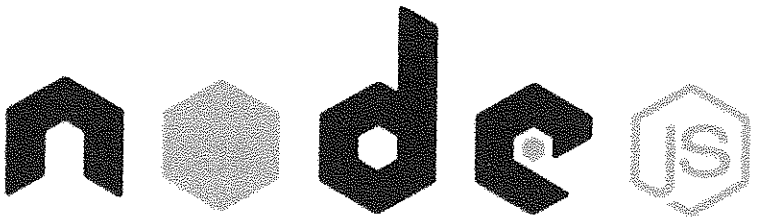


Curso de Node.js

Northware®
Software Development

Indice

Northware®
Software Development

Introducción a Node.js

¿Que es Node.js?

Node.js es un interprete javascript del lado del servidor esta basado en eventos y construido encima del motor V8 Javascript de Chrome, su objetivo es permitir que un programador escriba aplicaciones escalables y que manejen miles de conexiones simultáneamente en un solo servidor físico.

Node.js a diferencia de otros lenguajes de programación del lado del servidor como Java, PHP cambia la manera de conexión al servidor, ya que se ejecuta de manera asíncrona,.

Código Asíncrono

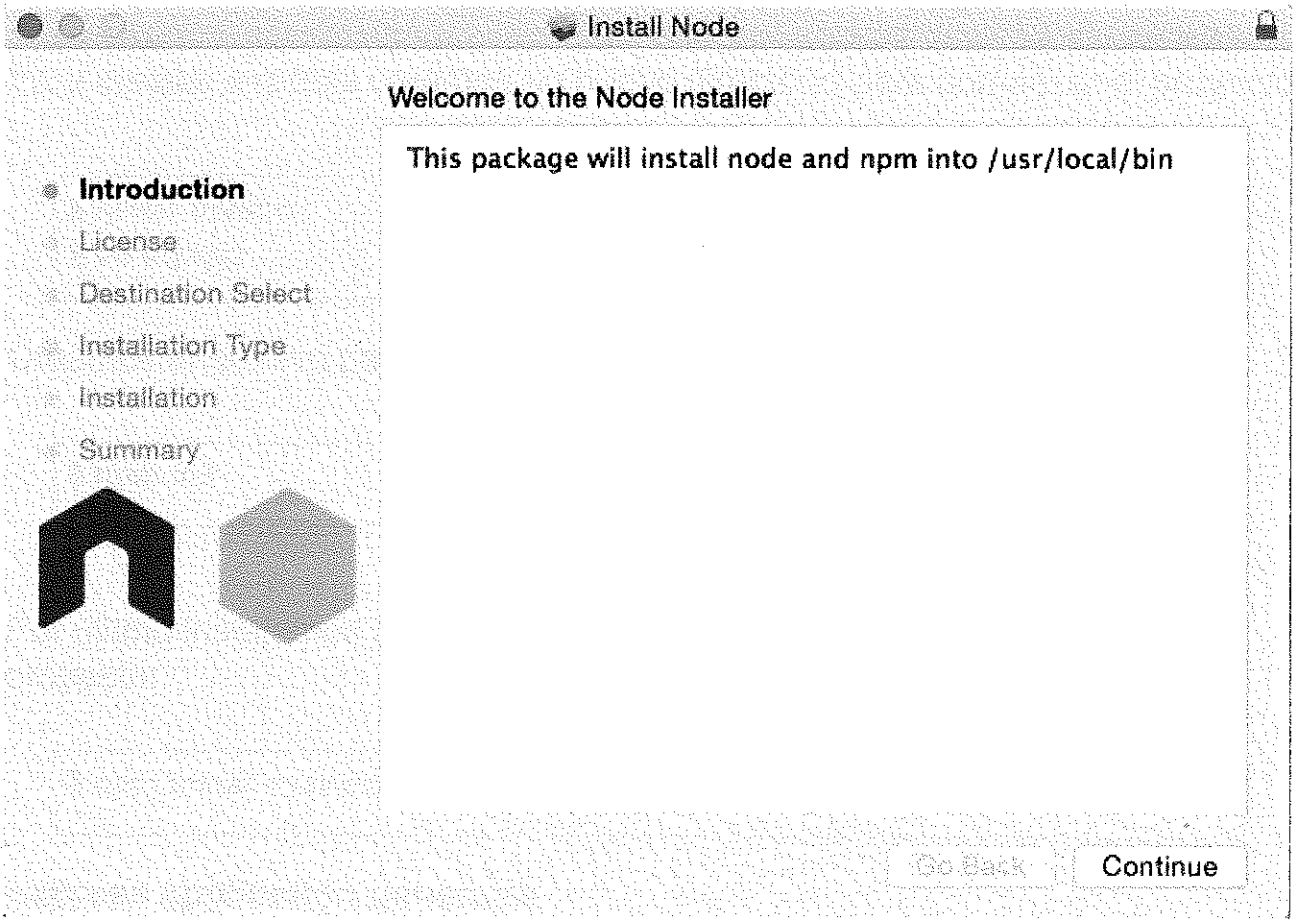
En el siguiente ejemplo realizado con Jquery hacemos una petición al servidor para obtener los usuarios, cuando se obtenga la respuesta se ejecutará la función callback que es el segundo parámetro.

```
72
73    $.post('/usuarios', function(usuarios){
74        console.log(usuarios);
75    })
76
```

Instalación de Node.js

Para realizar la instalación de Node.js es muy sencillo, tendremos que ira a la página oficial http://nodejs.org/ en la cual encontrarás un botón para instalarlo "INSTALL" el cual nos descargará un ejecutable dependiendo de nuestro sistema operativo, ya sea windows o mac como en este caso.

Posteriormente lo único que se tiene que hacer es seguir el asistente de instalación.



Una vez que tengamos instalado node ya podemos empezar a utilizarlo para ello necesitamos abrir nuestra terminal y ejecutamos el siguiente comando.

$ node —version

El cual nos mostrará la versión que tenemos instalada.

Instalación de Express Framework

Express

Express es un framework creado para hacer aplicaciones con node.js, a diferencia de otros framework express provee solo de configuraciones básicas, no te obliga a trabajar de cierta manera o con alguna estructura.

Para instalar express creamos un nuevo directorio para comenzar el proyecto

```
$ mkdir myapp
$ cd myapp
```

Creamos el archivo package.json con la configuración inicial.

```
$ npm init
```

Por ultimo instalamos express y lo agregamos a las dependencias del proyecto.

```
$ npm install express --save
```

Express application generator

Podemos utilizar el generador de aplicaciones el cual nos provee de una estructura básica y configuraciones básicas de la aplicación, como el motor de plantillas, el procesador css, etc. Para la instalación ejecutamos el siguiente comando:

```
$ npm install express-generator -g
```

Una vez hecho esto podemos utilizar el comando “express” para generar nuestros proyectos. Si ejecutamos el comando:

```
$ express -h
```

Nos mostrara una serie de opciones que podemos utilizar, como la creación del proyecto, consultar la versión que tenemos instalada.

```
Northwares-MBP:myapp oscargracia$ express -h

  Usage: express [options] [dir]

  Options:

    -h, --help          output usage information
    -V, --version       output the version number
    -e, --ejs           add ejs engine support (defaults to jade)
        --hbs           add handlebars engine support
    -H, --hogan         add hogan.js engine support
    -c, --css <engine>  add stylesheet <engine> support (less|stylus|compass) (defaults to plain css)
    -f, --force         force on non-empty directory
```

Ya con esto tenemos todas las herramientas necesarias para comenzar nuestro proyecto.

Node.js en práctica

Mi Primera aplicación

Para crear nuestra primera aplicación debemos de ir al directorio donde crearemos nuestra aplicación y ejecutamos el siguiente comando en la terminal para generar nuestra primera aplicación.

```
$ express -e
```

```
Northwares-MBP:proyecto1 oscargracia$ express -e

   create : .
   create : ./package.json
   create : ./app.js
   create : ./public
   create : ./public/javascripts
   create : ./public/images
   create : ./public/stylesheets
   create : ./public/stylesheets/style.css
   create : ./routes
   create : ./routes/index.js
   create : ./routes/users.js
   create : ./views
   create : ./views/index.ejs
   create : ./views/error.ejs
   create : ./bin
   create : ./bin/www

   install dependencies:
     $ cd . && npm install

   run the app:
```

En la terminal nos muestra como se han creado una serie de archivos y directorios, una vez generado tenemos que instalar las dependencias.

```
Northwares-MBP:proyecto1 oscargracia$ cd . && npm install
```

Ahora si vamos al directorio de nuestra aplicación vamos a encontrar una serie de directorios y archivos dentro.

app.js: Es nuestro archivo principal es nuestro servidor donde inicia nuestra aplicación.

node_modules: Este directorio contiene todos los modelos y dependencias de nuestro proyecto.

package.json contiene un registro de las dependencias y algunas configuraciones de nuestra aplicación.

public: Contiene todos los archivos estáticos de nuestra aplicación como javascript, css así como imágenes.

routes: En este directorio se encuentran los archivos javascript relacionados a las rutas de nuestra aplicación.

Una vez que entendemos los archivos y directorios que componen nuestra aplicación lo que sigue es ejecutarla, lo cual se hace de la siguiente manera:

$ DEBUG=proyecto1 ./bin/www

Si recibimos el siguiente mensaje en la terminal quiere decir que todo esta funcionando correctamente.

proyecto1 **Express server listening on port 3000 +0ms**

Abrimos nuestro navegador y entramos a http://localhost:3000/ y vemos como nuestra aplicación esta funcionando correctamente.

Manejo de rutas y peticiones HTTP

Las rutas nos permiten direccionar peticiones a los controladores correctos, empecemos por un ejemplo sencillo, vayamos al directorio routes y abrimos el archivo índex.js este archivo contiene la ruta principal de nuestra aplicación.

```
1  var express = require('express');
2  var router = express.Router();
3
4  /* GET home page. */
5  router.get('/', function(req, res) {
6    res.render('index', { title: 'Express' });
7  })
8
9  module.exports = router;
```

Ahora creamos una ruta nueva la cual tiene que estar definida antes de la linea :

module.exports = router;

ya que esta linea exporta las rutas para ser ejecutadas por el servidor node.

```
router.get('/acerca', function(req, res){
    res.send("<h1>Mi nombre es Oscar Gracia</h2>");
});
```

Si corremos nuestra app en la consola node app.js y vamos a DEBUG=proyecto1 ./bin/www en nuestro explorador de preferencia, debemos ver el mensaje "Mi nombre es Oscar Gracia" lo cual lo veremos como título de la página, ya que pusimos las etiquetas <h1></h1>.

Recibiendo parámetros

Si queremos recibir algún parámetro en una ruta debemos especificar en el string el nombre del parámetro con ":"

```
router.get('/acerca/:nombre', function(req, res){
    var nombre = req.params.nombre;
    res.send("<h1>Mi nombre es "+nombre+"</h2>");
});
```

Como vemos en el script anterior en la ruta definimos ":nombre" el cual es nuestro parámetro , si corremos la aplicación y vamos a http://localhost:3000/acerca/Oscar veremos que toma el parámetro que enviamos por url y no concatena en la cadena de texto que estamos mostrando.

Para enviar datos por post a una ruta especifica es de la misma manera solo sustituimos el ***router.get()*** por el ***router.post(ruta, callback)***, esto indica que nuestra ruta solo recibirá datos a travez de peticiones post.

Una vez que enviemos los datos a travez de alguna formulario o alguna aplicación que nos permita enviar peticiones por post, lo que tenemos que hacer para poder parsear los datos es llamar a la propiedad del cuerpo de la peticion a travez del request.

var datos = req.body.username;

A continuación se muestra la creación de una ruta que recibe datos a travez de post, en la función callback mostramos únicamente una cadena de texto con un mensaje simulando que se registro un usuario, mas adelante veremos la lógica que puede contener nuestras rutas.

```
14  router.post('/registar', function(){
15      var datos = req.body.nombre;
16      res.send('El usuario '+nombre+' se ha registrado correctamente');
17  });
```

Manejo de vistas y vistas parciales

Cuando creamos nuestro proyecto y ejecutamos el comando "express" indicamos que usaremos el motor **"EJS"** de plantillas. Este motor es similar a HTML utilizando etiquetas para generar la estructura, a diferencia del motor **"JADE"** que viene por defecto de no haber puesto el comando "-e" cuando generamos la aplicación.

Una vez que tenemos definido nuestro motor de plantillas en este caso EJS, en el directorio de views creamos todas nuestras vistas HTML que vayamos a necesitar, en este caso usaremos las rutas que ya habíamos creado, las cuales se encargaran de llamar a nuestras vistas.

Creamos nuestro archivo ***sobremi.ejs*** en el directorio views el cual será nuestra vista que mostraremos en pantalla una vez que accedamos a la ruta.

```
1  <html>
2      <head>
3          <title>Mi Primera aplicacion en Node.js</title>
4      </head>
5      <body>
6          <h1>Informacion Personal</h1>
7
8          <p><b>Mi Nombre es:</b> Oscar Gracia</p>
9          <p><b>Trabajo en:</b> Northware</p>
10
11     </body>
12 </html>
```

Una vez creada nuestra vista html lo que tenemos que hacer es modificar la ruta que habíamos construido previamente, la cual únicamente cambiaremos el método ***send()*** por el método ***render()*** el cual se encarga de renderizar nuestra vista html.

```
router.get('/acerca', function(req, res){
    res.render('sobremi');
})
```

Pasar parámetros a una vista

En la mayoría de los casos cuando llamamos a una vista HTML dinámicamente es por que necesitamos enviar datos del servidor al cliente, lo cual en node.js se hace de la siguiente manera:

```
 9  router.get('/acerca', function(req, res){
10      var persona = {
11              nombre:'Oscar A. Gracia', trabajo: 'Northware'
12          };
13
14      res.render('sobremi', persona);
15  })
```

Si nos damos cuenta lo único que tenemos que hacer es enviar un segundo parámetro al método ***render()***, el cual consiste en un objeto.

Para poder hacer uso de los datos en la vista, solamente tenemos que utilizar las expresiones <%= %>

```
 1  <html>
 2      <head>
 3          <title>Mi Primera aplicacion en Node.js</title>
 4      </head>
 5      <body>
 6          <h1>Informacion Personal</h1>
 7
 8          <p><b>Mi Nombre es:</b> <%= nombre %></p>
 9          <p><b>Trabajo en:</b> <%= trabajo %></p>
10
11      </body>
12  </html>
```

Vistas parciales

El uso de vistas parciales es esencial ya que permite reutilizar código html en bloques sin necesidad de volver a tener que escribir la estructura.

Para usar las vistas parciales solamente tenemos que crear nuestros archivos .ejs que contendrán la estructura html que vamos a manejar.

En este caso crearé un directorio que se llame "parcial" en el cual pondremos las 2 vistas parciales que crearemos **header.js** y **footer.ejs**

header.ejs

```
<header>
    Esta es la cabezera de la aplicacion.
</header>
```

footer.ejs

```
<footer>
    Este es el pie de pagina
</footer>
```

Una vez que tengamos nuestras vistas parciales solo tendremos que incluirlas en nuestra página principal lo cual haremos con la expresión ***include*** seguido de la ruta donde se encuentre.

```
<html>
    <head>
        <title>Mi Primera aplicacion en Node.js</title>
    </head>
    <body>

        <% include parcial/header.ejs %>

        <h1>Informacion Personal</h1>

        <p><b>Mi Nombre es:</b> <%= nombre %></p>
        <p><b>Trabajo en:</b> <%= trabajo %></p>

        <% include parcial/footer.ejs %>

    </body>
</html>
```

Si ejecutamos nuestra aplicación podremos ver que el resultado es el siguiente:

Esta es la cabezera de la aplicacion.

Informacion Personal

Mi Nombre es: Oscar A. Gracia

Trabajo en: Northware

Este es el pie de pagina

Podemos ver que muestra la información de los bloques correctamente.

¿Que es mongoDB?

MongoDB es una base de datos NoSQL lo que implica no ser una base de datos relacional si no una base de datos orientada a documentos de esquema libre. Esto indica que cada registro o "documento" en este caso puede contener un esquema de datos diferente, con columnas y atributos diferentes que no tienen por que repetirse de un registro a otro.

En mongoDB cada registro se denomina documento el cual contiene el formato JSON y un conjunto de documentos representan una colección, lo cual comparando con una base de datos relacional vendrían siendo las tablas.

Instalación de MongoDB

Para la instalación podemos entrar al sitio oficial descargar los paquetes y seguir los pasos dependiendo de nuestro sistema operativo.

http://docs.mongodb.org/manual/tutorial/install-mongodb-on-os-x/?_ga=1.106687235.1487510377.1416854091

Primeros pasos con Mongo

Una vez que tengamos instalado podemos abrir la terminal y empezar a usarlo. Es importante aclarar que para el uso de esta base de datos necesitaremos 2 terminales abiertas, ya que una ejecuta nuestro servidor y la otra nos permitirá manejar las consultas.

Para correr nuestro servidor ejecutamos el comando :

$ ***mongod***

```
Northwares-MacBook-Pro:~ oscargracia$ mongod
mongod --help for help and startup options
2014-11-24T15:01:14.539-0600 [initandlisten] MongoDB starting : pid=23999 port=27017 dbpath=/data/db 64-bit host=No
rthwares-MacBook-Pro.local
2014-11-24T15:01:14.540-0600 [initandlisten] db version v2.6.5
2014-11-24T15:01:14.540-0600 [initandlisten] git version: nogitversion
2014-11-24T15:01:14.540-0600 [initandlisten] build info: Darwin miniyosemite.local 14.0.0 Darwin Kernel Version 14.
0.0: Fri Sep 19 00:26:44 PDT 2014; root:xnu-2782.1.97~2/RELEASE_X86_64 x86_64 BOOST_LIB_VERSION=1_49
2014-11-24T15:01:14.540-0600 [initandlisten] allocator: tcmalloc
2014-11-24T15:01:14.540-0600 [initandlisten] options: {}
2014-11-24T15:01:14.601-0600 [initandlisten] exception in initAndListen: 10310 Unable to lock file: /data/db/mongod
.lock. Is a mongod instance already running?, terminating
2014-11-24T15:01:14.601-0600 [initandlisten] dbexit:
2014-11-24T15:01:14.601-0600 [initandlisten] shutdown: going to close listening sockets...
2014-11-24T15:01:14.601-0600 [initandlisten] shutdown: going to flush diaglog...
2014-11-24T15:01:14.601-0600 [initandlisten] shutdown: going to close sockets...
2014-11-24T15:01:14.601-0600 [initandlisten] shutdown: waiting for fs preallocator...
2014-11-24T15:01:14.601-0600 [initandlisten] shutdown: lock for final commit...
2014-11-24T15:01:14.601-0600 [initandlisten] shutdown: final commit...
2014-11-24T15:01:14.601-0600 [initandlisten] shutdown: closing all files...
2014-11-24T15:01:14.601-0600 [initandlisten] closeAllFiles() finished
2014-11-24T15:01:14.601-0600 [initandlisten] dbexit: really exiting now
```

Esta respuesta en la terminal nos indica que nuestro servidor de base de datos esta corriendo correctamente.

Una vez que el servidor de mongo se encuentre corriendo abrimos una segunda terminal simultáneamente y ejecutamos el comando **$ mongo**, nos mostrara la versión que tenemos instalada y el esquema al que se conecta por default.

```
Northwares-MacBook-Pro:~ oscargracia$ mongo
MongoDB shell version: 2.6.5
connecting to: test
>
```

Primero necesitamos crear un esquema en nuestra base de datos lo cual haremos con el comando ***use***

$ use prueba

El comando indica sobre que esquema estaremos trabajando en caso de no existir crea uno nuevo con el mismo nombre.

Para poder listar las colecciones (tablas) que contiene nuestro esquema utilizamos el siguiente comando:
> show collections

Al ejecutarlo nos damos cuenta que no muestra nada ya que no hemos creado una colección todavía.

> db.usuarios.insert({nombre: "Oscar Gracia", edad: 25})

La linea anterior nos permite registrar documentos en la colección. ¿Pero que colección? Bien, al igual que el esquema mongo nos crea automáticamente la colección dependiendo del nombre que le hayamos puesto al registro en este caso fue ***"usuarios"***

Si consultamos nuevamente las colecciones que tenemos nos mostrara una colección llamada **"usuarios"** ya que se creo al haber registrado el primer documento.

Podemos consultar los documentos de la colección utilizando la siguiente linea:

> db.usuarios.find();

```
> db.usuarios.find()
{ "_id" : ObjectId("5473a00b7dd27cc0c783d21a"), "nombre" : "Oscar Gracia", "edad" : 25 }
```

Node.js y Moongose

Para poder utilizar mongodb en Node.js existe una gran cantidad de librerías, pero una de las mas estables es Mongoose. Esta es un ORM que nos permitirá asociar los registros de la base de datos a objetos dentro de nuestra aplicación.

$ npm install --save mongoose

Instalamos la librería a la aplicación y agregamos las dependencias.

Creando los modelos

En este caso crearemos un directorio nuevo llamado ***models***
$ mkdir models

Una vez que tengamos el directorio creamos un archivo usuarios.js el cual contendrá el modelo de usuarios de nuestra aplicación.

```
1  module.exports = function(mongoose){
2
3      var Schema = mongoose.Schema;
4
5      var UsuarioSchema = new Schema({
6        nombre       : String,
7        apellido     : String,
8        correo       : String
9      });
10
11     return mongoose.model('usuarios',UsuarioSchema);
12   }
```

En este caso definimos que nuestro modelo de usuarios solo contendrá 3 propiedades, nombre, apellido y correo.

Ahora dentro del mismo directorio de **models** creamos un archivo llamado **index,js** este archivo contendrá nuestra conexión a mongodb y nuestra configuración global.

```
1  if( global.hasOwnProperty('db')){
2
3    var mongoose = require('mongoose');
4    var dbName = 'curso'
5
6    mongoose.connect('mongodb://localhost/' dbName);
7
8    global.db = {
9      mongoose: mongoose,
10     Usuario: require('./usuarios')(mongoose)
11   };
12 }
13
14 module.exports = global.db;
```

Northware®
Software Development

El objeto global, se encuentra disponible globalmente, como su nombre lo dice. A este objeto se le pueden agregar propiedades como lo hemos hecho en este ejemplo, le agregamos una llamada db, ésta se encargará de llevar una instancia del objeto Mongoose y los modelos de nuestra app.
Para esto funcione tenemos que correr este código cuando inicie nuestra app. Abrimos el archivo app.js y en cualquier parte agregamos ***require('./models');.***

CRUD con MongoDB en Node.js

Registro de usuarios

Ahora que contamos con el modelo de usuarios y la conexión a mongo, nos toca registrar usuarios a través de un formulario HTML.

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>

  </head>
  <body>
    <h1><%= title %></h1>

    <form action="guardar" method="post">
        <div>
            Nombre: <br>
            <input type="text" name="nombre">
        </div>
        <div>
            Apellido: <br>
            <input type="text" name="apellido">
        </div>
        <div>
            Correo Electronico: <br>
            <input type="text" name="correo">
        </div>

    <input type="submit" value="Guardar">
    </form>

  </body>
</html>

```

Con lo que vimos anteriormente sabemos que tenemos que crear el formulario html dentro de nuestro directorio de views, en este caso se llamará ***registro.ejs.***

En el ***action*** del formulario indicamos que la ruta a la cual enviará los datos se llama ***/guardar***

Después de haber construido nuestro formulario tenemos que generar 2 rutas en nuestra aplicación una será la encargada de renderizar la vista y la otra será la encargada de recibir la petición a POST y generar el registro a mongodb.

Ruta encargada de mostrar la vista

```
8
9   router.get('/add', function(req, res){
10      console.log('usuarios');
11      res.render('registro', {title: 'Registro de Usuarios'});
12  });
```

Ruta encargada de registrar los datos en mongo.

```
14  router.post('/guardar', function(req, res){
15      var datos = req.body;
16
17      var nuevoUsuario = new db.Usuario({
18          nombre: datos.nombre,
19          apellido: datos.apellido,
20          correo: datos.correo
21      });
22
23      nuevoUsuario.save(function(error, usuario) {
24          // Se valida que no haya ocurrido ni un error, de haber ocurrido lo mostrara en pantalla
25          if (error) response.json(error);
26
27          // Una vez que el usuario se haya registrado, se redireccionara
28          response.redirect('/usuario');
29      });
30
31  });
```

Como vemos en el script anterior creamos un objeto llamado nuevoUsuario el cual cada una de sus propiedades ya esta definida en el modelo de usuario, una vez que tengamos cada uno de los datos en sus respectivos propiedades solo es cuestión de llamar al método ***save*** que nos proporciona mongoose y en la función callback podemos ejecutar cualquier proceso posterior así como el objeto recién agregado.

Si abrimos nuevamente la terminal en la cual estamos ejecutando mongodb y realizamos una consulta, podemos ver que los usuarios que registremos se están almacenando.

```
> db.usuarios.find()
{ "_id" : ObjectId("5473a00b7dd27cc0c783d21a"), "nombre" : "Oscar Gracia", "edad" : 25 }
{ "_id" : ObjectId("54740c0a0100fd3363c7f838"), "nombre" : "Alberto", "apellido" : "Espinisa", "correo" : "oscar.gracia@gmail.com", "__v" : 0
}
{ "_id" : ObjectId("547656bc514fabd574a595bb"), "nombre" : "Roberto", "apellido" : "Lara", "correo" : "rlara@gmail.com", "__v" : 0 }
{ "_id" : ObjectId("54765772514fabd574a595bc"), "nombre" : "Victor", "apellido" : "Martinez", "correo" : "vip@gmail.com", "__v" : 0 }
{ "_id" : ObjectId("5476b2212b658d147cbcdb40"), "nombre" : "Claudia", "apellido" : "Gracia", "correo" : "oscar.gracia@gmail.com", "__v" : 0 }
```

Listado de usuarios

Para consultar los usuarios tenemos que crear un nuevo HTML que se encarga de listar cada uno de los registros.

```
1
2   <% include parcial/header.ejs %>
3
4   <a href="add">Nuevo Usuario</a>
5
6   <div>
7       <table>
8           <tr>
9               <th>Nombre</th>
10              <th>Apellido</th>
11          </tr>
12          <% usuarios.forEach(function(usuario){ %>
13              <tr>
14                  <td><%= usuario.nombre %></td>
15                  <td><%= usuario.apellido %></td>
16                  <td><%= usuario.correo %></td>
17                  <td><a href="edit/<%= usuario._id %>">Editar</a></td>
18                  <td><a href="delete/<%= usuario._id %>">Eliminar</a></td>
19              </tr>
20          <% }) %>
21      </table>
22  </div>
23
24  <% include parcial/footer.ejs %>
```

Podemos notar que el HTML hace referencia a las vistas parciales creadas anteriormente, así como un enlace a la ruta que creamos anteriormente para mostrar el formulario de registro de usuarios.

Después de eso creamos una tabla HTML para listar ahí los usuarios, EJS nos permite utilizar un ***forEach*** junto con las etiquetas de HTML el cual nos ayudara a recorrer el objeto que le pasemos a nuestra vista, en este caso

uno de usuarios. Ahora hay que construir la ruta que se encargara de consultar y enviar los usuarios a la vista.

```
4
5   router.get('/list', function(req, res) {
6       db.Usuario.find().exec(function(err, usuarios){
7           if (err) return res.json(error);
8
9           res.render('list', {usuarios: usuarios});
10      });
11  });
```

Lo que hacemos primero es definir la ruta, en este caso será cuando llames a **list** desde la URL, después de eso ejecutamos el callback que se encargara de hacer la peticiona a mongo, en el ejemplo anterior de registro utilizamos el método ***save*** el cual nos permitió guardar los datos, en este caso utilizaremos el método ***find()*** el cual se encarga de consultar todos los registros de la colección ***usuarios,*** una vez que se haya ejecutado, el callback se encargara de llamar a la vista ***list*** la cual es la que contiene la tabla HTML donde se listaran los usuarios. Como segundo parámetro del método ***render()*** enviamos valores a la vista como lo vimos anteriormente, solo que en esta coacción enviaremos toda la consulta que nos regreso mongodb.

Actualización de usuarios

En el listado de usuarios tenemos 2 enlaces por registro, a los cuales les pasamos el **id** que mongodb se encargo de asignarles, uno de esos enlaces nos enviara a otra pagina la cual será un formulario similar al de registro que nos permitirá actualizar los datos, con el otro podemos eliminar un usuario.

Para la funcionalidad tenemos que hacer lo mismo que anteriormente, crearemos 3 rutas mas. Una ruta nos enviara al formulario, otra ruta se encargara de generar el proceso de actualización y la tercera de eliminar el usuario.

Ruta para mostrar el formulario HTML, que es similar al que usamos para mostrar el formulario de registro. La única diferencia es que en este caso hacemos una consulta a mongo especificado que queremos obtener un documento con un id que recibimos como parámetro desde el enlace.

```
router.get('/editar', function(req, res){
    var idUsuario = req.params.id;
        db.Usuario.findById(idUsuario, function(err, usuario){
            if(err) res.json(err);
            res.render('form_actualizacion', {title: 'Editar Usuario', usuario: usuario});
        });
});
```

El formulario de edición contiene pequeñas diferencias comparado con el de registro.

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>

  </head>
  <body>
    <h1><%= title %></h1>

    <form action="/usuarios/editar/'"+<%= usuario._id %>+"'?_method=PUT" method="post">

      <div>
      Nombre: <br>
        <input type="text" name="nombre" value="<%= usuario.nombre %>">
      </div>
      <div>
        Apellido: <br>
        <input type="text" name="apellido" value="<%= usuario.apellido %>">
      </div>
      <div>
        Correo Electronico: <br>
        <input type="text" name="correo" value="<%= usuario.correo%>">
      </div>

    <button type="submit">Guardar</button>
    </form>

  </body>
</html>
```

En el **action** del formulario al final de la ruta especificada enviamos un parámetro adicional **_method=PUT** este parámetro le indicara a la alicantino que el submit que se esta realizando no es un método **POST** si no **PUT** lo que Node.js interpretara como una edición y no un nuevo registro.

```
router.put('/editar/:id', function(req, res){
    var usuario     = req.body,
        idUsuario   = req.params.id;

    db.Usuario.findByIdAndUpdate(idUsuario, usuario, function (error, usuario) {
        if (error) return response.json(error);

        res.redirect('/usuarios/list');
    });
});
```

Una vez que ejecutemos el formulario la ruta identificara que es un método PUT lo cual nos permitirá recibir todo el objeto de usuario el cual pasaremos como parámetro al método **findByIdAndUpdate()** el cual realizara la actualización del registro.

Por ultimo especificamos la ruta que se encargara de eliminar el registro, que es similar al de actualización solo que en este caso solo pasamos como parámetro el id del usuario y llamamos al método **findByIdAndRemove()** que nos provee mongoose.

```
router.get('/delete/:id', function(req, res){
    db.Usuario.findByIdAndRemove(req.params.id, function(error, usuarios){
        if (error) return response.json(error);

        res.redirect('/usuarios/list');
    });
});
```

Aplicaciones en tiempo Real

Que es socket y socket.io

Socket es una nueva tecnología incluida en HTML5 que nos permite crear sesiones de comunicación bidireccional entre un servidor y un cliente, lo cual establece una conexión que permanece abierta que no necesita que un cliente este preguntando al servidor si ha habido un cambio.

El problema con los websockets es que no soportan determinadas características necesarias, por lo cual nace la librería para node.js llamada Socket.IO, una librería javascript que nos permite manejar eventos en tiempo real mediante una conexión TCP y nos provee de todas las funcionalidades necesarias, así como los problemas de compatibilidad con todos los navegadores.

Node.js y Socket.io

Utilizar Socket.IO en node.js es muy sencillo solo es cuestión de instalar un modulo de node con el comando npm, así como se realizo con express.

$ npm install socket.io

En el servidor lo primero que tenemos que hacer es indicarle que socket.io va a escuchar en el mismo puerto que node.js y que emitirá un mensaje, que el cliente recibirá.

```
var server = app.listen(app.get('port'), function() {
  debug('Express server listening on port ' + server.address().port);
});

var io = require('socket.io')(server);

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

    socket.broadcast.emit('mensaje', {
        text: 'Se ha conectado un nuevo usuario'
    });
});
```

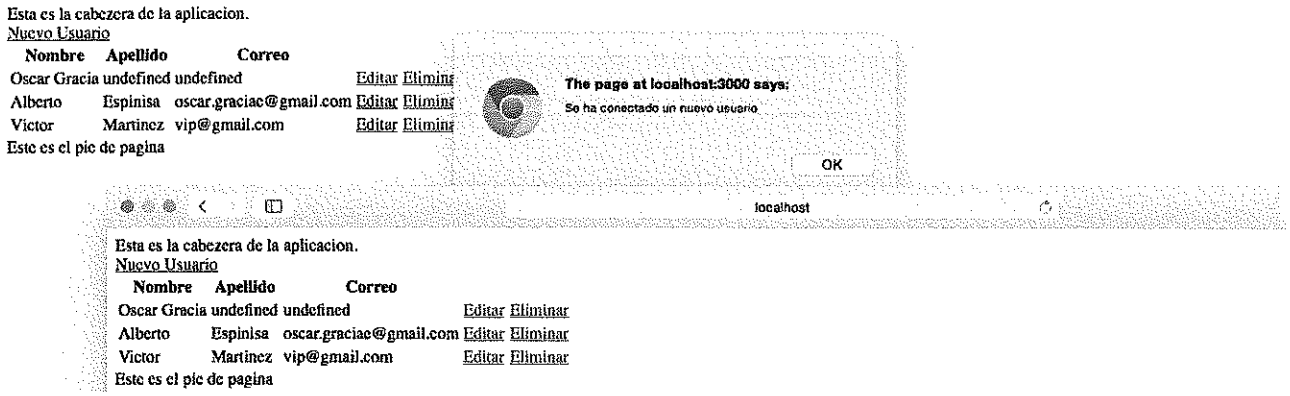
En este ejemplo mostramos un pequeño script que genera un sistema de notificaciones de usuarios conectados.

En el cliente lo único que tenemos que hacer es importar socket.io e indicarle a que servidor estamos escuchando. En este caso es u servidor local por lo cual solo tenemos que poner localhost.

```
<script src="/socket.io/socket.io.js"></script>
<script>
  var socket = io.connect('http://localhost');
  socket.on('mensaje', function (data) {
    console.log(data);
    alert(data.text);
  });
</script>


<header>
    Esta es la cabezera de la aplicacion.
</header>
```

Al realizar la prueba podemos ver que si abrimos la aplicación en un sitio no pasa nada, pero al abrir la aplicación en otro navegador nos aparece un mensaje que nos indica que otro usuario se a conectado a la aplicación todo en tiempo real.



En el ejemplo anterior ingrese a la aplicación en el navegador safari y por consecuencia en el navegador chrome me arrojo un mensaje indicando que un usuario se conecto.