Web Socket API

Quick Guides for Masterminds

J.D Gauchat
www.jdgauchat.com

Cover Illustration by **Patrice Garden**
www.smartcreativz.com

The content of this guide is a collection of excerpts from the book HTML5 for Masterminds. For more information, visit www.formasterminds.com.

The source code for this eBook is available at **www.formasterminds.com**

Copyright Registration Number: 1140725

1st Edition 2018

What is Inside

This guide will teach you how to use the Web Socket API to connect your website or web application to a Web Socket server. After reading this guide, you will know how to establish communication with a Web Socket server, how to send and receive messages, and how to control the process with events.

About this Guide

This guide is a collection of excerpts from the book HTML5 for Masterminds. The information included in this guide will help you understand a particular aspect of web development, but it will not teach you everything you need to know to develop a website or a web application. If you need a complete course on web development, read our book HTML5 for Masterminds. For more information, visit our website at www.formasterminds.com.

What Do You Need

This guide assumes that you have a basic knowledge of HTML, CSS and JavaScript, and you know how to create files and upload them to a server. If you don't know how to program in HTML, CSS or JavaScript, you can download our guides Introduction to HTML, Introduction to CSS, and Introduction to JavaScript. For a complete course on web development, read our book HTML5 for Masterminds.

IMPORTANT: We recommend you to execute the examples in this guide with the latest versions of Google Chrome and Mozilla Firefox (www.google.com/chrome and www.mozilla.com). You can also check the state of the current implementations at www.caniuse.com. To find examples, resources, links and videos, visit our website at www.formasterminds.com.

The Basics: To create the files for the examples of this guide, you can use a text editor (Notepad or Text Edit), or a professional editor like Atom (www.atom.io). If you do not have a server to test the files, you can install a server in your computer with a package like MAMP (www.mamp.info). For more information, read our free guide Web Development.

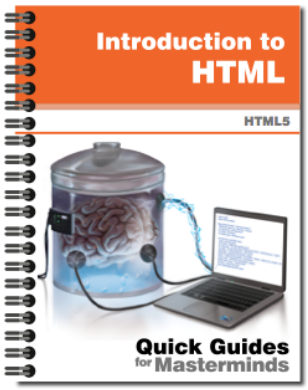
Recommendations

Introduction to JavaScript

Quick Guides for Masterminds

This guide will teach you how to program with JavaScript. After reading this guide, you will know how to create a program in JavaScript, how to define functions and objects, and how to read and modify an HTML document dynamically.

More Information

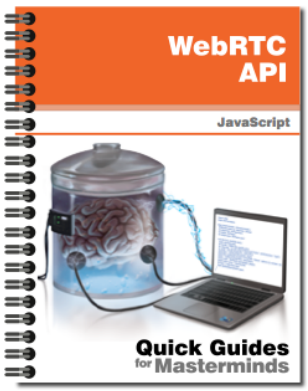


Introduction to HTML

Quick Guides for Masterminds

This guide will teach you how to create your website's documents with HTML. After reading this guide, you will know how to work with HTML elements, how to define a document's structure, and how to organize its content.

More Information



WebRTC API

Quick Guides for Masterminds

This guide will teach you how to use the WebRTC API to establish peer-to-peer communications. After reading this guide, you will know how to connect your users with each other, how to create a system to let your users perform video calls, and how to transfer data from one user to another.

More Information

More Guides Available at www.formasterminds.com

Table of Contents

WebSocket API

Web Sockets

The WebSocket API provides support for faster and more efficient bi-directional communication between browsers and servers. The connection is established through a TCP socket without sending HTTP headers, thus reducing the size of data transmitted in every call. The connection is also persistent, allowing servers to keep clients updated without a previous request, which means we do not have to call the server for updates. Instead, the server itself automatically sends us information about the current situation.

WebSocket may look like an improved version of Ajax, but it is an entirely different way of communication that allows the construction of real-time applications in a scalable platform, such as multiplayer video games, chat rooms, etc.

The API is simple; a few methods and events are included to open and close the connection and send and listen to messages. However, no server is configured to provide this service, and the response has to be customized to our needs, so we have to install our own WS server (WebSocket server) to establish the communication between the browser and the server that allocates our application.

WebSocket Server

Although we can build our own WS server, there are several programs available to configure a server and have it ready to process requests. Depending on our preferences, we can opt for codes written in PHP, Java, Ruby or other languages. For the purpose of this guide, we are going to use a PHP server. There are several versions available in this language, but the easier to install and configured is a server called *phpws*, developed by Chris Tanaskoski.

> **IMPORTANT:** The phpws server requires at least PHP version 5.3 to run correctly, and your hosting account has to include shell access to be able to communicate with your server and run the PHP code (you can ask your hosting provider to enable shell access if you do not have it by default).

The phpws server includes several files to create the classes and methods we need to run the server. The library is available at https://github.com/Devristo/phpws/, but for testing purposes, we have included a package on our website already configured to install the WS server. The package includes a file called demo.php that contains a function called **onMessage()** where all the messages received by the server are processed. If we want to provide our own response, we have to modify this function. The following is a version of the function we have developed for the examples of this guide.

Listing 1: *Adapting the* onMessage() *function to our application (demo.php)*

```
public function onMessage(IWebSocketConnection $user,
IWebSocketMessage $msg){
  $msg = trim($msg->getData());
  switch($msg){
    case 'hello':
      $msgback = WebSocketMessage::create("hello human");
      $user->sendMessage($msgback);
      break;
    case 'name':
      $msgback = WebSocketMessage::create("I don't have a
name");
      $user->sendMessage($msgback);
      break;
    case 'age':
      $msgback = WebSocketMessage::create("I'm old");
      $user->sendMessage($msgback);
      break;
    case 'date':
      $msgback = WebSocketMessage::create("today is
".date("F j, Y"));
      $user->sendMessage($msgback);
      break;
    case 'time':
      $msgback = WebSocketMessage::create("the time is
".date("H:iA"));
      $user->sendMessage($msgback);
      break;
    case 'thanks':
      $msgback = WebSocketMessage::create("you're
welcome");
      $user->sendMessage($msgback);
      break;
    case 'bye':
      $msgback = WebSocketMessage::create("have a nice
day");
      $user->sendMessage($msgback);
      break;
    default:
      $msgback = WebSocketMessage::create("I don't
understand");
      $user->sendMessage($msgback);
      break;
  }
```

```
}
```

Once we have all these files on our server, it is time to run the WS server. WebSocket uses a persistent connection, so the WS server has to be running all the time, receiving and sending messages to users. To execute the PHP file, we have to access our server using an SSH connection, find the folder where the demo.php file is located and insert the command **php demo.php**.

Do It Yourself: Download the ws.zip file from our website, decompress it, and upload the ws folder and all of its files to your server. Inside this folder, you will find the demo.php file with the **onMessage()** function already updated with the code of Listing 1. Connect to your server with SSH using your favorite program (Terminal, Putty, etc.), find the ws folder, and execute the command **php demo.php** to run the WS server.

IMPORTANT: You can also use a public WS server, such as ws://echo.websocket.org/ (more information at http://websocket.org/echo.html). Usually, these servers do not include any built-in commands and return the message received to the same user, but are useful for testing purposes.

The Basics: SSH is a network protocol (Secure Shell) that you can use to access your server and control it remotely. It allows you to work with folders and files on your server and run programs. The most popular applications that provide a console for Shell access are Terminal for Apple and *PuTTY for Windows* (available for free at www.chiark.greenend.org.uk/~sgtatham/putty/).

Connecting to the Server

With the server running, we now have to program the JavaScript code that will connect to it. For this purpose, the API offers an object called **WebSocket** with a few properties, methods, and events to set up the connection. The following is the object's constructor.

WebSocket(url**)**—This constructor initiates a connection between the application and the WS server targeted by the **url** attribute. It returns a **WebSocket** object referencing the connection. A second attribute may be specified to provide an array with communication sub-protocols.

The connection is initiated by the constructor, so there are only two methods to work with it.

send(data**)**—This method sends a message to the WS server. The **data** attribute is a string with the information to be transmitted.

close()—This method closes the connection.

A few properties inform about the connection's configuration and status.

url—This property returns the URL of the file to which the application is connected.

protocol—This property returns the sub-protocol used in the connection.

readyState—This property returns a number representing the state of the connection: 0 meaning the connection has not yet been established, 1 meaning the connection is opened, 2 meaning the connection is being closed, and 3 meaning the connection was closed.

bufferedAmount—This property reports the amount of data requested by the connection but not yet sent to the server. The value returned helps us regulate the amount of data and frequency of the requests in order not to saturate the server.

The API provides the following events to know the status of the connection and listen to messages sent by the server.

open—This event is fired when the connection is opened.

message—This event is fired when there is a message from the server available.

error—This event is fired when an error occurs.

close—This event is fired when the connection is closed.

The demo.php file we prepared for these examples has a method called **onMessage()** that processes a small list of predefined commands and sends back the proper answer (see Listing 1). For testing purposes, we are going to use a form to insert and send these commands to the server.

Listing 2: *Creating the document to insert commands*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>WebSocket</title>
  <link rel="stylesheet" href="websocket.css">
  <script src="websocket.js"></script>
</head>
<body>
  <section id="formbox">
    <form name="form">
      <label for="command">Command: </label><br>
      <input type="text" name="command" id="command"><br>
```

```
      <button type="button" id="button">Send</button>
    </form>
  </section>
  <section id="databox"></section>
</body>
</html>
```

We will also need a CSS file with the following styles to design the boxes on the page.

Listing 3: *Defining the styles for the boxes*

```
#formbox {
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#databox {
  float: left;
  width: 500px;
  height: 350px;
  overflow: auto;
  margin-left: 20px;
  padding: 20px;
  border: 1px solid #999999;
}
```

Of course, the JavaScript code is responsible for the entire process. The following example establishes a simple communication with the server to test the API.

Listing 4: *Sending messages to the server*

```
var databox, socket;
function initiate() {
  databox = document.getElementById("databox");
  var button = document.getElementById("button");
  button.addEventListener("click", send);

  socket = new
WebSocket("ws://YOUR_IP_ADDRESS:12345/ws/demo.php");
  socket.addEventListener("message", received);
}
function received(event) {
  var list = databox.innerHTML;
  databox.innerHTML = "Received: " + event.data + "<br>" +
list;
}
```

```
function send() {
  var command = document.getElementById("command").value;
  socket.send(command);
}
window.addEventListener("load", initiate);
```

IMPORTANT: You have to replace **YOUR_IP_ADDRESS** with the IP of your server. You can always use your domain instead, but using the IP eludes DNS translation. You should always use this technique to access your application to avoid the time spent by the network translating the domain into the corresponding IP address. Your server has to have the port 12345 open to use the JavaScript codes and the WS server provided for this guide. If you are not sure how to open this port, ask your hosting provider.

In the code of Listing 4, we add a listener for the **load** event to the window to execute a function called **initiate()** as soon as the document is loaded. In this function, the **WebSocket** object is constructed and stored in the **socket** variable. The **url** attribute declared in the constructor points to the location of the demo.php file in our server. This URL includes the port of connection. Usually, the host is specified with the IP number of the server, and the value of the port is 12345, but that depends on our needs, the configuration of our server, the ports available, the location of the file, etc.

After we get the **WebSocket** object, we add a listener for the **message** event. Every time the WS server sends a message to the browser, the **message** event is fired, and the **received()** function is called to respond. The object sent by this event to the function includes the **data** property that contains the content of the message. In the **received()** function, we use that property to show the message on the screen.

The **send()** function is included to send messages to the server. The value of the **<input>** element is taken by this function and sent to the WS server using the **send()** method.

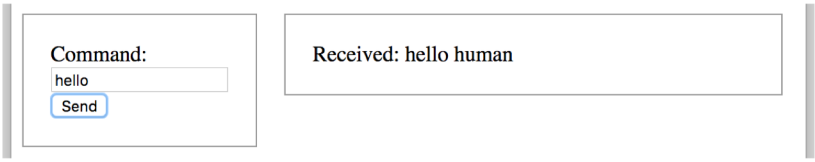


Figure 1: *Application talking with the WS server*

Do It Yourself: Create a new HTML file with the document of Listing 2, a CSS file called websocket.css with the code in Listing 3, and a JavaScript file called websocket.js with the code in Listing 4. Open the document in your browser, insert the command **hello** in the input field, and press the Send button. You should see a message on the right box with the response from the server, as illustrated in Figure 1 (your WS server has to be installed and running, as explained before).

IMPORTANT: The **onMessage()** function we prepared for these examples checks the message received and compares its value with a list of predefined commands (see Listing 1). The commands available are **hello**, **name**, **age**, **date**, **time**, **thanks** and **bye**.

The last example illustrates how the communication process works for this API. The connection is started by the **WebSocket()** constructor, the **send()** method sends every message we want to be processed to the server, and the **message** event informs the application of the arrival of new messages from the server. However, we did not close the connection, check for errors or even detect when the connection was ready to work. The following example listens to all the events provided by the API to inform the user about the status of the connection at every step of the process.

Listing 5: *Informing the user about the state of the connection*

```
var databox, socket;
function initiate() {
  databox = document.getElementById("databox");
  var button = document.getElementById("button");
  button.addEventListener("click", send);

  socket = new
WebSocket("ws://YOUR_IP_ADDRESS:12345/ws/demo.php");
  socket.addEventListener("open", opened);
  socket.addEventListener("message", received);
  socket.addEventListener("close", closed);
  socket.addEventListener("error", showerror);
}
function opened() {
  databox.innerHTML = "CONNECTION OPENED<br>";
  databox.innerHTML += "Status: " + socket.readyState;
}
function received(event) {
  var list = databox.innerHTML;
  databox.innerHTML = "Received: " + event.data + "<br>" +
list;
}
function closed() {
  var list = databox.innerHTML;
  databox.innerHTML = "CONNECTION CLOSED<br>" + list;

  var button = document.getElementById("button");
  button.disabled = true;
}
function showerror() {
```

```
  var list = databox.innerHTML;
  databox.innerHTML = "ERROR<br>" + list;
}
function send() {
  var command = document.getElementById("command").value;
  if (command == "close") {
    socket.close();
  }else{
    socket.send(command);
  }
}
window.addEventListener("load", initiate);
```

There are a few improvements in the code in Listing 5 from the previous example. Listeners for all the events available in the **WebSocket** object are added, and the proper functions to handle the events are created. We also show the status when the connection is opened using the value of the **readyState** property, close the connection using the **close()** method when the "close" command is sent from the form, and disable the Send button when the connection is closed (**button.disabled = true**).

Figure 2: *Connection closed*

Do It Yourself: This last example requires the HTML document and CSS styles of Listings 2 and 3. Open the document in your browser, insert a command in the form, and click the Send button. You should get answers from the server according to the command inserted (**hello**, **name**, **age**, **date**, **time**, **thanks** or **bye**). Send the command "close" to close the connection.

IMPORTANT: Remember that your WS server has to be running all the time to process requests. As we have discussed, if you wish, you may test these examples with a public WS server, such as ws://echo.websocket.org/. This server sends back to the browser the same text inserted in the form.

Quick Reference

Methods

WebSocket(url**)**—This constructor initiates a connection between the application and the WS server targeted by the **url** attribute. It returns a **WebSocket** object referencing the connection. A second attribute may be specified to provide an array with communication sub-protocols.

send(data**)**—This method sends a message to the WS server. The **data** attribute is a string with the information to be transmitted.

close()—This method closes the connection.

Properties

url—This property returns the URL of the file to which the application is connected.

protocol—This property returns the sub-protocol used in the connection.

readyState—This property returns a number representing the state of the connection: 0 meaning the connection has not yet been established, 1 meaning the connection is opened, 2 meaning the connection is being closed, and 3 meaning the connection was closed.

bufferedAmount—This property reports the amount of data requested by the connection but not yet sent to the server. The value returned helps us regulate the amount of data and frequency of the requests in order not to saturate the server.

Events

open—This event is fired when the connection is opened.

message—This event is fired when there is a message from the server available.

error—This event is fired when an error occurs.

close—This event is fired when the connection is closed.

For Masterminds

Book Series

for more Books and Quick Guides visit

www.formasterminds.com