5

Securing Your Data

In this chapter, we will cover the following recipes:

* Implementing Basic Authentication
* Doing Token-Based Authentication
* Handling Server-Side Validation
* Locking Down the HTTP Referrer
* Using Secure Web-Sockets

# Introduction

Since the Web-Socket protocol opens up so much opportunity to communicate more directly between the client and the server, people often wonder if Socket.IO is actually as secure as something like the HTTP protocol. The answer to that question is that it depends entirely on how you implement it. Web-Sockets can easily be locked down to prevent malicious or accidental security holes, but as with any API interface, your security is only as tight as your weakest link.

In this chapter, we will explore several topics related to security in Socket.IO applications. These topics will cover the gambit from authentication and validation to how to use the wss:// protocol for secure Web-Sockets.

# Implementing Basic Authentication

Most applications need a way to authenticate users. In this recipe, we will create a simple form to create and authenticate users. We will keep our authenticated users in the session so that we can maintain our authenticated state even when the page is refreshed. In order to get our users, we will pass a random token to the socket once it authenticates that it can use to retrieve the authenticated profile.

## Getting Ready...

For this recipe, we will be using MongoDB to persist our users. We will also be using the md5 npm module to hash our passwords before we save them to the database.

## How To Do It...

To implement basic authentication, follow these steps:

1. First, we will need to create our server. The server will require several additional modules that we will build next.

var express = require('express'),

app = express(),

http = require('http'),

socketIO = require('socket.io'),

server, io;

app.get('/', function (req, res) {

res.sendFile(\_\_dirname + '/index.html');

});

server = http.Server(app);

server.listen(5000);

io = socketIO(server);

var getUser = require('./lib/getUser'),

loginUser = require('./lib/loginUser'),

createUser = require('./lib/createUser'),

authenticateUser = require('./lib/authenticateUser');

global.userSessions = {};

io.on('connection', function (socket) {

// Get the authenticated user

socket.on('user.get', function (token) {

getUser(socket, token);

});

// Create a new user

socket.on('user.create', function (data) {

console.log('user.create');

createUser(socket, data);

});

// Login

socket.on('user.login', function (data) {

authenticateUser(socket, data);

});

// Log the authenticated user out

socket.on('user.logout', function (token) {

delete userSessions[token];

});

});

1. We will create a getUser module in our /lib folder that will be responsible for retrieving authenticated users that match the token we pass into it:

// Get a user who matches the token

module.exports = function getUser (socket, token) {

// Emit an error if the token doesn't exist

if (!userSessions[token]) {

return socket.emit('user.get.error', {

message: 'This user is not authenticated'

});

}

// Emit a message with the profile and token

// if the token does exist

return socket.emit('user.get.success', {

profile: userSessions[token],

token: token

});

};

1. We will need to create a loginUser module in the same directory. This will be responsible for setting a user in the session if it authenticates. Note that we are using the crypto module to encode our token.

var crypto = require('crypto'),

getUser = require('./getUser');

// Logs in a user

module.exports = function loginUser (socket, user) {

// Create a token with crypto

var token = crypto.randomBytes(64).toString('base64');

// Save the user session

userSessions[token] = user;

// Get the user belonging to the token and emit it

return getUser(socket, token);

};

1. We won’t be able to log a user in until we can create a user first, so we will add a createUser module. This will use the MD5 module to hash our password. You can install it by running npm install MD5 –save in your terminal.

var md5 = require('md5'),

User = require('./userModel'),

loginUser = require('./loginUser');

// Creates a new user

module.exports = function createUser (socket, data) {

// Hash the password

data.password = md5(data.password);

// Create a new user in MongoDB

var user = new User(data);

// Save the MongoDB Model

user.save().then(function (data) {

return loginUser(socket, data);

});

};

1. We will need to add an authenticateUser module to help us to log our users in. This will attempt to find a user with a matching username and password to the combination that the user passes in. If a user is found, they will be logged in. If no user is found, we will emit an error message. Note that we are hashing the password before we attempt to find a matching user. The password in the database is hashed, so we will want to match that.

var md5 = require('md5'),

User = require('./userModel'),

loginUser = require('./loginUser');

// Authenticates a user and logs them in

module.exports = function authenticateUser (socket, data) {

// Hash the password

data.password = md5(data.password);

User.findOne(data, null, function (err, data) {

// If the username and password are correct, log the user in

if (data) {

return loginUser(socket, data);

// If the username or password are incorrect, emit an error

} else {

return socket.emit('user.login.error', err || {

message: 'Invalid email or password.'

});

}

});

};

1. Now we will create our userModel. This will be a MongoDB model that we can interact with to add and retrieve users.

var mongoose = require('mongoose');

var db = mongoose.connect('mongodb://localhost/basicauthapp');

var userSchema = db.Schema({

firstname: String,

lastname: String,

password: { type: String, select: false },

email: String

});

module.exports = db.model('User', userSchema);

1. Now for the client-side. We will need a container for our views to get rendered to.

<div class="container" id="main-container"></div>

1. Now for our client-side JavaScript. This mainly just listens for messages from Socket.IO and does some poor-mans routing when the hashtag changes.

function renderTemplate (tpl) {

document.getElementById('main-container').innerHTML = tpl;

}

function userLoggedIn (data) {

localStorage.token = data.token;

renderTemplate(getProfileTemplate(data.profile));

profileController(data);

}

functiongetRoute () {

if (document.location.hash === '#/create-account') {

renderTemplate(getCreateAccountTemplate());

createAccountController();

return;

}

renderTemplate(getLoginTemplate());

loginController();

}

function showError (data) {

alert(data.message)

}

(function () {

socket.on('user.create.error', showError);

socket.on('user.get.success', userLoggedIn);

socket.on('user.login.error', showError);

window.addEventListener('hashchange', getRoute);

getRoute();

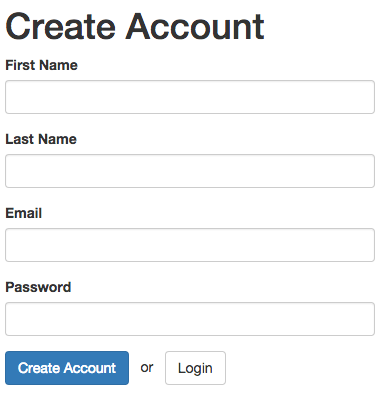
// If we have a token, send it to the server to authenticate

if (localStorage.token) {

socket.emit('user.get', localStorage.token);

}

}());



**Insert Image B04893\_05\_01.png**

function getCreateAccountTemplate () {

return `

<form id="create-account-form">

<h1>Create Account</h1>

<div class="form-group">

<label>First Name</label>

<input id="firstname" class="form-control" />

</div>

<div class="form-group">

<label>Last Name</label>

<input id="lastname" class="form-control" />

</div>

<div class="form-group">

<label>Email</label>

<input id="email" type="email" class="form-control" />

</div>

<div class="form-group">

<label>Password</label>

<input id="password" type="password" class="form-control" />

</div>

<div class="form-group">

<button class="btn btn-primary">Create Account</button>

&nbsp; or &nbsp;

<a href="#/" class="btn btn-default">Login</a>

</div>

<div id="messages"></div>

</form>

`;

}

function createAccountController () {

document.getElementById('create-account-form')

.addEventListener('submit', function (e) {

e.preventDefault();

var user = {

email: document.getElementById('email').value,

password: document.getElementById('password').value,

firstname: document.getElementById('firstname').value,

lastname: document.getElementById('lastname').value

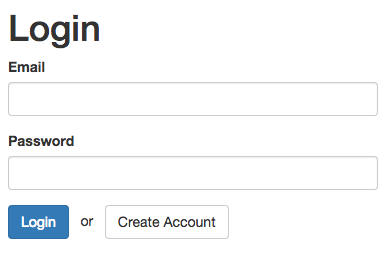
};

socket.emit('user.create', user);

});

}

1. We will also create a template to display our login form.



**Insert Image B04893\_05\_02.png**

function getLoginTemplate () {

return `

<form id="login-form">

<h1>Login</h1>

<div class="form-group">

<label>Email</label>

<input id="email" type="email" class="form-control" />

</div>

<div class="form-group">

<label>Password</label>

<input id="password" type="password" class="form-control" />

</div>

<div class="form-group">

<button class="btn btn-primary">Login</button>

&nbsp; or &nbsp;

<a href="#/create-account" class="btnbtn-default">Create Account</a>

</div>

<div id="messages"></div>

</form>

`;

}

function loginController () {

document.getElementById('login-form')

.addEventListener('submit', function (e) {

e.preventDefault();

var user = {

email: document.getElementById('email').value,

password: document.getElementById('password').value

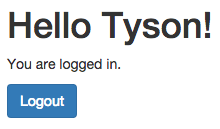
};

socket.emit('user.login', user);

});

}

1. Finally, we will need a template for when the user logs in. We will take an argument to the template function so we can render some data to it.



**Insert Image B04893\_05\_03.png**

function getProfileTemplate (profile) {

return `

<div>

<h1>Hello ${profile.firstname}!</h1>

<p>You are logged in.</p>

<div class="form-group">

<button id="logout" class="btn btn-primary">Logout</button>

</div>

</div>

`;

}

function profileController () {

document.getElementById('logout')

.addEventListener('click', function (e) {

e.preventDefault();

socket.emit('user.logout');

delete localStorage.token;

getRoute();

});

}

## How It Works...

After our user was authenticated, we added our user to the session by appending them to an object of users with a token as our key. We also emitted the token back to the client-side where we stored it in localStorage. Using that same token, we are able to prove that we are the authenticated user that we claim to be, so our session is maintained.

# Doing Token-Based Authentication

Now that we are able to do basic authentication with Socket.IO, let’s look at a token-based approach that handles authentication more securely, namely JSON Web Tokens, or JWT.

JSON Web Tokens are a URL-safe means of representing claims to be transferred between two parties. The claims in a JSON Web Token are encoded as a JSON object that is digitally signed using JSON Web Signature. Using this approach, we can securely send a salted web token to the client to use on subsequent requests.

## Getting Ready...

For this recipe, we will be using the “jsonwebtoken” npm package to create secure JSON web tokens. The package can be installed by running npm install jsonwebtoken –save in your terminal.

## How To Do It...

To do token-based authentication, follow these steps:

1. We will create our server and expose Socket.IO events to authenticate and receive a JSON Web Token. In this example, we are hard-coding a profile to be associated with the JWT, but you would typically retrieve a record from your database to represent the authenticated user.

var express = require('express'),

app = express(),

http = require('http'),

socketIO = require('socket.io'),

jwt = require('jsonwebtoken'),

server, io;

app.get('/', function (req, res) {

res.sendFile(\_\_dirname + '/index.html');

});

server = http.Server(app);

server.listen(5000);

io = socketIO(server);

io.on('connection', function (socket) {

// Our JWT secret

var jwtSecret = 'my b1g $3CR3T';

// Verifies our JWT token and emits a profile if it checks out

function getProfile (data) {

jwt.verify(data.token, jwtSecret, function(err, decoded) {

// Send an error message

if (err) {

return socket.emit('profile.error', err);

}

// Send a success message

socket.emit('profile.success', decoded);

});

}

// Get the profile

socket.on('profile', getProfile);

// Log the user in

socket.on('login', function () {

var profile = {

firstName: 'Peter',

lastName: 'Parker',

email: 'peterparker@spiderman.com',

id: 12

};

var token = jwt.sign(profile, jwtSecret, {

expiresInMinutes: 60

});

socket.emit('login.success', {

token: token

});

getProfile({

token: token

});

});

});

1. On the client-side, we will create methods to log in and log out of our application. The client will listen for a login.success response from the server. When it receives the login success, it will save the JWT in localStorage to authenticate future requests with.

function renderTemplate (template, data) {

document.getElementById('main-container').innerHTML = templates[template](data);

}

function renderLoggedOut () {

renderTemplate('loggedOut');

document.getElementById('login').addEventListener('click', function (e) {

socket.emit('login');

});

}

function renderLoggedIn (data) {

renderTemplate('loggedIn', data);

document.getElementById('logout').addEventListener('click', function (e) {

delete localStorage.jwtToken;

renderLoggedOut();

});

}

var templates = {

loggedOut: function () {

return `

<div>

<h1>You are not logged in.</h1>

<button id="login">Login</button>

</div>`;

},

loggedIn: function (data) {

return `

<div>

<h1>You are logged in as ${data.firstName} ${data.lastName}.</h1>

<button id="logout">Logout</button>

</div>`;

}

};

socket.on('profile.success', function (data) {

renderLoggedIn(data);

});

socket.on('profile.error', function (err) {

renderLoggedOut();

});

socket.on('login.success', function (data) {

localStorage.jwtToken = data.token;

});

if (localStorage.jwtToken) {

socket.emit('profile', {

token: localStorage.jwtToken

});

} else {

renderLoggedOut();

}

## How It Works...

JSON Web Tokens are a secure means of representing an authenticated session because they are signed with a secret passphrase along with the profile. The token looks like a long string of random characters, but the server-side is able to verify the token and decode it as needed.

# Handling Server-Side Validation

When we are writing data to a database, it is important to do validation on the server-side to ensure that the data is in the type and format that we expect it to be in. In this recipe, we will demonstrate how we can emit data to the server and emit back messages if there is a success or an error.

## Getting Ready...

We are going to be using promises to handle success and error states. Depending on your version of Node, you may need to install the promis module using npm install promise –save. Promises are a feature of ES6, so they will eventually be native in Node.

## How To Do It...

To handle validation on the server-side, follow these steps:

1. First, we will create a method to do some validation on the server-side. Typically, your validation will take place in your ORM, but we will do it in plain JavaScript here to exhibit the concept. We will resolve a promise if the data is valid and reject the promise if it is invalid.

function validatePerson (person) {

return new Promise (function (resolve, reject) {

if (!person.firstname.length) {

return reject({

firstname: 'Please provide a first name.'

});

}

if (!person.lastname.length) {

return reject({

lastname: 'Please provide a last name.'

});

}

if (person.firstname === person.lastname) {

return reject({

lastname: 'Why is your last name the same your first name? That seems unlikely...'

});

}

// We aren't really saving anything here, but we can still pretend ;)

return resolve(person);

});

}

1. Now, we will call our validatePerson() method any time the client emits an event to save a person. If the data is valid, we can safely save the data and emit an event announcing that the data was saved. A good way to keep track of your success and error messages is to postfix .success or .error onto the end of the original message name when you create the new message from the server. In this way, your client will always know to listen for the correct message name and you won’t have to spend as much time trying to think of clever message names. You can always expect that when you save data, the server will reply with a success or error when it is done.

io.on('connection', function (socket) {

socket.on('person.save', function (person) {

validatePerson(person).then(function (data) {

socket.emit('person.save.success', data);

}).catch(function (data) {

socket.emit('person.save.error', data);

});

});

});

1. Now, on the client-side we will need a template to render our form.

<div class="container">

<form id="personform">

<div class="form-group">

<label for="firstname">First Name</label>

<input name="firstname" id="firstname" class="form-control" />

<small class="text-danger" id="firstname-error"></small>

</div>

<div class="form-group">

<label for="lastname">Last Name</label>

<input name="lastname" id="lastname" class="form-control" />

<small class="text-danger" id="lastname-error"></small>

</div>

<div class="form-group">

<button class="btn btn-primary">Save</button>

</div>

</form>

</div>

1. Finally, we will add some client-side JavaScript to send data to the server and update the view with messages when it receives a response from the server.

socket.on('person.save.success', function (data) {

console.log('success', data);

alert(`${data.firstname} ${data.lastname} was successfully saved`);

});

socket.on('person.save.error', function (data) {

for (vari in data) {

if (data.hasOwnProperty(i)) {

document.getElementById(i + '-error').innerHTML = data[i];

}

}

});

document.getElementById('personform').addEventListener('submit', function (e) {

e.preventDefault();

var person = {

firstname: document.getElementById('firstname').value,

lastname: document.getElementById('lastname').value

};

for (vari in person) {

document.getElementById(i + '-error').innerHTML = '';

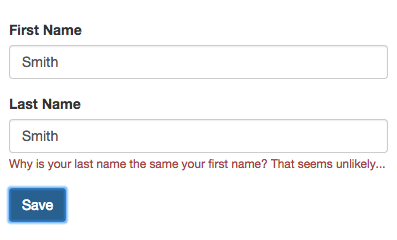
}

socket.emit('person.save', person);

});

## How It Works...

The key to good server-side validation with Socket.IO is to always respond with either a success or error message for every request to submit a form. Socket.IO doesn’t have a callback like an Ajax call would, so it is important to listen for these events on the client-side so that the update can be updated as needed.



**Insert Image B04893\_05\_04.png**

# Locking Down the HTTP Referrer

Socket.IO is really good at getting around cross-domain issues when you are making a request from a client in a different domain than the domain your server lives on. You can easily include the Socket.IO script from a different domain in your page and it will just work as you might expect it to.

There are some instances where you may not want your Socket.IO events to be available to every other domain. Not to worry! We can easily whitelist only the http referrers that we want so that some domains will be allowed to connect and other domains won’t.

## How To Do It...

To lock down the HTTP referrer and only allow events to whitelisted domains, follow these steps:

1. Create two different servers that can connect to our Socket.IO instance. We will let one server listen on port 5000, and let the second server listen in on port 5001:

var express = require('express'),

app = express(),

http = require('http'),

socketIO = require('socket.io'),

server, server2, io;

app.get('/', function (req, res) {

res.sendFile(\_\_dirname + '/index.html');

});

server = http.Server(app);

server.listen(5000);

server2 = http.Server(app);

server2.listen(5001);

io = socketIO(server);

1. When the connection is established, check the referrer in the headers. If it is a referrer we want to give access to, we can let our connection do its work and build up events as normal. If a blacklisted referrer, such as the one on port 5001 that we created, attempts a connection, we can politely decline and perhaps throw an error message back to the client.

io.on('connection', function (socket) {

switch (socket.request.headers.referer) {

case 'http://localhost:5000/':

socket.emit('permission.message', 'Okay, you\'re cool.');

break;

default:

return socket.emit('permission.message', 'Who invited you to this party?');

break;

}

});

1. On the client-side, we can listen to the response from the server and react as appropriate:

<h1></h1>

<script src="http://localhost:5000/socket.io/socket.io.js"></script>

<script type=”text/javascript”>

var socket = io.connect(5000);

socket.on('permission.message', function (data) {

document.querySelector('h1').innerHTML = data;

});

</script>

## How It Works...

The referrer is always available in the socket.request.headers object of every socket, so we were able to inspect it there to see if it was a trusted source.

In our case, we are using a switch statement to whitelist our domain on port 5000, but we could really use any mechanism at our disposal to do the job. For example, if we need to dynamically whitelist domains, we might store a list of them in our database and search for it when the connection is established.

# Using Secure Web-Sockets

WebSocket communications can either take place over the ws:// protocol or the wss:// protocol. They can be thought of in similar terms to the HTTP and HTTPS protocols in that one is secure and one isn’t. Secure WebSockets are encrypted by the transport layer, so they are safer to use when handling sensitive data. The main feature of https (and wss) is that socket is encrypted from client to server, so if we're in the same networks and we try to sniff the content you won't see anything legible.

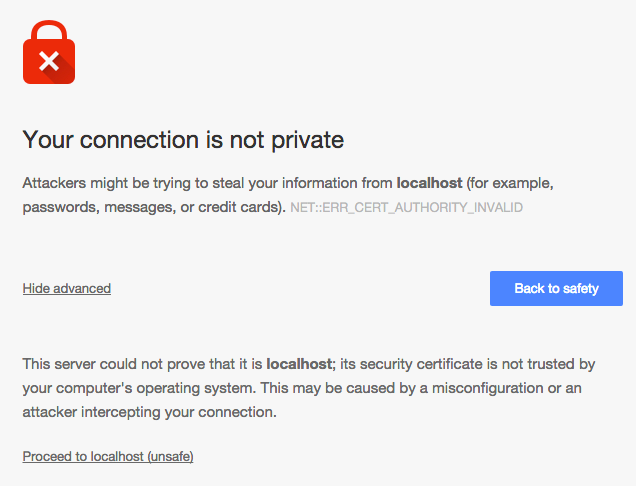
If your application uses the https protocol, you will also need to use wss protocol for your webSockets. Many browsers do not allow un-secure content when they use https.

In this recipe, we will learn how to force our Socket.IO communications to happen over the wss:// protocol for an extra layer of encryption.

## Getting Ready...

In this recipe, we will need to create a self signing certificate so that we can serve our app locally over the HTTPS protocol. For that, we will need an npm package called Pem. Pem allows us to create a self-signed certificate that we can provide to our server. Of course, in a real production environment, we would want a true SSL certificate instead of a self-signed one. To install Pem, simply call npm install pem –save.

Since our certificate is self-signed, you will probably see something like this when you go to your secure server:



**Insert Image B04893\_05\_05.png**

In production, when you purchase an SSL certificate, the warning in your browser will go away. For now, just take a chance by clicking the “proceed to localhost” link. You’ll see your application load using the HTTPS protocol.

## How To Do It...

To use the secure wss:// protocol, follow these steps:

1. First, create a secure server using the built-in Node HTTPS package. We can create a self-signed certificate with the pem package so that we can serve our application over HTTPS instead of HTTP.

var https = require('https'),

pem = require('pem'),

express = require('express'),

app = express(),

socketIO = require('socket.io');

// Create a self-signed certificate with pem

pem.createCertificate({

days: 1,

selfSigned: true

}, function (err, keys) {

app.get('/', function(req, res){

res.sendFile(\_\_dirname + '/index.html');

});

// Create an https server with the certificate and key from pem

var server = https.createServer({

key: keys.serviceKey,

cert: keys.certificate

}, app).listen(5000);

var io = socketIO(server);

io.on('connection', function (socket) {

var protocol = 'ws://';

// Check the handshake to determine if it was secure or not

if (socket.handshake.secure) {

protocol = 'wss://';

}

socket.emit('hello.client', {

message: 'This is a message from the server. It was sent using the ' + protocol + ' protocol'

});

});

1. In your client-side JavaScript, specify secure: true when you initialize your WebSocket. This will force the webSocket connection to be secure. If this is not set, Socket.IO will determine whether or not to use secure webSockets based on the protocol of your app.

<script src="//localhost:5000/socket.io/socket.io.js" type="text/javascript"></script>

<script type="text/javascript"></>

var socket = io('//localhost:5000', {

secure: true

});

socket.on('hello.client', function (data) {

console.log(data);

});

</script>

1. Now start your server and go to <https://localhost:5000>. Proceed to the page and you should see a message in your browser developer tools that says “**This is a message from the server. It was sent using the wss:// protocol**.”  
   We should note that opening servers in non-standard ports can cause problems with users behind firewalls, which only allow traffic to ports 80 and 443. I would advise using port 80 whenever possible. If you are unable to use either if the standard ports, you may want to do some further reading on reverse proxies to help bypass this problem.

## How It Works...

The protocol of our WebSocket is actually set automatically based on the protocol of the page that it sits on. This means that a page that is served over the HTTP protocol will send WebSocket communications over ws:// by default and a page that is served of HTTPS will default to using the wss:// protocol.

However, by setting the secure option to true, we told the WebSocket to always serve over wss:// no matter what.