Web app real time con Node.js, JSON, Backbone y PonyExpress

Para este tutorial necesitarás tener instalado Node.js y el resto será magia :).
(Si no sabes como, devuelvete a la clase #2 en donde te enseñamos a instalarlo para usarlo con stylus)

Ten en cuenta que para empezar con este tutorial debiste primero haber leído la lectura "¿Qué es pony express?" y la lectura sobre sistemas de templates, desarrollaremos un ToDo el cual tendrá la habilidad de cambiar su estado de completo o no así que empecemos con el desarrollo de nuestro ToDo, el objetivo de nuestro ToDo será crear una aplicación de tareas en donde agregaremos cada ítem y tendrá la capacidad de hacer check de completa o incompleta y poder eliminarla si así queremos.

Empezaremos por crear el package.json con los requerimientos del sistema que serán los siguientes:

Package.json

```
{
  "name"    : "ToDo",
  "version" : "0.0.1",
  "dependencies" : {
    "socket.io"   : "0.9.14",
    "express"     : "3.1.1",
    "swig"        : "0.13.5",
        "consolidate" : "0.9.0",
        "node-uuid"   : "1.4.0",
        "socket.io-client" : "0.9.11"
  }
}
```

Luego de esto creamos un archivo llamado server.js que será el archivo del servidor en el que manejamos todo el backend de nuestra aplicación.

Server.js

Debemos empezar importar las librerías necesarias para que todo esto sirva, en este caso será express, el server http, socket.io, consolidate, swig y node-uuid.

Express.js: Framework de node.js para crear aplicaciones web de manera sencilla.
Socket.io: Conexión con sockets para cada cliente con interacción para cada cliente en tiempo real
Consolidate: Encargado del render de nuestros templates para el frontend.
Swig: sistema de templates a utilizar.
Node-uuid: encargado de dar un id único a cada tarea

Y adicionalmente creamos un arreglo llamado ToDoTask.

```
var express  = require('express'),
  app      = express(),
  server   = require('http').createServer(app),
  io       = require('socket.io').listen(server),
  swig     = require('swig'),
  cons     = require('consolidate'),
  uuid     = require('node-uuid'),
  ToDoTask = [];
```

Luego debemos decirle a nuestro servidor por que puerto escuchar y adicionalmente mandaremos un mensaje a consola indicando donde podemos ver el server corriendo (Seamos personas cool).

```
server.listen(3000);
console.log('visita http://localhost:3000 para ver el ToDo');
```

Luego procedemos a establecer las propiedades de consolidate y swig en donde primero quitamos el cache de swig cosa que solo hacemos en

local pero a paso de producción lo activamos.

```
swig.init({
  cache : false
});
```

Luego indicamos a engine que trabajara con consolidate para renderizar los templates de swig.

```
app.engine('.html', cons.swig);
app.set('view engine', 'html');
```

Establecemos la carpeta estática en donde tenemos todos los archivos css, javascript, dependencias, y el maravilloso y atractivo PonyExpress.

```
app.use(express.static('./static'));
```

Y ahora habilitamos las peticiones POST a nuestro server.

```
app.use(express.bodyParser());
app.use(express.cookieParser());
app.use(express.methodOverride());
```

Habilitamos una petición .get a la raíz de nuestro server indicando que archivo .html se renderiza.

```
app.get('/', function (req, res) {
  res.render('index');
});
```

Habilitamos una solicitud POST a /ToDoTask en donde mostraremos el JSON con la información que tomara PonyExpress para mostrar del lado del cliente.

```
app.post('/ToDoTask', function (req, res){
  req.body.id = uuid.v1();
  console.log('body', req.body);
  ToDoTask.push(req.body);
  io.sockets.emit('ToDoList::create', req.body);
  res.send(200, {status:"Ok"});
});
```

En este primero indicamos un uuid al id del body, luego mandamos un mensaje a consola con el request del body, procedemos a hacer un push al arreglo ToDoTask con los requerimientos del body, emitimos un socket con el evento create (Evento de PonyExpress para crear elementos u objetos) y luego indicamos que el estado de nuestro elemento está Ok.

Luego debemos crear lo que pasará cuando tengamos un evento de eliminar un mensaje, en este caso creamos un app.delete hacia el id del mensaje que queremos eliminar.

```
app.delete('/ToDoTask/:id', function (req, res){
  var ToDoList;
  for (var i = ToDoTask.length - 1; i >= 0; i--) {
    ToDoList = ToDoTask[i];
    if(ToDoList.id === req.params.id){
      ToDoTask.splice(i,1);
    }
  };
  io.sockets.emit('ToDoList::delete', {id:req.params.id});
  res.send(200, {status:"Ok"});
});
```

Entonces, primero definimos la variable ToDoList, recorremos mediante un for al arreglo ToDoTask y le asignamos su valor a ToDoList, verificamos si el mensaje en cuestión es la misma que el que se quiere eliminar, en caso de ser así eliminamos el mensaje y corremos los elementos que están después del que se eliminó, luego emitimos a nuestros clientes de los sockets el evento que se eliminó un elemento indicando que mensaje se eliminó y decimos que la operación tuvo un

estatus Ok.

Ahora configuramos un update para cada mensaje en donde lo que hacemos es un .put que es exactamente lo mismo que un .post solo que es el utilizado por backbone hacia cada mensaje.

```
app.put('/ToDoTask/:id', function (req, res){
  var ToDoList;
  for (var i = ToDoTask.length - 1; i >= 0; i--) {
    ToDoList = ToDoTask[i];
    if(ToDoList.id === req.params.id){
      ToDoTask[i] = req.body;
    }
  };
  io.sockets.emit('ToDoList::update', req.body);
  res.send(200, {status:"Ok"});
});
```

Ahora como hicimos en el evento de elimina vamos a crear la variable ToDoList y por un for recorremos a ToDoTask y cuando encontramos el mensaje en cuestión lo reemplazamos por la información que nos lanza el request del body emitimos un evento update (Evento de PonyExpress) a cada cliente informando del cambio e informamos que su estatus final fue ok.

Ahora debemos crear una variable connection con la información de cada evento de PonyExpress y encendemos el socket para que cada conexión tenga estos mismos eventos.

Y con esto terminamos todo nuestro servidor para nuestro ToDo y es la primera parte de nuestro proyecto.

Ahora empezaremos con el desarrollo de nuestro archivo html en donde tendremos el como se verá nuestra app y de aquí derivamos el javascript de nuestra app y luego quedará a nuestra creatividad el estilo de nuestra página.

Index.html

Entonces, para crear nuestro index.html primero en la raíz del documento debemos crear una carpeta llamada views y dentro de esta colocaremos nuestro index.html.

En el head debemos importar todas las dependencias que en este caso seria:

```
<script type="text/javascript" src="dependencies/jquery-
1.9.1.min.js"></script>
<script type="text/javascript" src="dependencies/neon.js"></script>
<script type="text/javascript"
src="dependencies/CustomEvent.js"></script>
<script type="text/javascript"
src="dependencies/CustomEventSupport.js"></script>
<script type="text/javascript" src="socket.io/socket.io.js"></script>
<script type="text/javascript" src="dependencies/underscore-
min.js"></script>
<script type="text/javascript" src="dependencies/backbone.js"></script>
```

Para aclarar de donde salen todos estos archivos, debemos crear una carpeta static en la raíz de la pagina y dentro de esta debemos descargar la carpeta de dependencias para PonyExpress y colocarla dentro de static, de aqui salen todos los archivos de dependencias.

Ahora procedemos a importar PonyExpress, archivo que descargamos y colocamos en static en una carpeta llamada lib.

```
<script type="text/javascript" src="lib/PonyExpress.js"></script>
```

Y por último importamos el archivo de JavaScript en donde luego escribiremos todo el código que recoge la información del json y se encarga de renderizar y en caso de un evento crear, actualizar o eliminar, le informe a el socket y sea actualizado el server.

Ahora nos vamos al Body en donde crearemos la siguiente estructura:

```
<h1>ToDo</h1>
    <div id="Tasks"></div>
    <script type="text/template" id="AgregarTask">
      <p class="write"> ✎ </p>
      <input class="inputext" id="user" type="text"/>
      <input class="inputext" id="text"></input>
      <button id="submit">Enviar</button>
      <div class="TaskIncomplete"/>
      <div class="TaskComplete"/>
    </script>
    <script type="text/template" id="ToDoList-template">
      <div class="IndTask">
        <label class="highlight">✓</label>
        <div class="Task">
          <h3><%= user %></h3>
          <p><%= text %></p></div>
        <label class="remove">✗</label>
        </div>
      </div>
    </script>
    <nav>
      <p id="all">All</p>
      <p id="complete">Complete</p>
      <p id="incomplete">Incomplete</p>
    </nav>
```

En el div con id Tasks tendrá el contenido de toda la página.

El script con id AgregarTask es en donde tenemos los campos para agregar los datos en cuanto a usuario y tarea.

En el script con id ToDoList-template indicamos la vista de cada mensaje y utilizamos urderscore como motor de template para cargar los datos necesarios, adicionalmente tenemos dos label con clase highlight y remove los que se encargan de cambiar el estado de la tarea o removerla.

Teniendo todo esto listo y entendido empezamos con el desarrollo del JavaScript en un archivo ubicado en static/js y nombre demo.js.

Demo.js

Bueno ahora vamos a la parte de la magia y donde veremos realmente el valor que tiene pony express para nuestros proyectos y el nivel de

optimización en el frontend con más JavaScript.

Primero vamos a crear el modelo de PonyExpress indicando donde escuchara el socket, luego crearemos un modelo de backbone extendido llamado ToDoTaskModel indicando la url en donde veremos el JSON, el siguiente punto sería crear una colección llamada ToDoTaskCollection con una colección de backbone extendida con el nombre ToDoList e indicando el modelo creado anteriormente ToDoTaskModel.

Todo esto lo haremos con el siguiente código:

```
window.ponyExpress = new PonyExpress({
  io : "http://localhost:3000"
});
window.ToDoTaskModel = Backbone.Model.extend({
  urlRoot : "/ToDoTask",
});
window.ToDoTaskCollection =  Backbone.Collection.extend({
  name  : "ToDoList",
  model : window.ToDoTaskModel
});
```

Ahora tenemos que crear una lista de colecciones en base a la colección de backbone creada anteriormente y procedemos a crear un Pony Express bind ya que esta función sera la que se disparara al momento de cada evento y generara un emit a socket io y este se encargara de mandar el evento a los navegadores, entonces en este se realizará un get a /ToDoTask que es el lugar donde recogemos el json y en caso de que salga bien el get le añadiremos el data de la solicitud, luego creamos con PonyExpres un BackbonePLug quien será el que informe de eventos a pony para luego disparar el bind e indicaremos la colección a mandar junto el evento que en este caso sería ToDoListCollection.

y todo esto lo hacemos con el siguiente código:

```
window.ToDoListCollection = new ToDoTaskCollection();
window.ponyExpress.bind('connect', function(){
  var xhrToDoTasks = $.get('/ToDoTask');
  xhrToDoTasks.done(function(data){
```

```
    window.ToDoListCollection.add(data);
    window.ToDoListPlug  = new PonyExpress.BackbonePlug({
      collection : window.ToDoListCollection
    });
  });
});
```

Ahora empezaremos con el código de jquery y manejamos todos los eventos de crear, actualizar y eliminar dependiendo de lo que haga el usuario.

Entonces empezaremos por como se verá cada tarea y la creación de la misma, primero como siempre debemos decir que cuando el documento esta listo empezaremos con un window.ToDoView para lo cual extenderemos el elemento vista de backbone en donde definiremos el template a renderizar y el evento send sobre #submit y la acción recíproca a esta.

```
$(document).ready(function(){
  window.ToDoView = Backbone.View.extend({
    tpl: _.template( $('#AgregarTask').html() ),
    events : {
      "click #submit" : "send"
    },
```

Luego de esto creamos una función inicializar que será la que se encargue de agregar los ToDos que ya estén creados según su status, entonces primero creamos una variable todoView e indicamos que es la misma función de donde se ejecuta, luego decimos que this.$el tendrá un elemento o sera el mismo y luego decimos que this.$el será igual al primer elemento del arreglo, luego pasamos a renderizar y lo agregamos al id Tasks.

```
    initialize : function(config){
      var todoView = this;
      this.$el = this.targetElement || this.$el;
      this.el = this.$el[0];
      this.render();
      this.$el.appendTo('#Tasks');
    }
```

Ahora procedemos a agregar la colección de tareas (Tareas ya creadas) a nuestra lista de completas o incompletas según su estado en donde creamos una nueva vista de tarea llamada ToDoListView e indicaremos que modelo es y el id que llevará cada div de cada tarea para luego hacer un prependTo a cada tarea; con esto cerramos la función initialize.

```
    window.ToDoListCollection.on('add', function(ToDoListModel){
        var ToDoListView = new ToDoTaskView({
          model: ToDoListModel,
          id: "tarea-" + ToDoListModel.id,
        });
        if(ToDoListModel.get('TaskStatus')){
          ToDoListView.$el.prependTo( todoView.$el.find('.TaskComplete')
);
        }else{
          ToDoListView.$el.prependTo(
todoView.$el.find('.TaskIncomplete') );
        }
      });
    },
```

Ahora crearemos la función que se ejecutará al hacer click en #submit en donde será una función en donde user tendrá el valor del id user y text tendrá el valor del id text, luego diremos que en caso de que usuario y texto están llenos continúe la función, en caso de que no, mate la función, entonces como en un mundo de maravilla y placer vamos a tener tanto a text como a user con valor continuaremos con la función, en donde creamos un modelo guardando a user y text (el status será por default false) y le hacemos un model.save que guardara el modelo para luego renderizarlo.

```
send : function(){
      var user = this.$el.find('#user').val(),
        text = this.$el.find('#text').val();
      if( !user || !text ){
        return;
      }
      var model = new ToDoTaskModel({user: user, text: text});
      model.save();
```

```
      this.$el.find('#text').val("");
    },
```

Y para acabar con la vista de ToDoView tendremos a render en donde indicamos que renderizamos en $el con el template definido desde el inicio (tpl).

```
    render : function(){
      this.$el.html( this.tpl({}) );
    }
  });
```

Ahora crearemos la vista extendida de backbone para cada tarea en caso de cambiar el status o removerla, entonces empezamos definiendo a tpl con id #ToDoList-template y los eventos de click en highlight el cual ejecutará la función highlightHandler y click en remove que ejecutará removeHandler con una función initialize en donde indicamos que ToDoListView será sí misma y renderizamos y luego emitimos un cambio en el modelo y genera un render de ToDoListView, luego indicamos una función destroyHandler que mostrará en consola lo que se destruye y ejecutará un .remove en ToDoListView, por último emitimos un evento destroy y retornamos a this.

```
  window.ToDoTaskView = Backbone.View.extend({
    tpl: _.template( $('#ToDoList-template').html() ),
    events : {
      "click .highlight" : "highlightHandler",
      "click .remove"    : "removeHandler"
    },
    initialize : function(config){
      var ToDoListView = this;
      this.render();
      this.model.on('change', function(){
        ToDoListView.render();
      });
      this.destroyHandler = function(){
        console.log('destroying', this.toJSON() );
        ToDoListView.remove();
      }
      this.model.on('destroy', this.destroyHandler);
```

```
      return this;
    },
```

Ahora pasaremos a crear highlightHandler en donde diremos que en caso de dar click, si tenía el estado en true, cambie a false y viceversa, luego guardaremos y renderizamos.

```
    highlightHandler : function(){
      if(this.model.get('TaskStatus')){
        this.model.set('TaskStatus', false);
      }else{
        this.model.set('TaskStatus', true);
      }
      this.model.save();
      this.render();
    },
```

Ahora pasaremos a crear la función en caso de destruirse, la cual sería que si se da click en remove le haga un model.off a destroy y ejecute destroyHandler (Función creada anteriormente), luego hacemos un model.destroy y eliminamos el modelo.

```
  removeHandler : function(){
      this.model.off('destroy', this.destroyHandler);
      this.model.destroy();
      this.remove();
    },
```

Y por último creamos a render en donde indicamos que en $el.html con el template y el modelo .toJSON (Json donde recoge información) y que cuando le de click en highlightHandler agregue las clases correspondientes al estado, y con esto terminamos con todo el JavaScript de la interacción en cuanto a Pony Express y el usuario.

```
    render : function(){
      this.$el.html( this.tpl( this.model.toJSON() ) );
      if(this.model.get('TaskStatus')){
        this.$el.addClass('highlighted');
        this.$el.appendTo(".TaskComplete");
```

```
      }else{
        this.$el.removeClass('highlighted');
        this.$el.appendTo(".TaskIncomplete");
      }
    }
  });
```

El ejemplo completo y funcionando lo pueden descargar en github[https://github.com/Mejorandola/PonyExpress] esta en el folder de examples > folder todo, junto con un chat hecho de la misma manera y una lista de tareas mas avanzada que la que vimos en este ejemplo.

Ya que da a creatividad del usuario como maneja su frontend, yo les dejo un ejemplo en donde navegamos entre las completas, incompleta y todas, y cuando das click en el lápiz se completan todas las tareas, solo es que dejes volar tu imaginación.

Espero te halla gustado, empieces a usarlo y si se te ocurren nueva ideas siempre estés aportando al repositorio. Bienvenido a Mejorando.la Open Source.