4 Messaging: communicating to and from scripts in HTML5

This chapter covers

- Server-sent events and event-driven communications from the server
- WebSockets for bidirectional, event-driven communication
- Client-side messaging between pages from different domains

In the last decade, the web has moved from communication based on uploading static content, similar to the traditional print publishing model, to a real-time communication system where tweets and friendings are instantly announced to hundreds of followers. We've all become so used to dynamically updating web pages that we don't realize most of this is built as a series of hacks on top of HTML4 and HTTP 1.0/1.1. HTML5 cleans up these hacks by providing well-defined APIs for messaging—between the browser and web servers and between different iframes or other objects loaded in the browser.

Because messaging is a complex subject, this will be a complex chapter. You're going to do a lot and learn a lot. Specifically, you're going to

- Learn how to use server-sent events (SSE). This new client-server API allows communication from the server without a specific client request.
- Learn how to use WebSockets.
- Dabble in one of the new event-driven, server-side technologies: Node.js.
- Learn about cross-document messaging, an API for communication between pages and scripts already loaded in the browser.

Why build this chapter's chat and planning board applications?

- You'll build a chat application based on a traditional LAMP/WIMP (Linux, Apache, MySQL, PHP/Windows, IIS, MySQL, PHP) server stack to learn about SSE.
- You'll build a collaborative agile planning board with WebSockets and Node.js.

After you build those two applications, we'll show you how to integrate them on the client using cross-document messaging.

If you need background on the principles of computer networking, take a side trip through appendix D. It'll help you understand the performance trade-offs to using the new HTML5 client-server APIs, as well as define terms like *protocol*, *network stack*, *latency*, *throughput*, *polling*, and *event-driven*. The appendix will also give you the background to understand why and when to use the new approaches we introduce in this chapter, such as server-sent events, which we cover in the next section.

4.1 Server-sent events (SSE)

Server-sent events (SSE) allow the web server to create an event in the browser. The event can contain raw data or it can be a notification or a ping. The API for SSE in the browser is the event listener in JavaScript, created using the same `addEventListener()` method you'd use for any other event listener. The only difference is that instead of adding a listener to the `document` object or an element, you add it to an instance of the new `WebSocket` object. Why is this any better than requesting new data with AJAX? SSE offers two main advantages:

- The server drives communication.
- There's less overhead of repeatedly creating a connection and adding headers.

In this section you'll learn how to use SSE as you build a simple chat application. As the section winds down, you'll also learn when it's good to use SSE and when another tool might be better.

4.1.1 A simple SSE chat application

Server-sent events are delivered to the browser in the form of a special file the browser requests by creating an `EventSource` object. Instead of a regular HTML file or image, the browser requests an event stream. Normally, the server attempts to deliver any file as fast as possible, but with the event stream the file is purposely delivered slowly. The browser stays connected to the server for as long as the file takes to be delivered, and the server can add data to the file at any time. This approach is identical to that used by the forever frame technique (defined in appendix D) except that instead of developers having to decide for themselves how to format the response, the format is laid down in the HTML5 standard. In return for following SSE conventions, you use the familiar `addEventListener()` approach you'd use for any other events.

Server-sent events	9	6	N/A	11	5

As we discuss how to build an SSE chat application, we'll focus on the front-end code, because we're not trying to teach PHP or MySQL. That said, the easiest way forward is to download the server files, listed in the "Chat application prerequisites" sidebar.

Chat application prerequisites

You'll need the following programs to make the application in this section work:

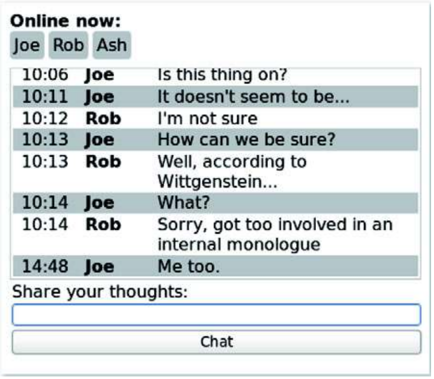
- *A web server that can host PHP*—We used Apache (http://apache.org/) for the example, but IIS on Windows also should work.
- *PHP*—Download from http://php.net/ with PDO support.
- *MySQL*—Download from http://dev.mysql.com/.
- *jQuery*—Download from http://jquery.com/ (included in code download).

The other files you need are available in the code download section of our book's website. If you don't want to do the setup yourself, you can also get all the needed components as part of most inexpensive web-hosting packages.

Figure 4.1 shows a screenshot of the finished application.

As you can see, the user types a message into the text input and hits Enter or the Chat button, and his words of wisdom are immediately distributed to everyone else online. The chat shown in figure 4.1 is, of course, entirely manufactured. Rest assured; the authors are not that corny in real life.

As you might guess from the name, server-*sent* events, the server sends events to the browser; it can't receive information via SSE. Communication from the browser back to the server, new chat messages entered by the



Online now:
Joe Rob Ash

10:06	**Joe**	Is this thing on?
10:11	**Joe**	It doesn't seem to be...
10:12	**Rob**	I'm not sure
10:13	**Joe**	How can we be sure?
10:13	**Rob**	Well, according to Wittgenstein...
10:14	**Joe**	What?
10:14	**Rob**	Sorry, got too involved in an internal monologue
14:48	**Joe**	Me too.

Share your thoughts:

Figure 4.1 The simple chat application in action

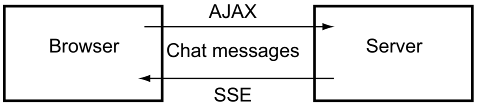


Figure 4.2 The conceptual flow of chat messages in this section's application. Messages will be sent back to the server using standard AJAX techniques, but chat messages will be received from the server through server-sent events.

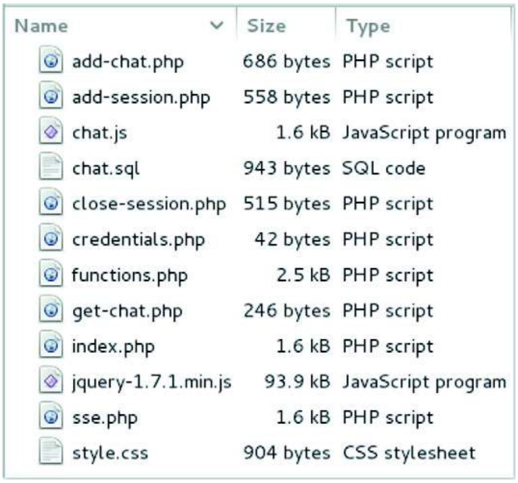
user, will use traditional AJAX methods. Figure 4.2 illustrates the flow of chat messages in the application.

The file structure you'll create, and which is provided in the companion source code for this book, is illustrated in figure 4.3.

For everything to work, these files will need to be located in a directory where your web server can find them. For Apache, this will likely be under /var/www/html, and for IIS, this will be C:\Inetpub\WWWRoot; check the details in the documentation for your OS and web server. Usually these folders have restricted access, so either create and edit the files in your home directory and copy them across or run your editor with appropriate permissions. Through the following steps we'll refer to this directory as the *working directory*.

We'll walk you through the build in eight steps:

- Step 1: Create a database in which to store chat messages.
- Step 2: Create a chat form.
- Step 3: Create a login form.
- Step 4: Implement a login process.
- Step 5: Send new chat messages to the server with AJAX.
- Step 6: Store new chat messages in the database.
- Step 7: Build an SSE stream at the server.
- Step 8: Connect to an SSE stream in the browser.



Name	Size	Type
add-chat.php	686 bytes	PHP script
add-session.php	558 bytes	PHP script
chat.js	1.6 kB	JavaScript program
chat.sql	943 bytes	SQL code
close-session.php	515 bytes	PHP script
credentials.php	42 bytes	PHP script
functions.php	2.5 kB	PHP script
get-chat.php	246 bytes	PHP script
index.php	1.6 kB	PHP script
jquery-1.7.1.min.js	93.9 kB	JavaScript program
sse.php	1.6 kB	PHP script
style.css	904 bytes	CSS stylesheet

Figure 4.3 The file layout for the chat application

SSE on older browsers

Server-sent events are a rationalized version of the forever-frame hack discussed in appendix D. The required server-side code is similar, so the most obvious approach for fallback in older browsers is to use the forever frame if SSE isn't available. An alternative is to use one of the prebuilt libraries, which implement a fallback transparently. One such library is Remy Sharp's EventSource.js polyfill: https://github.com/remy/polyfills/blob/master/EventSource.js.

STEP 1: CREATE A DATABASE IN WHICH TO STORE CHAT MESSAGES

Use your MySQL administration tool to create a database called `ssechat` (see appendix C). Included in the code download is a chat.sql file, which, when run, will create two tables in the database called `sessions`, to record who is logged in, and `log`, to record a log of the chat messages. Get the file credentials.php from the source code download and edit it to contain your database connection details. The example expects `$user`, `$pass`, and `$db` to define strings for the username, password, and connection string, respectively. The `$db` variable will look something like `"mysql:host=localhost;dbname=ssechat"`.

STEP 2: CREATE A CHAT FORM

Create the index.php page and the markup that users will see. The markup will contain two forms that will be visible or not, depending on the status of the user. In this step you'll create the list of chat messages and a form for adding new ones; in the next step you'll create a form for logging in. The following listing shows the PHP source for the form shown in figure 4.1. It's a simple HTML template that makes a couple of function calls to render the main content, and it contains a form to allow new chat messages to be added.

Listing 4.1 index.php body content

```
<body>
    <strong>Online now:</strong>
    <ul class="chatusers">
        <?php
        print_user_list($dbh);
        ?>
    </ul>
    <div class="chatwindow">
    <ul class="chatlog">
        <?php
        print_chat_log($dbh);
        ?>
    </ul>
    </div>
    <form id="chat" class="chatform" method="post"
          action="add-chat.php">
        <label for="message">Share your thoughts:</label>
        <input name="message" id="message" maxlength="512" autofocus>
```

The print_user_list function outputs an unordered list (the HTML element) of currently logged-on users.

The print_chat_log function outputs an unordered list of chat messages.

The chatform has an action defined that allows it to work, in a limited sense, without JavaScript enabled, but JavaScript will be used to override the default action in listing 4.6.

```
        <input type="submit" value="Chat">
    </form>
</body>
```

You'll also need to set up basic links in the <head> section of index.php. The required code is shown in the next listing.

Listing 4.2 index.php head

```
<?php
session_start();
include_once "credentials.php";
include_once "functions.php";
try {
    $dbh = new PDO($db, $user, $pass);
} catch (PDOException $e) {
    print "Error!: " . $e->getMessage() . "<br/>";
    die();
}
?><!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>SSE Chat</title>
    <link href="style.css" rel="stylesheet">
    <script src="jquery-1.7.1.min.js"></script>
    <script>var uid='<?php print session_id(); ?>';</script>
    <script src="chat.js"></script>
</head>
```

This enables the standard PHP session tracking.

Common variables and functions are included from separate files.

You'll be using PHP Data Objects (PDO) to connect to the database.

Make the PHP session ID easily available to JavaScript (saves reading the cookie).

chat.js is the file where you'll later implement the client-side code for SSE.

STEP 3: CREATE A LOGIN FORM

In order to track which user is which, you need to have them log in, which means recording their chat handle along with their PHP session ID. As mentioned in step 2, rather than create a separate page for this, you're going to add another form into the index.php file, then use conditional statements to turn the visibility of the form on and off. You're not going to do anything fancy—the index.php page with the login form enabled is shown in figure 4.4.

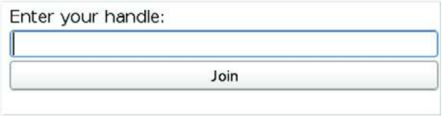


Figure 4.4 A simple login page for the chat application

As we just discussed, you don't need to create a separate PHP file for displaying the previous form—instead, you'll add conditional functionality to your existing index.php page. The following listing contains the code that determines whether to show the login form or the chat form. It should go immediately after the <body> tag in listing 4.1.

Listing 4.3 Check to see if the user is logged on

```
<?php
try {
    $checkOnline = $dbh->prepare(
        'SELECT * FROM sessions WHERE session_id = :sid');
    $checkOnline->execute(array(':sid' => session_id()));
    $rows = $checkOnline->fetchAll();
} catch (PDOException $e) {
    print "Error!: " . $e->getMessage() . "<br/>";
    die();
}
if (count($rows) > 0) {
?>
```

Look up all the sessions in the database with a session_id equal to the current session_id().

If one is found, assume the user is logged in. (This is intended to be the simplest code that will work—it's not best practice, secure PHP.)

The rest of the code from listing 4.1, starting at <strong>, will continue here.

Now that you've added a conditional statement before the code for the chat form, you have to close the first block of the condition, then add the code for the login form inside an `else` block after the chat page code. The code for the login form is shown in the next listing. It should be placed immediately before the closing </body> tag in index.php.

Listing 4.4 Display a login form

```
<?php
} else {
?>
<form id="login" class="chatlogin"
  method="post" action="add-session.php">
    <label for="handle">Enter your handle:</label>
    <input name="handle" id="handle" maxlength="127" autofocus>
    <input type="submit" value="Join">
</form>
<?php
}
?>
```

This else statement corresponds to the if at the end of listing 4.3.

The add-session.php file will deal with inserting the user into the database.

Try it out

You should now be able to see the login form by browsing to the index.php file on your local server. It won't do anything yet, because you haven't created a PHP file to process the logins. In order to get users logged in, you'll need a working add-session.php file.

Step 4: Implement a login process

The add-session.php file is shown next. Put this file in the same directory as index.php, as per the file layout in figure 4.3.

Listing 4.5 The add-session.php file

```
<?php
session_start();
include_once "credentials.php";
try {
    $dbh = new PDO($db, $user, $pass);
    $preparedStatement = $dbh->prepare(
        'INSERT INTO `sessions`(`session_id`, `handle`, `connected`)
         VALUES (:sid,:handle,NOW())');
      $preparedStatement->execute(
        array(':sid' => session_id(), ':handle' => $_POST["handle"] ));
      $rows = $preparedStatement->fetchAll();
      $dbh = null;
  } catch (PDOException $e) {
      print "Error!: " . $e->getMessage() . "<br/>";
      die();
  }
  header("Location: index.php");
  ?>
```

You're not doing anything more complex than recording the submitted handle in the database with the session_id().

Redirect to index.php when finished.

Now that you have the user's basic details sorted out, it's time to implement the application functionality.

STEP 5: SEND NEW CHAT MESSAGES TO THE SERVER WITH AJAX

You accomplish the transport of data back to the server with traditional AJAX techniques. The next listing shows the code for processing the chat form `submit`—nothing surprising for experienced front-end developers. Create a file chat.js in your working directory to contain all of your JavaScript code; as per figure 4.3 you can create it in the same directory as index.php and put the code from the following listing in it.

Listing 4.6 Add a chat message (client code)

```
$(document).ready(
    function() {
        var chatlog = $('.chatlog');
        if (chatlog.length > 0) {
            var chatformCallback = function() {
                chatform.find('input')[0].value = '';
            }
            chatform.bind('submit', function() {
                var ajax_params = {
                    url: 'add-chat.php',
                    type: 'POST',
                    data: chatform.serialize(),
                    success: chatformCallback,
                    error: function () {
                        window.alert('An error occurred');
                    }
                };
                $.ajax(ajax_params);
                return false;
            })
```

You'll close the function and the condition in listing 4.9.

A simple function to clear the chat input after the message has been successfully sent to the server.

The add-chat.php takes the message and adds it to the database, along with some information from the session; check the download files for more details.

Because the form is submitted by AJAX, you don't want the page to reload.

STEP 6: STORE NEW CHAT MESSAGES IN THE DATABASE

On the server you'll need a script to insert the chat messages in the database as they're created. The next listing shows the source code for add-chat.php, which grabs the message from a `POST` request and stores it with the appropriate details.

Listing 4.7 Add a chat message (server code)

```
<?php
session_start();
include_once "credentials.php";
$dbh = new PDO($db, $user, $pass);
$preparedStatement = $dbh->prepare('
    INSERT INTO `log`(`session_id`,`handle`, `message`, `timestamp`)
    VALUES (
      :sid,
      (SELECT `handle` FROM `sessions` WHERE `session_id` = :sid),
      :message,NOW()
    )');
$preparedStatement->execute(
    array(':sid' => session_id(),
          ':message' => $_POST["message"] ));
$rows = $preparedStatement->fetchAll();
$dbh = null;
session_write_close();
header("HTTP/1.1 200 OK");
echo "OK";
ob_flush();
flush();
die();
?>
```

The database details are stored in a separate file.

The message table is simple: an ID, a user handle, and a time (the user handle is being stored for convenience).

All database access in this example is using PHP's PDO database library—this should be part of your standard PHP install.

You've created a simple interface and a way to add new chat messages—now at last you're ready to start using SSE. What you need next is a way to get the chat messages of other users to appear in your browser as they're entered by your fellow chatters. This is the sort of task SSE is designed for.

Step 7: Build an SSE stream at the server

The following snippet shows an excerpt from an SSE event stream like the one you're about to create. It's all plain text and should be served with the MIME type `text/event-stream` (typically, because you're generating the event stream dynamically, you'll set the MIME type in your server-side code). A sample of the event stream you'll be generating is shown here:

```
event: useradded
data: Rob

event: message
data: <time datetime="2011-10-24 10:13:17">10:13</time>
 <b>Joe</b> <span>How can we be sure?</span>

event: message
data: <time datetime="2011-10-24 10:13:40">10:13</time>
 <b>Rob</b> <span>Well, according to Wittgenstein...</span>
```

An event is defined by the keyword event, followed by a colon, followed by the name of the event.

On the following line, the data keyword gives the text to be associated with the event.

Any string can be used to define the event name, but note that the events captured by any script in the browser will have the same name as the events being emitted.

The data can also be any string; the script in the browser is responsible for interpreting it correctly.

The event stream itself is similar to the forever-frame approach (see appendix D). A connection is opened and kept open, and the chat.js script periodically adds content to it. Each time new content arrives at the browser, it's converted into the simple event-driven JavaScript programming model with which we're all familiar.

The code on the server is straightforward. Create a file sse.php to generate the event stream in the same directory as index.php and add the same `session_start()`, `include_once`, and PDO creation code that starts off index.php. You don't need to add a `!DOCTYPE` declaration because you're not generating an HTML page. Then add code to loop, constantly looking for new messages. If you already have a forever-frame script, it's likely you can easily adapt it. The code for sse.php is shown in the following listing.

Listing 4.8 sse.php key code loop

```
<?php
session_start();
include_once "credentials.php";
include_once "functions.php";
try {
    $dbh = new PDO($db, $user, $pass);
} catch (PDOException $e) {
    print "Error!: " . $e->getMessage(). "<br/>";
    die();
}
header('Content-Type: text/event-stream');
header('Cache-Control: no-cache');
$uid = $_REQUEST["uid"];
$lastUpdate = time();
$startedAt = time();
session_write_close();
var $lastupdate = now();
while (is_logged_on($dbh, $uid)) {
    $getChat = $dbh->prepare(    'SELECT `timestamp`,`handle`, `message`
     FROM `log`
     WHERE `timestamp` >= :lastupdate
     ORDER BY `timestamp`');
    $getChat->execute(
     array(':lastupdate' => strftime("%Y-%m-%d %H:%M:%S", $lastUpdate))
    );
    $rows = $getChat->fetchAll();
    foreach($rows as $row) {
        echo "event: message\n";
        echo "data: <time datetime=\"".$row['timestamp']."\">";
        echo strftime("%H:%M",strtotime($row['timestamp']));
        echo "</time> <b>".$row['handle']."</b> <span>";
        echo $row['message']."</span>\n\n";
        ob_flush();
        flush();
    }
    $lastUpdate = time();
    sleep(2);
}
?>
```

A quirk of PHP is that the session is single-threaded; if you leave it open in this script, it'll block any other pages using it.

Set the correct content-type.

Ensure the stream isn't cached.

Loop here until the user logs out. Nearly all web server configurations limit execution time to between 30 and 90 seconds to allow the script to time out, but the browser will automatically reconnect.

In a real application, you'd factor this inline SQL into a function. This example tries to keep all the logic visible.

In a real application you'd invoke some rendering logic here that's shared among your application files.

Fetch all chat messages added to the database since the last update; to keep things simple you'll worry about only the message event for now.

Send the data as HTML. You could also send it as a JSON-encoded object.

Stores the last time you updated, and sleeps for two seconds. This is necessary in this example because the MySQL timestamp column is only accurate to the closest second. Implementing a millisecond-accurate time field in MySQL is possible but has been avoided here to keep the code simple.

Like the forever frame, you gain a low overhead of passing data from the server to the client. Once the connection is open, the only data that needs to be transferred is that which is pertinent to the application. No headers need to be sent with each update.

Step 8: Connect to an SSE stream in the browser

Core API

Core API

To retrieve chat messages, you'll connect your index.php page to the event stream using an `EventSource` object. The next listing shows the relevant JavaScript. You should add it to the chat.js you created in step 5. In this listing the `EventSource` is established and event listeners are added. The annotations explain the key points.

Listing 4.9 Client code for connecting to an event stream

```
            var evtSource = new EventSource("sse.php?uid=" + uid);

            evtSource.addEventListener("message", function(e) {
                var el = document.createElement("li");
                el.innerHTML = e.data;
                chatlog.appendChild(el);
            })
            evtSource.addEventListener("useradded", function(e) {
                var el = document.createElement("li");
                el.innerHTML = e.data;
                chatusers.appendChild(el);
            })
        }
    }
)
```

An EventSource is declared by linking to the script on the server that provides the event stream; the uid is a value passed via the host page to link users to their PHP session on the server side.

Event listeners can be added to the EventSource using normal DOM methods.

What the events will be called is determined in the server script; "message" and "useradded" aren't regular DOM events but the ones defined in the server-side code (see listing 4.8).

This closes the function and conditional opened in listing 4.6.

Try it out!

Everything is now in place for you to try the application. If you haven't already, copy all the files to a location where your web server can access them (as discussed earlier in this chapter, this is likely to be either /var/www/html or C:\Inetpub\WWWRoot) and have a go. You can use a couple of different browsers to simulate multiple users and try talking to yourself.

Controlling the default server timeout

There's one thing to bear in mind if you're using PHP on Apache, as in this example: The default script timeout is 30 seconds. This means that after 30 seconds the script on the server will be terminated and the connection will be dropped.

This isn't a problem on the client side, because it should automatically reconnect to the event source. By default, a reconnection will be attempted every 3 seconds, but it's also possible to control this from the event stream by emitting a retry directive:

```
retry: 10000
```

The number is a time in milliseconds. This should force the browser to wait 10 seconds before attempting a reconnect. Controlling the retry time would be useful if you knew the server was going to be unavailable or under high load for a short time.

4.1.2 When to use SSE

Before we move on to WebSockets, let's step back to consider why it was worth bothering with SSE. After all, server-sent events do have some obvious disadvantages:

- You can only communicate from the server to the client.
- SSE offers little advantage over long-polling or forever frame.

If your application implemented one of the older hacks, it would probably not be worth updating just to take advantage of an event-driven interface consistent with other HTML5 APIs. SSE won't dramatically lower the communication overhead compared to these hacks. If you're starting from scratch, SSE does have advantages over WebSockets (which we'll talk about in the next section):

- It's an extremely simple wire protocol.
- It's easy to implement on cheap hosting.

If you're working on a hobby project, SSE will probably be a good fit for you. But if you're working on high-load, web-scale startups where you're constantly tweaking the infrastructure, you'll want to look closely at WebSockets, the pièce de résistance of the HTML5 communication protocols.

In the next section you'll use Node.js web server (also commonly referred to as just plain *Node*) to write an application using WebSockets. Node is well suited to SSE and WebSockets because it's designed from the ground up to do event-driven communication (frequent, small, but irregular message sending; see appendix D). If you're used to web servers like Apache or IIS, it works differently than you might expect. It's therefore worth spending time becoming familiar with the basics.

4.2 Using WebSockets to build a real-time messaging web app

WebSockets allow bare-bones networking between clients and servers with far less overhead than the previously more common approach of tunneling other protocols through HTTP. With WebSockets it's possible to package your data using the appropriate protocol, XMPP (Extensible Messaging and Presence Protocol) for chat, for example, while also benefiting from the strengths of HTTP.

The WebSockets Protocol, which describes what browser vendors and servers must implement behind the scenes, is used at the network layer to establish and maintain socket connections and pass data through them. The WebSockets API describes the interface that needs to be available in the DOM so that WebSockets can be used from JavaScript. Appendix D more fully describes the protocol and API, so if you'd like more information before you build the next piece of this chapter's sample application—an agile planning board—detour to section D.6, "Understanding the WebSockets Protocol," now.

When you return, we'll give you an overview of the application you're going to build and help you get your prerequisites in order, have you create and test a WebSocket with Node.js, and build the planner application.

4.2.1 Application overview and prerequisites

In section 4.1 you built a simple chat system based on SSE. In this section you'll use WebSockets and Node.js to build an agile planning board which is intended to be a simple way to group tasks according to their status so that progress on the overall project can be discerned at a glance. Tasks, originally represented by sticky notes on a notice board (figure 4.5), are slotted into three or more simple categories such as to do, in progress, and done.

Agile methodologies are a particularly attractive target for tools based on messaging because agile is intended to be collaborative rather than dictatorial. So it's expected

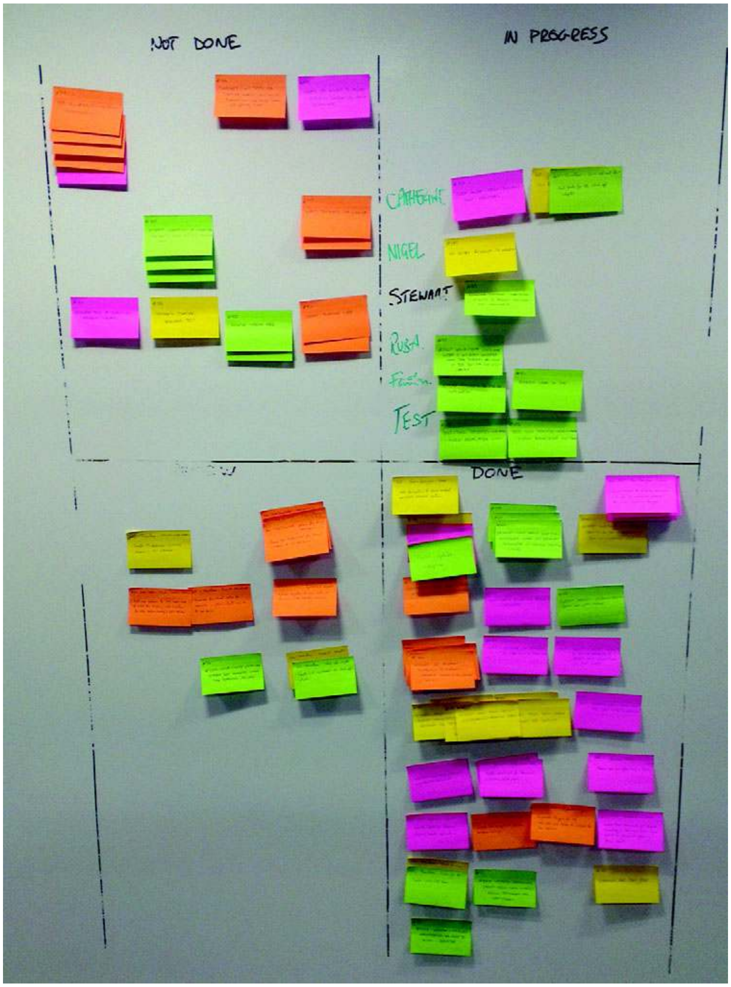


Figure 4.5 A real-life agile planning board at the TotalMobile offices in Belfast. The sticky notes describe tasks to be done, and the four quadrants are labeled, from top-left clockwise, NOT DONE, IN PROGRESS, DONE, and REVIEW. In this section you'll develop an electronic version of this board.

that you might have a bunch of people online trying to update the same plan at the same time.

BEFORE YOU PROCEED: PREREQUISITES

Before you begin this portion of the application, you'll need certain prerequisites to make the application in this section work. Specifically, you'll need the following:

- *The chat app*—See section 4.1.
- *Node.js*—Download from http://nodejs.org/; see appendix E for install instructions.
- You'll also need to install four Node modules (see appendix E for details of how to install):
  - *Director*—Download from https://github.com/flatiron/director or install with NPM; for handling routing (assigning requested URLs to handlers).
  - *Session.js*—Download from https://github.com/Marak/session.js or install with NPM; for handling user sessions.
  - *Mustache*—Download from http://mustache.github.com/ or install with NPM; for generating HTML from combining objects and templates, both within Node and in client-side JavaScript.
  - *WebSocket-Node*—Download from https://github.com/Worlize/WebSocket-Node or install with NPM; for extending Node to support WebSockets.
- *jQuery*—Download from http://jquery.com/.
- *EventEmitter.js*—Download from https://github.com/Wolfy87/EventEmitter.

The rest of the files you need are available in the code download from the Manning .com website; we won't list them here because they're not relevant to the WebSockets logic. You'll need to either create your own or grab the ones from the download.

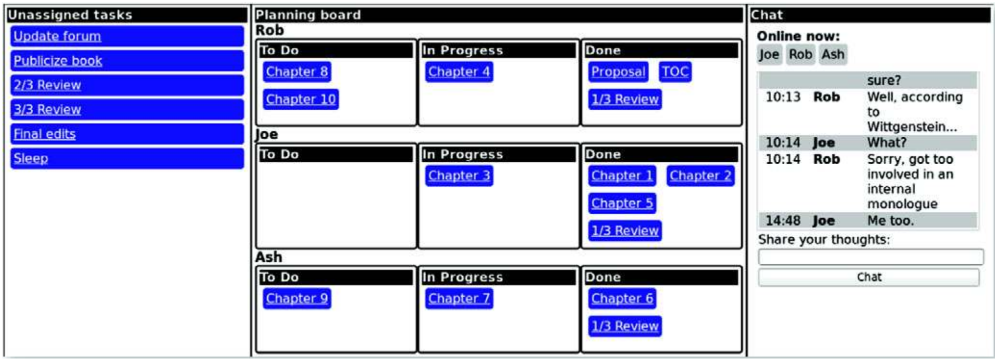
AN OVERVIEW OF THE BUILDING PROCESS

After you load your prerequisites and test your installation, the building process will flow like this:

1. Create a template page.
2. Build planner logic that can be used both in the client and on the server.
3. Create browser event listeners to deal with incoming WebSocket events and update the plan.
4. Create server logic to listen to incoming messages, update the plan, and send updates to other clients.

The finished application (figure 4.6) won't look quite like the real-life example, but it will feature of the main components. To simulate the experience of a bunch of people all standing around a real notice board, sipping their coffee, and arguing about where to put particular tasks, the chat application from section 4.1 is provided in an iframe. All participants will still have to provide their own coffee.

The final file layout you'll create during the build is shown in figure 4.7.



Unassigned tasks

- Update forum
- Publicize book
- 2/3 Review
- 3/3 Review
- Final edits
- Sleep

Planning board

	To Do	In Progress	Done
Rob	Chapter 8 Chapter 10	Chapter 4	Proposal TOC 1/3 Review
Joe		Chapter 3	Chapter 1 Chapter 2 Chapter 5 1/3 Review
Ash	Chapter 9	Chapter 7	Chapter 6 1/3 Review

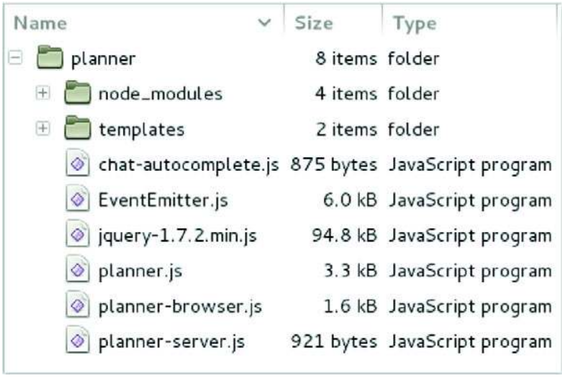
Chat

Online now: Joe Rob Ash

		sure?
10:13	**Rob**	Well, according to Wittgenstein...
10:14	**Joe**	What?
10:14	**Rob**	Sorry, got too involved in an internal monologue
14:48	**Joe**	Me too.

Share your thoughts:

Figure 4.6 The finished planning application



Name	Size	Type
planner	8 items	folder
node_modules	4 items	folder
templates	2 items	folder
chat-autocomplete.js	875 bytes	JavaScript program
EventEmitter.js	6.0 kB	JavaScript program
jquery-1.7.2.min.js	94.8 kB	JavaScript program
planner.js	3.3 kB	JavaScript program
planner-browser.js	1.6 kB	JavaScript program
planner-server.js	921 bytes	JavaScript program

Figure 4.7 Planner application file layout

With prerequisites installed, but before you build the planner application, let's make sure that WebSockets are working for you. In the next section you'll write a quick test page to confirm that WebSockets are working correctly in Node and in the browser, before it all is obscured by your application logic.

4.2.2 Creating a WebSocket with Node.js

Rather than deal with all the low-level, bit-by-bit data manipulation required by the WebSockets Protocol, you'll be using the WebSocket-Node module. It allows you to concentrate on the APIs involved rather than the mundane details of packing bits together in the correct format—details described for you in appendix C. In this section you'll create two files:

- A JavaScript file to be run with Node.js
- An HTML page, which will be sent to the browser

WebSocket API 3 6 10 11 5

The `onmessage` event is used in every other messaging API in HTML5, so it should come as no surprise to you that it gets used in WebSockets, too. For WebSockets you need to create a `WebSocket` object and attach a function to the `message` event listener.

The code you write will dump information to the console as it receives it; sample console output is shown in the following listing.

Listing 4.10 Server output for a simple WebSocket test

```
Sun Nov 27 2011 23:59:13 GMT-0800 (PST) Server is listening on port 8080
Sun Nov 27 2011 23:59:24 GMT-0800 (PST) Connection accepted.
Received Message: My Message
```

As each message is received, it's reflected back in a message to the client.

Figure 4.8 shows the corresponding output in the browser developer console. The browser requests the page; then it upgrades the connection to a WebSocket. It sends the message "My Message" before receiving the response from the server; in this case the same "My Message" string is sent back as a message.

Core API

The next listing shows JavaScript that opens a WebSocket, then listens for messages from the server. You should create a page named websocket-sample.html and include this listing in a `<script>` block. The page doesn't need to do anything or have any content; you'll determine success by examining the JavaScript console (see step C in the listing).

Listing 4.11 A simple JavaScript WebSockets client

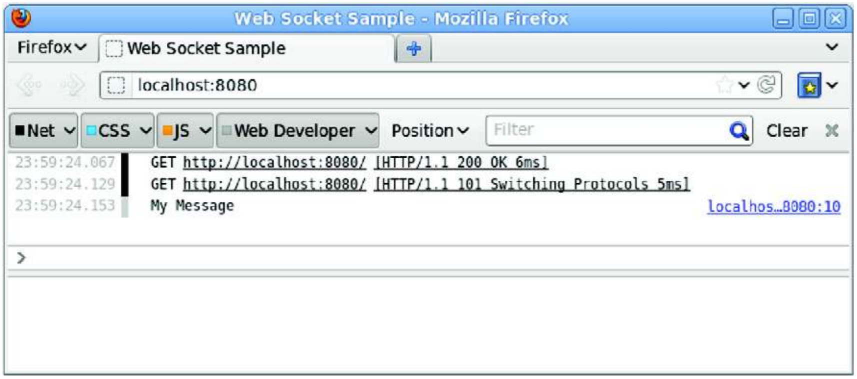
```
var ws = new WebSocket('ws://localhost:8080');
ws.onmessage = function(e) {
    console.log(e.data);
};
ws.onopen = function() {
    ws.send('My Message');
};
```

This line creates a WebSocket object; note that the URL uses the ws:// protocol.

The familiar onmessage event

Log the data to the console so you can see it.

The onopen event fires when the socket created in the first step is successfully opened by a browser—this function then sends a message to the server.



```
23:59:24.067   GET http://localhost:8080/ [HTTP/1.1 200 OK 6ms]
23:59:24.129   GET http://localhost:8080/ [HTTP/1.1 101 Switching Protocols 5ms]
23:59:24.153   My Message                                          localhos…8080:10
>
```

Figure 4.8 The simple WebSocket client running in the browser

On the server, the WebSocket-Node library is used to extend the base HTTP server. Appendix E provides the steps you need to take to install this module in Node; if you're following along step-by-step, please take that detour now.

With the module installed, you're ready to continue. Our next listing shows a Node.js app that will accept a WebSocket request and echo back any message sent to it. Save it as websocket-sample.js in the same directory as the file from listing 4.11.

Listing 4.12 A simple Node.js WebSockets server

```
var http = require("http");
var fs = require('fs');
var WebSocketServer = require('websocket').server;

function handler (req, res) {
    fs.readFile(__dirname + '/websocket-sample.html',
    function (err, data) {
        if (err) {
            res.writeHead(500);
            return res.end('Error loading websocket-sample.html');
        }
        res.writeHead(200);
        res.end(data);
    });
}

var app = http.createServer(handler);

app.listen(8080, function() {
    console.log((new Date()) + " Server is listening on port 8080");
});

wsServer = new WebSocketServer({
    httpServer: app
});

wsServer.on('request', function(request) {
    var connection = request.accept(null, request.origin);
    console.log((new Date()) + " Connection accepted.");
    connection.on('message', function(message) {
        console.log("Received Message: " + message.utf8Data);
        connection.sendUTF(message.utf8Data);
    });
});
```

This handler function will be run in response to any HTTP request.

Create a basic HTTP server object.

Start the server listening on port 8080.

The WebSocket-Node module is designed to extend an existing HTTP server; the HTTP server object is passed to the WebSocket server object as a parameter.

This handler function will be run in response to any WebSocket request.

When a client connects to the WebSocket, add a handler for received messages.

The handler will echo any message received back to the socket it was received from.

TRY IT OUT

Run listing 4.12 with Node (enter `node websocket-sample.js` on the command line). Now open your browser and connect to http://localhost:8080/ and check the console for the output.

4.2.3 Building the planner application

Now that you've confirmed that WebSockets are functioning both in Node and in your browser, and you know how to implement the WebSocket API in the client and how to set up Node.js to service those WebSockets, you're ready to build a real application that takes advantage of all of these features.

The steps you'll follow to build the planner application are these:

- Step 1: Create a template page.
- Step 2: Build multipurpose business logic in JavaScript to create and update plans.
- Step 3: Handle updates in the browser.
- Step 4: Handle updates on the server.

STEP 1: CREATE A TEMPLATE PAGE

The markup for the application page, index.html as normal, is shown in the following listing, though most of the interesting things in this application will be in the linked JavaScript files.

Listing 4.13 The planner index.html file

```
<body>
    <strong>Online now:</strong>
    <ul class="chatusers">
        <?php
        print_user_list($dbh);
        ?>
    </ul>
    <div class="chatwindow">
    <ul class="chatlog">
        <?php
        print_chat_log($dbh);
        ?>
    </ul>
    </div><!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Planner</title>
    <link rel="stylesheet" href="style.css">
    <script src="jquery-1.7.2.min.js"></script>
    <script src="EventEmitter.js"></script>
    <script src="planner.js"></script>
</head>
<body>
    <div id="plan">
        <div class="taskqueue">
            <strong>Unassigned tasks</strong>
        </div>
        <div class="grid">
            <strong>Planning board</strong>
            <div class="user">
                <div class="who">
                </div>
                <div>
                    <div class="todo">
```

This section will contain a list of tasks that are currently unassigned.

This section will have one or more resources; each has a section for to-do, in-progress, and completed tasks.

```
                        <strong>To Do</strong>
                    </div>
                    <div class="inprogress">
                        <strong>In Progress</strong>
                    </div>
                    <div class="done">
                        <strong>Done</strong>
                    </div>
                </div>
            </div>
        </div>
        <div class="external">
            <strong>Chat</strong>
            <iframe src="http://localhost/sse-chat/index.php">
            </iframe>
        </div>
    </div>
</body>
</html>

    <form id="chat" class="chatform" method="post"
          action="add-chat.php">
        <label for="message">Share your thoughts:</label>
        <input name="message" id="message" maxlength="512" autofocus>
        <input type="submit" value="Chat">
    </form>
</body>
```

The final section embeds the chat application from section 4.1.2.

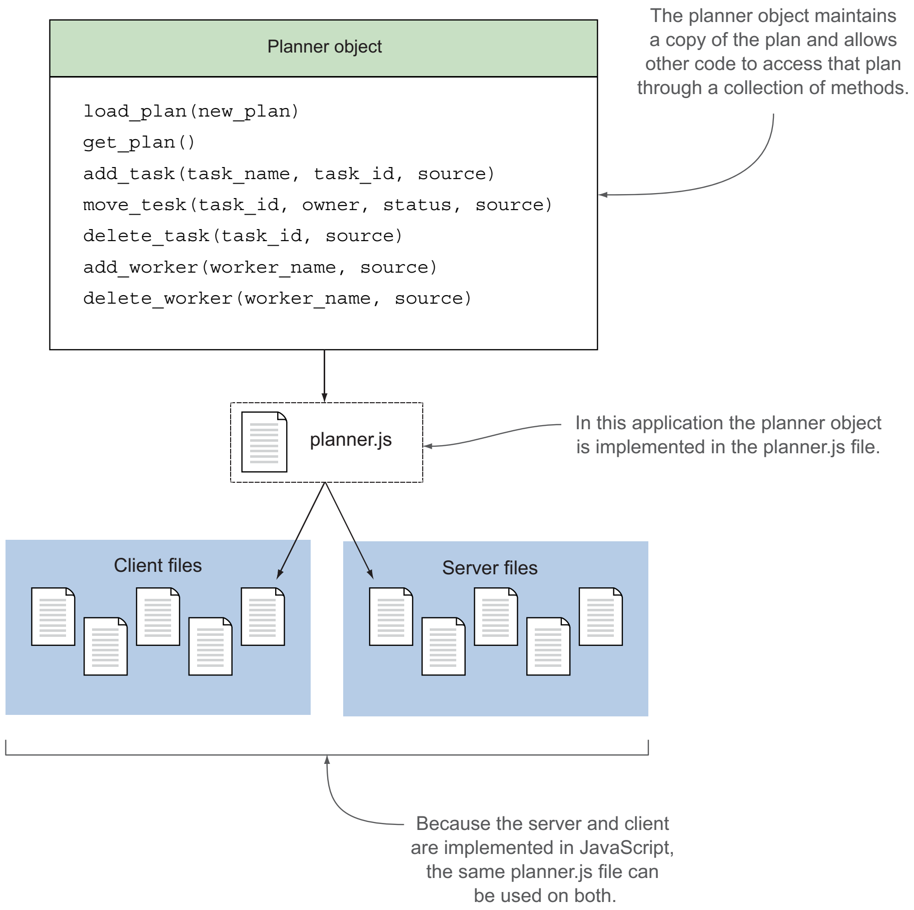
You now have the basic page structure out of the way, so let's delve into the JavaScript APIs that will make it all work.

Step 2: Build multipurpose business logic in JavaScript to create and update plans

A key advantage of having the server use the same programming language as the client is that they can share code. Instead of implementing the same functionality once in the server-side language and then again in JavaScript, implement it only one time. Figure 4.9 shows how this works.

Figure 4.10 shows the architecture of the application on the server and in two identical connected clients. As you can see, the structure on both client and server is similar. As each user makes changes, the same methods get fired on their local copy of the planner object as will be fired on the server planner object and on the planner objects used by other clients as the messages are passed between them using WebSockets.

Your model (the object containing the plan) will make use of the events framework, EventEmitter.js, as mentioned in the prerequisites. This is a browser-compatible version of the events module that comes as standard with Node. As methods are called on the model object, events will be fired. You'll then attach listeners to those events; when the model is run in the browser, those events will update the UI and send the changes back to the server. When the model is run on the server, those events will update all the other connected clients. The following listing shows the basic outline of the object you'll be using to store the plan, including some types and some utility functions. Add it to a file called planner.js. In the next listing you'll add some functionality.


Planner object
load_plan(new_plan)
get_plan()
add_task(task_name, task_id, source)
move_tesk(task_id, owner, status, source)
delete_task(task_id, source)
add_worker(worker_name, source)
delete_worker(worker_name, source)
planner.js
Client files
Server files

The planner object maintains a copy of the plan and allows other code to access that plan through a collection of methods.

In this application the planner object is implemented in the planner.js file.

Because the server and client are implemented in JavaScript, the same planner.js file can be used on both.

Figure 4.9 By using the same model (the plan held by the planner object) in the browser and on the server, the business logic (the methods in the planner object) can be the same in both places.

Listing 4.14 Creating the plan object and utility functions in planner.js

```
var Planner = function(ee) {
    var plan = {};
    plan.tasks = [];
    plan.workers = [];
    plan.statuses = ['todo','inprogress','done'];
    var Task = function(task_name, task_id) {
        var that = {};
        that.name = task_name;
        if (typeof task_id === 'undefined') {
            that.id = guidGenerator();
        } else {
            that.id = task_id;
        }
```

The planner expects an EventEmitter object to be passed in when it's created.

This first section sets up a few private variables.

A utility function to create a new task.

```
        that.owner = '';
        that.status = '';
    }
    function get_task(task_id) {
        return plan.tasks[get_task_index(task_id)];
    }
    function get_task_index(task_id) {
        for (var i = 0; i < plan.tasks.length; i++) {
            if (plan.tasks[i].id == task_id) { return i; }
        }
        return -1;
    }
    function guidGenerator(){
        var S4 = function() {
           return (
               ((1+Math.random())
                 *0x10000)|0).toString(16).substring(1);
        };
        return (S4()+S4()+"-"+S4()+"-"+S4()+"-"+S4()+"-"+S4()+S4()+S4());
    }
    var that = { }
    return that;
}
```

A couple of utility functions for picking out tasks from the plan.

A utility function to return a pseudo-GUID (Globally Unique Identifier), so that every object created in the plan can have a unique ID.

You'll populate this object in listing 4.16; it will contain all the public properties and methods.

As mentioned in the previous step, the that object is returned.

Using the EventEmitter library allows the event code to be identical on both server and client. The model, your plan object, emits events as the methods on it are called. On the client side, you'll listen to these events and update the display appropriately.

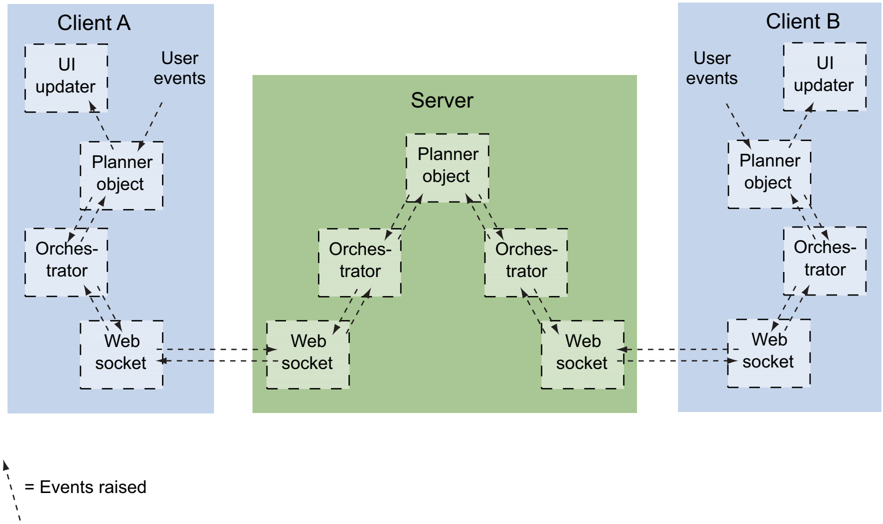


Figure 4.10 Planner application architecture following through from User events in Client A: Events are generated by the client and update the local plan; an orchestrator monitors the plan and sends those updates through a WebSocket to the server. An orchestrator on the server updates the server planner object; then those updates are sent out via other WebSockets to the other connected clients, culminating in the UI of the other clients being updated.

The model itself will be updated from two sources:

- User input
- Messages from the server

The next listing is the part of planner.js that creates the plan object (`that`), which will be returned when the planner is initialized; it should replace `var that = { }` in listing 4.14.

Listing 4.15 More planner.js

```
var that = {
    load_plan: function(new_plan) {
        plan = JSON.parse(new_plan);
        ee.emit('loadPlan',plan);
    },
    get_plan: function() {
        return JSON.stringify(plan);
    },
    add_task: function(task_name, task_id, source) {
        var task = Task(task_name, task_id);
        plan.tasks.push(task);
        ee.emit('addTask',task, source);
        return task.id;
    },
    move_task: function(task_id, owner, status, source) {
        var task = get_task(task_id);
        task.owner = owner;
        task.status = status;
        ee.emit('moveTask', task, source);
    },
    delete_task: function(task_id, source) {
        var task_index = get_task_index(task_id);
        if (task_index >= 0) {
            var head = plan.tasks.splice(task_index,1);
            head.concat(plan.tasks);
            plan.tasks = head;
            ee.emit('deleteTask', task_id, source);
        }
    },
    add_worker: function(worker_name, source) {
        var worker = {};
        worker.name = worker_name;
        worker.id = guidGenerator();
        plan.workers.push(worker);
        ee.emit('addWorker', worker, source);
    },
    eachListener: ee.eachListener,
    addListener: ee.addListener,
    on: ee.on,
    once: ee.once,
    removeListener: ee.removeListener,
    removeAllListeners: ee.removeAllListeners,
    listeners: ee.listeners,
```

Once the that object is returned by planner constructor (listing 4.14), it will access private functions of planner (i.e., get_task()) via JavaScript's closure feature.

In a real application you'd add validation logic here to check to see if the JSON string constitutes a valid plan.

A corresponding method to allow the plan's current state to be saved outside the object.

Each method will follow a similar pattern. Let's look at the add_task method in detail. Note that the task_id parameter is optional—it's not needed when the task is created, but it will be needed when this event is replicated back on the server and in other clients.

The task is created with the utility function and then pushed into the task's array on the plan.

An event is emitted containing the new task.

For brevity, the corresponding delete_worker() method isn't shown here; it will move all the worker's tasks back to the queue and delete the worker.

The EventEmitter methods are now monkey-patched onto the return object.

You'll be able to use the on method to add event listeners to the plan object.

```
        emit: ee.emit,
        setMaxListeners: ee.setMaxListeners
    };
```

The UI is mostly drag and drop. We covered this HTML5 API in great detail in chapter 3, so there's no need to go over it all again. Similarly, updating the display uses the standard jQuery DOM manipulation methods you're already familiar with. More interesting to us right now is what happens when the plan object is updated by these UI actions and events that arrive via a WebSocket. In the next step, you'll look at the code that handles this; in the following step, you'll look at the server-side code to handle the updates.

Step 3: Handle updates in the browser

Now create the client orchestrator code; for this create a new file called planner-browser.js in your working directory. The next listing shows the event listeners on the WebSocket that will update the model and the event listeners on the planner object that trigger messages to be sent through the WebSocket.

Core API

The WebSocket listeners are added by setting `ws.onmessage`. And listeners on the planner object are added with `plan.on()`.

Listing 4.16 The planner-browser.js (partial) browser code

```
function init() {
    var ee = new EventEmitter();
    var planner = new Planner(ee);
    var render;
    if (typeof  MozWebSocket !== 'undefined') {
        WebSocket = MozWebSocket;
    }
    var ws = new WebSocket('ws://localhost:8080');
    ws.onmessage = function(msg_json) {
        var msg = JSON.parse(msg_json);
        switch (msg.type) {
            case 'loadPlan':
                planner.load_plan(msg.args.plan);
                render = new Renderer(planner);
                break;
            case 'addTask':
                planner.add_task(msg.args.task_name,
                                 msg.args.task_id,
                                 'socket');
                break;
            case 'moveTask':
                planner.move_task(msg.args.task_id,
                                  msg.args.task_owner,
                                  msg.args.task_status,
                                  'socket');
                break;
            case 'deleteTask':
                planner.delete_task(msg.args.task_id,
                                    'socket');
```

Because this code creates the planner object, it also has to create the EventEmitter.

Create a new planner object using the EventEmitter.

In Firefox the WebSocket object is called MozWebSocket and will be until the spec is finalized. For practical use, MozWebSocket is identical to WebSocket, so map one to the other.

Add a listener to the WebSocket.

Assume that anything received on the WebSocket is a JSON-encoded object.

The type property of the decoded message object will be used to determine the correct action.

When the client first connects, expect the server to deliver a JSON-encoded planner object with the latest version of the plan.

The rest of the potential messages are mapped onto their equivalent planner actions.

```
                break;
        }
    };
    ws.onerror = function(e) {
        console.log(e.reason);
    }
    planner.on('addTask', function(task, source) {
        if (source !== 'socket') {
            var msg = {};
            msg.type = 'addTask';
            msg.args = { 'task_name': task.name, 'task_id': task.id };
            ws.send(JSON.stringify(msg));
        }
    });
    planner.on('moveTask', function(task, source) {
        if (source !== 'socket') {
            var msg = {};
            msg.type = 'moveTask';
            msg.args = { 'task_id': task.id, 'owner': task.owner,
                     'status': task.status };
            ws.send(JSON.stringify(msg))
        }
    });
    planner.on('deleteTask', function(task_id, source) {
        if (source !== 'socket') {
            var msg = {};
            msg.type = 'deleteTask';
            msg.args = { 'task_id': task_id };
            ws.send(JSON.stringify(msg))
        }
    });
}
```

Log any errors to the console to aid any debugging.

The on method on the planner object attaches an event listener. When events are raised by the in-browser planner object, they are detected and sent to the server.

Because adding a task will trigger an addTask event, there's no need to do anything if the original source of the event was this code.

STEP 4: HANDLE UPDATES ON THE SERVER

Similarly on the server, the model will be updated by incoming messages from various clients. Create a file called planner-server.js in your working directory for this code, or grab the version from the code download. In this file you'll need to set up listeners on the model to send those same updates to any other connected client. The key part of the code for responding to a moveTask message is shown in the following listing. Check the planner-server.js file in the code download for the rest of the code.

Listing 4.17 planner-server.js server code

```
planner.on('moveTask', function(task, source) {
    var msg = {};
    msg.type = 'moveTask';
    msg.args = { 'task_id': task.id, 'owner': task.owner,
                 'status': task.status };
    var jMsg = JSON.stringify(msg);
    for (var i=0; i<clients.length; i++) {
        if (source !== clients[i].client_id) {
```

This part of the code is the same as the equivalent in listing 4.16. In a more complex application, you may want to extract it to a separate shared module.

There's no need to send the message to the client that originated it.

The clients variable is an array of objects representing connected clients. Each time a connection is created, an entry is added to the array.

```
                clients[i].ws.send(jMsg)
            }
        }
});
```

The WebSocket is also stored in the clients array.

Security and validation

In a real application, the server has additional responsibilities in terms of validating data and persistence. A general tenet of server-side development is to never trust data you've received over the wire. In order to concentrate on using WebSockets, those features have been left out of the sample application in this section.

If you've followed along and either downloaded or re-created the UI logic, you should now have a working planning-board application. In this model of web application development, the server becomes another client. The bulk of the code involved is identical to what's running in all the users' browsers. You should also have the chat application from section 4.1 sitting in an iframe alongside it, but so far they're independent applications on different domains. We assume you have the chat application on port 80 from a standard web server, and the planning board is running on port 8080 from Node. Normally, the browser wouldn't allow scripts on either page to exchange data with each other. In the next section, you'll learn about some HTML5 APIs that enable client-side communication between scripts from different domains.

4.3 Messaging on the client side

Client-side messaging refers to the communication between windows and scripts that are loaded in the browser. These could be browser windows, iframes, framesets, or worker threads; the HTML5 specification refers to these with the umbrella term *script contexts*.

Before HTML5, communication between different script contexts has been done by direct DOM manipulation. If you want to build web pages out of loosely coupled components, this isn't a good approach for two reasons:

- Changes to the structure of one component could easily break all the components that try to communicate with it.
- Each component needs access to the full DOM of the hosting page and vice versa. You can't share only a limited set of information. Often it's easier to communicate via the server. In the new world of disconnected web applications, that's sometimes no longer an option.

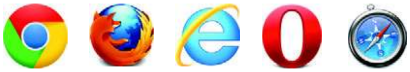
Cross-document versus cross-domain

You'll often hear cross-document messaging referred to as cross-domain messaging. It's not a requirement to have the two documents served from different domains. Messaging will work just as well if the two pages are on the same domain. But that option doesn't represent new functionality in HTML, rather a different way of doing something we've been doing for years. As a result, people tend to focus on the cross-domain aspect.

In this section you'll have a brief introduction to HTML5's cross-document messaging API, and then you'll look at how to use it to connect the applications from sections 4.1 and 4.2.

4.3.1 Communicating across domains with postMessage

Web browsers usually restrict communication between windows according to the Same Origin Policy: Scripts on pages loaded from one domain can't access content in windows loaded from another domain. This is a sound security approach. Without it, a website could create an iframe, load your Facebook page into it, and steal your personal details or post on your wall. But you'll find plenty of situations where you'll want to embed content from other sites in web pages; for example, Google ads and analytics, Facebook Like buttons, and Twitter feed widgets. You can implement all these examples by loading JavaScript from other sites using `<script>` elements. When scripts are included this way, they have as much access to your content as scripts on your own domain; they bypass the Same Origin Policy.



Cross-document messaging	1	3	8	9.5	4

Until HTML5, the options for any foreign domain content embedded in your pages were these:

- No access to any of your content
- Complete access to all of your content

It would be nice to have a middle ground between these extremes. Although there may be some sources you don't trust at all, it's likely you have plenty you trust a little bit. HTML5 satisfies this demand for flexibility with cross-document messaging. The cross-document messaging API allows a controlled messaging channel to be created between two pages by using the `postMessage` method and the `onmessage` event.

Core API

The `postMessage` method should be passed two parameters:

- The message itself
- The domain of the page being targeted:

```
windowRef.postMessage('The message', 'http://domain2.com');
```

The domain parameter is important because it ensures that if a different page is loaded into the iframe, either by the user clicking a link or through some other activity, the message won't be passed. It's possible to pass a wildcard, `'*'`, as the second parameter and avoid all the security, but be careful because you could end up sending your user's information to a malicious website.

Core API

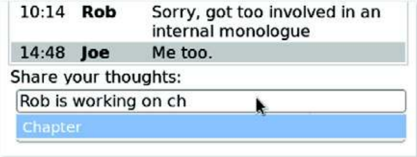
When a window receives a message, the aptly named `message` event is fired. As usual, with DOM events this handler can either be attached declaratively using an `onmessage` attribute on the body element or with `addEventListener`:

```
window.addEventListener('message', receiver, false);
```

The receiver function will accept the event as a parameter. The message passed will be in the data property of the event. In the next section, you'll implement receiver functions in the context of the planner and chat apps you built in previous sections.

4.3.2 Joining the applications with cross-document messaging

At this point, you have two applications, from different servers, coexisting in the same web page. In this section, you'll use the cross-domain messaging API to allow the data in the planner object to be used to feed an auto-complete feature in the chat window. This will offer user names and task titles in a drop-down list to speed up typing while retaining accuracy, as shown in figure 4.11.



10:14 Rob Sorry, got too involved in an internal monologue

14:48 Joe Me too.

Share your thoughts:

Rob is working on ch

Chapter

Figure 4.11 As the user types into the chat, the letters will be compared to words in the plan and matches will be shown in a drop-down list, where they can be selected using the down arrow.

> **Auto-complete prerequisite**
>
> This section relies on having a JavaScript auto-completer script. In order to concentrate on the HTML5 features, this section won't cover the details; a suitable script is included in the code download. Add the file to the working directory of the chat application.

To implement auto-complete, you need to set up message handlers on both the planner and the chat applications. The chat application will wait for the user to start typing and then send the letters of each word as they are typed to the parent window. The parent window will receive the message, compare the typed letters to the labels existing within the plan object, and send a message back with a list of matching words. The code for the chat application part of this is shown in the following listing; add it to the chat.js file in the SSE chat application.

Listing 4.18 Auto-complete interface for the chat application

```
function getWords(letters) {
    var msg = {};
    msg.type = 'getWordList';
    msg.params = {};
    msg.params.letters = letter;
    parent.postMessage(JSON.stringify(msg), 'http://localhost');
}

window.addEventListener('message', receiver, false);
function receiver(e) {
```

Called from an onKeyPress listener on the chat text input.

Create an object to contain the message; the variety of message types sent by the chat app is what defines the services provided by the parent window and is what defines the interface expected in the parent window. In more complex applications, you might want to create a function to define the interface explicitly.

Encode the object to a string and send it in the message to the parent window.

The standard onmessage listener.

```
    if (e.origin == 'http://localhost:8080') {
        var msg = JSON.parse(e.data);
        switch (msg.type) {
            case 'wordList':
                showAutocompleter(msg.params.words);
                break;
        }
    }
}
```

The messages accepted here define the interface for the calling page.

In the sample, there's only one domain you expect to receive messages from, but more complex checking could be inserted here to allow dynamic registration of components.

For brevity, the code to create an element containing the list of words isn't shown here, but it's much the same as the hundreds of auto-complete scripts available on the web. Download the sample code for further details.

Note that the chat application code is entirely generic—it doesn't matter what application has embedded it as long as it can return a list of words when sent a message in the correct form. The corresponding code in the planner application is necessarily specific to the planner. The following listing shows a new method for the planner object; add it to the planner.js file.

Listing 4.19 Word-completion service in the planner application

```
get_words: function(letters) {
    var words = [];
    for (var i=0; i<plan.tasks.length; i++) {
        var tokens = plan.tasks[i].name.split(' ');
        for (var j=0; j<tokens.length; j++) {
            if (tokens[j].length > 3 &&
                tokens[j].indexOf(letters) > -1) {
                words.push(tokens[j]);
            }
        }
    }
    return words;
}
```

This method goes inside the planner object from listing 4.16.

Go through each task in the plan. . .

. . .and each word in the task name.

Add them to the list if they are at least two letters long and contain the requested letters.

This is the list of words that will end up getting passed to listing 4.18.

The planner.get_words method needs to be hooked up to the window's onmessage event. The next listing shows the code for this, still in planner.js.

Listing 4.20 Listening to the onmessage event in the planner application

```
window.addEventListener('message', receiver, false);
function receiver(e) {
    if (e.origin == 'http://localhost') {
        var msg = JSON.parse(e.data);
        switch (msg.type) {
            case 'getWordList':
                var words = planner.get_words(msg.params.letters);
                var el = document
                    .getElementsByTagName('iframe')[0]
                    .contentWindow;
                var response = {};
                response.type = 'wordList';
                response.params = {};
                response.params.words = words;
```

Check that the message came from the page you expected it to come from.

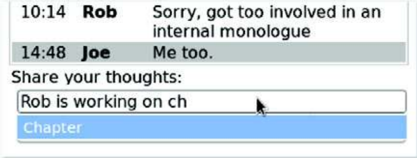
Create an object to contain the message, as in listing 4.20. For more complex applications, you might want to create a function to define this.

```
                el.postMessage(JSON.stringify(response),
                     'http://localhost:8080');
                break;
        }
    }
}
```

Encode the object to a string and send it in the message to the iframe.

This value needs to match the return words; line in listing 4.19.

With all this code in place, your work is complete. You should now be able to re-create the drop-down, shown again here for your convenience in figure 4.12.



10:14 **Rob** Sorry, got too involved in an internal monologue
14:48 **Joe** Me too.
Share your thoughts:
Rob is working on ch
Chapter

Figure 4.12 Planner-chat auto-complete one more time

Cross-document versus channel messaging

HTML5 has a more general-purpose alternative to cross-document messaging known as channel messaging. It allows you to create as many message ports as you want, not only between windows but also between any sorts of JavaScript object. Channel messaging wasn't necessary to complete the application in this chapter, but if you think you'll find it useful in your own applications, we've included a short introduction in appendix F.

4.4 Summary

In this chapter, you've learned about the new messaging APIs in HTML5, between pages in different windows on the client, with cross-document messaging, and between client and server, with server-sent events and WebSockets. You've also gained a practical understanding of how to use one of the new wave of web servers optimized for event-driven communication, Node.js. With all this new knowledge you're well equipped to build the next generation of web applications, based on lightweight, event-driven data communication between client and server, and you'll be able to join several such applications together in client browsers in a lightly coupled way thanks to cross-document messaging.

In the next chapter, you'll move on to consider an application environment where saving every byte makes a real difference: mobile web applications. HTML5 offers new capabilities that allow your application to keep working when no network is available.

Chapter 5 at a glance

Topic	Description, methods, and so on	Page
Web storage and management of simple key/value pair data on client-side local storage	Methods:	
	■ `getItem()`	140
	■ `localStorage`	140
	■ `removeItem()`	141
	■ `clear()`	142
Indexed database	Complex, indexed client-side database functionality	
	Database/object store methods:	
	■ `open()`	145
	■ `createObjectStore()`	145
	■ `createIndex()`	146
	■ `loadTasks`	150
	■ `objectStore()`	152
	■ `deleteDatabase()`	155
	Cursor method	
	■ `continue`	150
Application cache	Enable web applications to be used when client is offline	
	Method:	
	■ `swapCache()`	160

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