2

Creating Real-Time Dashboards

In this chapter, we will cover the following recipes:

* Loading Static Data from the Server
* Creating a Server-Side Clock
* Loading Data from MongoDB
* Real-Time Analytics
* Handling Connection Timeouts

# Introduction

While there is a great deal of power in bidirectional communication, Socket.IO is also a perfect tool for creating unidirectional real-time experiences. Many applications have need of some sort of dashboard interface, whether to display analytical data or to show the state of the application data. For an application state that frequently changes, or in cases where there are multiple users changing the state, making the dashboard real-time makes for a much better experience.

In this chapter, we are going to implement various techniques to display and maintain real-time dashboards that only harness a unidirectional data flow from to pass data from the server to the client.

# Loading Static Data from the Server

In order to understand how Socket.IO pushes data to the client, we will create an application that emits a static object. While this exercise may seem contrived, the only difference between static data and dynamic data is that static data only needs to be emitted once whereas dynamic data needs to be emitted on each mutation. The client-side doesn’t concern itself with the frequency of state changes, so the difference between emitting data once and emitting data frequently is inconsequential. If we can re-render the state of a dashboard once, we can re-render it a million times.  
  
Once we’ve completed this recipe, our dashboard will look something like this:



**Insert Image B04893\_02\_01.png**

## Getting Ready...

In addition to Socket.IO and Node, we will be using Express for this exercise. Make sure that you have installed Express by running npm install express socket.io in your terminal.

## How To Do It...

To load static data with Socket.IO, follow these steps:

1. Create your **server.js** Node file. This will be responsible for starting up your server, setting up Socket.IO and instantiating your individual socket controllers.

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);

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

var controllers = ['comments', 'posts'];

for (var i = 0; i<controllers.length; i++) {

require('./controllers/' + controllers[i] + '.controller')(socket);

}

});

1. Now create your comments controller file. This file should live in **controller/comments.controller.js**. It will loop over all of our comments and emit a comments.add event that the client can listen for, to display users in the dashboard. It will also emit a comments.count event for each comment in your static array.

var comments = [{

user: 'Batman',

comment: 'Great post!'

}, {

user: 'Robin',

comment: 'Interesting ideas...'

}, {

user: 'Joker',

comment: 'Thanks, Batman!'

}, {

user: 'Bruce Wayne',

comment: 'I agree with Batman'

}];

module.exports = function (socket) {

// Recent Comments

for (var i = 0; i<comments.length; i++) {

socket.emit('comment.add', comments[i]);

socket.emit('comments.count', {

count: i + 1

});

}

};

1. Now you can create your posts controller. This is very similar to our comments controller. It loops over a static array of posts and emits an event to add the post. It also emits an event to update the posts count.

var posts = [{

user: 'Two-Face',

title: 'How to Flip a Coin'

}, {

user: 'Joker',

title: 'Top 5 Jokes of 2015'

}];

module.exports = function (socket) {

// Recent Posts

for (var i = 0; i<posts.length; i++) {

socket.emit('post.add', posts[i]);

socket.emit('posts.count', {

count: i + 1

});

}

};

1. Now that our server-side code is done, we need to create our client-side template to render our data when it comes in. We’ll use Bootstrap to style the page and make it look nice. The markup will look like this:

<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.4/css/bootstrap.min.css" />

<div class="container">

<div class="row bg-primary">

<div class="col-md-4">

<h1 id="users-count">0</h1>

<p>Users</p>

</div>

<div class="col-md-4">

<h1 id="posts-count">0</h1>

<p>Posts</p>

</div>

<div class="col-md-4">

<h1 id="comments-count">0</h1>

<p>Comments</p>

</div>

</div>

<div class="row">

<div class="col-md-6">

<h3 class="text-primary">Recent Comments</h3>

<table class="table">

<thead>

<tr>

<th>User</th>

<th>Comment</th>

</tr>

</thead>

<tbody id="recent-comments">

<!-- Recent Comments -->

</tbody>

</table>

</div>

<div class="col-md-6">

<h3 class="text-primary">Recent Posts</h3>

<table class="table">

<thead>

<tr>

<th>User</th>

<th>Title</th>

</tr>

</thead>

<tbody id="recent-posts">

<!-- Recent Posts -->

</tbody>

</table>

</div>

<div>

</div>

<script src="http://code.jquery.com/jquery-2.1.4.min.js"></script>

<script src="/socket.io/socket.io.js"></script>

1. Last of all, we need to add the client-side Socket.IO event listeners. They can either go directly in the index.html file or they may be included in as an external resource. They will listen for posts and comments and render them to the page. We’re using jQuery for simplicity, but you could easily use any other client-side framework if you need more structure:

var socket = io('http://localhost:5000');

// Update the users count

socket.on('users.count', function (data) {

$('#users-count').text(data.count);

});

// Update the comments count

socket.on('comments.count', function (data) {

$('#comments-count').text(data.count);

});

// Updae the posts count

socket.on('posts.count', function (data) {

$('#posts-count').text(data.count);

});

// Add a comment

socket.on('comment.add', function (data) {

var $row = $('<tr>' +

'<td>' + data.user + '</td>' +

'<td>' + data.comment + '</td>' +

'</tr>');

$('#recent-comments').append($row);

});

// Add a post

socket.on('post.add', function (data) {

var $row = $('<tr>' +

'<td>' + data.user + '</td>' +

'<td>' + data.title + '</td>' +

'</tr>');

$('#recent-posts').append($row);

});

## How It Works...

Our server is including the comments and posts controllers. These controllers are emitting the static arrays as soon as they are initialized. We are emitting the records one at a time, so we could easily emit more data one row at a time if we switched from using static data to using real dynamic data.

## There’s More...

Since we are emitting our records individually, they don’t have to get emitted all at once. For another contrived example, we could send our posts one at a time every two seconds. This can be achieved by amending out the end of our posts.controller file to something like this:

var i = 0;

var addingPosts = setInterval(function () {

if (posts[i]) {

socket.emit('post.add', posts[i]);

socket.emit('posts.count', {

count: i + 1

});

i++;

} else {

clearInterval(addingPosts);

}

}, 2000);

# Creating a Server-Side Clock

It is sometimes useful to get the date and time from the server-side instead of the client-side. We’re usually interested in the client-side time zone, but that isn’t always the case. For example, if we are creating a dashboard for a web-hosting platform, it could be helpful to display the time as it appears on the server.

In this recipe we are going to create a server-side clock that updates the user interface in real-time so we always know what time the server thinks it is.

## How To Do It...

To create a server-side clock that emits data to the client-side, follow these steps:

1. First, create a **server.js** file that emits a new date string every second. To do this, we will set an interval for 1000ms and emit a new Date() on that interval. In JavaScript, when we create a new Date() with no arguments, it will always be set to the current date and time.

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);

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

setInterval(function () {

socket.emit('seconds.update', {

time: new Date()

});

}, 1000);

});

2. Now, we need to add some client-side code to render the date as it comes in. The date object that the server-side sends will automatically be converted to a string before it is sent, so we will need to cast it back into a date object so we can display it in a human-readable format.

<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.4/css/bootstrap.min.css" />

<div class="container">

<div class="row bg-primary">

<div class="col-md-12">

<h1></h1>

</div>

</div>

</div>

<script src="http://code.jquery.com/jquery-2.1.4.min.js"></script>

<script src="/socket.io/socket.io.js"></script>

<script>

var socket = io('http://localhost:5000');

socket.on('seconds.update', function (data) {

var time = new Date(data.time);

$('h1').text(

time.getMonth() + '\/' + time.getDate() + '\/' + time.getFullYear() + ' ' +

time.getHours() + ':' + time.getMinutes() + ':' + time.getSeconds());

});

</script>

3.We should now be able to navigate to our page and see a clock that updates every second:

## How It Works...

The server-side date may be different from the client-side date if the server and the client are in different timezones. We are able to get around that by emitting the server-side date to the client. By emitting a new date each second, the clock interface will always be accurate.

## Loading Data from MongoDB

Since we are already able to send static data to our client with Socket.IO, we have all the tools we need to send dynamic data as well. MongoDB is a NoSQL database that stores data as JSON documents. In this recipe, we will send data from our database to the client-side to get rendered.

## Getting Ready...

Before you begin, you will need to install MongoDB on your machine. MongoDB can be installed by visiting [https://www.mongodb.org](https://www.mongodb.org/) and following the installation steps there. Once MongoDB is installed, you can start the MongoDB server by entering mongod into your terminal window. Leave the mongod process running, you’ll need to have it available to access your database.

You will also need to install a Node adapter for MongoDB. We will be using Mongoose which can be installed from NPM by typing npm install mongoose into your terminal.

## How To Do It...

To send dynamic data from a MongoDB database via SocketIO, follow these steps:

1. First, we will create a **server.js** file. This will be the file that starts the server and loads all of our server-side dependencies.

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);

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

var controllers = ['comments', 'posts'];

for (var i = 0; i < controllers.length; i++) {

require('./controllers/' + controllers[i] + '.controller')(socket);

}

});

2. You might have noticed that we are importing a **lib/mongo.js** file in our **server.js** file. This file will simply connect to a MongoDB database and export the mongoose instance for us to use throughout the app.

var mongoose = require('mongoose');

mongoose.connect('mongodb://localhost/dashboard');

module.exports = mongoose;

3. Now, we will need to include a MongoDB model for our comments. This can go in **models/comment.js**. This file will be responsible for creating a schema for our model and then returning the model itself.

var mongoose = require('../lib/mongo');

var commentSchema = mongoose.Schema({

user: String,

comment: String

});

module.exports = mongoose.model('Comment', commentSchema);

4. Now that we have a comments model, we can access it in our **controllers/comments.controller.js** file.

var Comment = require(‘../models/comment’);

module.exports = function (socket) {

var stream = Comment.find().stream();

stream.on('data', function (comment) {

socket.emit('comment.add', comment);

});

};

5. Now, we need to make our index.html template. This will listen for any new comments from the server and append them to the DOM.

<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.4/css/bootstrap.min.css" />

<div class="container">

<div class="row">

<div class="col-md-12">

<h3 class="text-primary">Recent Comments</h3>

<table class="table">

<thead>

<tr>

<th>User</th>

<th>Comment</th>

</tr>

</thead>

<tbody id="recent-comments">

<!-- Recent Comments -->

</tbody>

</table>

</div>

<div>

</div>

<script src="http://code.jquery.com/jquery-2.1.4.min.js"></script>

<script src="/socket.io/socket.io.js"></script>

<script>

var socket = io('http://localhost:5000');

// Add a comment

socket.on('comment.add', function (data) {

var $row = $('<tr>' +

'<td>' + data.user + '</td>' +

'<td>' + data.comment + '</td>' +

'</tr>');

$('#recent-comments').append($row);

});

</script>

6. Finally, we need to seed some data to display. We can create a lib/seed.js file to handle seeding and call it by running node lib/seed in the terminal:

var Comment = require('../models/comment');

// New comments

var comments = [{

user: 'Batman',

comment: 'Great post!'

}, {

user: 'Robin',

comment: 'Interesting ideas...'

}, {

user: 'Joker',

comment: 'Thanks, Batman!'

}, {

user: 'Bruce Wayne',

comment: 'I agree with Batman'

}];

// Loop over new comments and create them

for (var i = 0; i < comments.length; i++) {

new Comment(comments[i]).save();

}

console.log('Database Seeded');

process.exit(0);

## How It Works...

We are using the Mongoose stream method in our controller by calling Comment.find().stream(). With the stream method, we are able to pipe our comments to Socket.IO one by one as they come in.

Other than the implementation specific to listening for database items to stream, what we are doing here is really not far removed from just piping in static data.

# Real-Time Analytics

Socket.IO excels at creating rich real-time analytic dashboards. In this recipe we are going to display the count of users currently on our page, but the same concept could be used to show much more detailed analytical data if it is provided.

## How To Do It...

To show a real-time count of the users currently on a page, follow these steps:

1. Create a **server.js** file that emits the count of active users on the page whenever the count changes.

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 count = 0;

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

count++;

io.emit('users.count', count);

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

count--;

io.emit('users.count', count);

});

});

2. Now, display the results in an **index.html** template.

<!DOCTYPE html>

<html>

<head></head>

<body>

<h1 id="users-count">?</h1>

<p>Active Users On This Page</p>

<script src="/socket.io/socket.io.js"></script>

<script>

var socket = io('http://localhost:5000');

socket.on('users.count', function (number) {

document.getElementById('users-count').innerHTML = number;

});

</script>

</body>

</html>

3. To test this, open this page in a few different browser windows. Every time you add or remove a browser window, the count of active users will change.

## How It Works...

On the server, we are keeping a variable with the number of currently connected sockets. We are emitting an event every time the count changes. This allows us to always have the ability to display the number of current users on this page.

## There’s More…

To display real-time data with more detailed information, you would need to emit a custom event from the client instead of just listening for a socket connection. The custom event would include any information you could only get from the browser. It might contain browser and demographic information that isn’t available on the server. Here is an example of slightly richer analytics tracking:

socket.emit(‘analytics.log’, {

userAgent: window.navigator.userAgent,

location: window.location,

track: !window.navigator.doNotTrack

});

# Handling Connection Timeouts

When we are doing real-time application development, it is important to be aware of when the server-side WebSocket connection is dropped and we are no longer able to communicate with our server. This could allow us to provide an offline mode for our apps where we keep a record of all of the events that need to be emitted to the server once the connection is reestablished.

Socket.IO has some really great built-in functionality to reestablish a connection once it has been dropped. This is accomplished by making recurring polling requests to the server until a new connection is found or until the number of reconnection attempts we allow are exceeded.

## Getting Ready...

For this recipe, we will be using Express to serve up our Socket.IO application. Most of the magic in this recipe is happening on the client-side, so the server is really not as important as long as it is a functional server hosting Socket.IO.

## How To Do It...

To handle connection timeouts in Socket.IO, follow these steps:

1. First, we will create a basic server that spawns a Socket.IO connection.

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);

2. All of the connection timeout handling will happen on the client-side. Socket.IO gives us several socket lifecycle events that we can tap into. We can use those events to know when we lose a connection, when we successfully reconnect and much more.

<!DOCTYPE html>

<html>

<head></head>

<body>

<p>Open up your developer console, kill your server and let the fun begin!</p>

<script src="/socket.io/socket.io.js"></script>

<script>

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

'reconnection': true,

'reconnectionDelay': 500,

'reconnectionAttempts': 5

});

socket.on('reconnect', function (number) {

console.info('After attempting ' + number + ' times, we finally reconnected!');

});

socket.on('reconnect\_attempt', function (number) {

console.info('Reconnect attempt number ' + number);

});

socket.on('connect\_error', function () {

console.warn('Error connecting to Socket.IO');

});

socket.on('reconnect\_failed', function () {

console.error('We failed to reconnect to Socket.IO. We give up.');

});

</script>

</body>

</html>

3. Finally, start your server, navigate to the page in your browser and then kill your server to simulate a connection being dropped. You may also want to set localStorage.debug = ‘\*’; in your console so that you can see debugging information confirming that the socket connection has been dropped as expected.

## How It Works...

We are able to pass in an object of options as our second argument when we call the io() function on the client-side.

The reconnection option is set to true by default. If we set reconnection to false, Socket.IO will not attempt to reconnect when a connection is dropped.

The reconnectionDelay option specifies how many milliseconds are allowed to pass before we ping the server for a reconnection. The pinging will continue to happen until the number of reconnectionAttempts we specified is satisfied or until the connection is reestablished. By default, the reconnection attempts are set to Infinity.