `protocols` (either a single protocol name or an array of protocol names that the server must include in its response to establish the connection). Examples of protocols you can use in the `protocols` argument are XMPP (Extensible Messaging and Presence Protocol), SOAP (Simple Object Access Protocol), or a custom protocol.

Listing 2-1 illustrates the one required argument in the WebSocket constructor, which must be a fully qualified URL starting with the `ws://` or `wss://` scheme. In this example, the fully qualified URL is `ws://www.websocket.org`. If there is a syntax error in the URL, the constructor will throw an exception.

Listing 2-1. Sample WebSocket Constructor

```
// Create new WebSocket connection

var ws = new WebSocket("ws://www.websocket.org");
```

When connecting to a WebSocket server, you can optionally use the second argument to list the protocols your application supports, namely for protocol negotiation.

To ensure that the client and the server are sending and receiving messages they both understand, they must use the same protocol. The WebSocket constructor enables you to define the protocol or protocols that your client can use to communicate with a server. The server in turn selects the protocol to use; only one protocol can be used between a client and a server. These protocols are used over the WebSocket Protocol. One of the great benefits of WebSocket, as you'll learn in Chapters 3 through 6, is the ability to layer widely used protocols over WebSocket, which lets you do great things like take traditional desktop applications to the Web.

■ **Note** The WebSocket Protocol (RFC 6455) refers to protocols you can use with WebSocket as "subprotocols," even though they are higher-level, fully formed protocols. Throughout this book, we'll generally refer to protocols that you can use with WebSocket simply as "protocols" to avoid confusion.

Before we get too far ahead of ourselves, let's return to the WebSocket constructor in the API. During the initial WebSocket connection handshake, which you'll learn more about in Chapter 3, the client sends a `Sec-WebSocket-Protocol` header with the protocol name. The server chooses zero or one protocol and responds with a `Sec-WebSocket-Protocol` header with the same name the client requested; otherwise, it closes the connection.

Protocol negotiation is useful for determining which protocol or version of a protocol a given WebSocket server supports. An application might support multiple protocols and use protocol negotiation to select which protocol to use with a particular server. Listing 2-2 shows the WebSocket constructor with support for a hypothetical protocol, "myProtocol":

Listing 2-2. Sample WebSocket Constructor with Protocol Support

```
// Connecting to the server with one protocol called myProtocol

var ws = new WebSocket("ws://echo.websocket.org", "myProtocol");
```

■ **Note** In Listing 2-2, the hypothetical protocol "myProtocol" is a well-defined, perhaps even registered and standardized, protocol name that both the client application and the server can understand.

The WebSocket constructor can also include an array of protocol names that the client supports, which lets the server decide which one to use. Listing 2-3 shows a sample WebSocket constructor with a list of protocols it supports, represented as an array:

Listing 2-3. Sample WebSocket Constructor with Protocol Support

```
// Connecting to the server with multiple protocol choices

var echoSocket = new
WebSocket("ws://echo.websocket.org", ["com.kaazing.echo",
"example.imaginary.protocol"])

echoSocket.onopen = function(e) {
   // Check the protocol chosen by the server
   console.log(echoSocket.protocol);
}
```

In Listing 2-3, because the WebSocket server at `ws://echo.websocket.org` only understands the `com.kaazing.echo` protocol and not `example.imaginary.protocol`, the server chooses the `com.kaazing.echo` protocol when the WebSocket `open` event fires. Using an array gives you flexibility in enabling your application to use different protocols with different servers.

We'll discuss the WebSocket Protocol in depth in the next chapter, but in essence, there are three types of protocols you can indicate with the protocols argument:

- Registered protocols: Standard protocols that have been officially registered according to RFC 6455 (The WebSocket Protocol) and with the IANA (Internet Assigned Numbers Authority), the official governing body for registered protocols. An example of a registered protocol is Microsoft's SOAP over WebSocket protocol. See `http://www.iana.org/assignments/websocket/websocket.xml` for more information.
- Open protocols: Widely used and standardized protocols like XMPP and STOMP, which have not been registered as official standard protocols. We will examine how to use these types of protocols with WebSocket in the subsequent chapters.
- Custom protocols: Protocols that you've written and want to use with WebSocket.

In this chapter, we focus on using the WebSocket API as you would for your own custom protocol and examine using open protocols in the later chapters. Let's take a look at the events, objects, and methods individually and put them together into a working example.

WebSocket Events

The WebSocket API is purely event driven. Your application code listens for events on WebSocket objects in order to handle incoming data and changes in connection status. The WebSocket Protocol is also event driven. Your client application does not need to poll the server for updated data. Messages and events will arrive asynchronously as the server sends them.

WebSocket programming follows an asynchronous programming model, which means that as long as a WebSocket connection is open, your application simply listens for events. Your client does not need to actively poll the server for more information. To start listening for the events, you simply add callback functions to the WebSocket object. Alternatively, you can use the `addEventListener()` DOM method to add event listeners to your WebSocket objects.

A WebSocket object dispatches four different events:

- Open
- Message
- Error
- Close

As with all web APIs, you can listen for these events using `on<eventname>` handler properties, as well as using the `addEventListener();` method.

WebSocket Event: Open

Once the server responds to the WebSocket connection request, the `open` event fires and a connection is established. The corresponding callback to the `open` event is called `onopen`.

Listing 2-4 illustrates how to handle the event when the WebSocket connection is established.

Listing 2-4. Sample Open Event Handler

```
// Event handler for the WebSocket connection opening
ws.onopen = function(e) {
   console.log("Connection open...");
};
```

By the time the open event fires, the protocol handshake has completed and the WebSocket is ready to send and receive data. If your application receives an open event, you can be sure that a WebSocket server successfully handled the connection request and has agreed to communicate with your application.

WebSocket Event: Message

WebSocket messages contain the data from the server. You may also have heard of WebSocket frames, which comprise WebSocket messages. We'll discuss the concept of messages and frames in more depth in Chapter 3. For the purposes of understanding

how messages work with the API, the WebSocket API only exposes complete messages, not WebSocket frames. The `message` event fires when messages are received. The corresponding callback to the message event is called `onmessage`.

Listing 2-5 shows a message handler receiving a text message and displaying the content of the message.

Listing 2-5. Sample Message Event Handler for Text Messages

```
// Event handler for receiving text messages
ws.onmessage = function(e) {
   if(typeof e.data === "string"){
      console.log("String message received", e, e.data);
   } else {
      console.log("Other message received", e, e.data);
   }
};
```

In addition to text, WebSocket messages can handle binary data, which are handled as Blob messages, as shown in Listing 2-6 or as ArrayBuffer messages, as shown in Listing 2-7. Because the application setting for the WebSocket message binary data type affects incoming binary messages, you must decide the type you want to use for incoming binary data on the client before reading the data.

Listing 2-6. Sample Message Event Handler for Blob Messages

```
// Set binaryType to blob (Blob is the default.)
ws.binaryType = "blob";

// Event handler for receiving Blob messages
ws.onmessage = function(e) {
   if(e.data instanceof Blob){
      console.log("Blob message received", e.data);
      var blob = new Blob(e.data);
   }
};
```

Listing 2-7 shows a message handler checking and handling for ArrayBuffer messages.

Listing 2-7. Sample Message Event Handler for ArrayBuffer Messages

```
// Set binaryType to ArrayBuffer messages
ws.binaryType = "arraybuffer";

// Event handler for receiving ArrayBuffer messages
ws.onmessage = function(e) {
   if(e.data instanceof ArrayBuffer){
      console.log("ArrayBuffer Message Received", + e.data);
      // e.data is an ArrayBuffer. Create a byte view of that object.
      var a = new Uint8Array(e.data);
   }
};
```

WebSocket Event: Error

The `error` event fires in response to unexpected failures. The corresponding callback to the `error` event is called `onerror`. Errors also cause WebSocket connections to close. If you receive an error event, you can expect a close event to follow shortly. The code and reason in the close event can sometimes tell you what caused the error. The `error event handler` is a good place to call your reconnection logic to the server and handle the exceptions coming from the WebSocket object. Listing 2-8 shows an example of how to listen for `error` events.

Listing 2-8. Sample Error Event Handler

```
// Event handler for errors in the WebSocket object
ws.onerror = function(e) {
   console.log("WebSocket Error: " , e);
   //Custom function for handling errors
   handleErrors(e);
};
```

WebSocket Event: Close

The `close` event fires when the WebSocket connection is closed. The corresponding callback to the `close` event is called `onclose`. Once the connection is closed, the client and server can no longer receive or send messages.

■ **Note** The WebSocket specification also defines `ping` and `pong` frames that can be used for keep-alive, heartbeats, network status probing, latency instrumentation, and so forth, but the WebSocket API does not currently expose these features. Although the browser receives a `ping` frame, it will not fire a visible `ping` event on the corresponding WebSocket. Instead, the browser will respond automatically with a `pong` frame. However, a browser-initiated `ping` that is unanswered by a `pong` after some period of time may also trigger the connection `close` event. Chapter 8 covers WebSocket pings and pongs in more detail.

You also trigger the `onclose` event handler when you call the `close()` method and terminate the connection with the server, as shown in Listing 2-9.

Listing 2-9. Sample Close Event Handler

```
// Event handler for closed connections
ws.onclose = function(e) {
   console.log("Connection closed", e);
};
```

The WebSocket `close` event is triggered when the connection is closed, which can be due to a number of reasons such as a connection failure or a successful WebSocket closing handshake. The WebSocket object attribute `readyState` reflects the status of the connection (2 for closing or 3 for closed).

The close event has three useful properties you can use for error handling and recovery: wasClean, code, and error. The wasClean property is a boolean indicating whether the connection was closed cleanly. The property is true if the WebSocket closed in response to a close frame from the server. If the connection closes due to some other reason (for example, because underlying TCP connection closed), the wasClean property is false. The code and reason properties indicate the status of the closing handshake conveyed from the server. These properties are symmetrical with the code and reason arguments given in the WebSocket.close() method, which we'll describe in detail later in this chapter. In Chapter 3, we will cover the closing codes and their meanings as we discuss the WebSocket Protocol.

■ **Note** For more details about WebSocket events, see the WebSocket API specification at http://www.w3.org/TR/websockets/.

WebSocket Methods

WebSocket objects have two methods: send() and close().

WebSocket Method: send()

Once you establish a full-duplex, bidirectional connection between your client and server using WebSocket, you can invoke the send() method while the connection is open (that is, after the onopen listener is called and before the onclose listener is called). You use the send() method to send messages from your client to the server. After sending one or more messages, you can leave the connection open or call the close() method to terminate the connection.

Listing 2-10 is an example of how you can send a text message to the server.

Listing 2-10. Sending a Text Message Over WebSocket

```
// Send a text message
ws.send("Hello WebSocket!");
```

The send() method transmits data when the connection is open. If the connection is not available or closed, it throws an exception about the invalid connection state. A common mistake people make when starting out with the WebSocket API is attempting to send messages before the connection is open, as shown in Listing 2-11.

Listing 2-11. Attempting to Send Messages Before Opening a Connection

```
// Open a connection and try to send a message. (This will not work!)
var ws = new WebSocket("ws://echo.websocket.org")
ws.send("Initial data");
```

Listing 2-11 will not work because the connection is not yet open. Instead, you should wait for the open event before sending your first message on a newly constructed WebSocket, as shown in Listing 2-12.

Listing 2-12. Waiting for the Open Event Before Sending a Message

```
// Wait until the open event before calling send().
var ws = new WebSocket("ws://echo.websocket.org")
ws.onopen = function(e) {
   ws.send("Initial data");
}
```

If you want to send messages in response another event, you can check the WebSocket `readyState` property and choose to send the data only while the socket is open, as shown in Listing 2-13.

Listing 2-13. Checking the readyState Property for an Open WebSocket

```
// Handle outgoing data. Send on a WebSocket if that socket is open.
function myEventHandler(data) {
   if (ws.readyState === WebSocket.OPEN) {
      // The socket is open, so it is ok to send the data.
      ws.send(data);
   } else {
      // Do something else in this case.
      //Possibly ignore the data or enqueue it.
   }
}
```

In addition to the text (string) messages, the WebSocket API allows you to send binary data, which is especially useful to implement binary protocols. Such binary protocols can be standard Internet protocols typically layered on top of TCP, where the payload can be either a Blob or an ArrayBuffer. Listing 2-14 is an example of how you can send a binary message over WebSocket.

■ **Note** Chapter 6 shows an example of how you can send binary data over WebSocket.

Listing 2-14. Sending a Binary Message Over WebSocket

```
// Send a Blob
var blob = new Blob("blob contents");
ws.send(blob);

// Send an ArrayBuffer
var a = new Uint8Array([8,6,7,5,3,0,9]);
ws.send(a.buffer);
```

Blob objects are particularly useful when combined with the JavaScript File API for sending and receiving files, mostly multimedia files, images, video, and audio. The sample code at the end of this chapter uses the WebSocket API in conjunction with the File API, reads the content of a file, and sends it as a WebSocket message.

WebSocket Method: `close()`

To close the WebSocket connection or to terminate an attempt to connect, use the `close()` method. If the connection is already closed, then the method does nothing. After calling `close()`, you cannot send any more data on the closed WebSocket. Listing 2-15 shows an example of the `close()` method:

Listing 2-15. Calling the `close()` Method

```
// Close the WebSocket connection
ws.close();
```

You can optionally pass two arguments to the `close()` method: `code` (a numerical status code) and `reason` (a text string). Passing these arguments transmits information to the server about why the client closed the connection. We will discuss the status codes and reasons in greater detail in Chapter 3, when we cover the WebSocket closing handshake. Listing 2-16 shows an example of calling the `close()` method with an argument.

Listing 2-16. Calling the `close()` Method with a Reason

```
// Close the WebSocket connection because the session has ended successfully
ws.close(1000, "Closing normally");
```

Listing 2-16 uses code 1000, which means, as it states in the code, that the connection is closing normally.

WebSocket Object Attributes

There are several WebSocket Object attributes you can use to provide more information about the WebSocket object: `readyState`, `bufferedAmount`, and `protocol`.

WebSocket Object Attribute: readyState

The WebSocket object reports the state of the connection through the read-only attribute `readyState`, which you've already learned a bit about in the previous sections. This attribute automatically changes according to the connection state, and provides useful information about the WebSocket connection.

Table 2-1 describes the four different values to which the `readyState` attribute can be set to describe connection state.

Table 2-1. *readyState Attributes, Values, and Status Descriptions*

Attribute Constant	Value	Status
WebSocket.CONNECTING	0	The connection is in progress but has not been established.
WebSocket.OPEN	1	The connection has been established. Messages can flow between the client and server.
WebSocket.CLOSING	2	The connection is going through the closing handshake.
WebSocket.CLOSED	3	The connection has been closed or could not be opened.

(World Wide Web Consortium, 2012)

As the WebSocket API describes, when the WebSocket object is first created, its `readyState` is `0`, indicating that the socket is connecting. Understanding the current state of the WebSocket connection can help you debug your application, such as to ensure you've opened the WebSocket connection before you've attempted to start sending requests to the server. This information can also be useful in understanding the lifespan of your connection.

WebSocket Object Attribute: bufferedAmount

When designing your application, you may want to check for the amount of data buffered for transmission to the server, particularly if the client application transports large amounts of data to the server. Even though calling `send()` is instant, actually transmitting that data over the Internet is not. Browsers will buffer outgoing data on behalf of your client application, so you can call `send()` as often as you like with as much data as you like. If you want to know how quickly that data is draining out to the network, however, the WebSocket object can tell you the size of the buffer. You can use the `bufferedAmount` attribute to check the number of bytes that have been queued but not yet transmitted to the server. The values reported in this attribute do not include framing overhead incurred by the protocol or buffering done by the operating system or network hardware.

Listing 2-17 shows an example of how to use the `bufferedAmount` attribute to send updates every second; if the network cannot handle that rate, it adjusts accordingly.

Listing 2-17. bufferedAmount Example

```
// 10k max buffer size.
var THRESHOLD = 10240;

// Create a New WebSocket connection
var ws = new WebSocket("ws://echo.websocket.org/updates");

// Listen for the opening event
ws.onopen = function () {
```

```
   // Attempt to send update every second.
   setInterval( function() {
      // Send only if the buffer is not full
      if (ws.bufferedAmount < THRESHOLD) {
         ws.send(getApplicationState());
      }
   }, 1000);
};
```

Using the `bufferedAmount` attribute can be useful for throttling the rate at which applications send data to the server avoiding network saturation.

■ **Pro Tip** You may want to examine the WebSocket object's `bufferedAmount` attribute before attempting to close the connection to determine if any data has yet to be transmitted from the application.

WebSocket Object Attribute: protocol

In our previous discussion about the WebSocket constructor, we mentioned the `protocol` argument that lets the server know which protocol the client understands and can use over WebSocket. The WebSocket object `protocol` attribute provides another piece of useful information about the WebSocket instance. The result of protocol negotiation between the client and the server is visible on the WebSocket object. The `protocol` attribute contains the name of the protocol chosen by the WebSocket server during the opening handshake. In other words, the `protocol` attribute tells you which protocol to use with a particular WebSocket. The `protocol` attribute is the empty string before the opening handshake completes and remains an empty string if the server does not choose one of the protocols offered by the client.

Putting It All Together

Now that we've walked through the WebSocket constructor, events, attributes, and methods, let's put together what we have learned about the WebSocket API. Here, we create a client application to communicate with a remote server over the Web and exchange data using WebSocket. Our sample JavaScript client uses the "Echo" server hosted at `ws://echo.websocket.org`, which receives and returns any message you send to the server. Using an Echo server can be useful for pure client-side testing, particularly for understanding how the WebSocket API interacts with the server.

First, we create the connection, then display on a web page the events triggered by our code, which come from the server. The page will display information about the client connecting to the server, sending and receiving messages to and from the server, then disconnecting from the server.

Listing 2-18 shows a complete example of communication and messaging with the server.

Listing 2-18. Complete Client Application Using the WebSocket API

```
<!DOCTYPE html>
<title>WebSocket Echo Client</title>
<h2>Websocket Echo Client</h2>

<div id="output"></div>
<script>

// Initialize WebSocket connection and event handlers

function setup() {
   output = document.getElementById("output");
   ws = new WebSocket("ws://echo.websocket.org/echo");

// Listen for the connection open event then call the sendMessage function
   ws.onopen = function(e) {
      log("Connected");
      sendMessage("Hello WebSocket!")
   }

// Listen for the close connection event
   ws.onclose = function(e) {
      log("Disconnected: " + e.reason);
   }

// Listen for connection errors
   ws.onerror = function(e) {
      log("Error ");
   }

// Listen for new messages arriving at the client
   ws.onmessage = function(e) {
      log("Message received: " + e.data);
      // Close the socket once one message has arrived.
      ws.close();
   }
}

// Send a message on the WebSocket.
function sendMessage(msg){
   ws.send(msg);
      log("Message sent");
   }

// Display logging information in the document.
function log(s) {
   var p = document.createElement("p");
   p.style.wordWrap = "break-word";
```

```
   p.textContent = s;
   output.appendChild(p);

   // Also log information on the javascript console
   console.log(s);
}

// Start running the example.
setup();
</script>
```

After running the web page, the output should look similar to the following:

```
WebSocket Sample Client


Connected


Message sent


Message received: Hello WebSocket!


Disconnected
```

If you see this output, congratulations! You've successfully created and executed your first sample WebSocket client application. If the example does not work, you'll need to investigate why it has failed. You may find useful information in the JavaScript console of your browser. It is possible, though increasingly unlikely, that your browser does not support WebSocket. While the latest versions of every major browser contain support for the WebSocket API and protocol, there are still some older browsers in use that do not have this support. The next section shows you how to ensure your browser supports WebSocket.

Checking for WebSocket Support

Since (surprisingly) not all web browsers support WebSocket natively yet, it's good practice to include in your code a way to determine the browser support and, if possible, provide a fallback. Most modern browsers support WebSocket, but depending on your users, you'll likely want to use one of these techniques to cover your bases.

■ **Note** Chapter 8 discusses various WebSocket fallback and emulation options.

There are several ways to determine whether your own browser supports WebSocket. One handy tool to use to investigate your code is the web browser's JavaScript console. Each browser has a different way to initiate the JavaScript console. In Google Chrome, for example, you can open the console by choosing **View ➤ Developer ➤ Developer Tools**, then clicking **Console**. For more information about Chrome Developer Tools, see https://developers.google.com/chrome-developer-tools/docs/overview.

■ **Pro Tip** Google's Chrome Developer Tools also enables you to inspect WebSocket traffic. To do so, in the Developer Tools panel, click Network, then at the bottom of the panel, click WebSockets. Appendix A covers useful WebSocket debugging tools in detail.

Open your browser's interactive JavaScript console and evaluate the expression `window.WebSocket`. If you see the WebSocket constructor object, this means your web browser supports WebSocket natively. If your browser supports WebSocket but your sample code does not work, you'll need to further debug your code. If you evaluate the same expression and it comes back blank or undefined, your browser does not support WebSocket natively.

To ensure your WebSocket application works in browsers that do not support WebSocket, you'll need to look at fallback or emulation strategies. You can write this yourself (which is very complex), use a polyfill (a JavaScript library that replicates the standard API for older browsers), or use a WebSocket vendor like Kaazing, which supports WebSocket emulation that enables any browser (back to Microsoft Internet Explorer 6) to support the HTML5 WebSocket standard APIs. We'll discuss these options further in Chapter 8 as part of deploying your WebSocket application to the enterprise.

As part of your application, you can add a conditional check for WebSocket support, as shown in Listing 2-19.

Listing 2-19. Client Code to Determine WebSocket Support in a Browser

```
if (window.WebSocket){
    console.log("This browser supports WebSocket!");
} else {
    console.log("This browser does not support WebSocket.");
}
```

■ **Note** There are many online resources that describe HTML5 and WebSocket compatibility with browsers, including mobile browsers. Two such resources are http://caniuse.com/ and http://html5please.com/.

Using HTML5 Media with WebSocket

As part of HTML5 and the Web platform, the WebSocket API was designed to work well with all HTML5 features. The data types that you can send and receive with the API are broadly useful for transferring application data and media. Strings, of course, allow you to represent web data formats like XML and JSON. The binary types integrate with APIs like drag-and-drop, FileReader, WebGL, and the Web Audio API.

Let's take a look at using HTML5 media with WebSocket. Listing 2-20 shows a complete client application using HTML5 Media with WebSocket. You can create your own HTML file based on this code.

■ **Note** To build (or simply follow) the examples in this book, you can choose to use the virtual machine (VM) we've created that contains all the code, libraries, and servers we use in our examples. Refer to Appendix B for instructions on how to download, install, and start the VM.

Listing 2-20. Complete Client Application Using HTML5 Media with WebSocket

```
<!DOCTYPE html>
<title>WebSocket Image Drop</title>
<h1>Drop Image Here</h1>
<script>

// Initialize WebSocket connection
var wsUrl = "ws://echo.websocket.org/echo";
var ws = new WebSocket(wsUrl);
ws.onopen = function() {
 console.log("open");
}

// Handle binary image data received on the WebSocket
ws.onmessage = function(e) {
   var blob = e.data;
   console.log("message: " + blob.size + " bytes");
   // Work with prefixed URL API
  if (window.webkitURL) {
      URL = webkitURL;
   }

   var uri = URL.createObjectURL(blob);
   var img = document.createElement("img");
   img.src = uri;
   document.body.appendChild(img);
}
```

```
// Handle drop event
document.ondrop = function(e) {
   document.body.style.backgroundColor = "#fff";
   try {
      e.preventDefault();
     handleFileDrop(e.dataTransfer.files[0]);
     return false;
   } catch(err) {
     console.log(err);
   }
}

// Provide visual feedback for the drop area
document.ondragover = function(e) {
   e.preventDefault();
   document.body.style.backgroundColor = "#6fff41";
}
document.ondragleave = function() {
  document.body.style.backgroundColor = "#fff";
}

// Read binary file contents and send them over WebSocket
function handleFileDrop(file) {
   var reader = new FileReader();
   reader.readAsArrayBuffer(file);
   reader.onload = function() {
     console.log("sending: " + file.name);
     ws.send(reader.result);
   }
}
</script>
```

Open this file in your favorite modern browser. Take a look at your browser's JavaScript console while the WebSocket connection opens. Figure 2-1 shows the client application running in Mozilla Firefox. Notice that, at the bottom of this figure, we've displayed the JavaScript console, available in Firebug (a powerful web development and debugging tool available at http://getfirebug.com).

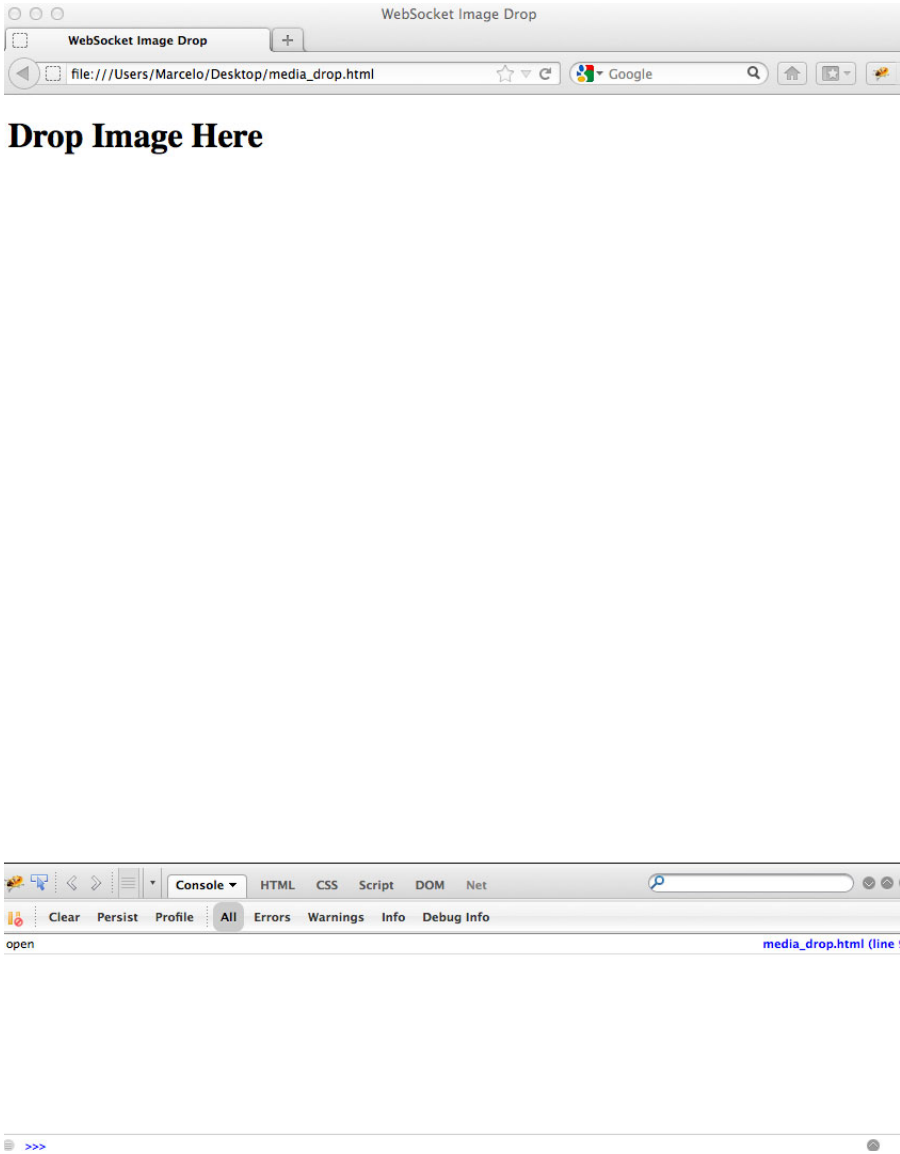

Figure 2-1. *Client application using HTML5 Media with WebSocket displaying in Mozilla Firefox*

Now, try dragging and dropping an image file onto this page. After you finish dropping the image file onto the page, you should see the image rendered on the web page, as shown in Figure 2-2. Notice how Firebug displays information about the image file being added to your page.

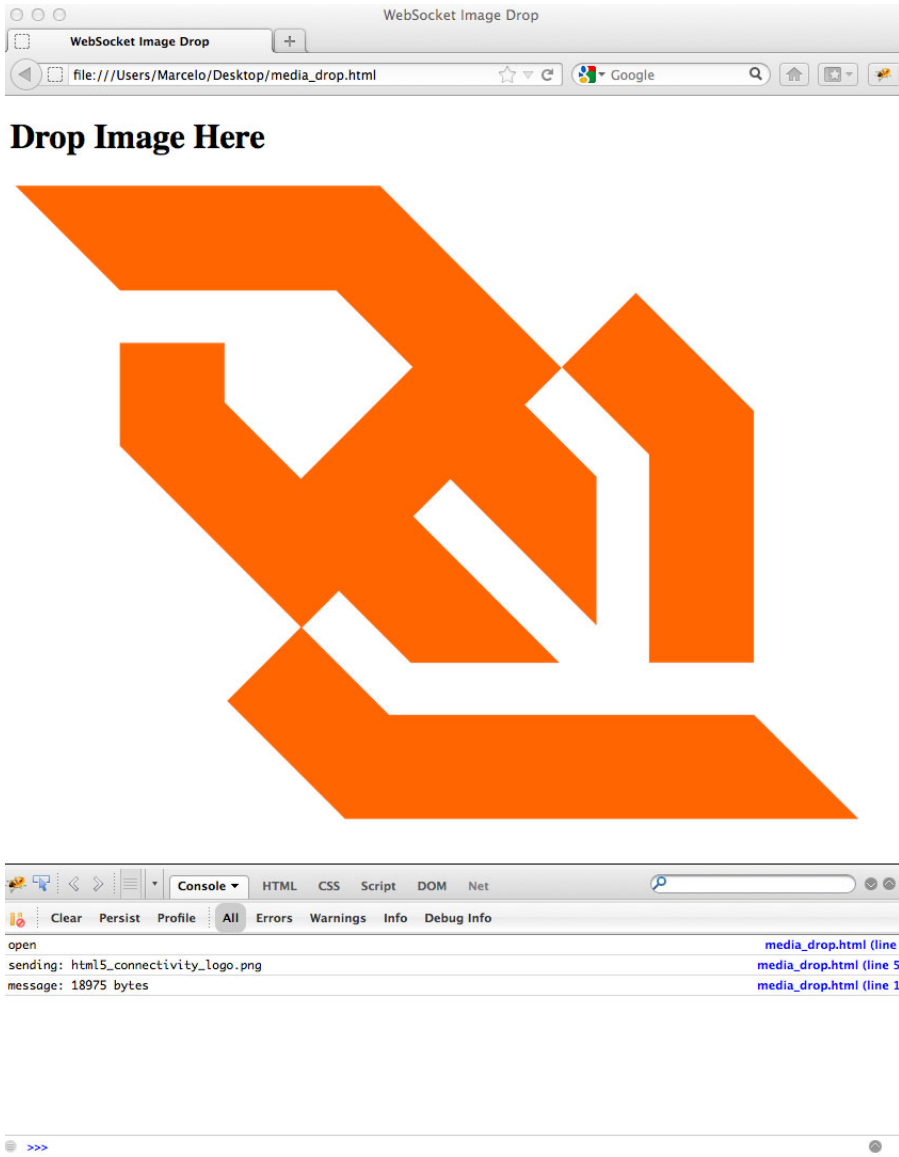

Figure 2-2. *Image (PNG) displayed in the client application using HTML5 Media with WebSocket in Mozilla Firefox*

■ **Note** The server `websocket.org` currently only accepts small messages, so this example will only work with image files less than 65kb in size, though this limit may change. You can experiment with larger media on your own servers.

The "wow" factor of this demo may be diminished by the fact that the media is originating from the same browser where it is ultimately displayed. You could accomplish the same visual result with AJAX or even without the network at all. Things get really interesting when a client or server sends some media data out that is displayed by a different browser—even *thousands* of other browsers! The same mechanics of reading and displaying binary image data work in a broadcast scenario just the same as in this simplified echo demo.

Summary

In this chapter, you learned about the various aspects of the WebSocket API, which enables you to initiate a WebSocket connection from a client application running in a browser and send messages from a server over a WebSocket connection to your client. You learned the basic concepts behind the WebSocket API, including events, messages, and attributes, as well as saw a few examples of the API in action. You also learned how to create your own WebSocket application with a publicly available WebSocket Echo server, which you can use for further testing of your own applications. For an authoritative definition of the interface, see the full WebSocket API specification at `http://www.w3.org/TR/websockets/`.

In Chapter 3, you will learn about the WebSocket Protocol and step through constructing your own basic WebSocket server.