Getting Started with HTML5 WebSocket Programming

Develop and deploy your first secure and scalable real-time web application

Vangos Pterneas

Getting Started with HTML5 WebSocket Programming

Develop and deploy your first secure and scalable real-time web application

Vangos Pterneas

[PACKT] PUBLISHING

open source*
community experience distilled

BIRMINGHAM - MUMBAI

Getting Started with HTML5 WebSocket Programming

Credits

Author
Vangos Pterneas

Reviewers
Sann-Remy Chea
Wayne Ye

Acquisition Editor
Rubal Kaur

Lead Technical Editor
Mohammed Fahad

Technical Editor
Manal Pednekar

Project Coordinator
Akash Poojary

Proofreader
Lucy Rowland

Indexer
Tejal Soni

Graphics
Abhinash Sahu

Production Coordinator
Aditi Gajjar

Cover Work
Aditi Gajjar

About the Author

Vangos Pterneas is a Software Engineer, passionate about natural user interfaces and modern innovative technologies. He loves developing smart clients for the Web and mobile devices. His professional experience includes iOS, Windows, Kinect, and HTML5 development for small and large-scale systems.

Vangos has worked as a Software Engineer and Consultant for Microsoft Innovation Center, where he participated in EU research projects and carried out numerous technical presentations and workshops. He is now running his own company, LightBuzz Software, introducing new concepts and software to the public. LightBuzz applications have won the first place in Microsoft's worldwide innovation competition, held in New York, and also the first place in TEDx's Rising Stars program.

Apart from this book, Vangos has reviewed *Augmented Reality with Kinect*, published by Packt Publishing.

When Vangos is not coding, he loves blogging about technical staff and providing the community with open-source utilities (`http://lightbuzz.com`)

> I would like to thank my kitty and all of my fluffy cats (Pixel, Vector, and Apollo) for their patience and support.

About the Reviewers

Sann-Remy Chea works as a Software Engineer at Ubisoft Owlient, a video game company specialized in Web games, based in Paris, France. During his Master's degree, he created two games, which have reached thousands of players. As an intern, he worked at Ubisoft and then joined IBM in the Media & Entertainment industry. Fond of Web application development, he specializes in HTML5 and mainly develops in JavaScript. You can follow Sann-Remy on Twitter: `@srchea`.

Wayne Ye is a Software Developer, Tech Lead, and also a Geek. He has immersed himself in software development for nearly 8 years, with an emphasis on C#/ASP. NET, Ruby on Rails, HTML5, JavaScript/jQuery, and nodejs. He is proficient in GOF Design Patterns, S.O.L.I.D principle, MVC/MVVM, SOA, REST, and AOP; he strongly believes in and masters Agile, Scrum, and TDD/BDD. He hacks daily with Vim. He is a CodeProject MVP (2012) and a certified PMP. In his spare time, he frequently writes tech/life blogs on `WayneYe.com`, and spends wonderful time with his dear wife and lovely son.

He works as a global leader in 3D design, engineering, and entertainment software. Autodesk helps people imagine, design, and create a better world. Autodesk offers an unparalleled depth of experience and a broad portfolio of software to give customers the power to solve their design, business, and environmental challenges. In addition to designers, architects, engineers, and media and entertainment professionals, Autodesk helps students, educators, and casual creators unlock their creative ideas through user-friendly applications.

Wayne is also the author of *Cucumber BDD How-To*, published by Packt Publishing.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

The WebSocket protocol is the art of handshaking in the HTML5 world. It defines a two-way communication between server and client, resulting in smoother, faster, and more efficient web applications. This book will guide you through the whole process of creating a modern web app, taking full advantage of the WebSocket's capabilities. You will learn, step-by-step, how to configure the client and server, transfer text and multimedia, add security layers, and provide fallbacks for older browsers. Moreover, you will get a taste of how these techniques work in a native mobile and tablet client, unleashing the complete power of the HTML5 WebSocket protocol.

What this book covers

Chapter 1, *WebSocket – a Handshake!*, provides a brief yet compact introduction to the WebSocket protocol, specifies the need for bi-directional communication in the Web, and showcases some inspiring real-world examples.

Chapter 2, *The WebSocket API*, highlights the fundamental concepts of the WebSocket API and demonstrates a WebSocket web client application.

Chapter 3, *Configuring the Server*, implements the server-side functionality, which is crucial for effectively achieving truly two-way communication.

Chapter 4, *Data Transfer – Sending, Receiving, and Decoding*, shows how the WebSocket handles different data types such as text, images, and multimedia.

Chapter 5, *Security*, examines some common security risks when running a WebSocket app and provides ways to ensure the stability of the system.

Chapter 6, *Error Handling and Fallbacks*, answers what to do when something goes wrong and how to emulate the WebSocket behavior when dealing with older browsers.

Chapter 7, *Going Mobile (and Tablet, Too)*, extends the WebSocket functionality to the mobile world and shows how a WebSocket app can run natively on an iPhone or iPad.

Appendix, provides some further resources, including interesting and controversial articles.

What you need for this book

To get the best out of this book, you need a modern web browser and a text editor. Just to make life easier, here are a few software requirements that will help you build and debug your WebSocket applications:

- The latest version of Google Chrome, Internet Explorer, Mozilla Firefox, or Opera, including their developer tools
- A text editor such as Aptana or WebMatrix

Considering the server-side examples, if you choose to use our C# code, you need:

- .NET Framework 3.5 or later
- Visual Studio 2010 or later

Finally, considering the mobile and tablet examples, if you choose to deploy on iOS, you need:

- Mac OS X 10.7 or later
- XCode 4.5 or later
- Apple developer license

Feel free to choose your preferred server-side, mobile and tablet technologies. The main methodologies and techniques remain the same regardless of the various tools and SDKs.

Who this book is for

This book is intended for professional software developers, researchers, and students who are interested in developing modern web applications. Basic knowledge of HTML, JavaScript, and at least one server-side technology is required. If you want to get the most out of the mobile and tablet chapter, a good knowledge of any mobile platform would be a plus. This book intends to guide you through the principles and fundamentals of WebSocket programming, so you can apply this knowledge on every platform you have expertise in.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: " You have probably noticed that we use the `echo.websocket.org` server for this demo".

A block of code is set as follows:

```
h1 {
  color: blue;
  text-align: center;
  font-family: "Helvetica Neue", Arial, Sans-Serif;
  font-size: 1em;
}
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: " On the **Solution Explorer** tab, right-click on the **References** icon and select **Add new reference**".

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
WebSocket – a Handshake!

In real life, handshaking is the act of gently grasping two people's hands, followed by a brief up and down movement. If you have ever greeted someone this way, then you already understand the basic concept of the HTML5 WebSocket protocol.

WebSockets define a persistent two-way communication between web servers and web clients, meaning that both parties can exchange message data at the same time. WebSockets introduce true concurrency, they are optimized for high performance, and result in much more responsive and rich web applications.

The following diagram shows a server handshake with multiple clients:
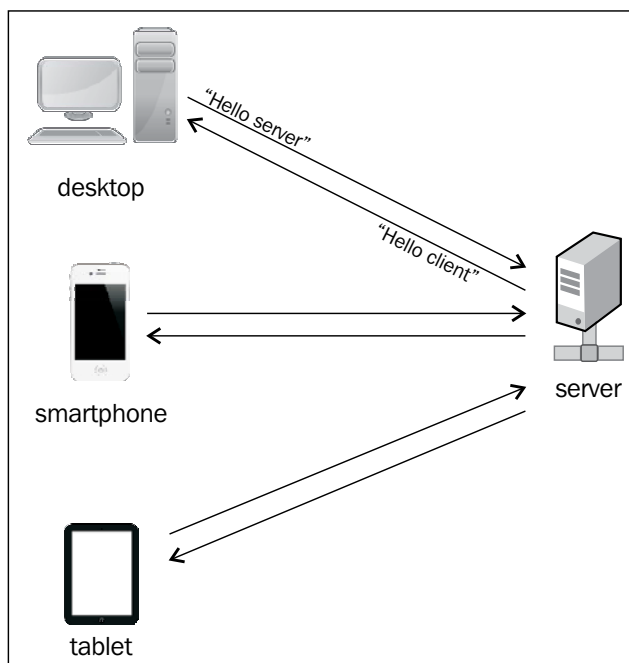
For the record, the WebSocket protocol has been standardized by the **Internet Engineering Task Force** (**IETF**) and the WebSocket API for web browsers is currently being standardized by the **World Wide Web Consortium** (**W3C**)—yes, it's a work in progress. No, you do not need to worry about enormous changes, as the current specification has been published as "proposed standard".

Life before WebSocket

Before diving into the WebSockets' world, let's have a look at the existing techniques used for bidirectional communication between servers and clients.

Polling

Web engineers initially dealt with the issue using a technique called polling. Polling is a synchronous method (that is, no concurrency) that performs periodic requests, regardless whether data exists for transmission. The client makes consecutive requests after a specified time interval. Each time, the server responds with the available data or with a proper warning message.

Though polling "just works", it is easy to understand that this method is overkill for most situations and extremely resource consuming for modern web apps.

Long polling

Long polling is a similar technique where, as its name indicates, the client opens a connection and the server keeps the connection active until some data is fetched or a timeout occurs. The client can then start over and perform a sequential request. Long polling is a performance improvement over polling, but the constant requests might slow down the process.

Streaming

Streaming seemed like the best option for real-time data transmission. When using streaming, the client performs a request and the server keeps the connection open indefinitely, fetching new data when ready. Although this is a big improvement, streaming still includes HTTP headers, which increase file size and cause unnecessary delays.

Postback and AJAX

The web has been built around the HTTP request-response model. HTTP is a stateless protocol, meaning that the communication between two parts consists of independent pairs of requests and responses. In plain words, the client asks the server for some information, the server responds with the proper HTML document and the page is refreshed (that's actually called a postback). Nothing happens in between, until a new action is performed (such as the click of a button or a selection from a drop-down menu). Any page load is followed by an annoying)(in terms of user experience) flickering effect.

It was not until 2005 that the postback flickering was bypassed thanks to **Asynchronous JavaScript and XML** (**AJAX**). AJAX is based on the JavaScript's `XmlHttpRequest` Object and allows asynchronous execution of JavaScript code without interfering with the rest of the user interface. Instead of reloading the whole page, AJAX sends and receives back only a portion of the web page.

Imagine you are using Facebook and want to post a comment on your Timeline. You type a status update in the proper text field, hit Enter and... voila! Your comment is automatically published without a single page load. Unless Facebook used AJAX, the browser would need to refresh the whole page in order to display your new status.

AJAX, accompanied with popular JavaScript libraries such as jQuery, has strongly improved the end user experience and is widely considered as a must-have attribute for every website. It was only after AJAX that JavaScript became a respectable programming language, instead of being thought of as a necessary evil.

But it's still not enough. Long polling is a useful technique that makes it seem like your browser maintains a persistent connection, while the truth is that the client makes continuous calls! This might be extremely resource-intensive, especially in mobile devices, where speed and data size really matter.

All of the methods previously described provide real-time bidirectional communication, but have three obvious disadvantages in comparison with WebSockets:

- They send HTTP headers, making the total file size larger
- The communication type is half duplex, meaning that each party (client/server) must wait for the other one to finish
- The web server consumes more resources

The postback world seems like a walkie-talkie—you need to wait for the other guy to finish speaking (half-duplex). In the WebSocket world, the participants can speak concurrently (full-duplex)!

The web was initially built for displaying text documents, but think how it is used today. We display multimedia content, add location capabilities, accomplish complex tasks and, hence, transmit data different than text. AJAX and browser plugins such as Flash are all great, but a more native way of doing things is required. The way we use the web nowadays bears the need for a holistic new application development framework.

Then came HTML5

HTML5 makes a huge, yet justifiable, buzz nowadays as it introduces vital solutions to the problems discussed previously. If you are already familiar with HTML5, feel free to skip this section and move on.

HTML5 is a robust framework for developing and designing web applications.

HTML5 is not just a new markup or some new styling selectors, neither is it a new programming language. HTML5 stands for a collection of technologies, programming languages and tools, each of which has a discrete role and all of these together accomplish a specific task—that is, to build rich web apps for any kind of device.

The main HTML5 pillars include Markup, CSS3, and JavaScript APIs, together.

The following diagram shows HTML5 components:
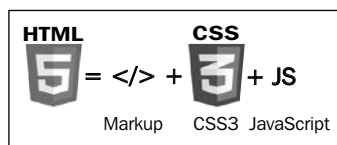
Here are the dominant members of the HTML5 family. As this book does not cover the whole set of HTML5, I suggest you visit `html5rocks.com` and get started with hands-on examples and demos.

Markup	Structural elements
	Form elements
	Attributes
Graphics	Style sheets
	Canvas
	SVG
	WebGL

Multimedia	Audio
	Video
Storage	Cache
	Local storage
	Web SQL
Connectivity	WebMessaging
	WebSocket
	WebWorkers
Location	Geolocation

Although Storage and Connectivity are supposed to be the most advanced topics, you do not need to worry if you are not an experienced web developer. Throughout this book, we will explain how to accomplish common tasks and we'll create some step-by-step examples, which you can later download and experiment with. Moreover, managing WebSockets via the HTML5 API is pretty simple to grasp, so take a deep breath and dive in with no fear.

The WebSocket protocol

The WebSocket protocol redefines full-duplex communication from the ground up. Actually, WebSockets, along with WebWorkers, take a really enormous step in bringing desktop-rich functionality to web browsers. Concurrency and multi-threading did not truly exist in the postback world. They were emulated in a rather restrictive manner.

The URL

HTTP protocol requires its own schemas (http and https). So does the WebSocket protocol. Here is a typical WebSocket URL example:

```
ws://example.com:8000/chat.php
```

The first thing to notice is the `ws` prefix. This is pretty normal, as we need a new URL schema for the new protocol. `wss` is supported as well and is the WebSocket equivalent to https for secure connections (SSL). The rest of the URL is similar to the plain old HTTP URLs and is illustrated in the following image.

The following image shows the WebSocket URL in tokens:

```
schema      host        port        server
ws://example.com:8000/chat.php
```

Browser support

For the time being, the latest specification of the WebSocket protocol is RFC 6455 and it's a blessing that the latest versions of every modern web browser support it. More specifically, the RFC 6455 is supported in the following browsers:

- Internet Explorer 10+
- Mozilla Firefox 11+
- Google Chrome 16+
- Safari 6+
- Opera 12+

It is worth mentioning that the mobile versions of Safari (for iOS), Firefox (Android), Chrome (Android, iOS), and Opera Mobile all support WebSockets, bringing the WebSocket power to smartphones and tablets!

But, wait. What about the older browser versions that many people still use worldwide? Well, no need to worry, as throughout this book, we'll have a look at some fallback techniques that make our websites accessible to the largest audience possible.

Who's using WebSockets

Although WebSocket is a brand-new technology, quite many promising companies utilize its various capabilities in order to deliver a richer experience to their users. The most well-known paradigm is Kaazing (`http://demo.kaazing.com/livefeed/`), a startup that raised an investment of 17 million dollars for its real-time communication platform.

Other businesses include the following:

Name	Website	Description
Gamooga	`http://www.gamooga.com/`	Real-time backend for apps and games

Name	Website	Description
GitLive	`http://gitlive.com/`	Notifications on GitHub projects
Superfeedr	`http://superfeedr.com/`	Real-time data pushing
Pusher	`http://pusher.com/`	Scalable real-time functionality API for web and mobile apps
Smarkets	`https://smarkets.com/`	Real-time betting
IRC Cloud	`https://www.irccloud.com/`	Chatting

Two great resources containing a large variety of WebSocket demos are as follows:

- `http://www.websocket.org/demos.html`
- `http://www.html5rocks.com/en/features/connectivity`

Mobile?

WebSockets, as the name indicates, are related to the web. As you know, the web is much more than a bunch of techniques for some browsers; rather, it's a broad communication platform for a vast number of devices, including desktop computers, smartphones, and tablets.

Obviously, any HTML5 app that utilizes WebSockets will work on (almost) any HTML5-enabled mobile web browser. Imagine you want to implement the same functionality using the enhanced features of a native mobile app. Is the WebSocket supported in the mainstream mobile operating systems? The short answer: yes. Currently, all key players in the mobile industry (Apple, Google, Microsoft) provide a WebSocket API you can use in your own native apps. iOS, Android, and Windows smartphones and tablets integrate WebSockets in a similar way to HTML5.

The future is now

New neuroscience research confirms the old adage about the power of a handshake: people do form a better impression of those who proffer their hand in greeting (`http://www.sciencedaily.com/releases/2012/10/121019141300.htm`). As a human handshake can lead to better deals, so a WebSocket handshake can lead to better user experience. We investigate user experience as a combination of performance (the user is waiting less) and simplicity (the developer builds straight and quick).

So, it's up to you: do you want to build modern, truly real-time web applications? Do you want to provide your users with the maximum experience? Do you want to offer a terrific performance boost to your existing web apps? If the answer to any of these questions is yes, then it's time to realize that the WebSocket API is mature enough to offer its goodies right here right now.

What are we going to make?

Throughout this book, we are going to implement a real-world project: a simple, multi-user, WebSocket-based, chatting application. Live chat is a very common feature among all modern social networks. We will learn, step-by-step, how to configure the web server, implement the HTML5 client, and transfer messages between them.

Apart from plain text messages, we'll see how WebSockets handle various types of data, such as binary files, images, and videos. Yeah, we'll demonstrate real-time media streaming, too!

Moreover, we are going to enhance the security of our app, examine some known security risks and find out how to avoid common pitfalls. Furthermore, we'll take a glance at some fallback techniques targeting those poor guys who cannot (or do not want to) update their browsers yet.

Last but not least, we'll get mobile. You chat using a desktop browser, a phone, or a tablet. Wouldn't it be nice if you could use the same techniques and principles across multiple targets? Well, through reading this book, you'll find out how to easily convert your web app into a native mobile and tablet application as well.

Summary

In this first chapter we introduced the WebSocket protocol, mentioned the existing techniques for real-time communication and determined the specific needs that WebSockets fulfill. Moreover, we examined its relationship with HTML5 and illustrated how the users can benefit from such enhancements. It's now time to introduce the WebSocket client API in more detail.

2
The WebSocket API

If you are familiar with HTML and JavaScript, you already know enough stuff to start developing HTML5 WebSockets right now. WebSocket communication and data transmission is bidirectional, so we need two parties to establish it: a server and a client. This chapter focuses on the HTML5 web client and introduces the WebSocket client API.

HTML5 basics

Any HTML5 web client is a combination of structure, styling, and programming logic. As we have already mentioned, the HTML5 framework provides discrete sets of technologies for each use. Although we assume that you are already slightly familiar with these concepts, let's have a quick look at them.

Markup

The markup defines the structure of your web application. It is a set of XML tags that lets you specify the hierarchy of the visual elements within an HTML document. Popular new HTML5 tags include the `header`, `article`, `footer`, `aside`, and `nav` tags. The elements have a specific meaning and help distinguish the different parts of a web document.

Here is a simple example of HTML5 markup code that generates the essential elements for our chatting app: a text field, two buttons, and a label. The text field is used for typing our message, the first button will send the message, the second button will terminate the chat, and the label will display the interactions coming from the server:

```
<!DOCTYPE html>
<head>
  <title>HTML5 WebSockets</title>
```

```
</head>
<body>
  <h1> HTML5 WebSocket chat. </h1>
  <input type="text" id="text-view" />
  <input type="button" id="send-button" value="Send!" />
  <input type="button" id="stop-button" value="Stop" />
  <br/>
  <label id="status-label">Status</label>
</body>
```

The first line of the preceding code (the DOCTYPE) indicates that we are using the latest version of HTML, which is HTML5.

For more information about the HTML5 markup, consider visiting `http://html5doctor.com/`. There is a complete reference for the supported HTML5 tags at `http://html5doctor.com/element-index/`.

Styling

In order to display colors, backgrounds, fonts, alignments, and so on, you need to be familiar with **Cascading Style Sheets** (**CSS**). CSS is quite self-explanatory, so, if you want to change the header style (for example color, alignment, and font), you would write something similar to the following code:

```css
h1 {
  color: blue;
  text-align: center;
  font-family: "Helvetica Neue", Arial, Sans-Serif;
  font-size: 1em;
}
```

`http://www.css3.info/` is a great resource for CSS3 and further reading.

Logic

The markup defines the structure and the CSS rules apply the styling. What about event handling and user actions? Well, here comes JavaScript! JavaScript is a scripting programming language that lets you control and alter the behavior of your web app according to the accompanying actions. Using JavaScript, you can handle button clicks, page loads, apply addition styling, add special effects, or even fetch data from web services. Using JavaScript, you can create objects, assign them properties and methods, and raise and catch events when something occurs.

Following is a simple JavaScript example:

```
var buttonSend = document.getElementById("send-button");

buttonSend.onclick = function() {
  console.log("Button clicked!");
}
```

The first line searches the document tree, finds the element named `action-button` and stores it in an object named `buttonSend`. Then, a function is assigned to the onclick event of the button. The body of the function is executed every time the button is clicked on.

The brand-new HTML5 features are heavily based on JavaScript, so a basic knowledge of this language is essential before implementing any web app. Most importantly, the WebSocket API is pure JavaScript, too!

A chatting application

The most popular kind of full-duplex communication is chatting. We'll start the development of a simple chatting application right here. First thing to do is configure the client side, which consists of three basic files:

- An HTML (`.html`) file containing the markup structure of the web page
- A CSS (`.css`) file containing all the styling information
- A JavaScript (`.js`) file containing the logic of the application

Currently, that's all you need to have for a full-featured HTML5 chat client. No browser plugins or other external libraries are required.

API overview

API, which stands for **Application Programming Interface**, is a set of objects, methods, and routines that let you interact with the underlying layer of functionality. Considering the WebSocket protocol, its API includes the WebSocket primary object, events, methods, and attributes.

Translating these characteristics into actions, the WebSocket API allows you to connect to a local or remote server, listen for messages, send data, and close the connection.

Here is a typical usage of the WebSocket API.

The following illustration shows the typical WebSocket workflow:

```
Check for WebSocket browser support ->
Create an instance of the WebSocket JS object ->
Connect to the WebSocket server ->
Register for the WebSocket events ->
Perform the proper data transmission according to the users'
actions ->
Close the connection
```

Browser support

The WebSocket protocol is a new HTML5 feature, so not every browser supports it yet. If you ever tried to run WebSocket-specific code on a browser that is not supported, nothing would happen. Think of your users: it wouldn't be nice for them to surf on an unresponsive site. Moreover, you wouldn't like to miss any potential customers!

As a result, you should check for browser compatibility before running any WebSocket code. If the browser cannot run the code, you should provide an error message or a fallback, such as AJAX or Flash-based functionality. There will be more on fallbacks in *Chapter 6*, *Error Handling and Fallbacks*. I also like providing messages that gently prompt my users to update their browser.

JavaScript provides an easy way to find out whether a browser can execute WebSocket-specific code:

```
if (window.WebSocket) {
  console.log("WebSockets supported.");

  // Continue with the rest of the WebSockets-specific functionality…
}
else {
  console.log("WebSockets not supported.");
  alert("Consider updating your browser for a richer experience.");
}
```

The `window.WebSocket` statement indicates whether the WebSocket protocol is implemented in the browser. The following statements are equivalent:

```
window.WebSocket
```

```
"WebSocket" in window
```

```
window["WebSocket"]
```

Each one of them results in the same validation check. You can also check about any feature support using your browser's developer tools.

Want to see which browsers do support the WebSocket protocol? There is an up-to-date resource available at `http://caniuse.com/#feat=websockets`.

At the time of writing, WebSocket is fully supported by Internet Explorer 10+, Firefox 20+, Chrome 26+, Safari 6+, Opera 12.1+, Safari for iOS 6+, and Blackberry Browser 7+.

The WebSocket object

It's now time to initialize a connection to the server. All we need is to create a WebSocket JavaScript object, providing the URL to the remote or local server:

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

When this object is constructed, it immediately opens a connection to the specified server. *Chapter 3*, *Configuring the Server*, will show us in detail how we can develop the server-side program. For now, just keep in mind that a valid WebSocket URL is necessary.

The example URL `ws://echo.websocket.org` is a public address that we can use for testing and experiments. The `Websocket.org` server is always up and running and, when it receives a message, it sends it back to the client! It's all we need in order to ensure that our client-side application works properly.

Events

After creating the `WebSocket` object, we need to handle the events it exposes. There are four main events in the WebSocket API: Open, Message, Close, and Error. You can handle them either by implementing the `onopen`, `onmessage`, `onclose`, and `onerror` functions respectively, or by using the `addEventListener` method. Both ways are almost equivalent for what we need to do, but the first one is much clearer.

Note that, obviously, the functions we'll provide to our events will not be executed consecutively. They will be executed asynchronously when a specific action occurs.

So, let's have a closer look at them.

onopen

The onopen event is raised right after the connection has been successfully established. It means that the initial handshake between the client and the server has led to a successful first deal and the application is now ready to transmit data:

```
socket.onopen = function(event) {
  console.log("Connection established.");

  // Initialize any resources here and display some user-friendly
messages.
  var label = document.getElementById("status-label");
  label.innerHTML = "Connection established!";
}
```

It's a good practice to provide your users with the appropriate feedback while they are waiting for the connection to open. WebSockets are definitely fast, but the Internet connection might be slow!

onmessage

The onmessage event is the client's ear to the server. Whenever the server sends some data, the onmessage event is fired. Messages might contain plain text, images, or binary data. It's up to you how that data will be interpreted and visualized:

```
socket.onmessage = function (event) {
  console.log("Data received!");
}
```

Checking for data types is pretty easy. Here is how we can display a string response:

```
socket.onmessage = function (event) {
  if (typeof event.data === "string") {
    // If the server has sent text data, then display it.
    var label = document.getElementById("status-label");
    label.innerHTML = event.data;
  }
}
```

We'll learn more about the supported data types in *Chapter 4, Data Transfer – Sending, Receiving, and Decoding*.

onclose

The `onclose` event marks the end of the conversation. Whenever this event is fired, no messages can be transferred between the server and the client unless the connection is reopened. A connection might be terminated due to a number of reasons. It can be closed by the server, it may be closed by the client using the `close()` method, or due to TCP errors.

You can easily detect the reason the connection was closed by checking the `code`, `reason`, and `wasClean` parameters of the event.

The `code` parameter provides you with a unique number indicating the origin of the interruption.

The `reason` parameter provides the description of the interruption in a string format.

Finally, the `wasClean` parameter indicates whether the connection was closed due to a server decision or due to unexpected network behavior. The following code snippet illustrates the proper usage of the parameters:

```
socket.onclose = function(event) {
  console.log("Connection closed.");

  var code = event.code;
  var reason = event.reason;
  var wasClean = event.wasClean;

  var label = document.getElementById("status-label");

  if (wasClean) {
    label.innerHTML = "Connection closed normally.";
  }
  else {
    label.innerHTML = "Connection closed with message " + reason +
      "(Code: " + code + ")";
  }
}
```

You can find a detailed list of the code values in the appendix of this book.

onerror

The `onerror` event is fired when something wrong (usually unexpected behavior or failure) occurs. Note that `onerror` is always followed by a connection termination, which is a close event.

A good practice when something bad happens is to inform the user about the unexpected error and probably try to reconnect:

```
socket.onclose = function(event) {
  console.log("Error occurred.");

  // Inform the user about the error.
  var label = document.getElementById("status-label");
  label.innerHTML = "Error: " + event;
}
```

Actions

Events are raised when something happens. We make explicit calls to actions (or methods) when we want something to happen! The WebSocket protocol supports two main actions: `send()` and `close()`.

send()

While a connection is open, you can exchange messages with the server. The `send()` method allows you to transfer a variety of data to the web server. Here is how we can send a chat message (actually, the contents of the HTML text field) to everyone in the chat room:

```
// Find the text view and the button.
var textView = document.getElementById("text-view");
var buttonSend = document.getElementById("send-button");

// Handle the button click event.
buttonSend.onclick = function() {
  // Send the data!!!
  socket.send(textView.value);
}
```

It's that simple!

But wait… The preceding code is not 100 percent correct. Remember that you can send messages only if the connection is open. This means that we either need to place the `send()` method inside the `onopen` event handler or check the `readyState` property. This property returns the state of the WebSocket connection. So, the previous snippet should be modified accordingly:

```
button.onclick = function() {
  // Send the data if the connection is open.
  if (socket.readyState === WebSocket.OPEN) {
```

```
    socket.send(textView.value);
  }
}
```

After sending the desired data, you can wait for an interaction from the server or close the connection. In our demo example, we leave the connection open, unless the stop button is clicked on.

close()

The `close()` method stands as a goodbye handshake. It terminates the connection and no data can be exchanged unless the connection opens again.

Similarly to the previous example, we call the `close()` method when the user clicks on the second button:

```
var textView = document.getElementById("text-view");
var buttonStop = document.getElementById("stop-button");

buttonStop.onclick = function() {
  // Close the connection, if open.
  if (socket.readyState === WebSocket.OPEN) {
    socket.close();
  }
}
```

It is also possible to pass the `code` and `reason` parameters we mentioned earlier:

```
socket.close(1000, "Deliberate disconnection");
```

Properties

The `WebSocket` object exposes some property values that let us understand its specific characteristics. We have already met the `readyState` property. Following are the rest:

Properties	Description
url	Returns the URL of the WebSocket
protocol	Returns the protocol used by the server

Properties	Description
`readyState`	Reports the state of the connection and can take one of the following self-explanatory values:
	`WebSocket.OPEN`
	`WebSocket.CLOSED`
	`WebSocket.CONNECTING`
	`WebSocket.CLOSING`
`bufferedAmount`	Returns the total number of bytes that were queued when the `send()` method was called
`binaryType`	Returns the binary data format we received when the `onmessage` event was raised

The complete example

Here are the complete HTML and JavaScript files we used. We have omitted the stylesheet file in order to keep the main points simple. However, you can download the complete source code at `http://pterneas.com/books/websockets/source-code`.

index.html

The complete markup code for our web app page is as follows:

```
<!DOCTYPE html>
<html>
<head>
  <title>HTML5 WebSockets</title>
  <link rel="stylesheet" href="style.css" />
  <script src="chat.js"></script>
</head>
<body>
  <h1> HTML5 WebSocket chat. </h1>
  <input type="text" id="text-view" />
  <input type="button" id="send-button" value="Send!"  />
  <input type="button" id="stop-button" value="Stop" />
  </br>
  <label id="status-label">Status</label>
</body>
</html>
```

chat.js

All the JavaScript code for the chatting functionality is as follows:

```javascript
window.onload = function() {
  var textView = document.getElementById("text-view");
  var buttonSend = document.getElementById("send-button");
  var buttonStop = document.getElementById("stop-button");
  var label = document.getElementById("status-label");

  var socket = new WebSocket("ws://echo.websocket.org");

  socket.onopen = function(event) {
    label.innerHTML = "Connection open";
  }

  socket.onmessage = function(event) {
    if (typeof event.data === "string") {
      label.innerHTML = label.innerHTML + "<br />" + event.data;
    }
  }

  socket.onclose = function(event) {
    var code = event.code;
    var reason = event.reason;
    var wasClean = event.wasClean;

    if (wasClean) {
      label.innerHTML = "Connection closed normally.";
    }
    else {
      label.innerHTML = "Connection closed with message: " +
        reason + " (Code: " + code + ")";
    }
  }

  socket.onerror = function(event) {
    label.innerHTML = "Error: " + event;
  }

  buttonSend.onclick = function() {
    if (socket.readyState == WebSocket.OPEN) {
      socket.send(textView.value);
    }
```

```
  }

  buttonStop.onclick = function() {
    if (socket.readyState == WebSocket.OPEN) {
      socket.close();
    }
  }
}
```

What about the server?

You have probably noticed that we use the `echo.websocket.org` server for this demo. This public service simply returns back the data you send. In the next chapter, we are going to build our own WebSocket server and develop a true chatting app.

Summary

In this chapter, we built our first WebSocket client application! We introduced the `WebSocket` object and explained its various methods, events, and properties. We also developed a basic chat client in a few lines of HTML and JavaScript code. As you noticed in the current examples, there is only a dummy server which echoes the messages. Read on to find out how we you can configure your own WebSocket server to do a lot more magic.

> **Downloading the example code**
>
> You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.
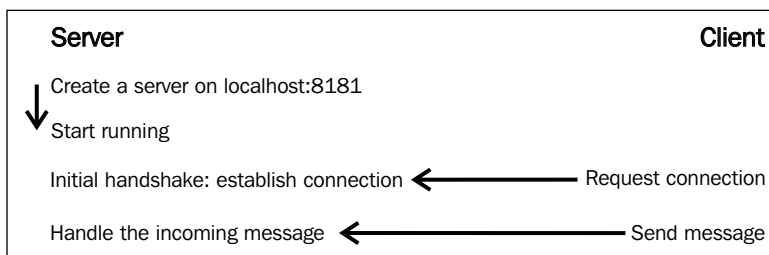
3
Configuring the Server

WebSocket stands for bidirectional, full-duplex communication. As a result, we need two parties for this kind of conversation. In the previous chapter, we implemented the WebSocket client application. Now it's time to establish the other side of the channel, which is the WebSocket server.

Why do I need a WebSocket server?

We assume that you have a minor familiarity with servers in general. A server is nothing but a remote computer that has specific hardware and software requirements in order to achieve high availability and up-time, enhanced security, and management of multiple concurrent connections.

A WebSocket server is nothing but a simple program that is able to handle WebSocket events and actions. It usually exposes similar methods to the WebSocket client API and most programming languages provide an implementation. The following diagram illustrates the communication process between a WebSocket server and a WebSocket client, emphasizing the triggered events and actions.

The following diagram shows WebSocket server and client event triggering:

Server	Client
Create a server on localhost:8181	
↓ Start running	
Initial handshake: establish connection ⟵	Request connection
Handle the incoming message ⟵	Send message

Do not get confused—a WebServer can either run on top of Apache or IIS—or it can be a completely different application by itself.

Considering the hardware, you could use a super-computer or your developer machine as a server. It's all down to the requirements and the budget of each project.

Setting up the server

Implementing a WebSocket server from scratch is not a hard process, though it requires specific knowledge and it's far from the purposes of this book. As a result, we are going to use one of the existing WebSocket implementations that are currently out there. Thanks to the large community of developers, we can easily pick the WebSocket server of our preferred programming language or framework. Furthermore, most of the implementations are open source, so you can even adjust them to your own needs if necessary!

Selecting the technology that suits you

We have listed some popular WebSocket server implementations. Here are some questions you should ask yourself before picking one:

- What technology are you most familiar with?
- What are the specific requirements of your project?
- Do you already have a solution you want to enhance?
- Is the server's documentation thorough and understandable?
- Is there an active support community for the server?

Let's now have a look at the most popular WebSocket server libraries for the most extensively used programming languages.

C/C++

Tufao	`https://github.com/vinipsmaker/tufao`
Wslay	`http://wslay.sourceforge.net/`
Libwebsockets	`http://libwebsockets.org/trac`
Mongoose	`https://code.google.com/p/mongoose/`

Java

Apache Tomcat	`http://tomcat.apache.org/`
JBoss	`http://www.jboss.org/`
GlassFish	`http://glassfish.java.net/`
Atmosphere	`https://github.com/Atmosphere/atmosphere`
Play Framework	`http://www.playframework.com/`
Jetty	`http://www.eclipse.org/jetty/`
jWebSocket	`http://jwebsocket.org/`
Migratory data	`http://migratorydata.com/`
Bristleback	`http://bristleback.pl/`

.NET

Internet Information Services 8	`http://www.iis.net/`
Fleck	`https://github.com/statianzo/Fleck`
SuperWebSocket	`http://superwebsocket.codeplex.com/`

PHP

Php-websocket	`https://github.com/nicokaiser/php-websocket`
Rachet	`http://socketo.me/`
Hoar	`https://github.com/hoaproject/Websocket`

Python

Tornado	`http://www.tornadoweb.org/en/stable/`
Pywebsocket	`https://code.google.com/p/pywebsocket/`
Autobahn	`http://autobahn.ws/`
txWS	`https://github.com/MostAwesomeDude/txWS`
WebSocket for Python	`https://github.com/Lawouach/WebSocket-for-Python`

Ruby

EM-WebSocket	`https://github.com/igrigorik/em-websocket`
Socky server	`https://github.com/socky/socky-server-ruby`

JavaScript

This is no joke. You can create a web server using JavaScript thanks to `Node.js`. `Node.js` (`http://nodejs.org`) is an event-driven framework that lets you build real-time web applications. It is also interpreted by Google's JavaScript engine, V8. Although the framework does not support WebSockets out-of-the-box, there are some quite good extensions which do so.

Socket IO	`http://socket.io/`
WebSocket-Node	`https://github.com/Worlize/WebSocket-Node`
Node WebSocket Server	`https://github.com/miksago/node-websocket-server`

`Node.js` is constantly getting more and more fans, so it might be worth a try.

Setting up the development environment

The environment where your server will be created depends on the technology, frameworks, and programming languages that you are planning to use. There is an amazingly huge variety of **Integrated Development Environments** (**IDEs**) and utilities that make your life easier!

Here is a list of some IDEs we propose, along with the web programming languages they support:

IDE	Operating System	Supported languages
Aptana	Windows, Mac, Linux	HTML5
		JavaScript
		PHP
NetBeans	Windows, Mac, Linux	HTML5
		C/C++
		Java

IDE	Operating System	Supported languages
Eclipse (with the Web Developer plugin)	Windows, Mac, Linux	HTML5
		JavaScript
		C/C++
		Java
Visual Studio	Windows	HTML5
		JavaScript
		.NET
WebMatrix	Windows	HTML5
		JavaScript
		PHP
		.NET

Throughout the book, we decided to use C#.NET and Fleck, though this should make no difference to you. Feel free to pick the language you prefer or the language your existing projects require.

For didactic purposes, C# has the following advantages:

- It runs on Windows using the .NET framework and on Mac and Linux using Mono
- It has an active community of developers, making it easier to find support
- It is easy-to-learn
- You can quickly setup a WebSocket server with minimum configuration

Fleck library was chosen because of three reasons:

- It is supported on both Windows and Unix-based operating systems
- It is extremely easy-to-use and configure
- It is well-maintained and well-documented

This is how you can quickly set up a Fleck WebSocket server using C#:

1. Download Visual Studio Express (It is freely available at `http://www.microsoft.com/visualstudio/eng/products/visual-studio-express-for-windows-desktop`).
2. Download Fleck (`https://github.com/statianzo/Fleck`).
3. Launch Visual Studio and click on **File** | **New** | **Project**.
4. Under Visual C#, select **Windows**.

5. Choose **Console Application** (yes, a console-based server is the easiest way to set up a WebSocket server).

6. Name your project whatever you like and click on **OK**.

7. On the **Solution Explorer** tab, right-click on the **References** icon and select **Add new reference**.

8. Click on **browse** and find the `Fleck.dll` file.

9. Click on **OK** and you are done!

Connecting to the web server

The WebSocket server works in a similar way to the WebSocket clients. It responds to events and performs actions when necessary.Regardless of the programming language you use, every WebSocket server performs some specific actions. It is initialized to a WebSocket address, it handles `OnOpen`, `OnClose` and `OnMessage` events, and sends messages to the clients, too.

Creating the WebSocket server instance

Every WebSocket server needs a valid host and port. Here is how we create a WebSocketServer instance in Fleck:

```
var server = new WebSocketServer("ws://localhost:8181");
```

You can type any valid URL you'd like and specify a port that is not in use.

It is very useful to keep a record of the connected clients, as you may need to provide them with different data or send different messages to each one.

Fleck represents the incoming connections (clients) with the `IWebSocketConnection` interface. We can create an empty list and update it whenever someone connects or disconnects from our service:

```
var clients = new List<IWebSocketConnection>();
```

After that, we can call the `Start` method and wait for the clients to connect. When started, the server is able to accept incoming connections.

In Fleck, the `Start` method needs a parameter which indicates the socket that raised the events:

```
server.Start(socket) =>
{
});
```

Some syntax explanation: what follows the `Start` declaration is called a C# Action and you can totally ignore it if you're using a different language. We'll handle all of the events inside the `Start` block.

Open

The `OnOpen` event determines that a new client has requested access and performs the initial handshake. We should add the client to the list and probably store any information related to it, such as the IP address. Fleck provides us with such information, as well as a unique identifier for the connection.

```
server.Start(socket) =>
{
  socket.OnOpen = () =>
  {
    // Add the incoming connection to our list.
    clients.Add(socket);
  }

  // Handle the other events here…
});
```

Close

The `OnClose` event is raised whenever a client is disconnected. We can remove that client from our list and inform the rest of the clients about the disconnection:

```
socket.OnClose = () =>
{
  // Remove the disconnected client from the list.
  clients.Remove(socket);
};
```

Message

The `OnMessage` event is raised when a client sends data to the server. Inside this event handler, we can transmit the incoming message to all of the clients, or probably select only some of them. The process is straightforward. Note that this handler takes a string named `message` as a parameter:

```
socket.OnMessage = () =>
{
  // Display the message on the console.
  Console.WriteLine(message);
};
```

Send

The `Send()` method simply transmits the desired message to the specified client. Using `Send()`, we can deliver text or binary data across the clients. Let's loop through the registered clients and transfer the messages to them. We need to modify the `OnMessage` event as follows:

```
socket.OnMessage = () =>
{
foreach (var client in clients)
{
  // Send the message to everyone!
  // Also, send the client connection's unique identifier in order
  to recognize who is who.
  client.Send(client.ConnectionInfo.Id + " says: " + message);
}
};
```

Obviously, you do not need to expose everyone's IP address or ID publicly! It's totally useless and makes no sense for your users (unless they are hackers). Of course, during a real chat conversation, users pick nicknames instead of string literals. We'll give them the nickname option in the next chapter.

Fleck accepts strings and byte arrays. Strings contain plain text, XML, or JSON messages. Byte arrays are handful when dealing with images or binary files.

Other methods

Depending on which WebSocket server implementation you use, there might be additional events or methods. For example, Fleck supports the `OnBinary` event, which is a binary-supporting equivalent of the `OnMessage` event.

Keep in mind that the web server stores the connections in a list and we need to loop through all of them in order to send messages.

The complete source code

Here is the complete server-side source code, with a couple of extra additions for better user experience. The screenshots display a Chrome and an Internet Explorer 10 window chatting side-by-side!

The following screenshot shows a user chatting using Chrome:
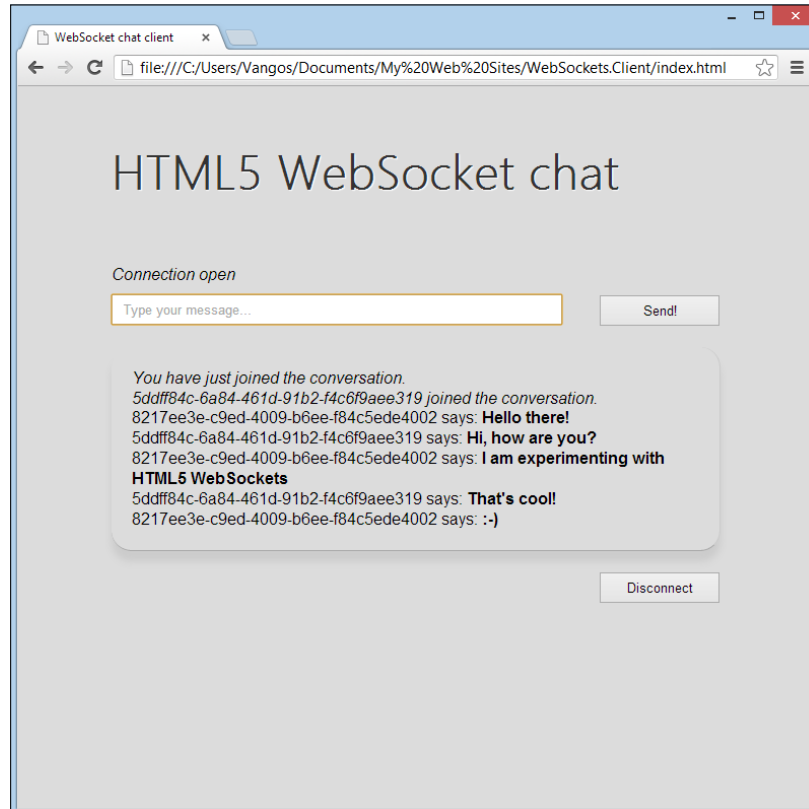
The following screenshot shows a second user chatting concurrently using Internet Explorer 10:
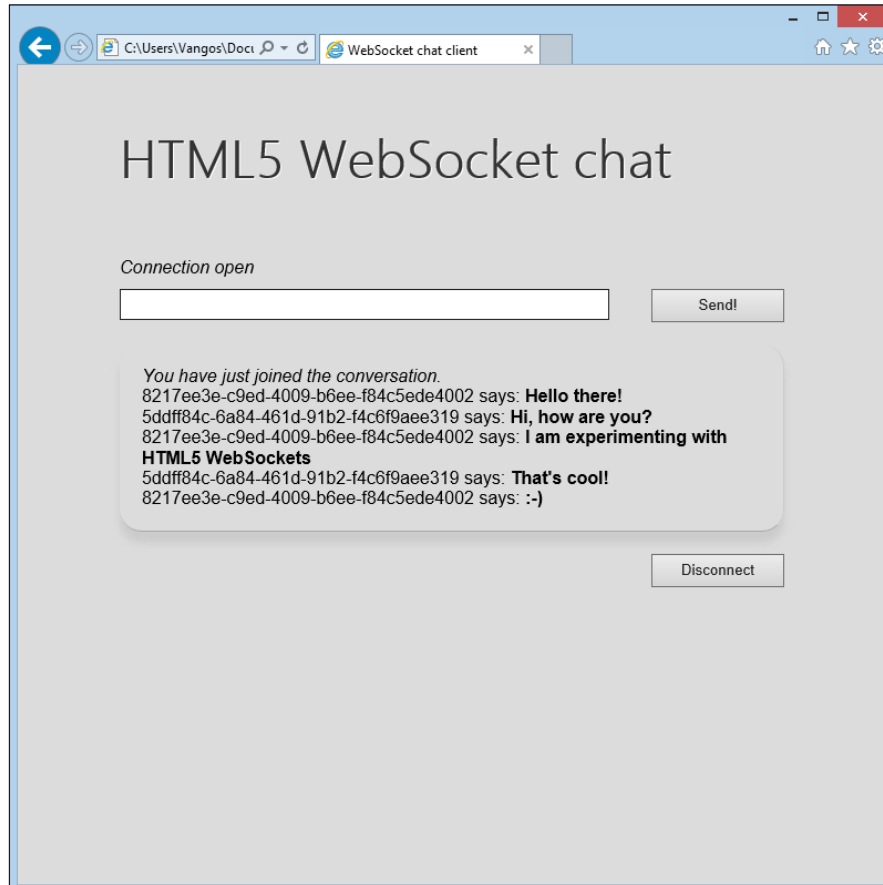

```csharp
namespace WebSockets.Server
{
  class Program
  {
    static void Main(string[] args)
    {
      // Store the subscribed clients.
      var clients = new List<IWebSocketConnection>();

      // Initialize the WebSocket server connection.
      var server = new WebSocketServer("ws://localhost:8181");

      server.Start(socket) =>
      {
```

```csharp
socket.OnOpen = () =>
{
  // Add the incoming connection to our list.
  clients.Add(socket);

  // Inform the others that someone has just joined the
  conversation.
  foreach (var client in clients)
  {
    // Check the connection unique ID and display a
    different welcome message!
    if (client.ConnectionInfo.Id !=
        socket.ConnectionInfo.Id)
    {
      client.Send("<i>" + socket.ConnectionInfo.Id + "
        joined the conversation.</i>");
    }
    else
    {
      client.Send("<i>You have just joined the
        conversation.</i>");
    }
  }
};

socket.OnClose = () =>
{
  // Remove the disconnected client from the list.
  clients.Remove(socket);

  // Inform the others that someone left the conversation.
  foreach (var client in clients)
  {
    if (client.ConnectionInfo.Id !=
        socket.ConnectionInfo.Id)
    {
      client.Send("<i>" + socket.ConnectionInfo.Id + "
        left the chat room.</i>");
    }
  }
};

socket.OnMessage = message =>
{
  // Send the message to everyone!
```

```
            // Also, send the client connection's unique
               identifier in order to recognize who is who.
            foreach (var client in clients)
            {
              client.Send(socket.ConnectionInfo.Id + " says:
                <strong>" + message + "</strong>");
            }
          };
        });

        // Wait for a key press to close...
        Console.ReadLine();
      }
    }
  }
```

Summary

By now, you should be able to create a complete WebSocket application! *Chapter 2,*
The WebSocket API, illustrated how to configure a client using JavaScript and
this chapter showed you how you can configure a WebSocket server using the
environment and programming language you are most familiar with. Moreover,
we had a look at the WebSocket server events and actions. In the upcoming chapters,
we are going to learn how we can effectively handle different data formats and
secure our WebSocket-based apps.

4
Data Transfer – Sending, Receiving, and Decoding

Modern web development is all about content. No matter what kind of application you are building, users will stop using it unless they get what they want. In the old days of the web, the content someone could publish on his/her website was extremely limited. Nowadays, content is a lot more than static text and images; you can exchange messages, watch videos, download programs, and much more. As a web developer, you should be able to deliver the desired content in a fast and efficient way. The WebSocket protocol supports a variety of transferable data, taking the burden to speed the whole process as much as possible.

In this chapter's demo, you are going to handle image and video data via WebSockets. Let's start!

What kinds of data can WebSockets transfer?

The WebSocket protocol supports text and binary data. In JavaScript, text is referred to as String, while binary data is represented by the ArrayBuffer and Blob classes (the first one is still experimental). Using plain text and binary format, you can transfer and decode almost any type of HTML5 media.

Always keep in your mind that WebSockets only support one binary format at a time and you have to explicitly declare it as follows:

```
socket.binaryType = "arraybuffer";
```

Another was to do it is as follows:

```
socket.binaryType = "blob"
```

Throughout this book, we'll demonstrate specific examples for using each and every data type.

String

You have already taken a glimpse of transmitting plain text data in the previous chapters, where you exchanged simple chat messages. Apart from this, strings are tremendously helpful when dealing with human-readable data formats such as **XML** and **JSON**.

Remember that whenever the `onmessage` event is raised, the client needs to check the data type and act accordingly. JavaScript can easily determine that a data type is of `string` type using the strict equal operator (that is, `===`).

```
socket.onmessage = function(event) {
  if (typeof event.data === "string") {
    console.log("Received string data.");
  }
}
```

If you have an average experience with core JavaScript, you'll probably notice that you could have used the following expression instead:

```
if (event.data instanceof String)
```

Although this code is pretty valid, it wouldn't work in your case. The reason is that the `instanceof` expression requires the object on the left to have been created using the JavaScript string constructor. In your case, the data is generated from the server, so you can only determine their underlying type instead of their JavaScript class.

JSON

JSON (**JavaScript Object Notation**) is a lightweight format for transferring human-readable data between computers. It is structured in key-value pairs, usually describing properties and values. Due to its efficiency, JSON is the dominant format for transferring data between server and client. The most popular RESTful APIs, including Facebook, Twitter, and Github, nowadays fully support JSON. Moreover, JSON is a subset of JavaScript, so you can parse it immediately without using external parsers!

Suppose that the web server somehow sends the following JSON string:

```
{
"name" : "Vangos Pterneas",
"message" : "Hello world!"
}
```

Obviously, the preceding notation contains two key-value pairs. Guess what? In your chat demo, it represents the chat data received from another user. You are going to use this information in a few minutes.

Following code shows how you can handle a JSON object and extract its properties:

socket.onmessage = function(event) {

```
    if (typeof event.data === "string") {
      // Create a JSON object.
      var jsonObject = JSON.parse(event.data);

      // Extract the values for each key.
      var userName = jsonObject.name;
      var userMessage = jsonObject.message;
    }
  }
```

The preceding code is straightforward. Using the `eval` function, you create a JSON object from the input string. What `eval` really does is invoke the JavaScript compiler and execute the enclosed string arguments. The properties of the generated object are the names of the JSON keys and each property holds its corresponding value.

XML

Similar to JSON, you can parse XML-encoded strings using JavaScript. We won't go deeper into XML parsing, as this would be out of this book's scope. Parsing XML is not difficult, though it requires different techniques for different browsers (**DOMParser** versus **ActiveXObject**). The best method is using a third-party library such as **jQuery**.

> In both XML and JSON cases, the server should send you a string value, not the actual XML/JSON file (which would be of binary type, of course)!

ArrayBuffer

ArrayBuffer contains structured binary data. The key term here is **structured**, which means that the enclosed bits are given in an order, so that you can retrieve portions of them. In order to manipulate an ArrayBuffer for specific formats, you need to create the corresponding `ArrayBufferView` object.

ArrayBuffers are really handy for storing image files. Suppose that your chat-room guests can exchange images by dragging and dropping image files on the chat window. Following code explains how JavaScript handles the drop event in HTML5 browsers:

```
document.ondrop = function(event) {
  var file = event.dataTransfer.files[0];
  var reader = new FileReader();

  reader.readAsArrayBuffer(file);

  reader.onload = function() {
    socket.send(reader.result);
  }

  return false;
}
```

In the preceding code snippet, you firstly create an event handler for the drop event. The event handler takes one parameter, which lets you access the dropped files. You only drop one single image, so you need the zero-indexed file. After that, you create a file reader that reads the file's data as an ArrayBuffer. When the reader has finished processing the file, you handle the `onload` event where you send the image to the web server using your WebSocket.

Learn more about FileReader at `http://www.html5rocks.com/en/tutorials/ file/dndfiles/`.

The following is a screenshot of the drop effect that raises the send method:
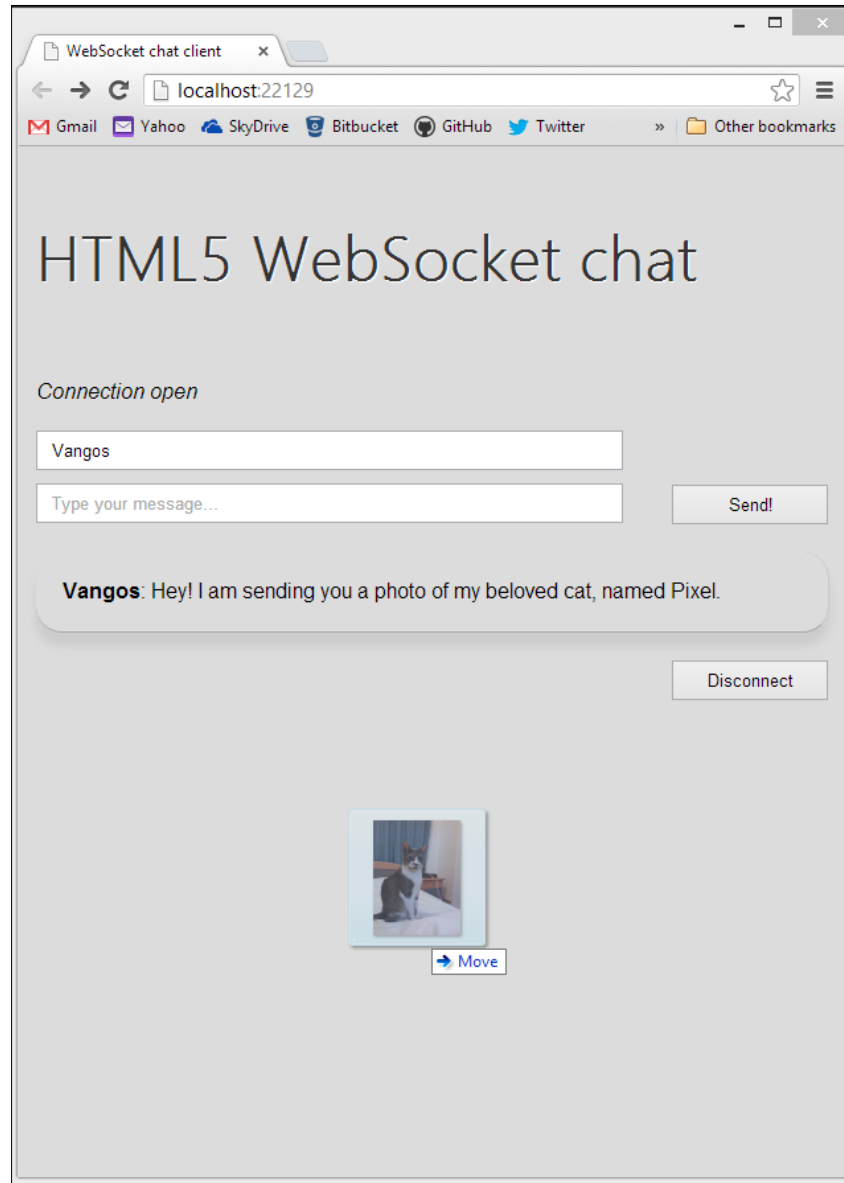
Dropping an image to the browser and sending it to the server

Receiving data as ArrayBuffers is fairly simple. Note that you check using `instanceof`, rather than the strict equal operator.

```
socket.onmessage = function(event) {
  if (event.data instanceof ArrayBuffer) {
    var buffer = event.data;
  }
}
```

Blobs

Blobs (**Binary Large Objects**) contain totally raw data in their most native form. A blob might theoretically be anything, even a non-JavaScript object. As a result, interpreting blob data might be quite tricky. As a thumb rule, you'd better know exactly what the server is supposed to send, otherwise you'll need to make fairly non-concrete assumptions.

However, the big advantage of blob data is their file size. Binary format is machine-level format, so there are almost no abstraction layers used that would increase its size.

When you transmit multimedia over the web, you need the highest speed possible, in order to achieve the best user experience. The WebSocket blobs do not create extra burden for your Internet connection and they rely on the client for proper interpretation.

Following code shows how you can display an incoming image, sent as a set of raw bits:

```
socket.onmessage = function(event) {
  if (event.data instanceof Blob) {
    // 1. Get the raw data.
var blob = event.data;

    // 2. Create a new URL for the blob object.
    window.URL = window.URL || window.webkitURL;
    var source = window.URL.createObjectURL(blob);

    // 3. Create an image tag programmatically.
    var image = document.createElement("img");
    image.src = source;
    image.alt = "Image generated from blob";
```

```
    // 4. Insert the new image at the end of the document.
    document.body.appendChild(image);
  }
}
```

The preceding code snippet generates an image by properly interpreting the incoming raw data. You have used some brand-new HTML5 JavaScript methods to easily handle the blob. Let's be more specific.

At first, you verify that the server message is an instance of blob, similar to the way you checked for the buffered array. Then, you store the raw data to a local variable, named `blob`.

In order to display the blob in an image format, you need to decode it properly. The new JavaScript API makes basic image manipulation a piece of cake. Instead of reading the bytes, you create a plain URL to the specified data source. This URL is alive as long as the HTML document is alive. That means you cannot retrieve it after closing your browser window.

The `window.URL` property is currently supported in all the major browsers, though **Google Chrome** has named it `window.webkitURL`. The `createObjectURL` method generates a URL for the temporary file specified as a parameter. You do not need to provide any further details or write any further code! JavaScript represents the blob you received as a normal browser URL!

Finally, using the DOM manipulation methods you already know, you create an image element, you provide it with the new URL, and you insert it right at the end of the HTML document.

> The `createObjectURL` method is supported in Chrome 23+, Firefox 8+, Internet Explorer 10+, Opera 16+ and Safari 6+, as well as in their mobile counterparts (except IE and Opera).

Try it out and you'll see something like the following screenshot:
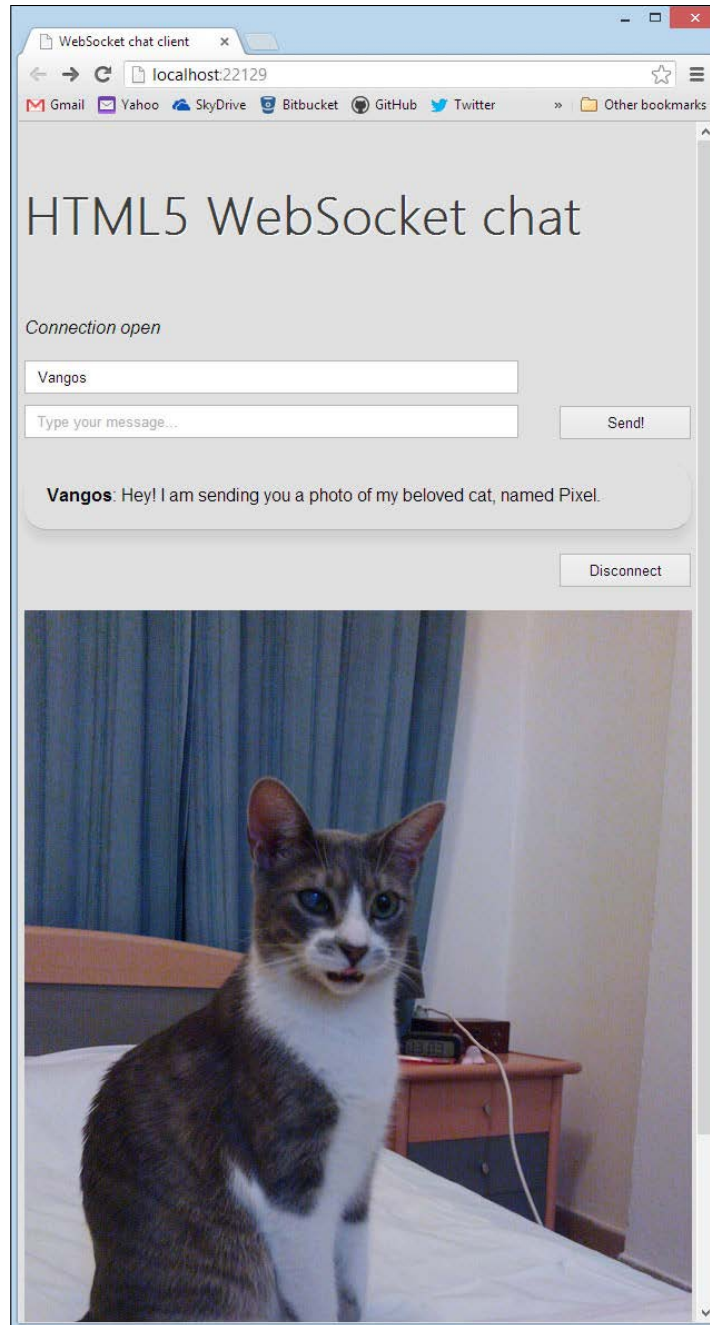
The incoming blob data, displayed as an HTML image

Video streaming

Many web designers and developers argue that the future of the web is video. Until now, video was delivered using third-party plugins and technologies such as Flash or Silverlight. Although these technologies worked pretty well on the desktop browsers, they required extra software and were a catastrophe (in terms of battery life) for mobile and tablet devices. After Apple decided to drop Flash support for iPhone and iPad, HTML5 became the only available gate for delivering videos and rich graphics over the web.

In terms of WebSockets, it makes sense to stream video across different clients using a fast and efficient way. Live video streaming is currently supposed to be one of the last reasons Flash is still alive. Let's see how you can stream live video data from the server to the clients in the WebSocket way.

A video is nothing more than a collection of consecutive images. Each of these images is called a **frame**. When a number of frames (usually more than 20) are displayed per second, the human eye cannot distinguish the distinct images and thinks of it like a continuous flow. That's the technique you are going to use for streaming a video file from the server to the clients.

The server sends 20 or more frames (images) per second, so that the client is constantly awaiting for new messages. Remember the code you wrote for displaying images? Well, in a real-time video stream context, you do not need to store the data as URLs until the web page is closed. Rather, it's a good practice to dispose the frame URLs when you do not use them any more. Also, there is no need to create the `<img>` element using JavaScript, as you can place it in our markup:

```
<img id="video" src="" alt="Video streaming" />
```

…and create a reference in your JavaScript code:

```
var video = document.getElementById("video");
```

So, here is the modified onmessage client event, which will be raised 20 or more times per second:

```
socket.onmessage = function(event) {
  if (event.data instanceof Blob) {
    // 1. Get the raw data.
var blob = event.data;

    // 2. Create a new URL for the blob object.
    window.URL = window.URL || window.webkitURL;
    var source = window.URL.createObjectURL(blob);

    // 3. Update the image source.
    video.src = source;
```

```
      // 4. Release the allocated memory.
      window.URL.revokeObjectURL(source);
    }
  }
```

The code is similar to the one you used to drop an image on the HTML document. There are two things to notice:

- You have created a reference for the `<img>` element, in order to constantly modify its `src` property.
- After every `src` assignment, you release the image by calling the `revokeObjectURL` function. This function cleans up the memory assigned to the specified URL, and lets the browser know it doesn't need to keep the URL's reference any more.

The following screenshot shows video streaming using consecutive frames:
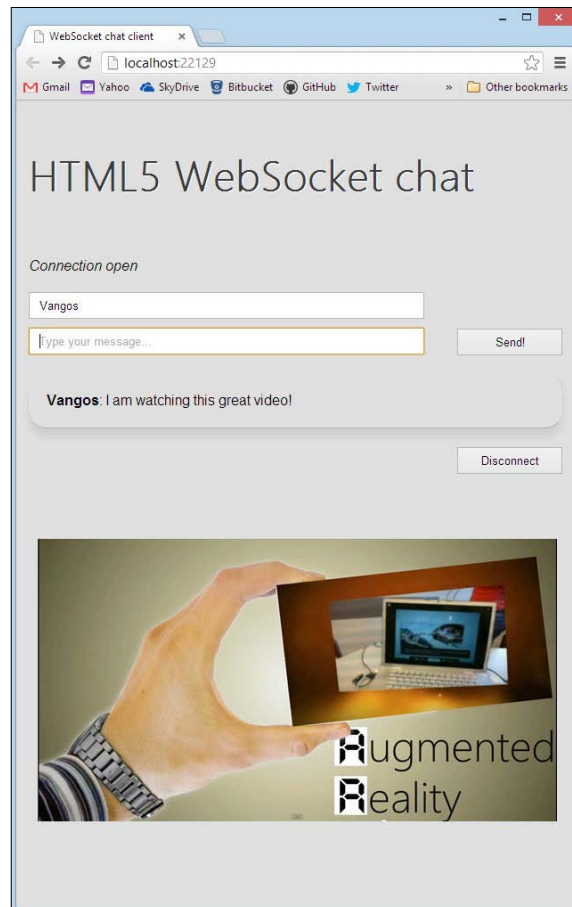
Although it makes the point, this might not be the optimal way to stream video. For a more professional approach, have a look at WebRTC (`http://www.webrtc.org`), a great multimedia development API, implemented by Google, Mozilla, and Opera.

Putting it all together

You might be wondering where is the server code that handles the requests, receives the images, and updates the video frames. We deliberately left out the server-side part in order to focus on the client-side JavaScript code. For the purposes of our chat demo web app, we'll now show you both the client and the server code. Once more, note that you can use the programming language and frameworks of your choice to implement the WebSocket server.

Let's have a close look at the new parts you'll implement.

Sending the nickname and message using JSON

At first, you'll add one more text field to the HTML document, in order for the user to type his/her preferred nickname. You'll send the nickname along with the text message by encoding them in JSON format.

Add a new text input just before the message input:

```
<label id="status-label">Status...</label>
<input type="text" id="name-view" placeholder="Your name" />
<input type="text" id="text-view" placeholder="Type your
  message..." />
```

Then, create a reference to the JavaScript code:

```
var nameView = document.getElementById("name-view");
```

And finally, send the nickname and the message to the server, as you did a few pages ago!

```
buttonSend.onclick = function (event) {
  if (socket.readyState == WebSocket.OPEN) {
    var json = "{ 'name' : '" + nameView.value + "', 'message' :
      '" + textView.value + "' }";
    socket.send(json);
    textView.value = "";
  }
}
```

The server now needs to transmit this message to the clients. Nothing is changed from the previous chapter:

```
socket.OnMessage = message =>
  {
    // Send the text message to everyone!
    foreach (var client in clients)
      {
        client.Send(message);
      }
  };
```

The clients decode the JSON string and display the message accordingly. You have added a prettier presentation style for showing the text in the chat area.

```
socket.onmessage = function (event) {
  if (typeof event.data === "string") {
    // Display message.
    var jsonObject = eval('(' + event.data + ')');
    var userName = jsonObject.name;
    var userMessage = jsonObject.message;

    chatArea.innerHTML = chatArea.innerHTML +
      "<p><strong>" + userName + "</strong>: " + userMessage +
        "</p>";
  }
}
```

Sending images to the server

Remember the `ondrop` event we discussed previously? For consistency reasons, here is the same functionality using Blobs instead of ArrayBuffers:

```
document.ondrop = function(event) {
  var file = event.dataTransfer.files[0];

  socket.send(file);

  return false;
}
```

When dealing with HTML5 drag-and-drop, remember to always prevent the default drag-and-drop behavior! Unless you explicitly define that you want to override the default functionality, whatever you implement will not be shown correctly. Fortunately, preventing the predefined actions from happening is quite simple:

```
document.ondragover = function (event) {
  event.preventDefault();
}
```

The server needs to distribute the blob image to all the clients. `Fleck` library introduces the `OnBinary` event, which is raised when binary data is received:

```
socket.OnBinary = data =>
  {
    // Send the binary data to everyone!
    foreach (var client in clients)
      {
        client.Send(data);
      }
  };
```

The method works similar to the `OnMessage` method. The only difference is that it takes a byte array (data) instead of string as a parameter. An array of bytes is the most native and efficient image representation.

When the rest of the clients receive the image, a new `<img>` element will be created. You have already seen the way, so you update the `onmessage` function accordingly:

```
socket.onmessage = function(event) {
if (typeof event.data === "string") {
  // Decode JSON, then display nickname and message.
  // …
}
  else if (event.data instanceof Blob) {
  // Get the raw data and create an image element.
var blob = event.data;

    window.URL = window.URL || window.webkitURL;
    var source = window.URL.createObjectURL(blob);

var image = document.createElement("img");
    image.src = source;
    image.alt = "Image generated from blob";

    document.body.appendChild(image);
  }
}
```

Summary

In this chapter, you had a detailed look at the various data formats the WebSocket protocol supports. You implemented various examples using string and binary data (text, images, and videos), found out how you can properly encode and decode the client-side data, and finally extended the chat demo to manipulate images and videos. The next chapter discusses security considerations over the web that will make your apps even more robust.

5
Security

Security is a crucial issue for web applications that exchange data. Every site or app that lives and breathes in the web is subject to attack by human or robot invaders. It's a sad but true reality, and we all have to live with it.

Of course, this does not mean that your web apps are totally unsafe. Fortunately, the native HTML5 security mechanisms protect you from the most common security attacks without any configuration. Moreover, the WebSocket protocol is designed to be a secure service, so a basic protection is guaranteed.

In this chapter, we are going to present some known security risks a WebSocket app may have, and also provide you with the tools and knowledge to prevent, confront, and overcome them, in favor of your users.

WebSocket headers

You normally don't shake hands with an unknown person or with someone who does not want to reveal his/her identity. In the WebSocket world, you need to be sure about the origin of the request. The **origin** is a header sent from the client and is essential for cross-domain communication, as it allows the web server to reject specific connections. Origin is the first and the most important security aspect introduced and documented in WebSockets.

There are a couple more headers required to allow a client upgrade to the WebSocket protocol. Such headers begin with a `Sec-` prefix and guarantees that every WebSocket request will be initialized via the WebSocket constructor, rather than any HTTP APIs, which might want to access the exchanged information.

The following is an example of WebSocket header sent from a client:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Origin: http://example.com
Pragma: no-cache
Cache-Control: no-cache
Sec-WebSocket-Key: AAf/gvkPw6szicrMH3Rwbg==
Sec-WebSocket-Version: 13
Sec-WebSocket-Extensions: x-webkit-deflate-frame
```

The `Sec-WebSocket-Version` parameter can help you identify the browser used. Take care if you need specific tweaks for specific browsers. The corresponding handshake from the server should look as follows:

```
HTTP/1.1 101 Switching Protocols
   Upgrade: websocket
   Connection: Upgrade
   Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

If you are interested in more theoretical details, feel free to read the complete specification of RFC 6455 at `http://tools.ietf.org/html/rfc6455`.

Common attacks

Currently, what you need to know is that the protocol is designed to be as secure as possible. Be careful though! WebSocket is a brand-new protocol and not all web browsers implement it correctly. For example, some of them still allow the mix of HTTP and WS, although the specification implies the opposite. Everything is subject to change, and while waiting for the browsers to mature, you can easily adopt some protection techniques yourself.

So, the old-school problems are not solved. Remember those bad people who sniffed the HTTP and intercepted into the web traffic? Well, the WS can be sniffed the same way.

Here are some common security attacks you need to be aware of, and consequently, some ways you can protect your app and your users.

Denial of Service

Denial of Service (**DoS**) attacks attempt to make a machine or network resource unavailable to the users that request it. Imagine that someone makes an infinite number of requests to a web server with no or tiny time intervals. Obviously, the server won't be able to handle every connection and will either stop responding or will keep responding too slowly. That's the simplest form of a DoS attack.

There is no need to mention how frustrating this might be for the end-users, who could not even load a web page.

DoS attack can even apply on peer-to-peer communications, forcing the clients of a P2P network to concurrently connect to the victim web server.

The following figure describes a DoS attack:
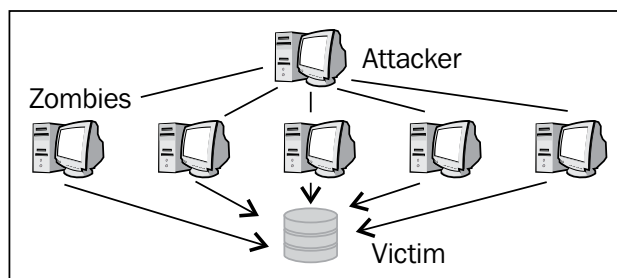
DoS attack

Man-in-the-middle

Suppose you are chatting with your girlfriend via an IM client. Her ex-boyfriend wants to view the messages you exchange, so he makes independent connections with both of you and sniffs your messages. He also sends messages to you and your girlfriend, as an invisible intermediate to your communication. That is known as a man-in-the-middle attack. The man-in-the-middle kind of attack is easier for unencrypted connections, as the intruder can read the packages directly. When the connection is encrypted, the information has to be decrypted by the attacker, which might be way too difficult.

From a technical aspect, the attacker intercepts a public-key message exchange and sends the message while replacing the requested key with his own.

Obviously, a solid strategy to make the attacker's job difficult is to use SSH with WebSockets. Mostly when exchanging critical data, prefer the WSS secure connection instead of the unencrypted WS.

The following figure describes how the spy interferes and acquires data:
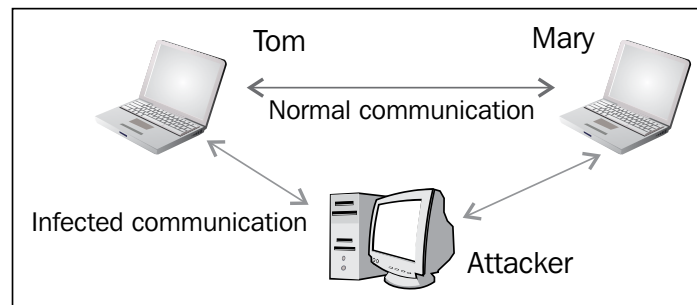
Man-in-the-middle attack

XSS

Cross-site scripting (**XSS**) is a vulnerability that enables attackers to inject client-side scripts into web pages or applications. An attacker can send HTML or JavaScript code using your application hubs and let this code be executed on the clients' machines.

You may encounter the simplest form of an XSS attack when filling a web form. Imagine that someone sends the following data using the chat application we developed:

```
<img src="http://www.therichest.org/wp-content/uploads/young-bill-
  gates.jpg" />
```

Try it out! Type the preceding lines in the message text field, click on **Send** and wait for the result.

The following image shows an XSS attack to our WebSocket chat application:
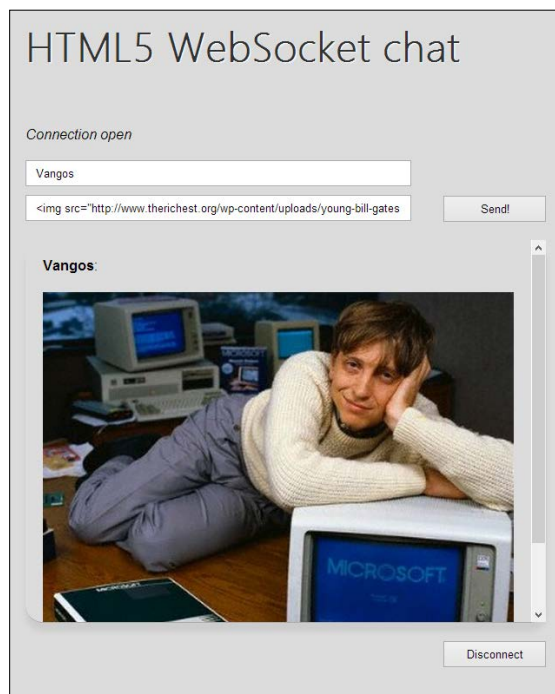
Although the image transmission is not at all bad during a chat application, the user sent the image by injecting HTML code. In a similar way, somebody could execute JavaScript code and harm the conversation.

What can we do? Taking into consideration the old rules about XSS attacks still works and is the best practice. Things you can do are checking your code for HTML entities or JavaScript syntax, and replacing them with the appropriate representation or simply rejecting them.

`https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet` contains a lot more information if you want to learn every aspect of XSS attacks, and how to avoid them.

WebSocket native defence mechanisms

By default, the WebSocket protocol is designed to be secure. In the real world, you might encounter various issues that might occur due to poor browser implementation. No need to worry though. As time goes by, browser vendors fix any issues immediately, and if you still feel afraid, you can always use some old-school fallback techniques (described in the next chapter).

SSH/TLS

As you have probably guessed, an extra layer of security is added when you use secure WebSocket connection over SSH (or TLS). Remember when you needed to decide between HTTP and HTTPS? You picked HTTPS only when it was absolutely necessary for your transactions (for example, bank account information, private data, and so on). Otherwise, HTTP was the way to go, as it is more lightweight and fast. HTTPS required more CPU resources and was quite slower than HTTP.

In the WebSocket world, you do not need to worry about the performance of a secure connection. Although there is still an extra TLS layer on top, the protocol itself contains optimizations for this kind of use, furthermore, WSS works more sleekly through proxies.

Client-to-Server masking

Every message transmitted between a WebSocket server and a WebSocket client contains a specific key, named **masking key**, which allows any WebSocket-compliant intermediaries to unmask and inspect the message. If the intermediary is not WebSocket-compliant, then the message cannot be affected. Masking is handled by the browser that implements the WebSocket protocol.

Security toolbox

Finally, we present some useful tools that help you investigate the flow of information between your WebSocket clients and server, analyze the exchanged data, and identify possible risks.

Fiddler

Fiddler is a great tool for monitoring the network activity and inspecting the traffic of any incoming or outgoing data.

The following screenshot shows fiddler in action, displaying the WebSocket headers:
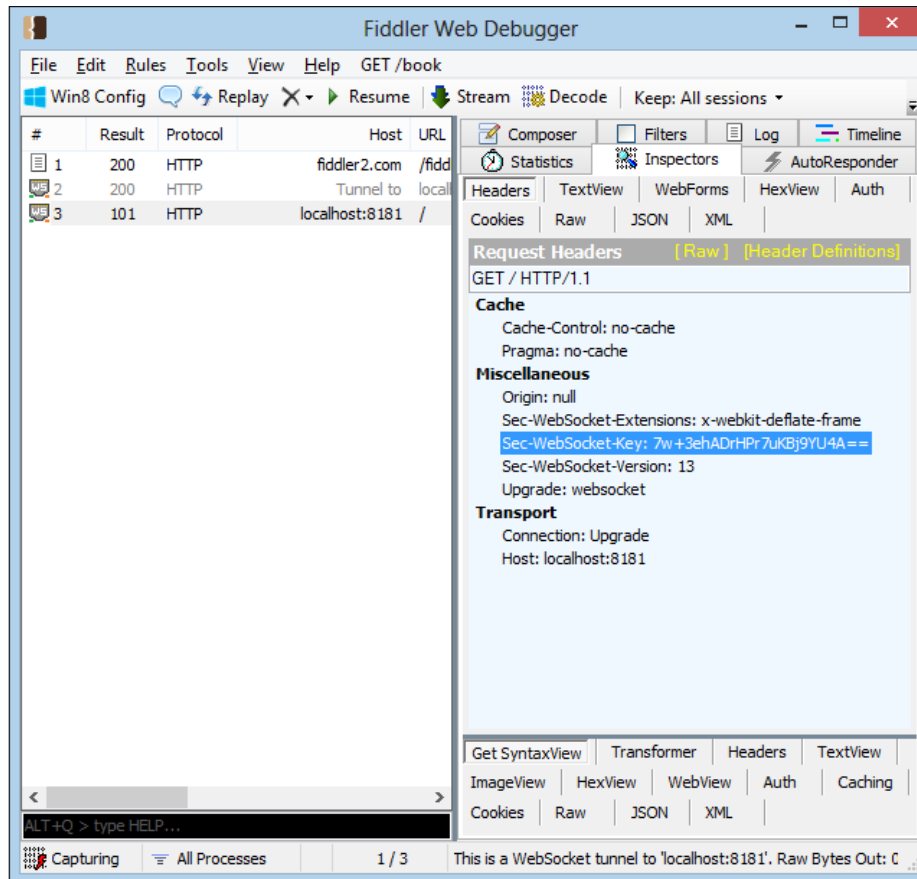
Fiddler can be downloaded from `http://www.fiddler2.com/fiddler2/`

Wireshark

Wireshark is a network packet analyzer that captures the packages and displays their data as accurately as possible.

The following screenshot shows wireshark in action:
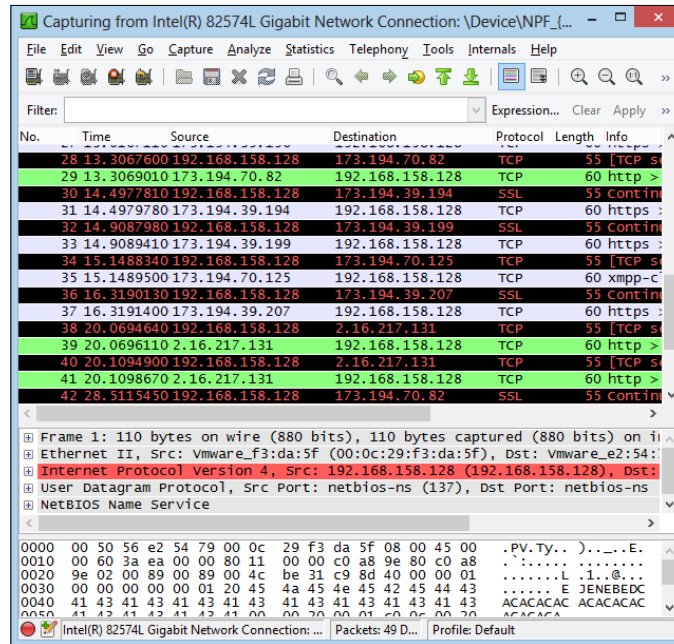
Wireshark can be downloaded from `http://www.wireshark.org/`

Browser developer tools

Chrome, Firefox, and Opera are great browsers in terms of developer support. Their built-in tools help us determine almost any aspect of client-side interactions and resources.

The following screenshot shows Chrome developer tools in action:
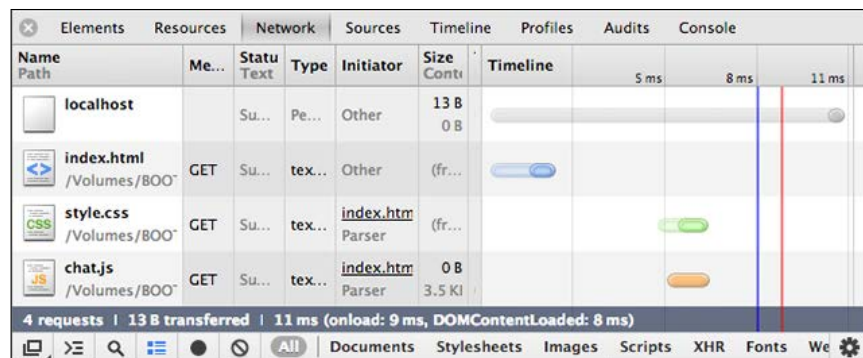
ZAP

ZAP is a penetration-testing tool for finding vulnerabilities throughout your web apps and sites, by performing an attack on them! Like all the preceding tools, ZAP comes with a handy GUI visualization.

The following screenshot shows ZAP in action:

ZAP can be downloaded from `https://code.google.com/p/zaproxy/`
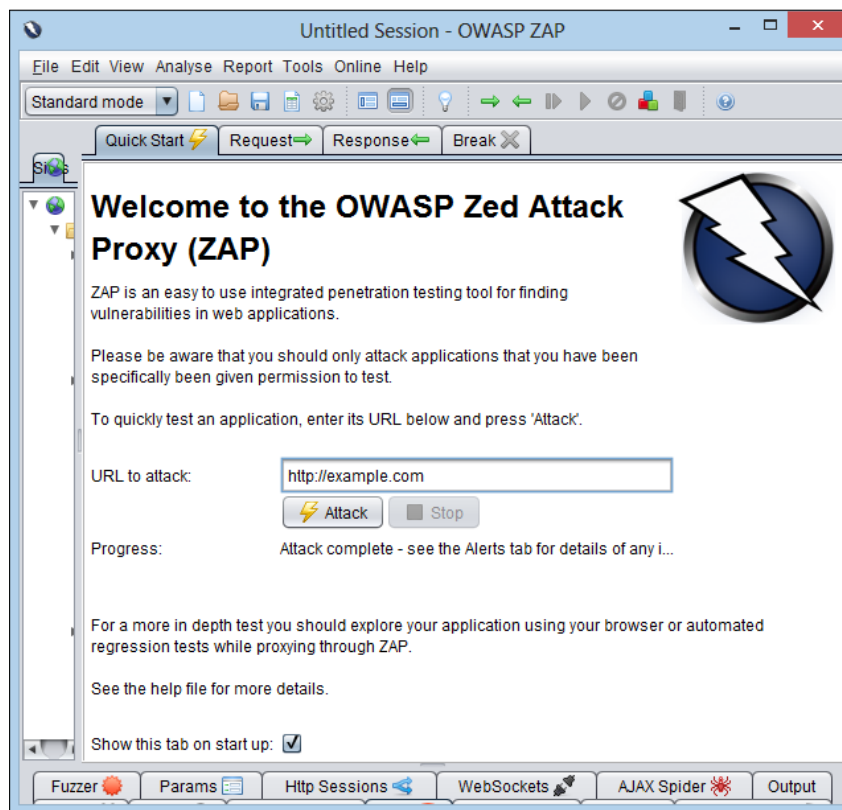
Summary

In this chapter, you investigated various security threats your web apps must be aware of, saw the built-in WebSocket security mechanism, and presented some popular tools that help us manage the network transfers better. Next, we are going to describe some fallback techniques for browsers that lack full or partial WebSocket support.

6
Error Handling and Fallbacks

By now, you must be familiar with the WebSocket capabilities and must have got an idea of the power of full-duplex communication. However, the WebSocket goodies are built on top of HTML5 and depend strongly on the browsers for full support. What happens when the features you want to implement are not supported by the means your audience is using? Would you let your customers leave? That doesn't sound like a good idea. Fortunately, with a little bit of extra effort, you can implement, mimic, and mostly emulate, the WebSocket behavior.

WebSocket is the future-friendly way to go, but you'll need some fallback techniques in order to support the widest audience possible.

Error handling

When it comes to error handling, you have to take both internal and external parameters into account. Internal parameters include errors that can be generated because of the bugs in your code, or an unexpected user behavior. External errors have nothing to do with the application; rather, they are related to parameters you have no control on. The most important one is the network connectivity. Any interactive bidirectional web application requires, well, an active Internet connection.

Checking network availability

Imagine that your users are enjoying your web app, when suddenly the network connection becomes unresponsive in the middle of their task. In modern native desktop and mobile applications, it is a common task to check for network availability. The most common way of doing so is simply making an HTTP request to a website that is supposed to be up (for example, `http://www.google.com`). If the request succeeds, the desktop or mobile device knows there is active connectivity.

Similarly, HTML has `XMLHttpRequest` for determining network availability. HTML5, though, made it even easier and introduced a way to check whether the browser can accept web responses. This is achieved via the navigator object:

```
if (navigator.onLine) {
  alert("You are Online");
}
else {

  alert("You are Offline");
}
```

Offline mode means that either the device is not connected or the user has selected the offline mode from his/her browser toolbar.

Here is how to inform the user that the network is not available and try to reconnect when a WebSocket close event occurs:

```
socket.onclose = function (event) {
  // Connection closed.
  // Firstly, check the reason.
  if (event.code != 1000) {
    // Error code 1000 means that the connection was closed normally.
    // Try to reconnect.
    if (!navigator.onLine) {
      alert("You are offline. Please connect to the Internet and try
        again.");
    }
  }
}
```

The preceding code is pretty simple. It checks the error code to determine whether the WebSocket connection was closed successfully. Error code 1000 would determine exactly this. If the close event was raised due to an error, the code would not be 1000. In this case, the code checks for connectivity and informs the user appropriately.

You might notice that this is an HTML5 feature. Later, we will discuss polyfills, so the following are two polyfills for checking network connectivity:

- `https://github.com/remy/polyfills/blob/master/offline-events.js`
- `http://nouincolor.com/heyoffline.js/`

The first one is using `XMLHttpRequest`, similar to what Smartphone APIs do.

Fallback solutions

In real life, physical contact is preferred, as it is more direct and efficient, but it shouldn't be the only way of meeting someone. There are numerous cases where you won't be able to handshake, so you'll need to find other methods of communication.

The sad reality of HTML5 is that every browser does not equally support it. Especially considering the new JavaScript APIs, major or minor differences still exist among different browsers. However, even if the browser vendors decided to provide the exact same features for their current releases, there would still be people who cannot or do not want to update. According to StatCounter and W3Counter, as of March 2013, the lion's share of desktop browsing belongs to Google Chrome, followed by Microsoft Internet Explorer and Mozilla Firefox.

Internet Explorer 8 still shares 7 percent, Internet Explorer 7 shares 5 percent, and Safari 5.1 shares 3 percent. A total of 15 percent is translated to a number of customers you probably do not want to miss.

Here come the fallback solutions, which can handle such situations and provide a gracefully scaled-down experience to the users of older browsers. There are two kinds of popular fallbacks nowadays, **Plugins** (such as Flash or Silverlight) and **JavaScript hacks**, formally known as **polyfills**.

JavaScript polyfills

We start by examining polyfills, as they are more close to the native web. JavaScript polyfills are solutions and libraries that mimic a future feature, by providing support for older browsers. Currently, there are polyfill solutions for almost all HTML5-specific feature (**canvas**, **storage**, **geolocation**, **WebSockets**, **CSS3**, and so on).

A polyfill solution should be used in parallel to the standards-based, HTML5-compliant API.

If you need to implement both an HTML5 and a polyfill solution, why not just implement the second one and save time and money? Well, here are four reasons you should use both:

1. Better user experience: When using HTML5, you serve your visitors the best and smoothest experience possible. Everything is handled by the browser, and you only need to focus on your application's requirements. When using a polyfill to address a specific issue, the end-product cannot be of the same quality. Surely, delivering something is better than delivering nothing, but a polyfill is just a patch for those who run poorer vehicles.

2. Performance: The most significant advantage between a native HTML5 solution and a polyfill plugin, is performance. When you request a JavaScript file, you require extra resources, which increase loading time. Moreover, a JavaScript plugin runs way slower than a native browser-implemented method. Regarding WebSockets, the protocol is designed to provide bidirectional full-duplex communication. That is the fastest way you can achieve this kind of staff. What a polyfill can do is to simply mimic full-duplex communication, using traditional **AJAX polling**. We have already seen that AJAX polling is way slower than WebSockets.

3. Future-friendly: Using HTML5 right now lets your website or app to be automatically enhanced from any future browser update. For example, someone who used canvas three years ago, benefitted automatically when Internet Explorer was updated to Version 9.

4. Standards-friendly: Although content, not web standards, should be our top priority, it is good to know that our current implementation consorts with the formal technical specifications. Moreover, the web standards propose what is known as "best practices". Although polyfills usually consist of valid JavaScript code, most of the time they need to address browser-specific bugs and inconsistencies by inserting necessary non-standard code.

Popular polyfills

Modernizr, a well-known library for detecting HTML5 and CSS3 features, provides a great list of HTML5 polyfills that can make your life much easier when it comes to supporting older browsers. Regardless of which HTML5 feature you are using, there is a corresponding polyfill available at `https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-Browser-Polyfills`

Regarding WebSockets, following are a some of the libraries that mimic the WebSocket behavior:

Name	Hyperlink
SockJS	`https://github.com/sockjs/sockjs-client`
socket.io	`http://socket.io/`
Kaazing WebSocket Gateway	`http://kaazing.com/products/kaazing-websocket-gateway.html`
web-socket-js	`http://github.com/gimite/web-socket-js/`
Atmosphere	`http://jfarcand.wordpress.com/2010/06/15/using-atmospheres-jquery-plug-in-to-build-applicationsupporting-both-websocket-and-comet/`

Name	Hyperlink
Graceful WebSocket	`https://github.com/ffdead/jquery-graceful-websocket`
Portal	`https://github.com/flowersinthesand/portal`
DataChannel	`https://github.com/piranna/DataChannel-polyfill`

Except Kaazing, all of the above libraries are open-source and free to use. Some of these libraries use the AJAX approach, while others rely on Flash, in order to emulate the WebSocket behavior.

Here is an example using the Graceful WebSocket library. We chose Graceful WebSocket because it is simple, lightweight, makes no use of Flash, and exposes functionality similar to the WebSocket API.

First of all, download the library, along with jQuery, and include them in your project:

```
<script src=" jquery-1.9.1.min.js"></script>
<script src="jquery.gracefulWebSocket.js"></script>
```

Structure your document as you would normally do and simply replace any reference to the WebSocket native class with the `gracefulWebSocket` once!

Replace this:

```
var socket = new WebSocket("ws://localhost:8181");
```

with this:

```
var socket = $.gracefulWebSocket("ws://localhost:8181");
```

It is that simple! The rest of the WebSocket events and methods remain the same:

```
socket.onopen = function (event) {
  // Handle the open event as previously.
};

socket.onclose = function (event) {
  // Handle the close event as previously.
};

socket.onmessage = function (event) {
  // Handle the message event as previously.
};

socket.onerror = function (event) {
  // Handle the error event as previously.
};
```

Sending data is equally easy and can be done as follows:

```
socket.send("Hello server! I'm a WebSocket polyfill.");
```

In normal mode, the preceding lines of code simply wrap the WebSocket object and execute the native methods. In fallback mode, the library changes the protocol from WS to HTTP, listens for messages by making HTTP GET requests, and sends messages using HTTP POST requests.
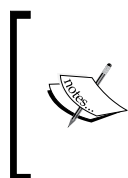
> The specific polyfill solution only required a minor change to our code. Other solutions might need you to make a lot of modifications or only work with specific server back-ends. You need to pay close attention to the requirements of each plugin, its usage, and documentation before using it for production.

Browser plugins

Browser plugins have been an extremely helpful solution for rich Internet applications in the pre-HTML5 era. To name but a few, developers used to offer desktop-rich functionality in their websites utilizing the capabilities of Flash (primarily), Silverlight, or Java. A few years ago, basic UX effects, transitions, and animations could not be made using plain HTML, CSS, or JavaScript.

To fill this gap, browser plugins provided the developers with a framework which could be installed in the client browser and allowed richer content.

Browser plugins have several drawbacks that make them deprecated day-by-day. They are resource-intensive, the user needs to wait more until a page is fully loaded, and they are mostly based on proprietary technologies. As a result, more and more companies (including Apple and Microsoft) are shifting away from browser plugins in favor of HTML5.

However, if your users browse using an old browser, it is likely that they have an old desktop PC with one or more such browser plugins installed. Some great WebSocket implementations use Flash in order to achieve bidirectional communication, and so do some of the polyfills mentioned previously.

The **websocket-as**, available at `https://github.com/y8/websocket-as`, is a popular utility, written in ActionScript, which implements a WebSocket API like the HTML5 approach. A similar example exists for Microsoft's Silverlight and WCF technologies too (`http://www.codeproject.com/Articles/220350/Super-WebSockets-WCF-Silverlight-5`).

If you are familiar with Flash or Silverlight, then you could implement a fallback solution based on your preferred browser plugin. Otherwise, you can stick to the JavaScript implementations.

Summary

Not all browsers support the WebSocket protocol natively. As a result, you need to provide some fallback solutions for those users who cannot sense the HTML5 goodies. Fortunately, the open-source community has provided us with various techniques, which emulate the WebSockets' features using plain HTTP or Flash internally. Implementing both the HTML5 and the fallback is critical for your web apps and is strongly related to the audience width you want to reach. In this chapter, we examined some popular fallback techniques and saw how to handle common connectivity errors in your WebSocket applications. That's all you need to know for the WebSocket and HTML part. In the last chapter, we are going to examine the WebSocket protocol in terms of native mobile experience.

7
Going Mobile (and Tablet, Too)

WebSockets, as their name implies, is something that uses the web. The web is usually interwoven with browser pages because that's the primary means of displaying data online. However, non-browser programs too use online data transmission. The release of the iPhone (initially) and the iPad (later) introduced a brand new world of web interconnectivity without necessarily using a web browser. Instead, the new smartphone and tablet devices utilized the power of native apps to offer a unique user experience.

Why mobile matters

Currently, there are one billion active smartphones out there. That is, millions of potential customers for your applications. Those people use their mobile phone to accomplish daily tasks, surf the Internet, communicate, or shop.

Smartphones have become synonymous to apps, and nowadays, there is an app for any usage you can think of. Most of the apps connect to the Internet in order to retrieve data, make transactions, gather news, and so on.

Wouldn't it be great if you were able to use your existing WebSocket knowledge and develop a WebSocket client running natively on a smartphone or tablet device?

Native mobile app versus mobile website

Well, this is a common conflict and as usual, the answer depends on your needs and your target audience. If you are familiar with the modern design trends, designing a website that is responsive and mobile friendly, is now a must. However, you should be sure that the content, which is what really matters, is equally accessible via a smartphone, as it is via a classic desktop browser.

Definitely, a WebSocket web app will run on any HTML5-compliant browser, including mobile browsers such as Safari for iOS and Chrome for mobile. So, no need to worry about compatibility issues on modern smartphones.

What happens though if your app utilizes device-specific information such as offline storage, GPS, notifications, or accelerometers, along with WebSockets? You need a more native implementation in a language other than HTML and JavaScript. W3C is defining some specifications that will let the client access hardware such as camera, GPS, and accelerometer. However, only a minority of modern web browsers currently support such specifications. At the time of writing, a native approach is the way to go, though the client-side future seems way more interesting! iOS uses Objective-C, Android uses Java, and Windows Phone uses C#. If you think that your mobile use-case scenario does not utilize any of the smartphone goodies, feel free to go with the browser-based app. If native functionality is required, then a native solution is necessary. That's exactly what are we going to build in this chapter!

Prerequisites

In order to develop a smartphone app, you need to install the development tools and SDKs of your preferred target. The philosophy behind the examples we'll demonstrate is fundamentally the same in the three major mobile operating systems, that is, iOS, Android, and Windows.

If you do not already have a mobile SDK installed, here is where you can pick one (they are all free):

Platform	Targets	SDK URL
iOS	iPhone, iPad	`https://developer.apple.com/devcenter/ios/`
Android	Android phones & tablets	`http://developer.android.com/sdk/`
Windows	Windows Phone, Windows 8	`http://developer.windowsphone.com/ develop` & `http://msdn.microsoft.com/ windows/apps`

We suppose that you are familiar with at least one of the above SDKs and technologies. If not, you can navigate to the corresponding developer portal, and follow the online resources and tutorials, which provide a great starting point.

Throughout this chapter, we have decided to provide code samples for iOS, but feel free to use the platform you are most familiar with.

Installing the SDK

After downloading the desired SDK, you follow an automated wizard that installs it in your system. Note that the iOS SDK can only run on a Mac operating system, the Windows SDK runs on a Windows operating system, and the Android SDK runs on Mac, Windows, or Linux. Along with the SDK, there are a couple of automatically installed goodies:

- Smartphone/tablet simulators
- An integrated development environment where you write and debug your code

Although you should always test your code on real devices (phones and tablets), the simulator is a really solid solution for constant debugging.

Considering iOS, here are the iPhone and iPad simulators, running iOS 6.

The following figure shows an iPhone simulator:

The following figure shows an iPad simulator:

Testing our existing code in the mobile browser

Remember the HTML and JavaScript code we wrote back in *Chapter 2, The WebSocket API*? Having the SDK and simulators installed, we can access the web using the mobile browser included in the simulator. We can also access our local HTML, CSS, and JavaScript files without uploading them to a web server. Here is the chat client running pretty well on an iPad.

The following image shows WebSocket web app on Safari for iOS (no modifications in code):
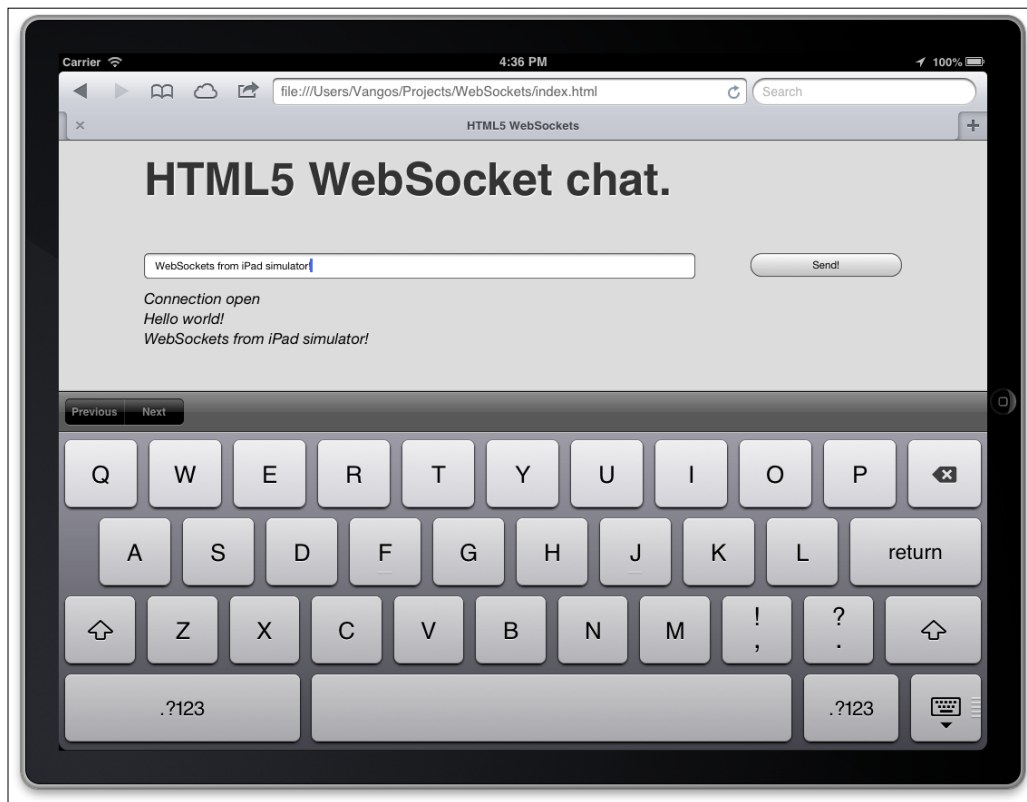
Going native

So, what if your application supports device-specific or offline features, and you still want to use WebSockets when the web is available?

You go native.

Fortunately, all of the major mobile platforms support WebSockets, so you need to make absolutely no changes in your server code! After all, HTML5 is a front-end client just like iPhone or iPad. Using the same techniques as the JavaScript samples, you are going to build the same application in Objective-C. The process is similar to any other mobile platform, so do not worry if you are not familiar with the Objective-C concept.

Creating the project

To begin with, open **XCode**, the development environment provided by Apple for building iOS apps. Eclipse and Visual Studio are the Android and Windows equivalents.

Follow the given steps for creating the project

1. Launch XCode and click on **Create a new XCode project**. The following screenshot shows XCode launch screen:

2. Create a single view application. Provide a name, along with company and organization identifiers if you want to. For example, name the app WebSocketsMobile. Then, select a local folder to place it into, as shown in the following screenshot:
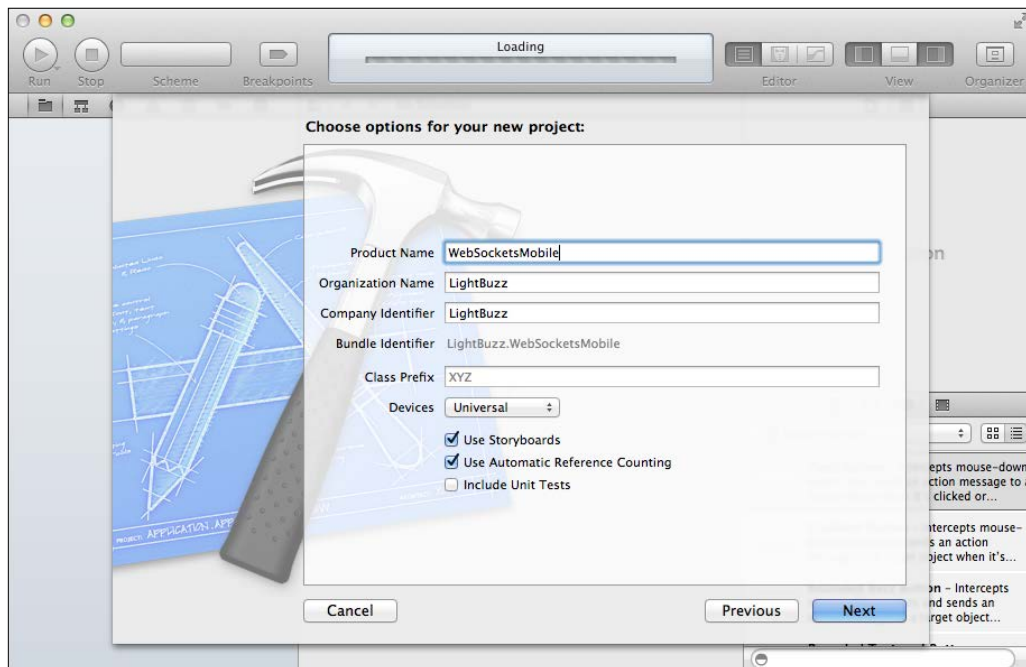
Creating the WebSocket iPhone app

If you need to deploy your app for production, you need to additionally specify some icons for the target platforms. We'll skip such stuff for now, but feel free to add any resources your app might require. XCode automatically creates some files for us. The storyboard files (one for iPhone and one for iPad) will let us create the user interface of our app and the `ViewController` file will handle all the programming logic.

The following screenshot shows the initial UI of our iPhone app:
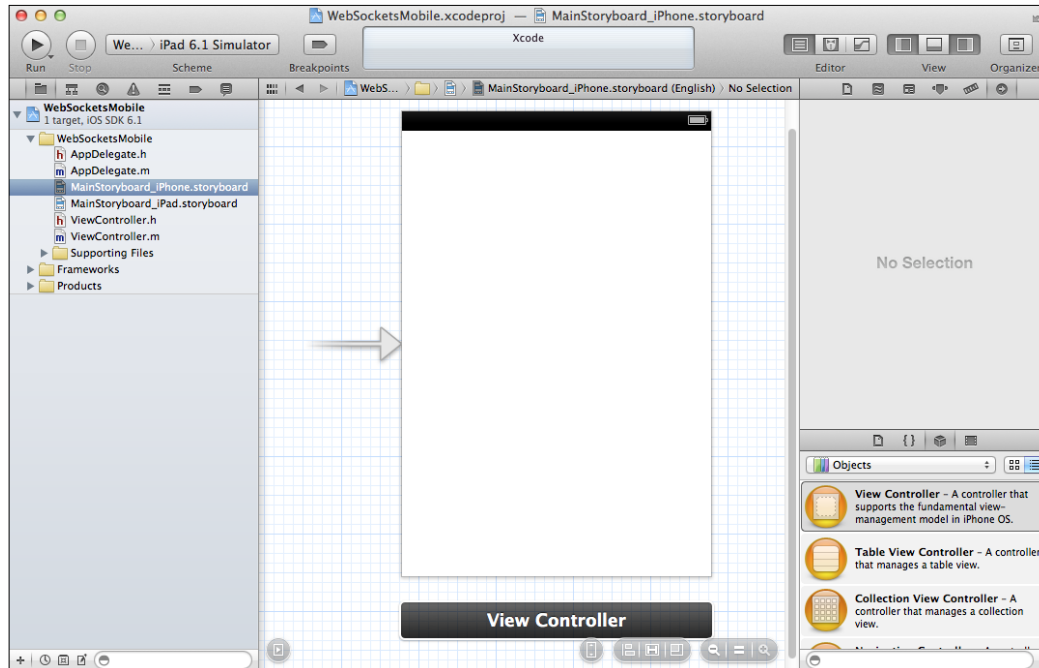
1. Add some controls to the user interface builder. For learning purposes, we'll try to keep it simple and only add `UITextField` for writing a message, a button for sending the message, and `UILabel` for displaying the chat messages. Remember to set the number of lines of the label to 0 (that is, infinite). Do not forget to connect the outlets with **View Controller,** using the assistant editor (`http://www.techotopia.com/index.php/Establishing_Outlets_and_Actions_using_the_Xcode_Assistant_Editor`). The following screenshot shows the iPhone app user interface:
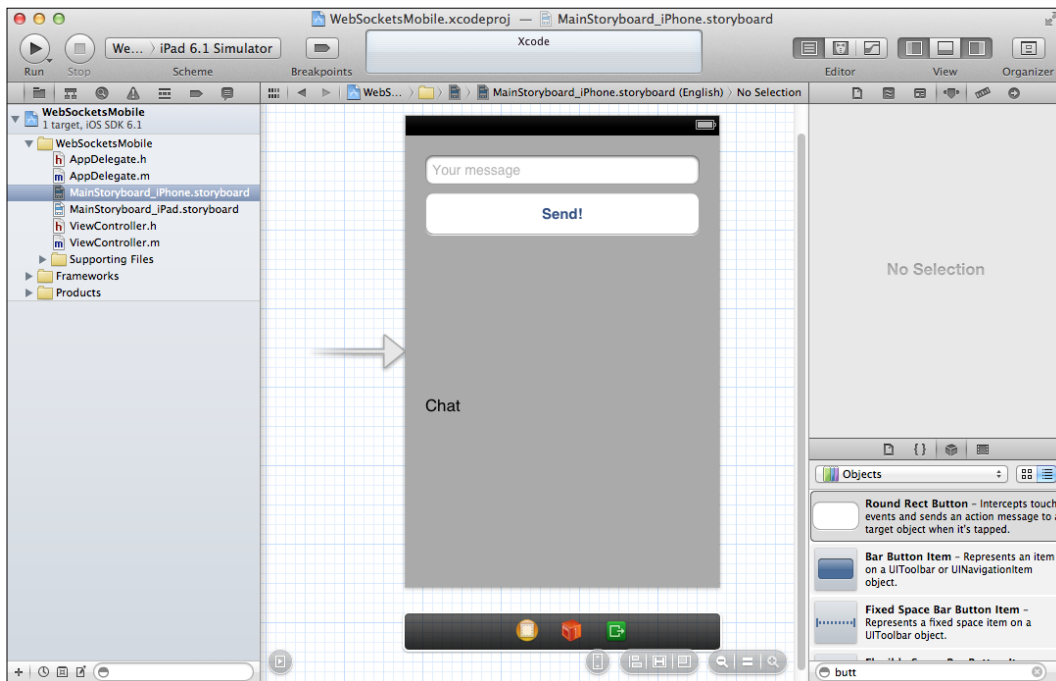
2. Download the `UnittWebSocketClient` library and include it in your
    project. This library handles most of the WebSocket functionality. You can
    pick another one or implement your own. Follow the directions specified at
    `https://code.google.com/p/unitt/wiki/UnittWebSocketClient`.

3. Include the header files of the library in your project and specify your View
    Controller as a `WebSocketDelegate`. Then subscribe for the corresponding
    events, which are identical to the JavaScript ones:

```
// ViewController.h

#import <UIKit/UIKit.h>
#import "WebSocket.h"
@interface ViewController : UIViewController <WebSocketDelegate>
@end

// ViewController.m

#import "ViewController.h"
```

```objc
@interface ViewController ()

@end

@implementation ViewController

- (void)viewDidLoad
{
  [super viewDidLoad];
}

- (void)didReceiveMemoryWarning
{
  [super didReceiveMemoryWarning];
}

- (void)didOpen
{
  // JavaScript event: onopen
}

- (void)didClose:(NSUInteger)aStatusCode message:(NSString *)
aMessage error:(NSError *)aError
{
  // JavaScript event: onclose
}

- (void)didReceiveError:(NSError *)aError
{
  // JavaScript event: onerror
}

- (void)didReceiveTextMessage:(NSString *)aMessage
{
  // JavaScript event: onmessage
}

- (void)didReceiveBinaryMessage:(NSData *)aMessage
{
  // JavaScript event: onmessage
}

@end
```

4. It is now time to populate the methods, as we did in the JavaScript samples. Here's what you need to do in order to set the app up and running:

```objc
// ViewController.h

@interface ViewController : UIViewController <WebSocketDelegate>
{
  // Create a new WebSocket object.
  WebSocket *socket;
}

// ViewController.m

- (void)viewDidLoad
{
  [super viewDidLoad];

  // Specify the WebSocket configuration. The only necessary
parameter is the URL.
  WebSocketConnectConfig *config = [WebSocketConnectConfig
    configWithURLString:@"ws://echo.websocket.org"
    origin:nil protocols:nil tlsSettings:nil headers:nil
      verifySecurityKey:YES extensions:nil];

  // Initialize the WebSocket object.
  socket = [WebSocket webSocketWithConfig:config
    delegate:self];

  // Open the WebSocket connection and start listening for
    events.
  [socket open];
}

- (void)didReceiveTextMessage:(NSString *)aMessage
{
  // JavaScript event: onmessage

  labelChat.text = [NSString stringWithFormat:@"%@\r%@",
    labelChat.text, aMessage];
}

- (IBAction)sendTapped:(id)sender
{
  [socket sendText:textMessage.text];
}
```

The following figure shows the native iOS WebSocket client up-and-running!

What about the iPad?

Although the app you created would run pretty fine on iPad devices, it is always better to provide a different interface for tablets. Simply navigate to the `MainStoryboard_iPad.storyboard` file, rearrange the UI elements, and provide tablet-specific functionality. Then, select the project targets and, while in the Summary tab, expand the `iPad Deployment Info` option and ensure that `MainStoryboard_iPad` is selected. You can select the iPhone storyboard if your app is not too complex or specify that the app target is iPhone only. This way, when someone runs your app on an iPad device, he/she will see a smaller screen centered in the iPad device.

Summary

In this chapter, we found out how WebSockets can act as a universal hub for transmitting messages between connected mobile and tablet clients. We implemented a native iOS application, which communicates with a WebSocket server just like the HTML5 JavaScript client.

Appendix

It is not possible to cover everything in a single book. There are some things that were intentionally or accidentally left out. So, here are some extra topics that will let you dive deeper into the WebSocket world.

Resources

The WebSocket API is expanding day-by-day. In order to keep up with the forthcoming changes, here are a few online resources you can bookmark for further reading.

Online sources

The following websites provide up-to-date content regarding the WebSocket protocol, applications, and real-world examples. Have a look at them, keep an eye, and bookmark the ones you like most.

WebSocket.org	`http://www.websocket.org/`
Web Platform docs	`http://docs.webplatform.org/wiki/apis/websocket/WebSocket`
HTML5 rocks	`http://www.html5rocks.com/en/features/connectivity`
HTML5 demos	`http://html5demos.com/`
Mozilla Developer Network	`https://developer.mozilla.org/en-US/docs/WebSockets`
The WebSockets API (W3C)	`http://www.w3.org/TR/2009/WD-websockets-20091222/`

Articles

Need more food for thought? These articles present personal opinions of well-known bloggers. You'll even read controversial subjects, but you'll surely find out that there is no black or white in the web industry.

WebSockets versus REST… fight!	`http://nbevans.wordpress.com/2011/12/16/` `websockets-versus-rest-fight/`
HTML% WebSocket cheat sheet	`http://refcardz.dzone.com/refcardz/` `html5-websocket`
Would You Let Your Grandma Use WebSockets?	`https://community.qualys.com/blogs/` `securitylabs/2012/08/15/would-you-let-` `your-grandma-use-websockets`
Your users don't care if you use WebSockets	`http://www.hanselman.com/blog/` `YourUsersDontCareIfYouUseWebSockets.aspx`
WebSockets and the risks of the unfinished standards	`http://news.cnet.com/8301-30685_3-` `20025272-264.html`

Source code

The source code we demonstrated in this book can be found online at `http://pterneas.com/books/websockets/source-code`. Note that the given link will always be up-to-date, following the current trends and standards.

You can download and modify all of the included files as you wish.

System requirements

Web Standards is a cross-platform mechanism. This means that the client-side source code will run on any HTML5-compliant browser. You only need a text editor such as Notepad or GEdit to modify the files.

The server-side code has been tested in Windows, though you can run it using any operating system that supports the Mono framework (http://www.mono-project. com/). Finally, regarding the iOS source code, you need a Mac computer, along with the XCode development environment.

Remember that you can use the operating systems, server-side libraries, and IDEs of your choice to build your own projects. The main logic and functionality remains the same.

Stay in touch

Found a bug or have any changes to propose? We would be glad to listen to your feedback and fix any issues as soon as possible. Simply send your message to `vangos@pterneas.com`.

Index

[PACKT] open source*
P U B L I S H I N G   community experience distilled

Thank you for buying
Getting Started with HTML5 WebSocket Programming

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

HTML5 Mobile Development Cookbook

ISBN: 978-1-84969-196-3   Paperback: 254 pages

Over 60 recipes for building fast, responsive HTML5 mobile websites for iPhone 5, Android, Windows Phone, and Blackberry

1. Solve your cross platform development issues by implementing device and content adaptation recipes

2. Maximum action, minimum theory allowing you to dive straight into HTML5 mobile web development

3. Incorporate HTML5-rich media and geo-location into your mobile websites
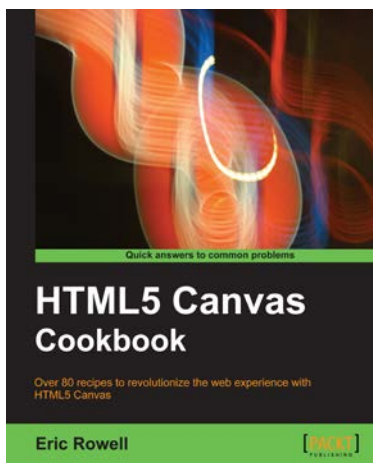
## HTML5 Canvas Cookbook

ISBN: 978-1-84969-136-9   Paperback: 348 pages

Over 80 recipes to revolutionize the web experience with HTML5 Canvas

1. The quickest way to get up to speed with HTML5 Canvas application and game development

2. Create stunning 3D visualizations and games without Flash

3. Written in a modern, unobtrusive, and objected oriented JavaScript style so that the code can be reused in your own applications

4. Part of Packt's Cookbook series: Each recipe is a carefully organized sequence of instructions to complete the task as efficiently as possible

Please check **www.PacktPub.com** for information on our titles
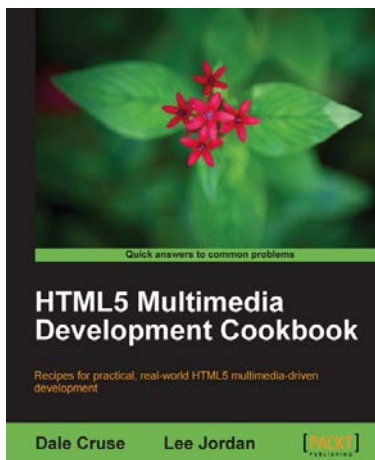
HTML5 Multimedia Development Cookbook

ISBN: 978-1-84969-104-8   Paperback: 288 pages

Recipes for practical, real-world HTML5 multimedia-driven developmnet

1. Use HTML5 to enhance JavaScript functionality. Display videos dynamically and create movable ads using JQuery

2. Set up the canvas environment, process shapes dynamically and create interactive visualizations

3. Enhance accessibility by testing browser support, providing alternative site views and displaying alternate content for non supported browsers

HTML5 for Flash Developers

ISBN: 978-1-84969-332-5   Paperback: 322 pages

Leverage your Flash skill set and learn to create content using a wide range of HTML5 web development features

1. Discover and utilize the wide range of technologies available in the HTML5 stack

2. Develop HTML5 applications with external libraries and frameworks

3. Prepare and integrate external HTML5 compliant media assets into your projects

Please check **www.PacktPub.com** for information on our titles