Lecture 6: Making Useful Web Applications

Nick Hynes

January 28, 2014

Topics

1 Congratulations!

Congratulations on making it to the end of the course! At this point, you should feel confident that you can go out into the world and create the next big web application! Just as a recap, we've covered creating dynamic user interfaces using JavaScript, sending and receiving data from servers, and creating servers that serve (no pun intended) as the backbone for the rest of the web app.

However, the previous class only covered the basics of creating server-side applications. A server-side application is generally used for storing and operating on stored data, especially data that must be shared among clients. As such, a server that simply returns the request variables doesn't really serve much purpose (serving up server humor. If you were to leave now, it would serve me right. Wait, come back...). Therefore, today's class will focus on different data-oriented technologies that can be used to make *useful*

web applications, namely WebSockets, databases, and external API calls. To do this, we'll go through a slightly contrived example of a chat server that uses WebSockets for real-time chat, Express for traditional request-response transactions, MySQL for user storage and authentication, and MongoDB for chat logging.

Note to run the code in this lecture, you will need to set up a MySQL server, Redis server, and MongoDB server. This isn't particularly challenging and guides are readily available, so it won't be covered during lecture.

2 Authentication using Express and MySQL

As everyone knows, any self-respecting chat server allows users to create accounts and log in (not strictly true). Since user management naturally uses a request-response communication format, we won't re-implement the wheel in WebSockets and, instead, use a REST API backed by Express. As such, when the user navigates to the home page, (s)he will be presented with a login form that has the option to create a new user.

Let's begin by assuming that the login form presented to the user is sane and contains validated fields for username and password. This form should POST the username and password to the server. Please note that you should never transmit unhashed/unencrypted passwords over a standard HTTP connection; use HTTPS instead.

The server is then responsible for the creation of new users (using a form similar to the login form) and the authorization of existing users. This will be handled by checking for data in a MySQL database, which you can think of as a set of columns which specify the fields of data stored (e.g. username, password) and rows, which are the data. The data format, or schema that we will be using will look very similar to this:

username	**salt**	**password**
user1	dWjRaTg/ZtT	a3fcd778c23ddbec0e9ebaa483385d75...
user2	Np7I/4BgHvm	75b9d59dddca9330af07f232ca5516dc...

The password doesn't look much like any password a real user would have. It has been *hashed* to prevent the user's text-password from being recovered if the database is compromised. The *salt* is a random value appended to the user's password before hashing to prevent the use of hash-password lookup

tables, known as *rainbow tables*. To authenticate the user, we will recompute the hash of the received password and compare it to the one in the database.

Here's what the client side login form might look like:

Example 1 – A sane login form

```
function login() {
  var username = $("#username").val();
  var password = $("#password").val();

  var $loginForm = $(".login-form");
  var $status = $loginForm.find(".response");
  $status.html("");

  $.post("/login", {username: username, password: password})
    .done(function(res) {
      showChatCentral();
    })
    .fail(function(res) {
      if(res.status === 401) { //Unauthorized
        $status.html("Incorrect username or password");
      } else {
        $status.html("Error");
      }
    });
}
```

Great, now on to the interesting part: the server.

Example 2 – Users using MySQL: The setup

```
var express    = require("express");
var app        = express();
var server     = require("http").createServer(app);
var mysql      = require("mysql");
var path       = require("path"); //to make native file paths
var crypto     = require("crypto");

//MySQL connection details
var sql = mysql.createConnection({
  host: "localhost",
  user: "root",
  password: "localhost",
  database: "chat-users"
});
sql.connect();

//serve static files using Node from /static
app.use(express.static(path.join(__dirname, "static")));

app.get("/", function(req, res) { //serve main page
  res.sendfile(path.join(__dirname, "static", "chat.html"));
});

//middleware to parse posted form data as parameters
app.use(express.urlencoded());
```

This chunk of code, which constitutes the "preamble" of the main Node.js file, does three things: it requires the modules we'll be using for the for the next parts (we will be using more), sets up a connection to the MySQL server running on the local machine, and sets up Express to serve static files from the /static directory with the default file being chat.html.

MySQL, a relational database, is well suited for the task of storing well-structured data with defined relationships between the columns. This makes it an ideal choice for user management.

Example 3 – Users using MySQL: Creating users

```
app.post("/users", function(req, res) {
  var username = sql.escape(req.param("username"));
  sql.query("SELECT * FROM users WHERE username="+username,
    function(err, rows) {
      if(rows && rows.length === 0) {
        var password = req.param("password");
        var salt = crypto.pseudoRandomBytes(64)
                   .toString("base64");
        var hash = crypto.createHash("sha512")
          .update(salt).update(password).digest("hex");

        sql.query("INSERT INTO users VALUES
                   ("+username+",\""+salt+"\",\""+hash+"\")",
         function(err, rows) {
          if(!err) {
            doLogin(res, username); //sets the login cookie
            res.status(200).send();
          } else {
            res.status(500).send(err);
          }
        });
      } else if(!err) { //If a row was found for the username
        res.status(409).send("username taken");
      } else { //If there was an error
        res.status(500).send(err);
      }
    }
  );
});
```

While this class will not cover the specifics of Structured Query Language and its syntax, you will, hopefully, be able to get an idea of its capabilities.

The first order of business is to register a `/user` POST route with Express. This makes sense since we want to create a new user sub-resource, which is the

definition of POST. We first query the database for a row with the supplied username to check if it already exists. If it doesn't, we create the password salt and hash and insert the username, salt, and hashed salt+password into the database as a new row. Otherwise, the server returns an HTTP 409, which the client will interpret as "Username taken." Of course, if there is an error, the server will return an HTTP 500 with the error as the response body.

One thing to always remember is to never store plain-text passwords in a database. Also, be sure to use a secure hashing algorithm (e.g. SHA2) since using an insecure algorithm (e.g. MD5, SHA1) can allow an attacker to find another "password" that hashes to the same value as the user's real password.

Okay, now that you've seen creating users, logins should be simple. This is mostly because a login only requires half of the steps required for a new user! For a login, the server only needs to look for the row that corresponds to the supplied username and, if it exists, checks the supplied password with the stored hash by prepending the salt and then hasing the password, and then completes the login. If the login information is incorrect, the server returns an HTTP 401 Unauthorized. And, as in the previous example, errors return HTTP 500s.

The `doLogin` function call seen in the login and new user sections is used to set the user's login cookie so that we can authenticate them before they connect using a WebSocket. This will be covered in more detail in just a bit.

Example 4 – Users using MySQL: Logins

```
app.post("/login", function(req, res) {
  var username = sql.escape(req.param("username"));
  var password = req.param("password");

  sql.query("SELECT * FROM users WHERE username="+username,
    function(err, rows) {
      if(err) {
        res.status(500).send(err);
        return;
      }
      if(rows && rows.length !== 0) {
        var salt = rows[0].salt;
        var hash = rows[0].password;
        var checkHash = crypto.createHash("sha512")
          .update(salt).update(password).digest("hex");

        if(checkHash === hash) {
          doLogin(res, username); //sets the login cookie
          res.status(200).send();
        } else { //Password was incorrect
            res.status(401).send(); //Unauthorized
        }
      } else { //Username wasn't found
        res.status(401).send();
      }
    }
  );
});
```

Example 5 – Setting cookies: doLogin

```
//these go in the "preamble"
var cookie   = require("express/node_modules/cookie");
var parseSignedCookie =
               require('express/node_modules/connect').utils
               .parseSignedCookie;

var SECRET = "0xdeadbeef"; //used to sign the cookies
app.use(express.cookieParser(SECRET));

/**
 * Sets the user's login cookie. The cookie is signed using
 * the app secret so that it is nontrivial to modify it on
 * the client.
 *
 * @param res the response object
 */
function doLogin(res, username) {
  var WEEK_IN_MSEC = 604800000;
  res.cookie('login', username,
  {signed: true, maxAge: WEEK_IN_MSEC});
}
```

Here we use Express's cookie parsing middleware to set signed cookies that we can use to keep the user logged in and, more importantly, authenticate the WebSocket connection. The `cookie` object is a property of the HTTP response variable. The cookie and parseSigned cookie variables that were required are hacky ways to access the cookie parsing middleware directly, without needing an Express HTTP response object.

In the next segment on WebSocket authentication using Socket.io, we will leave the realm of HTTP and return, once again, to WebSocket-world.

3 From HTTP to WebSockets

Example 6 – Authenticating Socket.io WebSockets

```
io.configure(function() {
  io.set("log level", 1); //suppresses Socket.io [debug]

  io.set("authorization", function(handshakeData, callback) {
    //callback is called to authorize/error the handshake
    //callback((null/Object) error, (Boolean) authorized)

    var cookies = handshakeData.headers.cookie;
    if(cookies) {
      var loginCookie = cookie.parse(cookies)["login"];
      var login = parseSignedCookie(loginCookie, SECRET);
      if(login) {
        handshakeData.username = login;
        callback(null, true);
      }
    } else {
      callback(null, false);
    }
  });
});
```

Although this block also sets Socket.io's debug level to 1 (suppresses debugging messages), the main event is the `authorization` component. Essentially, this function checks for the user's login cookie and sets the `username` property of the socket's `handshakeData` as the username so that it can be accessed later as a property of the `socket` itself.

It's useful to have the authentication happen before the socket ever connects since this process uses HTTP and, as such, we have access to data like the user's current page, browser, and cookies; the best part is that it can all be stored for use in WebSocket world as part of the `handshakeData` object.

The `callback` argument is called to let Socket.io know what it should do with the socket connection request (return an error, or (not) authorize it).

4 Chatting in WebSocket World

For this section, we're going to assume that we have a chat client that conforms to the following spec:

- The client should issue a `join` event containing an object with the `room` property set
- The client's messages to the server should be sent using the `message` event and should contain an Object with the `message` and `room` properties set.

You can look at the example implementation located in the *Additional resources* section.

Example 7 – A WebSockets chat server

```
io.sockets.on("connection", function(socket) {
  //username set during authentication
  var username = socket.handshake.username.replace(/\'/g,"");

  socket.on("join", function(data) {
    socket.join(data.room);
  });

  socket.on("message", function(data) {
    data.username = username;
    io.sockets.in(data.room).emit("message", data);
    logMessage(data); //Stores a chat log
  });
});
```

As is customary for a server that exists simply to pass messages from one place to another, the implmenetation is reasonably simple as well (mostly thanks to the great API provided by Socket.io). Although this example doesn't contain any bells and whistles like user join and leave notifications to chat rooms, all of the basic functionality—sending received messages to other clients in a room—is present.

5 Storing chat logs using MongoDB

However, for this chat server, we want to log everyone's messages so that we can go through them later and identify trends. We could store them in a relational database like MySQL, but we'll put them in the document-oriented, NoSQL database, MongoDB. Non-relational (NoSQL) databases are used over the standard relational databases when the data sets are very, very big (think Google big) or have a loosely-defined schema in which data entries do not have well-defined relationships. Since the data contained in non-relational databases is destructured, a database can be "sharded" and distributed across many readily-available servers to increase performance without great cost. This is known as *scaling out* or *horizontal scaling*, as opposed to *scaling up* which is associated with increasing the performance of a small number of servers, usually through hardware upgrades.

In the following example, we'll be using Mongoose, a Node.js driver for the MongoDB database that provides a clean API for schema creation and data validation. A Message document will be defined as having a `username`, `timestamp`, `room`, and `message`. Also, since we are going to process this data later, all we have to do for now is store it in the database, which is a very easy process.

Example 8 – Yummy data for the NSA to eat

```
//Add to preamble
var mongoose = require("mongoose");
mongoose.connect('mongodb://localhost/messages');

//Define a new Mongoose schema for a chat message
var messageSchema = new mongoose.Schema({
  timestamp: {type: Date, default: Date.now},
  room: String,
  message: String,
  username: String
});
var Message = mongoose.model('Message', messageSchema);

/**
 * Stores a message to MongoDB
 *
 * @param data the message data Object
 */
function logMessage(data) {
  var newMessage = new Message({
    room: data.room,
    message: data.message,
    username: data.username
  });

  newMessage.save(function(error) {
    if(error) {
      console.log("LOG ERROR:", error, data);
    }
  });
}
```

To use Mongoose, simply `require("mongoose")` (after installing it, of course!). In this example, the MongoDB server is running on the local ma-

chine and we're storing to the "messages" store.

As stated above, we create a Message schema and model using Mongoose's Schema constructor and model generator. If you're not familiar with these terms, a schema is a format for the data and a model is more of a concrete representation of the schema. We can then use the resulting model to very simply store messages into MongoDB. It's that easy!

This now concludes the slightly contrived example and now you don't have to make a boring chat server as an exercise!

6 Wrap up

This is by no stretch of the imagination a complete example of what Node.js can do (or even what a chat server can do...), but it is my hope that you now know