Criando uma Rede Social

Node.js

[Facebook](https://www.facebook.com/sharer/sharer.php?u=http://www.devmedia.com.br/node-js-criando-uma-rede-social/31037) [Twitter](https://twitter.com/home?status=Node.js:%20Criando%20uma%20Rede%20Social%0D%0Ahttp://www.devmedia.com.br/node-js-criando-uma-rede-social/31037)

 (20)  (0)

Aprenda nesse artigo a desenvolver uma mini rede social em tempo real utilizando a plataforma Node.js.

**Fique por dentro**

Neste artigo falaremos do aspecto mais importante do Node.js, e que foi fundamental para a sua propagação: a programação orientada a eventos. Veremos também uma das suas principais características em comparação com outras tecnologias semelhantes: a forma como ele lida com I/O não bloqueante. Após isso, teremos a oportunidade de ver um exemplo de construção de uma rede social que simule bem todos os aspectos mais importantes do Node.js, desde sua instalação e configuração, até conceitos mais aprofundados como a construção das camadas cliente e servidor e a API Socket.IO.

Nos últimos anos, o Node.js vem se tornando bastante popular entre programadores, entusiastas e empresas, e mesmo ainda estando em sua fase beta, há um grande esforço de seus desenvolvedores para mantê-lo o mais estável possível.

Essa estabilidade tem motivado pequenas, médias e grandes empresas a se aventurarem a criar projetos com Node.js em ambiente de produção.

Um dos motivos pelo qual o Node.js tem se tornado popular é o fato de utilizar JavaScript como linguagem de programação, uma vez que praticamente todo desenvolvedor web conhece ao menos os conceitos básicos e a sintaxe do JavaScript.

Além do mais, já era um sonho antigo dos entusiastas da linguagem poder trabalhar com JavaScript no lado do servidor. Inclusive, algumas outras tentativas de implementá-lo como linguagem server side no decorrer da história da web aconteceram, porém não acabaram se popularizando tanto.

Tão logo o Node.js começou a ficar conhecido, começaram a surgir implementações de frameworks comumente implementados em outras linguagens, como web servers, ferramentas de automatização e ORMs, multiplicando sua popularidade a cada dia.

Embora o Node.js tenha atingido um volume considerável de usuários, mantendo um grau confiável de estabilidade e com uma curva de aprendizagem razoavelmente pequena por utilizar JavaScript como linguagem de programação, sua característica mais importante é ser baseado em entrada e saída de dados não bloqueante, também chamado de *non blocking I/O* ou ainda I/O não bloqueante. Esta última expressão será a que usaremos para esse artigo.

Isso que dizer que, diferente do comportamento tradicional das tecnologias de programação, as requisições feitas ao Node.js que envolvem entrada ou saída de dados não permanecem presas ao processo até a sua conclusão.

Ao invés disso, são utilizadas solicitações com funções de callback, que não ficam presas ao processo. Quando o programa estiver pronto para entregar o resultado da solicitação ele fará uso destas funções de callback, entregando o conteúdo solicitado como parâmetro das mesmas.

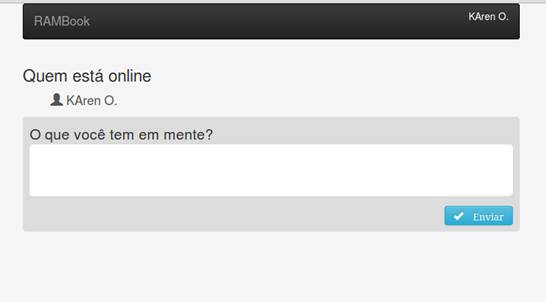
Essa programação orientada a eventos assíncronos é o que faz com que o Node.js tenha a característica de criar aplicações em tempo real.

**Aplicação exemplo**

Para demonstrar como o Node.js lida com aplicações em tempo real através da programação orientada a eventos e I/O não bloqueante, vamos apresentar uma aplicação web completa.

A aplicação é uma mini rede social volátil, oportunamente chamada de RAMBook, onde será possível criar posts, comentá-los e curti-los em tempo real, porém, cujos dados não serão persistidos em uma base de dados ou sistema de arquivos.

Devido a essa característica de não persistir os dados, qualquer usuário poderá se conectar à mini rede social sem precisar se autenticar, e sempre que um novo usuário entrar, será exibida a sua timeline vazia, um campo de texto para criar novos posts e a lista de usuários conectados, como apresentado na **Figura 1**.



**Figura 1**. Tela da mini rede social após o login do usuário.

**Ingredientes da aplicação**

Como deixamos a stack de persistência de dados de fora para este exemplo, ficamos com APIs da camada de apresentação e as regras do lado do servidor apenas.

Vale esclarecer que alguns códigos não estão necessariamente utilizando melhores práticas de desenvolvimento, em prol da didática.

Na camada cliente, serão utilizadas as seguintes ferramentas:

* Twitter Bootstrap;
* RequireJS;
* MustacheJS;
* jQuery;
* Bower.

Na camada server do Node.js utilizaremos:

* Socket.IO;
* ExpressJS;
* Nodemon.

**Preparação do ambiente de desenvolvimento**

O primeiro e mais óbvio passo é instalar o Node.js e uma vez que estiver instalado, vamos utilizar o NPM, o gerenciador de pacotes do Node.js, para instalar as demais dependências do projeto.

Se você ainda não tiver o Node.js instalado em sua máquina, ou se possui uma versão muito antiga, acesse o link de download do mesmo disponível na seção **Links** e efetue o download para o seu sistema operacional, realizando também a sua instalação.

Se for preciso, faça o mapeamento do diretório bin nas variáveis de ambiente do seu sistema operacional de maneira a poder acessar os binários node e npm de qualquer lugar.

**Instalando as dependências**

Uma vez que o Node.js estiver devidamente instalado e configurado, é hora de instalar as dependências do projeto através do NPM, o Node Packaged Modules.

O NPM é um repositório de módulos para Node.js repleto de APIs que podem ser facilmente baixadas, muito semelhante ao apt-get do Linux Debian, ou o Gem do Ruby.

Para nossa stack de backend vamos instalar o Socket.IO e o ExpressJS, e para as dependências da camada de front-end vamos utilizar o módulo Bower, que funciona de uma maneira extremamente semelhante ao próprio NPM. Porém o Bower gerencia APIs de JavaScript como jQuery e BackboneJS.

Existem outros módulos muito bons para a criação de projetos web como o Grunt, Gulp e Yeoman, que trabalham com geradores e utilizam o Bower e Grunt, colocando-os para trabalhar lado a lado de uma forma mais automatizada.

Porém, para o nosso exemplo a HTML, CSS e código JavaScript serão escritos “do zero”. Contudo, se você se sente confortável em trabalhar com ferramentas automatizadas, fique à vontade para utilizar o que preferir.

**Nota**: Todos os links para download das ferramentas citadas encontram-se disponíveis na seção **Links**.

**Baixando as dependências da camada server**

Crie um diretório chamado rambook e acesse-o para iniciar o download das dependências.

Vamos começar pelas dependências da camada server, utilizando o comando **npm install** conforme mostrado na **Listagem 1**.

**Listagem 1**. Baixando as dependências do backend.

npm install Socket.IO

npm install express

npm install nodemon -g

npm install bower -g

Note que para o download dos módulos Nodemon e Bower utilizamos o parâmetro **-g**, que é responsável por efetuar o download dos módulos e os instalar em um diretório de módulos globais, junto ao diretório onde você instalou o Node.js.

Ambos os módulos possuem arquivos binários, e podem ser executados como qualquer arquivo binário do seu sistema operacional.

Como já havia sido explicado anteriormente, o Bower funciona como um gerenciador de pacotes para APIs JavaScript, muito semelhante ao NPM.

O Nodemon é um “watcher”, que fica “escutando” quando qualquer arquivo dentro do diretório for alterado, e faz um “refresh” na aplicação Node.js quando isso acontece. É uma ferramenta que auxilia durante o desenvolvimento da aplicação, porém não é necessário (nem recomendado) sua utilização em ambiente de produção.

Há outras opções interessantes que podem ser usadas com o NPM, como preservar a referência das dependências em um arquivo package.json com a opção --save, e assim poder distribuir a aplicação apenas com os códigos fonte em controladores de versão como o Github, sem precisar fazer upload das dependências. Assim, outros desenvolvedores que fizerem download da aplicação precisarão apenas digitar o comando **npm install** e o NPM irá ler as dependências do arquivo package.json.

**Baixando as dependências da camada client**

Dentro do diretório rambook, crie um diretório chamado web e acesse esse diretório, pois é dentro dele que iremos baixar as dependências JavaScript/CSS, executando os comandos da **Listagem 2**.

Ainda no diretório web, crie os diretórios scripts e styles, onde ficarão nossos arquivos CSS e JavaScript da aplicação na stack de client.

**Listagem 2**. Baixando as dependências da camada client.

bower install jquery

bower install bootstrap

bower install mustache

bower install requirejs

Após o término dos downloads, você pode notar um novo diretório chamado bower\_components que possui uma estrutura de diretório para cada API instalada pelo Bower.

Por default, o Bower busca as APIs no Github, mas você pode especificar parâmetros para acessar outras fontes.

Além disso, assim como o NPM, o Bower possui um parâmetro --save que persiste a referência das dependências em um arquivo bower.json.

**Estrutura do projeto**

Até o momento temos uma estrutura de diretórios com todas as bibliotecas necessárias e prontas para trabalhar, conforme exibido na **Listagem 3**.

**Listagem 3**. Árvore de diretórios do projeto.

rambook

├── node\_modules

│ ├── express

│ └── Socket.IO

└── web

├── bower\_components

├── scripts

└── styles

**Preparando os arquivos do servidor**

Vamos utilizar o diretório “root” rambook para os arquivos da camada server, e o diretório web onde todo o conteúdo estático ficará.

No diretório rambook, crie os seguintes arquivos:

* **index.js**, que servirá como nosso “main program” e iniciará os servidores web e websocket;
* **server.js**, responsável por instanciar, configurar e exportar os servidores http e websocket;
* **userhandling.js**, onde residirá nossa regra de negócios para gerenciar as ações da mini rede social.

**Preparando os arquivos do client**

No diretório rambook/web crie os seguintes arquivos:

* index.html, que será a página da nossa aplicação. Este será o único HTML, pois faremos uma SPA, Single Page Application;
* styles/main.css, que conterá o estilo de nossa aplicação
* scripts/main.js, onde estará a configuração do RequireJS e servirá como nosso “main program”;
* scripts/client.js, onde estará nossa socket client que se comunicará com o servidor websocket;
* scripts/view.js, responsável por se comunicar com a client.js e fazer o link entre a websocket e os eventos dos elementos HTML da página.

Ao término da criação dos arquivos da camada client e server, devemos ter uma estrutura como exibido na **Listagem 4**.

**Listagem 4**. Árvore de diretórios e arquivos do projeto.

rambook

├── index.js

├── node\_modules

│ ├── express

│ └── Socket.IO

├── package.json

├── server.js

├── userhandling.js

└── web

├── bower\_components

│ ├── bootstrap

│ ├── jquery

│ ├── mustache

│ └── requirejs

├── bower.json

├── index.html

├── scripts

│ ├── client.js

│ ├── main.js

│ └── view.js

└── styles

└── main.css

Note que foram omitidos arquivos das dependências, como as do Socket.IO e ExpressJS.

Também foram omitidas as dependências baixadas pelo Bower, pois como ele trabalha com os repositórios das APIs do Github, muitos arquivos desnecessários para o projeto estão armazenados.

**Construindo a camada front-end da aplicação**

Primeiro, vamos escrever o conteúdo de nosso HTML, que foi montado utilizando o Twitter Bootstrap.

Caso você vá digitar o conteúdo desse arquivo, sugiro que utilize algum template, como os exemplos fornecidos pelo site do Bootstrap, ou utilize uma ferramenta de automatização como o Yeoman, ou ainda, utilize uma IDE que facilite a digitação de código com macros e “auto-complete text” como o Sublime Text, por exemplo.

**Entendendo a estrutura do HTML**

Se você já estiver familiarizado com o Bootstrap, não encontrará problemas para entender o código. Contudo, para facilitar a compreensão de todos, segue uma explicação de como o código HTML está organizado:

* No topo do código, temos o tradicional cabeçalho, onde estão sendo importados os estilos CSS e o tema do Bootstrap, o título da página e demais tags como a <meta>;
* A tag <nav> que marca o header da aplicação, onde reside o nome da aplicação e o formulário de login;
* A lista de usuários conectados a mini rede social apresentada na tag <ul>;
* Uma tela de boas vindas, solicitando ao usuário para que faça o login e possa então acessar o sistema;
* Um <div> oculto com o campo de texto para criar posts e a timeline do usuário;
* Tags <script> com o atributo type configurado para text/template, onde estão os templates a serem utilizados pelo Mustache na hora de renderizar os fragmentos da aplicação;
* E, finalmente, o include do RequireJS e seu arquivo principal.

O conteúdo de nosso arquivo HTML pode ser visto na **Listagem 5**.

**Listagem 5**. Conteúdo do documento index.html.

<!doctype html>

<html>

<head>

<meta charset="utf-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<title>RAMBook</title>

<meta name="description" content="">

<meta name="viewport" content="width=device-width">

<link rel="stylesheet" type="text/css" href="bower\_components/bootstrap/dist/css/bootstrap.min.css"/>

<link rel="stylesheet" type="text/css" href="bower\_components/bootstrap/dist/css/bootstrap-theme.min.css"/>

<link rel="stylesheet" href="styles/main.css">

</head>

<body>

<div class="container">

<nav class="navbar navbar-inverse" role="navigation">

<div class="navbar-header">

<span class="navbar-brand">RAMBook</span>

</div>

<div class="navbar-form navbar-right hidden" id="frm-join">

<div class="form-group">

<input type="text" id="username" maxlength="10" class="form-control input-sm" placeholder="Digite seu nome">

</div>

<button type="button" id="btn-entrar" class="btn btn-info input-sm">Entrar</button>

</div>

<div></div>

</nav>

<div class="row">

<div class="col-md-3">

<h3>Quem está online</h3>

<ul class="online-users"></ul>

</div>

<div class="col-md-9">

<div id="not-logged">

<h2>Você precisa se conectar</h2>

<h3>Digite seu nome e clique em "Entrar"</h3>

</div>

<div id="logged" class="hidden">

<div class="post-form">

<div class="action">O que você tem em mente?</div>

<textarea class="txt" id="post-content"></textarea>

<div class="pull-right">

<button class="btn btn-info glyphicon glyphicon-ok" id="btn-postar"> Enviar</button>

</div>

</div>

<div class="posts"></div>

</div>

</div>

</div>

</div>

<script type="text/template" id="userlist-template">

{{#users}}

<li data-user-id="{{id}}"><span class="glyphicon glyphicon-user"></span> {{username}}</li>

{{/users}}

</script>

<script type="text/template" id="post-template">

{{#post}}

<div class="post" data-post-id="{{id}}">

<div class="poster-info">

<span class="poster">{{author}}</span>

<span class="action">escreveu, as </span>

<span class="time">{{hora}} hs</span>

</div>

<div class="post-body">{{text}}</div>

<div class="post-actions">

<a class="glyphicon glyphicon-thumbs-up lnk-like-post" data-post-id="{{id}}" data-like="true">(0)</a>

<a class="glyphicon glyphicon-comment lnk-comment" data-post-id="{{id}}"></a>

</div>

<div data-post-to-comment-id="{{id}}" class="comment-form hidden">

<div class="action">Deixe seu comentário</div>

<textarea class="txt"></textarea>

<div class="pull-right">

<button class="btn btn-info btn-sm glyphicon glyphicon-ok btn-commentar"> Enviar</button>

<button class="btn btn-info btn-sm glyphicon glyphicon-remove lnk-cancel-comment" data-post-id="{{id}}"> Cancelar</button>

</div>

</div>

<div class="post-comments"></div>

</div>

{{/post}}

</script>

<script type="text/template" id="comment-template">

{{#comment}}

<div class="comment" data-comment-id="{{id}}">

<div class="poster-info">

<span class="poster">{{author}}</span>

<span class="action">comentou, as</span>

<span class="time">{{hora}} hs</span>

</div>

<div class="post-body">{{text}}</div>

<div class="post-actions">

<a class="glyphicon glyphicon-thumbs-up lnk-like-comment" data-comment-id="{{id}}" data-like="true">(0)</a>

</div>

</div>

{{/comment}}

</script>

<script src="bower\_components/requirejs/require.js" data-main="scripts/main"></script>

</body>

</html>

Note na listagem que ambos os includes de CSS e do RequireJS estão apontando para o diretório bower\_components, e em seguida, para os respectivos diretórios de cada biblioteca, com exceção de nosso próprio CSS, que está sendo direcionado para styles/main.css. O conteúdo do arquivo main.css pode ser visto na **Listagem 6**.

Vale destacar que, caso você decida fazer o deploy de sua aplicação com as APIs contidas no diretório bower\_components, você deve excluir os arquivos desnecessários, ou copiar as APIs para outro diretório, ou ainda, preferivelmente, utilize uma ferramenta de automatização como as tasks do Grunt para realizar essa tarefa para você.

**Listagem 6**. Conteúdo do arquivo de estilos styles/main.css.

body {

background-color: rgb(245, 245, 245);

overflow-y: scroll;

}

#frm-join {

color: #fff;

}

#log {

font-size: 25px;

text-align: center;

color: red;

}

.post {

border-bottom: 1px solid #aaa;

margin-top: 25px;

}

.poster-info {

font-size: 120%;

margin-bottom: 5px;

padding: 5px;

}

.poster {

color: #2aabd2;

font-weight: bold;

}

.post-body {

min-height: 30px;

color: #555;

padding: 5px;

}

.post-actions {

text-align: left;

padding: 10px;

}

.post-comments {

border-top: 1px dotted #bbb;

}

.comment {

font-size: 90%;

padding-left: 20px;

}

.post-form, .comment-form {

overflow: auto;

padding: 10px;

background-color: rgb(235, 235, 235);

border-radius: 5px;

}

.post-form {

font-size: 150%;

background-color: rgb(220, 220, 220);

}

.txt {

width: 100%;

height: 75px;

border: none;

border-radius: 5px;

margin-bottom: 10px;

}

.online-users {

list-style-type: none;

font-size: 130%;

color: #555;

}

a {

text-decoration: none !important;

cursor: pointer;

}

**Importando as dependências do JavaScript**

Como você pôde notar, a única dependência de JavaScript que temos é a do RequireJS, seguido de sua dependência de arquivo principal, main.js, no atributo data-main=”scripts/main”. O RequireJS pode ser tratado basicamente como uma API para modularização e carregamento assíncrono “on demand” de arquivos JavaScript.

O conteúdo do main.js aparece na **Listagem 7** e apresenta como os módulos de nossa aplicação na camada client estão configurados:

**Listagem 7**. Configurações do RequireJS.

requirejs.config({

basePath: './',

paths: {

'jquery': '../bower\_components/jquery/jquery.min',

'socketio': '/Socket.IO/Socket.IO',

'mustache': '../bower\_components/mustache/mustache'

},

shim: {

mustache: {

exports: 'Mustache'

}

}

});

require(['view'], function(view) {});

O código é autoexplicativo: trata-se basicamente de um objeto literal com as configurações do caminho base dos módulos, os caminhos para os módulos jQuery, Mustache e nossa websocket client, gentilmente fornecida pelo próprio Socket.IO server, como veremos mais à frente.

Ao final do código, é importado o módulo view, referente ao arquivo view.js, que pode ser analisado na **Listagem 8** e então este é executado.

**Listagem 8**. Arquivo view.js.

require(['jquery', 'client', 'mustache'], function($, client, Mustache) {

var username = $('#username'),

onlineUsers = $('.online-users'),

postContent = $('#post-content'),

frmJoin = $('#frm-join'),

posts = $('.posts'),

notLogged = $('#not-logged'),

logged = $('#logged'),

userlistTpl = $('#userlist-template').html(),

postTpl = $('#post-template').html(),

commentTpl = $('#comment-template').html(),

lnkComment = $('.lnk-comment');

//Formatador da hora para o mustache

function formatHour() {

var today = new Date(this.timestamp);

return today.getHours() + ':' + today.getMinutes();

}

/\*\*\*\*\*\*\*\*\*DECLARAÇÃO DE EVENTOS\*\*\*\*\*\*\*\*\*\*/

$('.container')

.on('click', '#btn-entrar', function(ev) {

var name = username.val();

if(name) {

client.userLogin(name);

}

})

.on('click', '.lnk-comment', function(ev) {

var postId = $(this).data('post-id');

$('[data-post-to-comment-id="' + postId + '"]').toggleClass('hidden');

})

.on('click', '#btn-postar', function(ev) {

client.makePost(postContent.val());

postContent.val('');

})

.on('click', '.btn-commentar', function(ev) {

var commentForm = $(this).parents('.comment-form'),

postId = commentForm.data('post-to-comment-id'),

textarea = $('.txt', commentForm),

commentData = {postId: postId, text: textarea.val()};

client.makeComment(commentData);

textarea.val('');

commentForm.addClass('hidden');

})

.on('click', '.lnk-cancel-comment', function(ev) {

var commentForm = $(this).parents('.comment-form');

commentForm.addClass('hidden');

})

.on('click', '.lnk-like-post', function(ev) {

var postId = $(this).data('post-id'),

like = $(this).attr('data-like');

if(like === 'true') {

like = true;

} else {

like = false;

}

client.likePost({postId: postId, like: like});

$(this).attr('data-like', !like);

})

.on('click', '.lnk-like-comment', function(ev) {

var commentId = $(this).data('comment-id'),

like = $(this).attr('data-like');

if(like === 'true') {

like = true;

} else {

like = false;

}

client.likeComment({commentId: commentId, like: like});

$(this).attr('data-like', !like);

});

/\*\*\*\*\*\*\*\*\*DECLARAÇÃO DOS LISTENERS\*\*\*\*\*\*\*\*\*\*/

client.events.connect = function(data) {

frmJoin.removeClass('hidden');

};

client.events.userlogin = function(data) {

frmJoin.html(username.val());

notLogged.addClass('hidden');

logged.removeClass('hidden');

};

client.events.userlist = function(data) {

var html = Mustache.render(userlistTpl, {users: data});

onlineUsers.html(html);

};

client.events.makepost = function(data) {

var html = Mustache.render(postTpl, {post: data, hora: formatHour});

posts.prepend(html);

};

client.events.makecomment = function(data) {

var html = Mustache.render(commentTpl, {comment: data, hora: formatHour}),

postComments = $('.post[data-post-id="' + data.postId + '"] .post-comments');

postComments.prepend(html);

};

client.events.likepost = function(data) {

var lnkLikePost = $('.post[data-post-id="' + data.postId + '"] .lnk-like-post');

lnkLikePost.html('(' + data.numLikes + ')');

};

client.events.likecomment = function(data) {

var lnkLikeComment = $('.comment[data-comment-id="' + data.commentId + '"] .lnk-like-comment');

lnkLikeComment.html('(' + data.numLikes + ')');

};

});

Apesar de ser um pouco longo, o arquivo view.js é muito simples de ser entendido, pois ele é a ligação entre a interação do usuário e a comunicação com a websocket client, que por sua vez se comunica diretamente com o nosso servidor websocket.

No topo do arquivo, pode-se notar a importação das dependências do jQuery, usado para ler elementos DOM do nosso index.html e realizar binds das ações do usuário, o MustacheJS usado para fazer a cola dos dados recebidos do servidor e os templates que vimos nas tags <script> na HTML, e, finalmente, a importação da nossa camada de negócios, client.js, que pode ser vista na **Listagem 9**.

**Listagem 9**. Regras de negócio da camada client – client.js.

define(['socketio'], function(io) {

var socket = io.connect('/'),

events = {};

socket.emit('likecomment', {});

socket.emit('unlikecomment', {});

//listeners

socket.on('connect', function(data) {

events.connect(data);

});

socket.on('msg', function(data) {

console.log(data);

});

socket.on('userlogin', function(data) {

events.userlogin(data);

});

socket.on('userlist', function(data) {

events.userlist(data);

});

socket.on('makepost', function(data) {

events.makepost(data);

});

socket.on('makecomment', function(data) {

events.makecomment(data);

});

socket.on('likepost', function(data) {

events.likepost(data);

});

socket.on('likecomment', function(data) {

events.likecomment(data);

});

//emits

function userLogin(username) {

socket.emit('userlogin', {username: username});

}

function makePost(post) {

socket.emit('makepost', post);

}

function makeComment(comment) {

socket.emit('makecomment', comment);

}

function likePost(likeData) {

socket.emit('likepost', likeData);

}

function likeComment(likeData) {

socket.emit('likecomment', likeData);

}

return {

socket: socket,

userLogin: userLogin,

makePost: makePost,

makeComment: makeComment,

events: events,

likePost: likePost,

likeComment: likeComment

};

});

Muito semelhante ao view.js, o arquivo client.js está dividido em listeners e eventos, chamados de emits pelo Socket.IO, o qual entraremos em mais detalhes a seguir.

Note que a única dependência do client.js é a API do Socket.IO, um framework responsável por se comunicar com o servidor de websocket.

**Entendendo o Socket.IO no lado do client**

Fazendo uso da programação orientada a eventos do Node.js, a proposta do Socket.IO é oferecer de forma transparente ao programador as diversas maneiras de se trabalhar com conexões persistentes entre o browser e o servidor.

A maneira mais coerente de fazer isso é através da API de Websocket, apresentada na especificação da HTML5.

A maioria dos browsers hoje em dia já possuem a implementação dessa especificação, porém, alguns browsers mais antigos não contam com a API da Websocket.

Nesses casos há diversas soluções como o polling, long polling, Flash Websocket e outros mais que tem a característica de simular de forma paliativa o comportamento de uma websocket.

Quando o programador precisa se preocupar com os vários browsers e versões, uma porção de validações precisam ser feitas para identificar qual é a solução mais adequada para lidar com conexões persistentes. E o pior, é ter que implementar todas elas de forma a atender todos os públicos.

Com o advento do combo Node.js + Socket.IO essa dor de cabeça terminou, e o programador pôde se concentrar nas regras de negócio da aplicação.

Quando o browser se conecta através da API do Socket.IO.js, este se encarrega de fazer as devidas validações, preparar o contorno mais coerente, se necessário, e se comunicar com o servidor passando todas essas informações.

Além do mais, caso o browser do usuário possua suporte a Websockets, uma série de passos precisam ser seguidos para conectar o usuário ao servidor, como o handshake, que é uma comunicação HTTP inicial responsável por validar e autenticar o usuário no servidor, até a conexão com o servidor da Websocket de fato.

**Aprendendo a lidar com a API do Socket.IO**

Diferentemente do protocolo HTTP tradicional, quando emitimos um evento ao servidor da Websocket, não ficamos “pendurados” ao servidor até que este retorne uma resposta de conteúdo ou uma mensagem de erro.

Ao invés disso, e devido ao seu comportamento orientado a eventos assíncronos e não bloqueantes, acabamos criando nosso próprio mini protocolo de comunicação. Ou seja, por um lado, emitimos uma mensagem ao servidor, e precisamos declarar uma escuta para “ouvir” a resposta do mesmo, que pode acontecer a qualquer momento, e não imediatamente após receber nossa mensagem.

Por isso, basicamente o Socket.IO trabalha com duas funções **emit** e **on**, como pôde ser visto no client.js. Ou seja, quando queremos emitir uma mensagem ao servidor, usamos a função emit, seguida de um nome de evento definido pelo programador e os dados a serem recebidos pelo servidor.

Quando queremos escutar um evento recebido do servidor, precisamos implementar a função on, cujos parâmetros são uma string com o nome do evento, e uma função de callback cujo parâmetro é a informação enviada pelo servidor.

Por exemplo, quando queremos saber se um usuário está logado, utilizamos o evento userlogin, como no exemplo abaixo:

socket.on('userlogin', function(data) {

events.userlogin(data);

});

Note que estamos escutando o evento userlogin, e os dados recebidos pelo parâmetro data são enviados para o objeto events, que será posteriormente lido pelo view.js.

Por outro lado, quando queremos notificar o servidor de que um novo usuário quer se conectar, emitimos o evento com o nome userlogin. Utilizamos o mesmo nome para eventos que representam a mesma ação para facilitar o entendimento:

function userLogin(username) {

socket.emit('userlogin', {username: username});

}

A maneira como a comunicação orientada a eventos funciona ficará mais clara quando virmos o código do servidor.

**Construindo a camada backend da aplicação**

Recapitulando a estrutura de arquivos da nossa camada do servidor, possuímos três arquivos: *index.js*,*server.js* e *userhandling.js*.

O conteúdo do index.js, exibido na **Listagem 10**, é bastante intuitivo.

**Listagem 10**. Conteúdo do index.js.

//Carrega o módulo server.js

var Server = require('./server'),

//Carrega o objeto io

socketServer = Server.socketServer,

//Carrega o objeto httpServer

webServer = Server.webServer,

UserHandling = require('./userhandling'),

userHandling = new UserHandling(socketServer);

//Inicia o web server na porta 9000

webServer.listen(9000);

O arquivo *index.js* é muito simples, pois apenas instancia os módulos de nossa aplicação e faz a união destes, iniciando finalmente o servidor.

Primeiro, são importados os módulos em variáveis, como socketServer que retém a referência ao Socket.IO, webServer com a referência ao servidor Web e o objeto userHandling, que possui toda a regra de negócio da nossa mini rede social volátil.

Então, o servidor web é iniciado na porta 9000 e o objeto userHandling é linkado ao servidor da Websocket.

O arquivo server.js, que pode ser analisado na **Listagem 11**, bem como o index.js, faz referência aos módulos do ExpressJS e Socket.IO. Então finalmente os instancia e os prepara para serem utilizados pelo index.js.

**Listagem 11**. Conteúdo do arquivo server.js.

process.title = 'TTT Server';

//Importando ExpressJS

var express = require('express'),

//Importando Socket.IO

socketio = require('Socket.IO'),

//Criando uma instancia do ExpressJS

app = express(),

//Criando um HTTP Server a partir do ExpressJS

httpServer = require('http').createServer(app),

//Utilizando a mesma porta do HTTP Server para o Socket.IO

io = socketio.listen(httpServer)

;

//Diz ao Express que o diretório web contém conteúdos estáticos

app.use(express.static(\_\_dirname + '/web'));

//Exporta os módulos

module.exports.socketServer = io;

module.exports.webServer = httpServer;

Basicamente, a maior diferença na estrutura de código do *server.js* e o *index.js* é a utilização do *proccess.title*, que registra o nome do processo no sistema operacional. Entretanto, essa funcionalidade pode não funcionar em todos os sistemas operacionais, mas também não afetará o funcionamento da aplicação.

Após a carga dos módulos, estes são preparados, configurados e atados uns aos outros.

Note que o servidor de websocket do Socket.IO consegue aproveitar o listener do web server do ExpressJS.

Por último, mas não menos importante, aliás, talvez o arquivo mais importante de nossa aplicação, a **Listagem 12** apresenta o conteúdo do userhandling.js, que retém toda a regra de negócio da mini rede social volátil.

É nele que estão todos os eventos que são emitidos e “escutados” pelo Socket.IO em relação ao client.

**Listagem 12**. Regras de negócio da rede social no arquivo userhandling.js.

var UserHandling = module.exports = function(io) {

//Mantem controle dos "likes" para cada post

this.posts = {};

this.io = io;

this.loggedUsers = {};

this.init();

};

UserHandling.prototype = {

init: function() {

var that = this;

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

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

that.onDisconnect(socket);

});

that.bindEvents(socket);

that.onUserList(socket);

});

},

bindEvents: function(socket) {

var that = this;

socket.on('userlogin', function(data) {

that.onUserLogin(socket, data);

});

socket.on('userlist', function(data) {

that.onUserList(socket, data);

});

socket.on('makepost', function(data) {

that.onMakePost(socket, data);

});

socket.on('makecomment', function(data) {

that.onMakeComment(socket, data);

});

socket.on('likepost', function(data) {

that.onLikePost(socket, data);

});

socket.on('likecomment', function(data) {

that.onLikeComment(socket, data);

});

},

onDisconnect: function(socket) {

delete this.loggedUsers[socket.id];

this.onUserList(socket);

},

sendMessage: function(socket, msg) {

socket.emit('msg', {text: msg, type: 'info'});

},

onUserLogin: function(socket, data) {

var that = this;

if(data.username) {

socket.handshake.username = data.username;

that.loggedUsers[socket.id] = socket;

socket.emit('userlogin');

that.onUserList(socket);

} else {

this.sendMessage(socket, 'vc precisa escolher um nome');

}

},

onUserList: function(socket, data) {

var that = this,

keys = Object.keys(this.loggedUsers),

userList = new Array(keys.length),

i = 0;

keys.forEach(function(k) {

userList[i++] = {

id: k,

username: that.loggedUsers[k].handshake.username

};

});

socket.emit('userlist', userList);

socket.broadcast.emit('userlist', userList);

},

onMakePost: function(socket, data) {

var id = Date.now(),

postData = {

author: socket.handshake.username,

authorId: socket.id,

timestamp: id,

id: id,

text: data

};

//cadastra o post no controle de likes

this.posts[id+''] = 0;

socket.emit('makepost', postData);

socket.broadcast.emit('makepost', postData);

},

onMakeComment: function(socket, data) {

var id = Date.now(),

commentData = {

author: socket.handshake.username,

authorId: socket.id,

timestamp: Date.now(),

text: data.text,

postId: data.postId,

id: id

};

//cadastra o post no controle de likes

this.posts[id+''] = 0;

socket.emit('makecomment', commentData);

socket.broadcast.emit('makecomment', commentData);

console.log(commentData);

},

onLikePost: function(socket, data) {

var likeCount = this.posts[data.postId+''];

if(data.like) {

likeCount += 1;

} else {

likeCount -= 1;

}

this.posts[data.postId+''] = likeCount;

socket.emit('likepost', {postId: data.postId, numLikes: likeCount});

socket.broadcast.emit('likepost', {postId: data.postId, numLikes: likeCount});

},

onLikeComment: function(socket, data) {

var likeCount = this.posts[data.commentId+''];

if(data.like) {

likeCount += 1;

} else {

likeCount -= 1;

}

this.posts[data.commentId+''] = likeCount;

socket.emit('likecomment', {commentId: data.commentId, numLikes: likeCount});

socket.broadcast.emit('likecomment', {commentId: data.commentId, numLikes: likeCount});

}

};

Embora um pouco mais comprido do que os outros arquivos da camada de backend, o arquivo userhandling.js se assemelha muito à sua versão da camada client.

Além dos eventos tradicionais referentes à rede social, como logar usuário, fazer post, comentários e likes, o userhandling possui dois eventos especiais: connection e disconnect.

O evento connection é disparado logo após o handshake, ou seja, logo após o client se comunicar com o servidor através de uma comunicação utilizando o protocolo HTTP.

Uma vez que ambos client e server “apertam as mãos” uns dos outros, o servidor permite ao client o acesso a websocket.

Ao mesmo passo em que, quando um usuário se desconecta do servidor, seja por ter fechado o browser ou ter disparado uma ação de desconexão, o evento disconnect é acionado e uma série de ações é realizada.

No nosso caso, quando o evento connection é disparado, fazemos o bind de todos os eventos que o server irá escutar do client e vice-versa, e também emitimos um evento userlist dizendo a todos os usuários conectados que um novo usuário entrou na rede social.

Da mesma maneira, o evento disconnect remove o usuário da lista de usuários online e avisa a todos os outros usuários, através do objeto socket.broadcast.emit com a nova lista de usuários online.

Outro ponto interessante é a maneira como estamos lidando com as curtidas de posts e comentários. Como não temos uma maneira de persistir os dados em disco, foi criado o objeto this.posts, como pode ser visto no começo do arquivo userhandling.js. Este objeto mantém um histórico do id dos posts e quantas curtidas eles receberam. Para se ter uma noção de como este controle funciona, dê uma olhada nas funções onLikeComment e onLikePost. Os ids dos posts foram feitos com o timestamp da hora do post.

Como havíamos comentado anteriormente, foram usados os mesmos nomes de eventos em ambos os lados do client e servidor, para garantir uma melhor legibilidade do código. Ou seja, se o server recebe um evento makepost, ao concluir a inclusão do post na timeline, ele informa o próprio usuário e os demais usuários com um evento de mesmo nome.

**Juntando as peças**

Agora que escrevemos ambas aplicações no lado cliente e servidor, é hora de executá-la!

Vá para o diretório raiz rambook e execute o comando nodemon index.js.

Como mencionamos anteriormente, o Nodemon é uma ferramenta que executa um programa Node.js, no nosso caso, index.js, e fica escutando por alterações no código. Toda vez que uma alteração é feita em um dos códigos contidos no diretório onde o Nodemon foi executado, ele recarrega toda a aplicação. Isso evita com que você tenha que parar o servidor e iniciá-lo novamente para cada alteração que fizer.

Após executar o Nodemon, você deverá ver uma tela semelhante à apresentada na **Listagem 13**.

**Listagem 13**. Servidor de websocket em ação.

chamb@selmy:~/Dropbox/projetos/tttserver$ nodemon index.js

20 Jan 21:14:06 - [nodemon] v0.7.10

20 Jan 21:14:06 - [nodemon] to restart at any time, enter `rs`

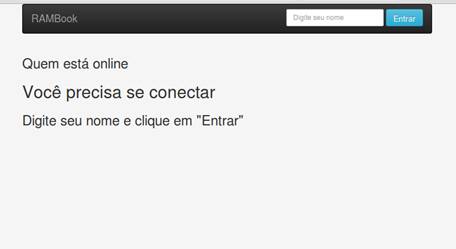
20 Jan 21:14:06 - [nodemon] watching: /home/chamb/Dropbox/projetos/tttserver

20 Jan 21:14:06 - [nodemon] starting `node index.js`

info - Socket.IO started

Agora podemos finalmente testar a aplicação, abrindo o browser e acessando a URL <http://localhost:9000/>.

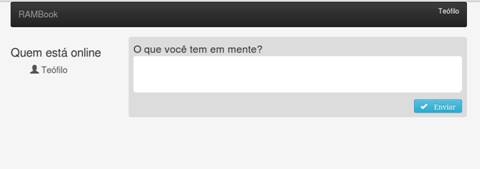
Você deverá ver uma tela conforme exibido na **Figura 2**.



**Figura 2**. Tela de login da mini rede social.

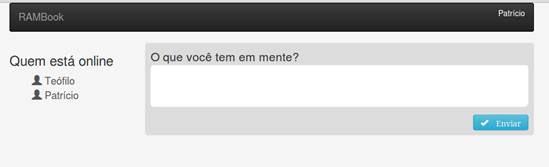
Perceba que o formulário de login leva um tempo para aparecer. Isso acontece pois inicialmente o elemento <div> que contém o formulário está oculto, e aguarda até que o client receba o evento on.('connect'). Veja como ele é semelhante ao evento connection no lado do servidor.

Após aparecer o formulário de login, digite seu nome e clique em “Entrar”, e você verá uma tela idêntica à exibida na **Figura 3**.



**Figura 3**. Tela da mini rede social após o login.

Se você abrir uma nova aba em seu browser e acessar a aplicação, poderá logar com um usuário de nome diferente e verá dois usuários conectados à mini rede social, conforme mostrado na **Figura 4**.



**Figura 4**. Usuários conectados simultaneamente na mini rede social.

Agora, tente criar posts com um usuário, fazer comentários nesse post com outro usuário, curtir e “descurtir” posts e comentários. Você poderá notar que as informações estarão atualizadas em tempo real em ambas as abas do browser.

Faça testes com mais usuários, outros browsers, peça para seus amigos se conectarem em sua mini rede social e divirta-se! Você poderá ter um resultado tão bacana quanto o exemplo da **Figura 5**.



**Figura 5**. Mostrando a aplicação em tempo real em ação.

Esta é uma pequena amostra do poder do Node.js e como ele, junto a APIs como o Socket.IO facilitam o desenvolvimento de aplicações em tempo real, graças a sua programação orientada a eventos.

Nossa aplicação de exemplo é bastante divertida, porém, está longe de estar completa.

Tente implementar novas funcionalidades, como avisar ao usuário postador quando um de seus posts foram curtidos, ou validar os dados de forma a impedir o usuário de cadastrar posts em branco.

Se você já estiver se sentindo confortável com o Node.js, tente criar a persistência dos dados de maneira a não perder os posts e comentários, e também criar um usuário e senha para impedir usuários de entrarem sem uma identidade bem definida. Quem sabe uma integração com a API do Facebook ou Twitter!

Em conclusão, pudemos ver uma amostra do que o Node.js é capaz de fazer, e como ele tem sido largamente utilizado pela comunidade graças a ferramentas como o Grunt, Bower e Yeoman.

E, embora o Node.js seja uma tecnologia muito empolgante, lembre-se de que ele não é uma bala de prata, e ainda que possa ser usado para desenvolver praticamente qualquer tipo de aplicação, cada tecnologia tem seu paradigma e especialidade, que, no caso do Node.js é trabalhar com a programação orientada a eventos e concorrência.

**Links**

**Download do Node.js**<http://nodejs.org/download/>

**Exemplos de uso do Twitter Bootstrap**<http://getbootstrap.com/getting-started/#examples>

**Site do Sublime Text**<http://www.sublimetext.com/>

**Site do Socket.IO**[http://Socket.IO/](http://socket.io/)

**Site do RequireJS**<http://requirejs.org/>

**Site do NPM**<https://npmjs.org/>