Oracle Press™

Java WebSocket Programming

About the Author

Danny Coward is a Chief Architect and Web Architect at Oracle. He is the Specification Lead for the Java API for WebSocket for Java EE and Java SE/JavaFX. Coward's work leading WebSockets at Oracle makes him the leading expert on Java WebSocket programming. Coward has specialized experience in all aspects of Java software—from Java ME to Java EE to the founding of the JavaFX technology.

About the Technical Editor

Dr. Santiago Pericas-Geertsen is a Principal Member of Technical Staff in the Sun Glassfish organization at Oracle, and an architect and technical lead in the Avatar project. Santiago is a Specification Lead for JSR 339, JAX-RS 2.0. While at Sun Microsystems, Santiago was a technical lead for the Glassfish Mobility Platform, a developer and lead in the Fast Web Services project, and a participant and editor in World Wide Web Consortium (W3C) initiatives. He holds two US patents, 7647415 and 7716577. Santiago blogs from Java.net, tweets from @spericas, and has presented at numerous academic and industry-oriented conferences.

Oracle Press™

Java WebSocket Programming

Danny Coward

New York Chicago San Francisco
Athens London Madrid Mexico City
Milan New Delhi Singapore Sydney Toronto

Cataloging-in-Publication Data is on file with the Library of Congress

Java WebSocket Programming

1 2 3 4 5 6 7 8 9 0 QFR QFR 1 0 9 8 7 6 5 4 3

ISBN 978-0-07-182719-5
MHID 0-07-182719-6

Sponsoring Editor
Brandi Shailer

Editorial Supervisor
Patty Mon

Project Manager
Harleen Chopra,
Cenveo® Publisher Services

Acquisitions Coordinator
Amanda Russell

Technical Editor
Santiago Pericas-Geertsen

Copy Editors
Emily Rader
Nancy Rapoport

Proofreader
Susie Elkind

Indexer
~~TBD~~

Production Supervisor
George Anderson

Composition
Cenveo Publisher Services

Illustration
Cenveo Publisher Services

Art Director, Cover
Jeff Weeks

This book is dedicated to Bill, Jared and Alex.

Contents at a Glance

Contents

Acknowledgments

Many thanks to Santiago for his thoughtful review comments during the writing of this book and to Brandi Shailer and Amanda Russell at McGraw-Hill Education for keeping me on track.

Introduction

The WebSocket protocol is a new networking protocol for web developers' burgeoning toolbox. Aside from its inclusion as a core technology in HTML5 and its rapid adoption by all the major browsers from desktop to tablets and smartphones, why would the web developer care about yet another web technology?

The Long Poll

By 2001, most major corporations worldwide had some kind of web presence. In the developing world, the personal computer revolution resulted in most households having Internet access through at least one channel. Businesses were rapidly building their presences on the Internet as a means of showcasing their products and services, and as a growing channel through which to deliver them. The basic technologies of the Web, such as HTTP, HTML, and JavaScript, were powering a revolution in how people interacted with one other, with their schools, and with their places of work; how they planned vacations; and even how they bought groceries.

Web sites grew from static and colorless catalog-style first efforts as developers sought new ways to make the web sites more interactive. They looked to add life to their web sites by injecting interesting information to their viewer at appropriate times and updating the information on the page as necessary. But developers found limits to basic HTTP and markup technologies. Developers needed to update the stock quote, the latest bid, the current list of friends logged into the same site, the new reduced price, the game result, and the latest celebrity to fall from grace. And they needed to do it all without relying on continuous interaction from the user. They needed to initiate data updates from the web servers to keep the web site fresh, engaging,

and interesting. They needed a visitor to their web site to be transformed into a viewer of their channel, and they needed visitors to do as little as possible to absorb the information that was pushed to them.

Over the next few years, developers looked to a variety of less-than-formal means to accomplish the task of updating all the current visitors to a web site with the latest information of a variety of kinds. The most obvious means was to have the browser poll the server for updates. Developers would embed a short snippet of JavaScript into the relevant web page to force the browser to refresh the whole page at a predefined interval. This method would refresh all the data, whether the data fetched needed a refresh or not. The latency of the network was apparent in this method, and so the user experience was not good, even apart from the problem of fetching unneeded data.

A slightly more sophisticated approach was to use the HTTP Keep Alive mechanism. In this approach, JavaScript code inside the web page would keep open a long-lived HTTP connection, like a never ending download, that would periodically update with new information. Large differences in how long browsers and servers would keep the connection open resulted in problems, and typically, the client's browser would need continually to reopen the connection, whether or not there was data to fetch from the server.

As developers approached these techniques, development frameworks such as Comet and AJAX grew to support them and encompass these basic techniques. To some extent, they could hide some of the deficiencies of these basic approaches. There were, however, two fundamental problems that even the best implementations could not overcome. First, HTTP is an expensive network protocol for sending simple information. Just to ask for a simple stock quote update, the context of the connection has to be recalled each time it is made: all the header information such as that which qualifies the client and server platform, the authentication properties, the descriptions of the payload, and so on. Second, and worse, the expensive connections were made whether the server had new information to convey or not.

Enter WebSockets

In 2009, work started on a technology that would allow the client to establish a lightweight connection with the server that would allow bi-directional communication and a lightweight content model. Servers would be able to push data to their connected clients only when they needed to. Once the

connection was established, client and server alike would have a way to send simple information without having to re-create the context of the connection each time a message was sent.

The days of needless updates were coming to an end.

To understand how wasteful of network resources polling approaches can be, consider an auction web site. Items are posted on the web site, and users can place bids on the item for a defined period of time, at the end of which, the item is sold to the highest bidder. Throughout the length of the auction, any user visiting the auction page is able to view the current bid and use that information to decide on a higher bid. Let's say that the site puts up a highly desirable item for auction—a slightly used iPod (it is, after all, 2003)—and the time period of the auction for this item is only one hour. Suppose that when a new bid is made for the iPod, the web site needs to convey only a short message with the new price and perhaps some accompanying information like the bidder's online name. Let's estimate that you can always fit that into 64 bytes. Let's say that given that all the users are logged in, there are some cookies that need to be conveyed in the HTTP request for new information. Together with the content type header, perhaps a couple of application-specific headers, the content length, the browser ID, and so on, let's estimate about 512 bytes for the header information. Now assume you have, on average, 100 users logged into this web site, and on average, a new bid is made every 30 seconds throughout the auction. And let's say that the frequency of bidding is not evenly distributed; at some points, bids may be a few seconds apart (close to the end) and at other times (at the beginning) the bids may be minutes apart. A bidder will always want the latest price information available, so it would be prudent to refresh the price every two seconds; otherwise, a bidder may become frustrated when a bid does not go through because another bid was made before the bidder got the update. Let's add up all the data sent in order to receive the updates:

30 updates per minute for 60 minutes = 1800 updates.

Each update carries about 512 bytes in header information each way.

Total header information sent and received = 1800 x 512 bytes = 921,600 = 900KB.

Now, we said that each of the updates would contain 64 bytes. And in this auction, we said that there were 120 bids. So the total update information = 120 x 64 bytes = 7.5KB.

So a rough efficiency ratio of (useful data) to (repeated data) is calculated at 7.5 / 900 = 0.8%.

Not a good score, and we won't even get into how this efficiency ratio decreases for longer auctions.

The goal of WebSockets is to dramatically increase this kind of network efficiency by sending contextual information only when setting up the connection, and once the connection is established, to allow either peer of the connection to send messages, even simultaneously, with a minimal amount of contextual information to identify the message.

In this way, web pages connected to web servers are able to receive updates only when the server decides they needed them, and when such updates are sent, it is not necessary to weigh down the message with a large payload of contextual information about the connection.

Introduction to the WebSocket Protocol

The WebSocket protocol is network protocol that allows two connected peers full duplex messaging over a single TCP connection. A good analogy for WebSockets is a phone call. When making a phone call, you initiate the call by dialing a number. If the party you are trying to call accepts the call by picking up the receiver, the connection is established. While the connection is active, both parties can speak at the same time if they want to (although it's not recommended for a free flowing conversation!) and both parties can hear what is being said even as they are talking themselves. This is what is meant by full-duplex communication. The connection remains active, whether or not anyone is talking, until either one of the parties decides to hang up.

In the case of WebSockets, the connection is established by means of an HTTP interaction WebSocket endpoint. The initiator of the connection sends a specially formulated HTTP request containing the URL of the WebSocket endpoint with which it wants to connect. That starts the ball rolling and is called the *opening handshake*. If the server is willing to accept the connection, the server formulates a special HTTP response called the *opening handshake response* and sends it back to the client. Now, the TCP connection may be established, and WebSocket messaging back and forth can ensue. The connection remains active until either party decides to terminate the connection, or until something external brings down the connection—such as a timeout due to inactivity or a problem with the physical network.

WebSockets are primarily used as a communication mechanism between web applications residing on a web server and browser clients, although there is nothing specific in the protocol that requires this deployment setup. Equally, WebSocket connections could be established between any two peers on the network, not necessarily a browser and a web server. However, because of the technology's roots, the most immediate opportunity for WebSocket technology is to bring to life otherwise static web sites, and easily enhance web sites and web applications with live data and activity.

In this setup, the WebSockets residing in the browser are created using a JavaScript API that has been standardized by the W3C, called the JavaScript WebSocket API. The WebSocket residing in the web server hosting the web site may be written in any number of languages. The popularity of WebSocket has been so sudden that many different languages and web platforms have begun to embrace it. In particular, the Java platform has been quick to build in support for WebSocket. The primary focus of this book is to explore the facilities of this new Java API, although some of the examples will rely on code that uses the JavaScript API for WebSockets.

The Java WebSocket API is a core feature of the latest Java EE 7 platform and any web developer already familiar with the other Java and/or Java-based technologies for building web applications, such as Java Servlets, JSPs, JavaServer Faces, or any of the many relatives of those technologies, should become familiar with WebSocket in the Java platform and think about bringing a fresh and modern feel to existing or future web applications by incorporating this technology. With this book, you will learn how to write WebSocket applications. You will learn all the major facilities of the Java WebSocket API, including the various messaging modes at your disposal and configuration options for your WebSocket applications, along with where you can store application state, how to configure WebSockets so that they can only be accessed securely, and how to integrate WebSockets into Java EE applications. The things you learn in each chapter of the book are backed up by example applications that illustrate the technical facilities.

This book contains eight chapters.

Chapter 1: Java WebSocket Fundamentals

In this chapter, you dive straight into your first WebSocket application, the Echo application. Though simple, this application and this first chapter introduce the main features of the Java WebSocket API, and in so doing, form the foundation for the rest of the chapters in the book.

Chapter 2: Java WebSocket Lifecycle

The second chapter examines the lifecycle of a WebSocket endpoint, the central component that you create in a WebSocket application. This lifecycle defines the framework by which you can manage resources used by the WebSocket endpoint, and most importantly, defines how to intercept WebSocket messages. The lifecycle is illustrated using the Lifecycle application, which presents the user with a set of traffic signals that highlight the key stages in the lifetime of a WebSocket endpoint.

Chapter 3: Basic Messaging

This chapter explores the fundamental aspects of sending and receiving messages in a web application, using a collaborative group drawing application as its example application. The example uses a Java application as a client, so this chapter also illustrates the use of the Java WebSocket API on the client side as well.

Chapter 4: Configurations and Sessions

Chapter 4 shows two of the most important objects in the Java WebSocket API, the WebSocket session object, which represents a conversation with a WebSocket peer, and the endpoint configuration object, which holds the configuration information for an endpoint. These objects are put into play to illustrate their features in an online chat example application.

Chapter 5: Advanced Messaging

This chapter makes a thorough examination of all the options available to developers for sending and receiving WebSocket messages. Building on Chapter 3, we take a look at advanced topics, such as message encoding and decoding strategies, and synchronous and asynchronous messaging modes. The example application in this chapter uses a user interface to illustrate the messaging options available in the API.

Chapter 6: WebSocket Path Mapping

Covering the nine rules of path mapping, this chapter explores all the options available in the Java WebSocket API for publishing a WebSocket endpoint to a URI so that peers are able to connect to it. Using a simple

stock portfolio application, the chapter shows techniques of exact path mapping, template mapping, and query strings, and discusses the kind of situations in which you might choose one technique over another.

Chapter 7: Securing WebSocket Server Endpoints

In this chapter, you learn how to limit access to a WebSocket endpoint only to certain users of a web application, and you learn how you can ensure that communication between WebSocket endpoints remains private. We revisit the stock portfolio application, applying security techniques to secure and personalize the application.

Chapter 8: WebSockets in the Java EE Platform

The last chapter starts an exploration of how to integrate WebSocket endpoints into a larger Java EE application. Providing ways to share application information between WebSocket endpoints and other Java web components, and with Enterprise JavaBeans, we rework the chat application of Chapter 4 to leverage Java Servlets and EJBs, the two key components of the Java EE platform.

Intended Audience

This book is suitable for the following readers:

Web developers who wish to enhance web applications with interactive features

Rich client application developers who wish to interact with a WebSocket server application

Java EE developers interested in developing applications for HTML5-enabled browsers

The book assumes you have a working knowledge of the Java programming language, some experience with the Java SE platform, and some knowledge of the JavaScript programming language. Some experience with web application development is probably useful in addition, although not necessary.

Retrieving the Sample Applications

All the sample application code can be downloaded from the Oracle Press web site at www.OraclePressBooks.com. The files are contained in a Zip file. Once you've downloaded the Zip file, you need to extract its contents. The samples run on the Glassfish 4.0 application server, which you can freely download from http://glassfish.java.net/, and can be built using the NetBeans IDE, which can be freely downloaded from http://netbeans.org/.

I hope this book inspires you to write applications no one has thought of yet!