UNIVERSIDAD DE LA PUNTA

GUIA DE DESARROLLO WEB CON NUEVAS TECNOLOGÍAS

Prof. Saez, Juan José.

Autor: Cordoba, Santiago Nahuel

San Luis, 2013

RESUMEN

Soy Santiago Nahuel Córdoba, nací un 24 de Abril de 1992 y soy un estudiante de la Tecnicatura en Desarrollo de Software en La Universidad de La Punta. Mi historia con Node.js comenzó un día normal de clases cuando mi profesor de Programación Web II (Marcelo Alaniz) nos comentó sobre esta tecnología y cuáles eran sus ventajas respecto a otros lenguajes, y eso me quedo latente en mi mente hasta la llegada del verano que pude hacer tiempo para aprender y termino siendo un verano de mucho JavaScript que finalmente me influencio totalmente cuando tuve que decidir un tema para mi trabajo final.

El objetivo de este trabajo es ayudarte a empezar con el desarrollo de aplicaciones web con Node.js, vas a aprender casi todo lo que necesites para realizar grandes proyectos con tecnologías novedosas como Node.js y MongoDB. Comenzando desde las primeras líneas de Node.js hasta la publicación de un proyecto.

ANTES DE COMENZAR

Previamente se explicaran algunos conceptos para comprender los términos que van a ser utilizados durante todo el desarrollo de la guía:

Recomiendo siempre seguir este texto junto con los archivos de código de los ejemplos.

Front-end: es la parte del software que interactúa con el usuario, es decir, del lado del cliente (HTML, CSS, HTML5, CSS3, Javascript, jQuery)

Back-end: es la parte del software que se encarga de interpretar datos que provienen del cliente o de bases de datos y genera plantillas para ser entregadas al cliente, es decir, del lado del servidor (Node.js)

I/O: input/output, entrada/salida.

Path: ruta de un archivo en disco.

Foo: es un término genérico para cualquier entidad informática que se ignora el nombre o no es necesario expresarlo. Para entenderlo se lo suele comparar con la “X” en matemática que puede significar cualquier número.

Framework: En el desarrollo de software, un framework o infraestructura digital, es una estructura conceptual y tecnológica de soporte definido, normalmente con artefactos o módulos de software concretos, con base a la cual otro proyecto de software puede ser más fácilmente organizado y desarrollado. Típicamente, puede incluir soporte de programas, bibliotecas, y un lenguaje interpretado, entre otras herramientas, para así ayudar a desarrollar y unir los diferentes componentes de un proyecto.(Wikipedia)

Arquitectura cliente servidor: es la clásica arquitectura de las páginas y los servicios web. Un cliente realiza una petición y el servidor procesa esa petición, le da una respuesta y finaliza la conexión (**figura n1**).

Se dará como ya sabido: HTML, CSS, jQuery, JavaScript, POO, Bases de datos relacionales, puertos de direcciones ip.

En ninguno de los casos se explicara cómo se instalan las herramientas necesarias.

JAVASCRIPT

Javascript es un lenguaje de programación interpretado orientado a objetos y débilmente tipado y dinámico. Inicialmente fue creado para operar únicamente del lado del cliente pero actualmente con las nuevas tecnologías puede ser utilizado también del lado del servidor.

Historia de JavaScript:

JavaScript fue desarrollado originalmente por Brendan Eich de Netscape con el nombre de Mocha, el cual fue renombrado posteriormente a LiveScript, para finalmente quedar como JavaScript. El cambio de nombre coincidió aproximadamente con el momento en que Netscape agregó soporte para la tecnología Java en su navegador web Netscape Navigator en la versión 2.002 en diciembre de 1995. La denominación produjo confusión, dando la impresión de que el lenguaje es una prolongación de Java, y se ha caracterizado por muchos como una estrategia de mercadotecnia de Netscape para obtener prestigio e innovar en lo que eran los nuevos lenguajes de programación web. (WikipediA)

Para poder desarrollar esta guía hay que tener conocimientos en algún lenguaje de programación y adaptarse a JavaScript no será costoso. Las únicas que pueden resultar raras de este lenguaje son: las funciones como parámetros y como se crean clases en JavaScript

Primero explicare las funciones como parámetros, En JavaScript las funciones pueden ser pasadas como parámetros, explicare esto con un ejemplo simple.

```
function suma(a,b){
    return a+b;
}

function resta(a,b){
    return a-b;
}

function operar(funcionOperadora,a,b){
    var resultado = funcionOperadora(a,b);
    console.log("Resultado: " + resultado);
}

operar(suma,2,5);
operar(resta,4,1);
```

En el ejemplo anterior se tienen dos funciones suma, resta y una tercera llamada operar que recibe una función (funcionOperadora) y dos números (a,b). En el primer llamado a operar se pasa la función suma, el número 2 y el número 5. En el segundo llamado a operar se pasa la función resta, el número 4 y el número 1. La salida a consola será:

Resultado: 7

Resultado: 3

Como se ve en ese caso se pasó la función como parámetro, algo importante para evitar errores es que solo se pone el nombre de la función sin los paréntesis y hay que asegurarse la cantidad de parámetros que recibe la función sea la correcta, porque si faltan parámetros puede hacer fallar la función y se caerá nuestro programa.

Para crear clases en JavaScript es medio complejo y realmente es una de las peores cosas que tiene JavaScript comparado con otros lenguajes. Pondré un ejemplo simple para entender cómo se realiza una clase:

```
var Animal = function(nombre){
    this.nombre=nombre; //atributo publico
    var vecesQueHaCorrido=1;  //variable privada

    this.correr=function(){  //funcion publica correr
        console.log(nombre+" corriendo por: "+vecesQueHaCorrido++ +" vez");
    }
    this.getVecesQueHaCorrido=function(){
        return vecesQueHaCorrido;
    }
    function privada(){
        //Función privada
    }
}
```

Uso de la clase:

```
var gatito=new Animal("Kat",2.5);
gatito.correr();
gatito.correr();
gatito.correr();
console.log(gatito.getVecesQueHaCorrido());
```

Salida por consola:

Kat corriendo por: 1 vez

Kat corriendo por: 2 vez

Kat corriendo por: 3 vez

4

JSON

JSON es la sigla de JavaScript Object Notation, es un formato rápido y simple de intercambio de datos, tiene ventajas con respecto a otros formatos de intercambio porque elimina bastante la redundancia de datos y en nuestro caso será muy útil para hacer intercambio de información entre el cliente y el servidor con Nodejs.

Ejemplo:

{

"nombre" : "juan"

```
  ,"apellido" : "perez"
  ,"edad" : "19"
}
```

En este caso vemos los datos de una persona reflejados en JSON, cualquier objeto JavaScript puede ser representado de esta manera.

Un documento JSON comienza con { (llave izquierda) y termina con } (llave derecha). Un elemento en JSON está compuesto por el par clave valor, la clave es un string y el valor puede ser: string, number, object, function, array, true, false y null. El par clave valor están separados por : (dos puntos), si un objeto tiene más de un elemento se separan con , (coma). Para representar un array como valor, se empieza con [(corchete izquierdo) y se finaliza con] (corchete derecho) y los valores se separan con , (coma). Si se escribe JSON es muy recomendable respetar la indentación para evitar confusiones.

Ejemplo complejo:

```
var persona={
   "nombres":["juan","gonzalo"],
   "apellidos":["perez","gomez"],
   "edad":19,
   "activo":true,
   "materiasaprobadas":[
      {"nombre":"Matematica","nota":10},
      {"nombre":"Lengua","nota":9},
      {"nombre":"Ingles","nota":7}
   ]
}
```

En el ejemplo complejo he asignado el documento JSON a una variable persona para demostrar cómo se accede a los datos desde JavaScript.

```
document.write(persona.nombres+"<br>");
```

```
for (i in persona.materiasaprobadas){
   materia = persona.materiasaprobadas[i];
   document.write(materia.nombre+" "+materia.nota+"<br>");
}
```

En ese ejemplo se demuestra cómo acceder a algunos datos del objeto persona. En la primera línea se escriben los nombres de la persona, luego se declara un for para recorrer las materias aprobadas y se escribe los datos de cada materia.

Una aclaración importante, cualquier objeto JavaScript es un documento JSON.

PROBLEMAS DE BASES DE DATOS RELACIONALES

Las bases de datos relacionales como MySql, SQLite, PostgreSQL tienen el problema de la baja escalabilidad, es decir, que mientras más grande se hace el sistema las consultas sobre la misma se hacen más complejas.

Cuando se estudia normalización de base de datos es fácil darse cuenta que las formas normales solo ahorran espacio de almacenamiento, pero ¿qué pasa con el procesamiento de las consultas? para explicar esto voy a poner un ejemplo simple de un sistema de clientes y de facturación (**ver figura n2**). Quiero hacer una consulta que me devuelva una fila con el nombre y apellido de una persona y el nombre y precio de un producto que esa persona haya comprado alguna vez, ejemplo:

Nombre	Apellido	Nombre	Precio
Juan	Pérez	Teclado Genius	45.00

Para llegar a este resultado hace falta un cuádruple JOIN entre las 4 tablas, la empresa tiene 147000 clientes, 3528000 facturas, 10584000 filas en la relación factura-producto y 500000 productos, cuando se hace un producto cartesiano para sacar las filas necesarias se obtienen 2,744515872e+24 filas y solo son pocas las necesarias para responder a la consulta. Los SGBD implementan índices para acelerar los JOIN, pero esto trae otros problemas cuando se agregan o se quitan filas, ya que hay que reordenar todos los índices de las tablas.

BASES DE DATOS NOSQL (mongoDB)

Las bases de datos No Sql aparecen con las exigencias de los servicios web, ya que tienen una alta escalabilidad y sobre todo una gran velocidad para realizar operaciones. Estas bases de

datos almacenan los datos de forma totalmente desnormalizada, pero esto ya no es un problema, hoy en día los medios de almacenamiento son enormes y no tiene importancia que la información este repetida. Es importante mencionar que las bases de datos NoSql no soportan SQL como lenguaje de consultas

MongoDB es un sistema de bases de datos NoSql que está orientado a documentos, es decir, guarda documentos con estructura JSON por ende cualquier documento JSON que provenga de una consulta puede ser instanciado como un objeto en JavaScript. El esquema es dinámico por lo que no todos los registros pueden tener la misma estructura y en cualquier momento se pueden agregar, eliminar, modificar o renombrar los campos del documento.

Cada documento JSON guardado contiene una clave "_id" con el valor formato ObjectId("5138abe865c37175685fa066")

Para comenzar con mongoDB vamos a trabajar con el Shell de mongo mediante la consola:

- Creando una nueva base de datos: lo único que hay que hacer para crear una base es seleccionar la base con la cual se estará trabajando. Ejemplo > use personal; apretamos enter y saldrá el mensaje switched to db personal pero hay que tener cuidado con esto porque todavía la base no está creada, esta se creara cuando se inserte un registro, para ver las bases creadas se puede ejecutar el comando show dbs y se mostraran todas las bases existentes.

- Creando una nueva colección: para crear una nueva colección solo es necesario insertar un registro. db.collection.operacion() db es la instancia de la base esto nunca cambia, collection es el nombre de la colección y operacion() hace referencia a cualquier operacion CRUD que veremos más adelante. Para ver las colecciones existentes se utiliza > show collections; y se mostraran las colecciones creadas y las necesarias para el funcionamiento de la base.

El core de mongoDB está compuesto por las operaciones CRUD (Create, Read, Update, Delete).

Operaciones Create: incluye las operaciones de insert y save, que sirven para guardar nuevos documentos en la colección.

- insert(<document>): guarda un nuevo JSON en la colección y le asigna el siguiente _id libre.

- save(<document>): guarda o reemplaza un JSON en la colección, cumple la misma función del insert pero si se especifica un _id que este en uso reemplaza el JSON viejo por el nuevo.

Ejemplos:

> db.test.insert({"name":"mongo","type":"DB"});

> db.test.find();

{ "_id" : ObjectId("5138b56365c37175685fa067"), "name":"mongo","type":"DB"}

Nos adelantamos con el comando find() que sirve para seleccionar registros de la colección

> db.test.save({ "_id" : ObjectId("5138b56365c37175685fa067"), "nombre":"juan"});

> db.test.find()

{ "_id" : ObjectId("5138b56365c37175685fa067"), "nombre" : "juan" }

En este caso reemplazamos el JSON que creamos anteriormente, sino se hubiera puesto la clave _id y su valor se hubiera insertado un nuevo registro en la colección.

Operaciones Read: incluye la operación find que selecciona documentos de una colección de la base.

- find(<query>,<projection>): selecciona documentos de la colección, si existen.
- findOne(<query>,<projection>): selecciona un solo documento de la colección, si existe

Operadores de consulta:

- $in: Cualquiera de los indicados.
- $or: Uno de los indicados.

Ejemplos:

> db.personas.find();

{ "_id" : ObjectId("513ccf12462d3eb8f1d6ead9"), "edad" : 71, "nombre" : "Alice", "apellido" : "Baudrix" }

{ "_id" : ObjectId("513ccf12462d3eb8f1d6eada"), "edad" : 23, "nombre" : "Bella", "apellido" : "Paliza" }

{ "_id" : ObjectId("513ccf12462d3eb8f1d6eadb"), "edad" : 56, "nombre" : "Cheyenne", "apellido" : "Medrano" }

Si no se pone una condición se seleccionan todos los documentos de la colección y si no se pone una proyección se respeta la proyección del documento original.

> db.personas.find({},{"_id":0,nombre:''});

{ "nombre" : "Bella" } { "nombre" : "Cheyenne" } { "nombre" : "Pancracio" }

No es necesario incluir una condición en el find, si se quiere proyectar pero seleccionar todos los documentos se debe poner { } las llaves vacías.

>db.personas.find({nombre:{$in:['Alice','Bella']}},{_id:0,edad:1,nombre:'',

apellido:''});

{ "edad" : 71, "nombre" : "Alice", "apellido" : "Baudrix" }

{ "edad" : 23, "nombre" : "Bella", "apellido" : "Paliza" }

Cualquier Persona que se llame Alice o Bella. Para evitar que se muestre el campo "_id" siempre se pone en la proyección _id:0.

> db.personas.find({$or:[{nombre:/c/},{nombre:/C/}]},{nombre:'',_id:0});

{ "nombre" : "Alice" } { "nombre" : "Cheyenne" } { "nombre" : "Carlina" }

{ "nombre" : "Pancracio" } { "nombre" : "Celedonio" } {"nombre": "Celeste" }

Cualquier persona que su nombre contenga c o C, como se ve se pueden utilizar expresiones regulares para comparar las cadenas.

Operación update: modifica uno, varios o ningún documentos de la colección.

- update(<query>,<update>,<options>): modifica los documentos seleccionados en el query con los criterios de update deseados. <options> se puede poner en modo múltiple { multi: true }

Operadores de Update: en esta parte se explicaran los operadores de la segunda parte del update (update(<query>,<update>,<options>)).

- Atributos:

1. $inc: incrementa o disminuye el valor de un atributo en una cantidad espefica.

$inc: { att1 : cant }

2. $rename: renombra un atributo

   $rename: { <old name1>: <new name1> }

3. $set: establece un valor a un atributo, si el atributo no existe se agrega al final.

   $set: { field1: value2 }

4. $unset: elimina un atributo del documento.

   $unset: { field1: "" }

- Array:

1. $pop: Elimina la primera o la última aparición de un elemento en un array, si se pasa 1 se elimina la primera aparición y si se pasa -1 se elimina la última aparición.

   $pop: { atributoArray : 1 }

   $pop: { atributoArray : -1 }

2. $pull: Elimina todas las apariciones de un elemento en un array.

   $pull: { atributoArray : valor1 }

3. $pullAll: Elimina todas las apariciones de todos los elementos determinados en un array.

   $pullAll: { atributoArray : [valor1, valor2, valor3] }

4. $push: Añade un elemento a un array.

   $push: { atributoArray : valor1 }

Ejemplos:

>db.usuarios.update({password:/1234/},{$set:{password:'InsecurityPassword'}},

{multi:true});

Le cambia el atributo password a los usuarios que su password contenga ‘1234’ por ‘InsecurityPassword’

> db.personas.find({edad:68},{edad:1,_id:0,nombre:''})

{ "edad" : 68, "nombre" : "Amanda" }

> db.personas.update({edad:68},{$inc:{edad:2}},{multi:true});

> db.personas.find({edad:70},{edad:1,_id:0,nombre:''})

{ "edad" : 70, "nombre" : "Amanda" }

Se le incrementa dos años de edad a las personas que tienen 68 años.

> db.personas.find({edad:70},{edad:1,_id:0,nombre:'',sueldos:[]})

{ "edad" : 70, "nombre" : "Amanda", "sueldos" : [1000, 1500] }

> db.personas.update({edad:70},{$push:{sueldos:2000}});

> db.personas.find({edad:70},{edad:1,_id:0,nombre:'',sueldos:[]})

{ "edad" : 70, "nombre" : "Amanda", "sueldos" : [1000, 1500, 2000] }

> db.personas.update({edad:70},{$pull:{sueldos:1000}});

> db.personas.find({edad:70},{edad:1,_id:0,nombre:'',sueldos:[]})

{ "edad" : 70, "nombre" : "Amanda", "sueldos" : [1500, 2000] }

Se le ingresa un nuevo sueldo de 2000 a una persona de 70 años y luego se le quita el sueldo de 1000.

Operaciones delete: incluye las operaciones remove y drop.

- remove(<query>,<justOne>): Elimina los documentos de la colección que satisfacen el query. justOne se utiliza si se quiere eliminar solo uno, de lo contrario se eliminan todos los que resulten del query. Solo elimina documentos completos, no sirve para eliminar atributos de un documento.
- drop(): Elimina todos los documentos de una colección y retorna true si se eliminó algún documento o false si no se eliminó ninguno.

Ejemplos:

> db.test.remove({nombre:'juan'});

Se eliminan todas las personas que se llaman 'juan'

> db.test.drop();

true

> db.test.find();

>

Se eliminan todos los documentos de la colección test.

Node.js

Node.js es una plataforma que permite correr código JavaScript con el intérprete "JavaScript V8", es orientado a eventos y sin bloqueo de I/O lo que lo hace ligero y eficiente, fue creado por Ryan Dahl en 2009. Se incluyen módulos que son utilizados para escritura, lectura de archivos, manipulación de peticiones, etc.

Previamente hay que aclarar que programar en esta plataforma no es tan simple como php, que se pone un archivo en una carpeta y apache hace todo el trabajo de montar el servidor, en node nosotros tenemos que crear el servidor y servir al cliente, en principio es difícil de entender este concepto para programadores muy familiarizados con php. Entonces si eres un programador de php tienes que pensar que no existe apache y tú lo debes crear apache. Por lo que comenzar con node puede resultar muy frustrante al comienzo y prácticamente inentendible. Pero no hay que desesperarse por esto, el esfuerzo se ve muy recompensado al final.

Node.js trae muchas funciones nativas muy útiles de JavaScript como console.log(), setInterval(), setTimeout().

Antes de ir al código es necesario explicar y ENTENDER bien el concepto de programación orientada a eventos ya que es muy importante para comenzar con node. Entonces es un paradigma de programación en el que tanto la estructura como la ejecución de los programas van determinados por los sucesos que ocurran en el sistema, definidos por el usuario o que ellos mismos provoquen. La diferencia con la programación secuencial es que en la programación secuencial el programador define el flujo del programa y en la programación orientada a eventos, los eventos definen el flujo del programa. Muchos lenguajes de programación orientados a objetos tienen la posibilidad de manejar eventos.

Comenzando con un ejemplo simple para verificar el funcionamiento de node.

```
santiago@santiago-laptop:~$ node
> 1+1
2
```

Se abre la consola, se escribe node y se presiona enter. Si node está instalado se entra en modo REPL (read-eval-print-loop) esto significa que se puede escribir código JavaScript y será ejecutado como lo muestra el ejemplo. En este caso voy a trabajar en un entorno Unix pero también se puede trabajar con node en Windows y no cambia en absolutamente nada. Para salir de modo REPL apretar Control+C dos veces.

Ahora si podremos realizar un ejemplo simple en un archivo

Archivo: ejemplo.js

```
console.log("archivo ejecutado");
```

Ejecutamos el archivo escribiendo en la consola ~$ node ejemplo.js y saldrá en la consola ‘archivo ejecutado’.

Entendido todo lo anterior ya podemos comenzar a crear nuestro servidor. Si nos dirigimos a la documentación de node, para crear un servidor hay que llamar a http.createServer([requestListener]) y esto retornara un objeto web server.

Explicando parte a parte:

1. http: es la variable del módulo ‘http’ que requerimos, esto se hace de la siguiente forma:

   ```
   var http = require('http');
   ```

2. http.createServer([requestListener]): Es el método que nos permite crear el servidor el parámetro requestListener es la función que se ejecutara cada vez que se reciba una nueva petición al servidor, esto es muy importante ya que cada vez que se recibe una petición esta

función se ejecutara. Se pone solo el nombre de la función no hay que poner el nombre y los ( ), **solo el nombre.**

```
var server = http.createServer(ejecutar);
```

3. Ahora debemos poner el servidor a escuchar en una ip determinada y en un puerto determinado.

```
server.listen(3000,'127.0.0.1');
```

4. Y por último creamos la función que se ejecutara cada vez que se reciba una petición.

```
function ejecutar(peticion,respuesta){
   console.log('Se ha recibido una petición');
}
```

Resumiendo todos los pasos nos queda algo como esto:

```
var http = require('http');

var server = http.createServer(ejecutar);

server.listen(3000,'127.0.0.1');

console.log('Servidor escuchando');

function ejecutar(peticion,respuesta){
   console.log('Se ha recibido una petición');
}
```

Ya hemos creado nuestro primer servidor, todavía sin funcionalidad pero un servidor en fin. Si lo ejecutamos por la consola escribiendo node primerserver.js y saldrá en la consola ”Servidor escuchando”. Nuestro servidor ya tendría que estar funcionando, para comprobarlo escribimos en el navegador 127.0.0.1:3000 y saldrá en la consola el mensaje 'Se ha recibido una petición' si hacemos esto varias veces el mensaje saldrá cada vez que entremos a la dirección. Listo nuestro servidor ya está vivo, pero ¿qué ha pasado en el navegador? En algunos navegadores saldrá el error que no se ha recibido ningún dato y en otros simplemente se quedara cargando siempre y no pasara nada. Esto sucede porque el servidor no responde nunca y el cliente se queda esperando una respuesta que nunca llegara. Para mandar una respuesta al cliente se debe llamar al método write(str) del objeto respuesta (el segundo parámetro de la función ejecutar) y cuando se termine de escribir todo se debe llamar al método end() también del objeto respuesta, esto es muy importante, ya que si no se llama al método end() el cliente sigue esperando que se escriban más cosas. El método end() también puede recibir un string que será lo último escrito. Ahora la función ejecutar quedara de la siguiente manera.

```
function ejecutar(peticion,respuesta){

   console.log('Se ha recibido una petición');

   respuesta.write("Hola mundo");

   respuesta.end(" Finalizar escritura");

}
```

Ahora si podremos ver los cambios reflejados en el navegador cuando entremos a la dirección se escribirá “Hola mundo Finalizar escritura”. Pero ¿qué pasa si escribimos por ejempo http://127.0.0.1:3000/info en el navegador? El resultado será siempre el mismo con cualquier sub ruta.

A continuación con un caso práctico explicare algo básico de ruteo y lectura de archivos. Debemos implementar un servidor simple con ruteo que sirva 3 archivos distintos en diferentes rutas. Para comenzar con esto unos consejos muy útiles para el desarrollo en node:

- console.log(): hay que hacer uso y abuso de esa función, esto nos sirve para ver lo que tiene cualquier variable. Solo con saber que contiene una variable se pueden realizar grandes cosas.
- Documentación online: cualquier tecnología de programación que estudiemos tiene documentación muy detallada de su funcionamiento, solo con buscar en cualquier buscador el nombre de la tecnología, entrando a la página y buscando la sección de documentación se puede conocer su funcionamiento, cosas que contiene, etc. No

hay que tenerle miedo a la documentación es la mejor fuente que puede tener un programador.

Ahora pondremos en práctica el primer consejo en la función ejecutar de nuestro servidor agreguemos una línea de código

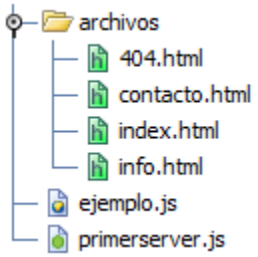
```
console.log(peticion);
```

No tiene importancia en qué lugar este, solo tiene que estar adentro de la función ejecutar. Ahora ejecutamos el script con la consola, entramos al navegador escribimos http://127.0.0.1:3000/info si nos dirigimos a la consola verán que se ha impreso una gran cantidad de líneas y el formato les parecerá familiar, si porque es la variable `peticion` expresada en formato JSON. Si miramos un poco el gran documento JSON escrito en la consola nos encontramos con un atributo llamado url y que tiene como valor '/info' ahí tenemos una pista de lo que necesitamos entonces modificamos la línea de código insertada anteriormente.

```
console.log(peticion.url);
```

Ejecutamos el script, escribimos en el navegador http://127.0.0.1:3000/contacto, y en la consola se escribirá '/contacto' espero que en este punto ya se hayan dado cuenta a que quiero llega, por si no se han dado cuenta lo que tienen en la atributo url es la sub ruta para el ruteo y esto es lo único que hace falta para ruteo, ahora solo tienen leer los archivos del disco y mandarlos al cliente.

Para leer archivos hay que usar el módulo File System o fs abreviado de node. Antes de ir al código mostrare cómo será la disposición de los archivos que serán mandados al cliente.



El servidor es el archivo primerserver.js y los archivos que serán enviados están en la carpeta archivos.

Para leer archivos se debe requerir el módulo fs de node y utilizar la función readFile()

```
var fs = require('fs');
```

Para entregar el archivo al cliente implementaremos una nueva función que recibe el path de un archivo y el objeto respuesta de la función ejecutar.

```
function enviarArchivoACliente(path, respuesta) {
  fs.readFile(path,function(err,data){
    if (err)
      respuesta.end('Error de servidor');
    else
      respuesta.end(data);
  });
}
```

La función readFile recibe como parámetros el path del archivo y la función que se ejecutara cuando el archivo este en memoria. Acá volvemos al tema de los eventos, la función que se ejecuta cuando se termina de leer el archivo en este caso es una función anónima.

Cuando llega una petición se elige el archivo correspondiente mediante un switch y se ejecuta la función enviarArchivoACliente con el path correspondiente y el objeto respuesta. Resultando algo como lo siguiente:

```
function ejecutar(peticion,respuesta){
  switch (peticion.url){
    case '/':
    case '/index':
            enviarArchivoACliente('archivos/index.html',respuesta);
            break;
      case '/contacto':
            enviarArchivoACliente('archivos/contacto.html',respuesta);
          break;
    case '/info':
          enviarArchivoACliente('archivos/info.html',respuesta);
```

```
                    break;

        default :

                    enviarArchivoACliente('archivos/404.html',respuesta);

                    break;

    }

}
```

Solo hay cambios en la función ejecutar y la nueva función enviarArchivoACliente Luego de montar el servidor lo testeamos.

127.0.0.1:3000/	127.0.0.1:3000/contacto	127.0.0.1:3000/info	127.0.0.1:3000/foo
GUIA DE DESARROLLO WEB CON NUEVAS TECNOLOGÍAS Temas a tratar: • javascript • json • problemas de las bases de datos relacionales • …	Tu nombre: Tu email: Comentarios: enviar	El objetivo de este trabajo es ayudarte a empezar con el desarrollo de aplicaciones web con Node.js, durante el desarrollo vas a aprender casi todo lo que necesites saber para realizar grandes proyectos con tecnologías novedosas como Node.js y MongoDB.	404 La ruta solicitada no existe.

Este simple servidor puede ser mejorado de un millón de formas, por ejemplo, almacenar en un buffer en memoria los archivos, comprobar que los archivos existan o no, ver si los archivos en buffer están actualizados o si fueron modificados, etc. no explico todos estos temas por que no tiene sentido hacer un servidor tan complejo que después no vamos a utilizar, ya que más adelante usaremos un framework que implementa un servidor web.

Algo muy importante de node que sirve para ordenar código y fragmentar la aplicación en partes son los **módulos**. Su funcionamiento es simple se crea el modulo en un archivo que puede contener variables, funciones, clases, etc. Se exportan los elementos que se desean y los demás serán privados al módulo y no podrán ser accedidos desde afuera. Luego desde otro archivo se declara una variable y se le asigna la función require con la ruta del archivo y el nombre.

Ejemplo práctico: crearemos el modulo foo.js que cuando se inicie muestre un mensaje en la consola, que tenga una función que imprima un texto en la consola y la cantidad de veces que fue ejecutada la función. La resolución a este problema queda de la siguiente manera.

```
console.log('Fue iniciado el modulo foo.js');

function mostrarTexto(str){
    console.log("Texto: "+str);
    console.log("Cantidad de veces ejecutada: " + cantVecesEjecutada++);
}

var cantVecesEjecutada=1;

//de esta forma se exporta la función, es importante que no tenga los
//paréntesis la función.
exports.mostrarTexto=mostrarTexto;

//el nombre con el cual se exporta no tiene que ser el mismo que tiene
la //función en el módulo.
//en este caso la exporte con el nombre log, pero la función se llama
//mostrarTexto.
exports.log=mostrarTexto;
```

Guardamos esto como foo.js y creamos otro archivo en la misma carpeta para hacer uso de nuestro modulo personalizado.

```
var foo = require("./foo");

foo.mostrarTexto("primer texto");

foo.mostrarTexto("segundo texto");

foo.log("tercer texto");
```

Como se ve cuando a la variable foo (por convención la variable siempre se llama igual que el módulo requerido para no confundirla con otra variable común) le asignamos el require del módulo foo.js se obtiene un objeto que contiene dos métodos mostrarTexto y log, en este caso las dos funciones tienen la misma utilidad pero podrían ser dos funciones distintas. No es recomendado exportar la misma función con dos nombres distintos porque se puede prestar a confusión. Por lo que llamando a cualquiera de las dos se obtiene el mismo resultado.

La salida a consola que se obtiene es la siguiente:

Fue iniciado el modulo foo.js

Texto: primer texto

Cantidad de veces ejecutada: 1

Texto: segundo texto

Cantidad de veces ejecutada: 2

Texto: tercer texto

Cantidad de veces ejecutada: 3

Un módulo puede contener la cantidad de código que sea necesario, inclusive puede requerir de otros módulos. En nuestro caso ya hemos hecho uso de módulos que vienen en el core de node como http y fs.

Algo importante que tiene que quedar bien en claro de node. Son los parámetros que se ejecutan cuando un usuario realiza una petición.

```
function ejecutar(req, res){
    res.end('contenido');
};
```

- req (peticion, request): contiene los parámetros de la página que el usuario solicita, por ejemplo: la ruta.
- res (respuesta o response): contiene los métodos necesarios para poder darle un respuesta al usuario.

  Desde aquí en adelante por convención y simplicidad utilizaremos los nombres req y res.

Modelo vista controlador

Previo a comenzar con el desarrollo de una aplicación compleja voy a hacer referencia a un patrón de diseño que utilizaremos con un framework. Este patrón consiste en separar en tres partes el desarrollo de un software: los datos, la interfaz de usuario y la lógica del negocio.

- Modelo: Esta es la representación específica de la información con la cual el sistema opera. En resumen, el modelo se limita a lo relativo de la vista y su controlador facilitando las presentaciones visuales complejas. El sistema también puede operar con más datos no relativos a la presentación, haciendo uso integrado de otras lógicas de negocio y de datos afines con el sistema modelado.
- Vista: Este presenta el modelo en un formato adecuado para interactuar, usualmente la interfaz de usuario.

- Controlador: Este responde a eventos, usualmente acciones del usuario, e invoca peticiones al modelo y, probablemente, a la vista.

Cuando explique el framework que vamos a utilizar voy a hacer referencia a estos tres ítems para ubicarlos en el proceso de desarrollo.

NPM (Node Package Manager)

NPM es una herramienta que utilizaremos mucho junto con node. Básicamente permite instalar programas creados en Node.js que han sido hechos por otras personas. También nos permite subir nuestros programas para que otras personas lo descarguen mediante npm.

Este programa no tiene interfaz gráfica solo se usara con la consola. No hay que preocuparse por su instalación, ya que se instala junto con node. Para comprobar que esté instalado escribir en la consola npm -v y esto retornara la versión instalada de npm.

Para utilizarlo hay que escribir npm en la consola y el comando que se quiera realizar. Para obtener una lista de comandos escribir npm --help y saldrá una lista de comandos. El comando que más utilizaremos será install por ejemplo: npm install nombrePaquete

Jade

Jade es un potente motor de plantillas implementado en JavaScript y especialmente para ser usado en node. Cuando nos referimos a un modelo vista controlador Jade es la vista, ósea va a encargarse de generar el html que será entregado al cliente.

Para instalar jade hay que utilizar npm de la siguiente forma:

npm install -g jade

Para compilar un archivo jade se debe situar la consola en el directorio del archivo y escribir: jade –p nombreArchivo.jade y si no hay errores saldrá el mensaje rendered nombreArchivo.html

La sintaxis es muy simple, más sencilla que la de html y está basado en la indentación. La mejor forma de entender este motor de plantillas es comparando el código original con el resultado, realmente es muy simple y se puede aprender en muy poco tiempo.

Primer ejemplo:

```
!!!5
html(lang="en")
    head
```

```
        title titulo de pagina
    body
        h1 Jade - node template engine
        div#container
            p div con clase personalizada.
        div.myid
            p div con id personalizado.
        button(onclick='alert("click");',style='background-color:blue')
boton con atributos
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>titulo de pagina</title>
  </head>
  <body>
    <h1>Jade - node template engine</h1>
    <div id="container">
      <p>div con clase personalizada.</p>
    </div>
    <div class="myid">
      <p>div con id personalizado.</p>
    </div>
    <button onclick="alert(&amp;quot;click&amp;quot;);"
style="background-color:blue">boton con atributos</button>
  </body>
</html>
```

Como se ve es muy simple lo único raro de este código es el principio "!!!5" esto genera el DOCTYPE de html5. No son necesarias las etiquetas de cierre y los elementos no llevan los paréntesis angulares < >. Jade basa el parentesco de los elementos mediante la indentación, por lo que es muy importante respetarla para conseguir los resultados deseados.

En Jade se puede escribir código como declaración de variables, sentencias de control e iteraciones. Las declaraciones de variables se preceden con un guion para indicarle al intérprete que es una declaración. Las sentencias de control pueden ser if, else if e if, y las iteraciones pueden ser while, each y for. La sintaxis es como se muestra en el ejemplo siguiente:

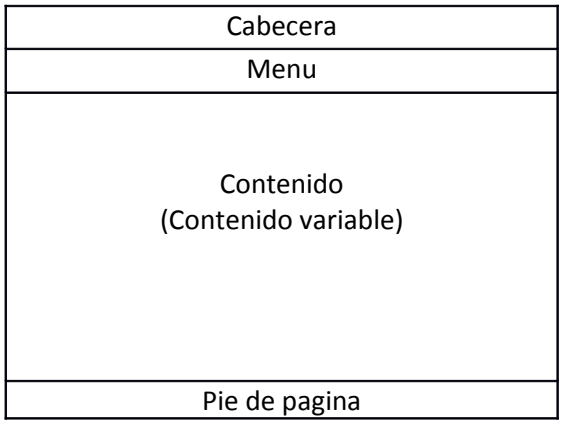
```
-var youAreUsingJade=true;
-var nombre='juan'

-var amigos=['foo','bar','foobar','baz']

-var parientes={madre:'Emma',padre:'Diego',hermano:'Pedro'}

if youAreUsingJade
    p You are amazing #{nombre}!
else
    p Get on it!
p tus amigos:
ul
    for amigo in amigos
        li=amigo
```

```
p tus parientes:
ul
    for i,key in parientes
        li Pariente: #{i}, parentesco: #{key}
p imprimir 3 veces:
    -var c=1;
    while c<=3
        p Impresión: #{c}
        -c++
```

```
<p>You are amazing juan!</p>
<p>tus amigos:</p>
<ul>
  <li>foo</li>
  <li>bar</li>
  <li>foobar</li>
  <li>baz</li>
</ul>
<p>tus parientes:</p>
<ul>
  <li>Pariente: Emma, parentesco: madre</li>
  <li>Pariente: Diego, parentesco: padre</li>
  <li>Pariente: Pedro, parentesco: hermano</li>
</ul>
<p>imprimir 3 veces:
  <p>Impresión: 1</p>
  <p>Impresión: 2</p>
  <p>Impresión: 3</p>
</p>
```

Obviamente luego de realizar la compilación, la declaración de variables y las sentencias se desaparecen para generar el resultado final. Como se ve en el ejemplo las variables pueden ser de cualquier tipo incluso JSON.

En jade un archivo puede extender a otro, esto es útil para los casos que solo hay que cambiar una sola sección de la página y el resto de la página se mantiene siempre igual. Con un ejemplo gráfico se entenderá lo que quiero decir:



Como se ve en el grafico lo único que cambiara de esta página es el contenido del medio, entonces haré un archivo que sea la estructura y otro que solo contenga el contenido.

Archivo layout.jade

```
!!!5
html
    head
        title Titulo
    body
        header
            h2 Cabecera
        div.menu
            a(href='#') inicio
            a(href='#') info
        block contenido
        footer Pie de pagina
```

Archivo inicio.jade

```
extends layout

block contenido
    h3 Inicio
    p El objetivo de este trabajo es ayudarte a empezar con el desarrollo de aplicaciones web con Node.js y MongoDB
```

Cuando se compila el archivo inicio.jade se obtiene el siguiente resultado:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Titulo</title>
  </head>
  <body>
    <header>
      <h2>Cabecera</h2>
    </header>
    <div class="menu"><a href="#">inicio</a><a href="#">info</a></div>
    <h3>Inicio</h3>
    <p>El objetivo de este trabajo es ayudarte a empezar con el desarrollo de aplicaciones web con Node.js y MongoDB</p>
    <footer>Pie de pagina</footer>
  </body>
</html>
```

Cuando un archivo extiende a otro, este solo debe establecer el contenido que en este caso el layout.jade determina, esto se realiza mediante la palabra clave block seguido el nombre de la sección, en este caso se llama “contenido” un archivo de layout puede contener todas las secciones variables que sean necesarias, si el archivo que lo extiende no rellena todas las secciones estas simplemente se van a ignorar.

Ahora crearemos otro archivo que extienda a layout.jade

Archivo autor.jade

```
extends layout

block contenido
    p Santiago Nahuel Cordoba
    p Universidad de La Punta
```

Resultado de la compilación:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Titulo</title>
  </head>
  <body>
    <header>
```

	Cabecera </header> inicioinfo Santiago Nahuel Cordoba Universidad de La Punta <footer>Pie de pagina</footer> </body> </html>

Como se aprecia en el ejemplo solo hay que cambiar unas pocas líneas de código para generar un resultado totalmente distinto y el archivo layout.jade no cambia en lo absoluto.

En jade se pueden incluir archivos externos mediante la palabra include, pero esto tiene una gran desventaja, jade no guarda los archivos en cache por lo que cada vez que se genera la plantilla el archivo incluido se vuelve a leer. Veamos con un ejemplo como se logra la inclusión.

Archivo includes.jade	Resultado includes.html
p texto incluido: p include texto.txt	texto incluido: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce cursus hendrerit rutrum.

Se puede incluir cualquier tipo de archivo que contenga texto. Los archivos jade que sean incluidos serán procesados para generar código html, y los archivos html incluidos serán pasados directamente.

Stylus

Stylus es una forma simplificada de escribir css muy útil para ahorrar unas cuantas líneas de código. Es presentado con el lema “CSS necesita un héroe” y sus características principales son:

- Escrito en JavaScript distribuido mediante NPM.
- Sintaxis limpia: sin llaves, sin paréntesis y sin punto y coma al final de un estilo.
- Indentación forzada: está basado en indentación por lo que esta es obligatoria.
- No te repitas (DRY): Tiene la posibilidad de definir variables y crear funciones.

Se instala con npm con el siguiente comando npm install -g stylus y para compilar un archivo .styl se utiliza el comando stylus -c nombreArchivo.styl

N.Ref	Archivo ejemplo1.styl	Resultado ejemplo1.css

1	`h1`
	`  border 2px solid blue`
		`h1 {`
		`  border: 2px solid #00f`
		`}`

2	`.texto`
	`  p`
	`  font-family Arial`
	`  ul`
		`.texto p {`
		`  font-family: Arial`
		`}`

3	`  li`
	`  background-color black`
	`  color white`
		`.texto ul li {`
		`  background-color: #000;`
		`  color: #fff`
		`}`

4	`vendor(prop,args)`
	`  -webkit-{prop} args`
	`  -moz-{prop} args`
	`  -o-{prop} args`
	`  {prop} args`
		`.publicacion {`
		`  -webkit-border-radius: 3px;`
		`  -moz-border-radius: 3px;`
		`  -o-border-radius: 3px;`
		`  border-radius: 3px;`

5	`.publicacion`
	`  vendor('border-radius', 3px)`
	`  border 1px solid black`
		`  border: 1px solid #000`
		`}`

6	`tamanio_fuente = 0.5em`	`p.like {` `  font-size: .5em;`

7	`p.like`
	`  font-size tamanio_fuente`
	`  vendor('border-radius', 3px)`
	`  border 1px solid black`
	`h3`
	`  font-size tamanio_fuente`
		`  -webkit-border-radius: 3px;`
		`  -moz-border-radius: 3px;`
		`  -o-border-radius: 3px;`
		`  border-radius: 3px;`
		`  border: 1px solid #000`
		`}`
		`h3 {`
		`  font-size: .5em`
		`}`

Con un simple ejemplo es suficiente para aprender las cosas útiles de Stylus. Atender a los números de referencia que puse en la tabla para las explicaciones.

1. Selector h2: los selectores de elementos son iguales que en css.
2. Selector de clase: el selector de clase e id son iguales que en css
3. Los selectores se pueden indentar uno adentro de otro.
4. Función: para declarar una función se pone el nombre y luego los parámetros adentro de los paréntesis. Si un parámetro es usado en el nombre de la propiedad este se debe encerrar con llaves y los parámetros usados como valor simplemente se ponen con el mismo nombre que en la definición de la función.
5. Uso de función: simplemente se pone el nombre de la función y los parámetros.
6. Declaración de variable: para declarar variables no es necesario el prefijo var.

7. Uso de variable.

Express

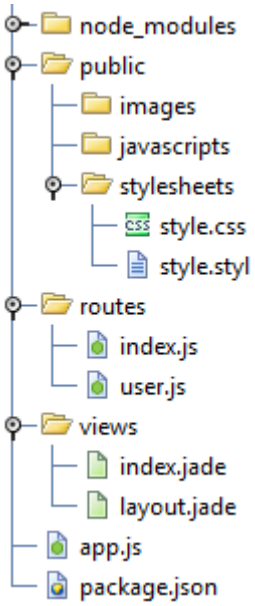
Express es un framework de desarrollo web escrito en JavaScript que sigue el patrón modelo vista controlador y está basado en el framework Sinatra para ruby. Este implementa un servidor web y todas las funciones de ruteo de forma simplificada.

Para su instalación se utiliza el siguiente comando npm install –g express y el comando para crear un nuevo proyecto en express es:

express nombreProyecto opciones

En las opciones se pueden incluir el soporte de sesiones (--sessions) y el soporte de motores de css (-c stylus) por el momento vamos a incluir el soporte de sesiones y stylus, luego explicare detalladamente que son las sesiones y como usarlas.

Para inciar nuestro proyecto escribimos express nombreProyecto --sessions -c stylus luego se pide que cambiemos el directorio y que instalemos las dependencias cd nombreProyecto y luego npm install probaremos la aplicación corriendo el script app.js mediante node (node app.js) Saldrá el mensaje Express server listening on port 3000 que nos confirma que express está corriendo en el puerto 400. Si ingresamos a http://localhost:3000/ podremos comprobar que express funciona correctamente. Cuando se inicia un nuevo proyecto se crea una estructura de archivos y carpetas parecida a la siguiente:



Vamos a analizar un poco esto que crea express:

- Carpeta node_modules: es necesario para el funcionamiento de node y no hay que modificarla de ninguna manera.
- Carpeta public: todo lo que contenga esta carpeta será público para los usuarios, se puede poner cualquier tipo de archivo.
- Carpeta routes: en esta carpeta pondremos todos los scripts que escribamos para generar las distintas páginas.
- Carpeta views: esta carpeta contendrá todos los archivos jade para ser utilizados.
- Archivo app.js: es el archivo que inicia las configuraciones y el servidor.

El ruteo en express esta implementado de forma simplificada, si vemos el archivo app.js, entes de la última función en el archivo estarán escritas las siguientes líneas:

```
app.get('/', routes.index);
app.get('/users', user.list);
```

La variable app es el servidor express y para agregarle nuevas rutas al servidor solo hay que setearlas con el método get de la variable app, el método get recibe un string con la ruta y una función con dos parámetros request y response (solicitud y respuesta) en estos casos usaremos módulos para organizar mejor el código. Como se ve en ese caso hay una ruta `'/'` esa ruta corresponde a el index de la página. Para agregar una ruta con un contenido que puede variar, como por ejemplo el nombre de un usuario, se hace de la siguiente manera.

```
app.get('/usuario/:nombreUsuario',user.verUsuario);
```

Cuando alguien entre mediante un link o escribiendo la dirección por ejemplo localhost:3000/usuario/juan express pondrá en algún lugar de la variable req el nombre del usuario, el lugar donde se guarda es en la variable params del objeto req que se accede como un objeto ejemplo:

```
console.log(req.params.nombreUsuario); //con el mismo nombre de la ruta
```

Entrando a esa página (localhost:3000/usuario/juan) saldrá en la consola el string “juan”.

Para procesar el query string de una página por ejemplo: http://localhost:3000/usuario/santiago?id=4&show=post esta información será guardada en la variable query del objeto req y será accedido como un objeto, por ejemplo:

```
console.log(req.params.nombreUsuario); //con el mismo nombre que en ruta
```

```
console.log(req.query.id);
console.log(req.query.show);
```

Por último para procesar peticiones POST, provenientes de formularios o de AJAX, se asignara la ruta de la misma manera pero el método se llama post y no get, es decir:

```
app.post('/registrar',user.registrar);
```

Y para procesar los datos que vienen por POST se acceden mediante la variable body del objeto req y será accedido como un objeto, por ejemplo: Se tiene un formulario con dos text con name usuario y password, cuando se envíe por el método post se accede de la siguiente forma a los datos que ingresan por post:

console.log(req.body.usuario);

```
console.log(req.body.password);
```

A continuación explicare un poco sobre el renderizado de vistas y como pasar parámetros a jade. Si buscamos el archivo index.js que se encuentra en la carpeta routes encontraran una función exportada y con la variable res (respuesta) que se está utilizando un método “render” (res.render('index', { title: 'Express' });), lo que hace express es agregar ese método que sirve para renderizar vistas. Los parámetros que recibe son:

1. El nombre de la vista: es decir el nombre del archivo en la carpeta views.
2. Los parámetros que recibe la vista: este es un documento JSON con todos los parámetros necesarios para generar la vista, sino están todos los parámetros puede generar un error jade y no hará correctamente el html. Esta es la manera de pasar datos a jade para que sean interpretados.

Suponiendo un caso práctico de un sistema, cualquiera sea, que requiera un inicio de sesión vamos a explicar el concepto de sesión y como se utilizan en express. El soporte de sesiones es una manera de guardar datos en el servidor para que ser utilizados y no se pierdan hasta que sean necesarios.

La aplicación será simple tendrá una página inicio, una de login, una de información, y una de contacto. El usuario cuando no este logueado no podrá ver ninguna página solo la de login. El procedimiento para procesar una petición seria el siguiente:

- Comprobar si esta logueado
    - Si: mostrar página pedida
    - Si no: mostrar página de login.

Cuando se loguea el usuario si es correcto guardar que ese usuario ya está como logueado y redireccionar a otra página,

El ruteo será definido de la siguiente manera:

```
app.get('/', routes.index);
app.get('/contacto',routes.contacto);
app.get('/informacion',routes.informacion);
app.get('/salir',routes.salir)
app.post('/login', routes.login);
```

Como usuarios vamos a considerar un arreglo con los nombres:

```
usuarios=['juan','pedro','jose','luca','baz','foo'];
```

La página con el método post que hace el logueo del usuario comprueba que el usuario exista, sí existe guarda como iniciada su sesión, su nombre y luego redirecciona, si no muestra de nuevo el formulario de login con un mensaje de error:

```
exports.login=function(req,res){
    if (usuarios.indexOf(req.body.usuario)!=-1){
        req.session.iniciada = true;
        req.session.usuario = req.body.usuario;
        res.writeHead(302,{"Location":'http://localhost:3000/'});
        res.end();
    }
    else{
        res.render("login",{title:'Usuario incorrecto'});
    }
}
```

Y cada página comprueba si esta logueado, si lo esta se muestra la página si no se muestra el formulario de login. Todas las paginas funcionan de la misma forma solo que cambia el archivo de render:

```
exports.index = function(req, res){
    if (req.session.iniciada){
        res.render("index",{title:
                'Mi pagina web inicio',usuario:req.session.usuario});
    }
    else{
        res.render("login",{title:'Mi pagina web'});
    }
};
```

Algo importante que mencionar en las sesiones de express es que cuando el servidor se cae toda la información de las sesiones se pierde. Por lo que siempre se mantiene en memoria y hay que tener cuidado de no poner tantos datos en la sesión para no saturar al servidor con datos en memoria. Por ejemplo si guardamos una consulta de una base de datos de diez mil filas sumado a la

redundancia de JSON y multiplicado por la cantidad de usuarios se puede ocupar un gran espacio en memoria. Mi recomendación es usar la sesión solo para guardar los datos justos y necesarios.

Para cerrar una sesión lo único que hay que hacer es borrar las variables que hemos creado con la palabra clave delete que sirve para borrar atributos de un objeto, y luego llevar al usuario a la página de inicio.

```
exports.salir=function(req,res){
    if (req.session.iniciada){
        delete req.session.iniciada;
        delete req.session.usuario;
    }
    res.writeHead(302,{"Location":'http://localhost:3000/'});
    res.end();
}
```

Previo a explicar las conexiones a bases de datos vamos a analizar el archivo package.json que se genera cuando se crea un proyecto en express. La función de este archivo es exclusiva para los servidores web node y para npm:

```
{
  "name": "application-name",     // 1
  "version": "0.0.1",             // 2
  "private": true,                // 3
  "scripts": {                    // 4
    "start": "node app" ,         // 5
    "start": "node sockets.js"    // 6
  },
  "dependencies": {               // 7
    "express": "3.1.0",           // 8
    "jade": "*",                  // 9
  }
}
```

//1. Nombre de la aplicación.

//2. Version de la aplicación.

//3. Si es true no se publicara en mpn, de lo contrario sí.

//4. Colección de scripts que se van a ejecutar cuando se inicie la aplicación.

//5. Se inicia el script app

//6. Se inicia el script sockets.js

//7. Colección de dependencias del paquete.

//8. Dependencia express versión 3.1.0

//9. Dependencia jade cualquier versión.

Por supuesto que el archivo package.json tiene que ser editado cuando se vaya a subir la aplicación a un servidor y se deben poner todas las dependencias necesarias para el correcto funcionamiento. Por ejemplo las dependencias de MySql, Mongoose (MongoDB), Stylus, etc. Cada vez que se modifiquen las dependencias deben dirigirse con la consola al directorio del proyecto mediante cd (change directory) y ejecutar el comando npm install para que se instalen las nuevas dependencias.

MONGOOSE

Para conectarse a una base de datos MongoDB con node se utilizara el paquete "mongoose" que se instala de la siguiente manera:

npm install -g mongoose

Mongoose es un modelador de objetos para node, entonces previo a conectarse a la base de datos hay que realizar los modelos de los documentos que queremos guardar. Los tipos de datos que acepta son: string, number, date, boolean, buffer, mixed y array. En el archivo modelos.js vamos a realizar el modelo bastante complejo de un usuario de un foro.

Archivo modelo.js:

```
var mongoose=require("mongoose");

var esquema=mongoose.Schema(
    {
        usuario:'string',
        nombre:'string',
        apellido:'string',
        amigos:'array',
        comentarios:[
            {
                coment:'string',
                date :{type:"date", default:Date.now},
                idpost:'string'
            }
        ],
        posts:[
            {
         title:{type:'string',uppercase:true,trim:true},

                post:'string',
                visitas:{type:'number',default:0},
                date :{type:'date', default:Date.now }
            }
        ]
    }
);
```

Como se observa es muy gráfico y simple el modelado de documentos JSON con mongoose. Lo único que varía un poco es cuando se le asigna propiedades a un campo como por el ejemplo el valor por defecto o una conversión de string. Ejemplos:

```
title:{type:'string',uppercase:true,trim:true},
visitas:{type:'number',default:0},
date :{type:'date', default:Date.now }
```

Un string puede ser convertido a varios formatos:

- lowercase: todo en minúsculas.
- uppercase: todo en mayúsculas.
- trim: elimina espacios en blanco adelante o al final del string

Por último se creara una clase en la variable que le asignemos y la exportamos:

```
var Usuario=mongoose.model("Usuario",esquema);

exports.Usuario=Usuario;
```

Esto es fundamental que sea hecho en un módulo por que los modelos solo se pueden declarar una vez si lo intentamos hacer más de una vez dará un error. Otra particularidad es que el nombre de la colección será en plural, por lo que si nosotros le pusimos usuario el nombre de la colección será usuarios.

Para conectar a la base y utilizar los modelos simplemente se requiere el módulo mongoose y el modulo donde se encuentren los modelos.

```
var modelos=require("../modelos.js");
var mongoose=require("mongoose");

mongoose.connect("localhost","test",function(err){
    if (err) console.log(err);
});
```

Algo particular pero muy bueno de mongoose es que el query se genera como un objeto y no como en otros sistemas de bases de datos que hay que generar un string inmenso con el query, lo que es muy complicado y siempre trae problemas por ejemplo con los quotes(') que faltan o que sobran, también está el problema de la inyección sql el cual puede representar un gran problema en la seguridad. Volviendo al tema de la siguiente manera se crea un modelo para ser guardado en la base:

```
var user=new modelos.Usuario(
    {
        usuario:"juanpablo",
```

```
        nombre:"juan",
        amigos:["lucas","mariano"]
    });

user.apellido='perez';


user.save(function(err){
    if (err) console.log(err);
    else console.log("usuario guardado");
});
```

Como se observa el modelo se puede crear directamente pasando el JSON con los datos al constructor o directamente como un objeto, en el ejemplo se realiza de las dos maneras. Si hacemos un find en la consola con el usuario “juanpablo” obtenemos el siguiente resultado:

{ "apellido" : "perez", "usuario" : "juanpablo", "nombre" : "juan", "posts" : [ ], "comentarios" : [ ], "amigos" : ["lucas", "mariano"] }

Para buscar elementos en la colección es muy simple:

```
modelos.Usuario.find({},{usuario:'',_id:0},function(err,docs){
    if (err) console.log(err);
    console.log(docs);
});
```

En este ejemplo buscamos todos los documentos y le pedimos que proyecte solo el usuario. Luego de la proyección podremos también poner las opciones, es decir, que el método find puede recibir los siguiente parámetros: find(conditions, [proyeccion], [opciones], [callback]) callback es la función que se ejecutara cuando la consulta se haya realizado, si bien no es obligatorio pero no tendría sentido hacer un find y no realizar ninguna operación con los datos obtenidos.

Algo bueno de Mongo es que su lenguaje de consultas es igual a javascript por lo que las consultas desde node son muy simples y no varía absolutamente en nada su sintaxis. Por ejemplo vamos a crear una consulta que agregue un post al usuario “juanpablo”:

```
modelos.Usuario.update(
    { usuario:"juanpablo" },
    {
        $push:{
            posts:{
                title:"    primerpost    ",
                post:'hola bienvenidos a mi primer post!!!'
            }
        }
    },
    function(err, numAffectedRows) {
        if (err) console.log(err);
        str=('filas afectadas: '+numAffectedRows);
        console.log(str);
```

```
    }
);
```

Directamente sobre la clase Usuario del módulo modelos utilizamos el método update y le agregamos con $push un post al atributo posts. He puesto el título del post con espacios en los extremos del string y con minúsculas para ver el efecto de trim y de uppercase (señalado con rojo), también se podrá observar los valores puestos por defecto de visitas y date (señalado con azul) Resultado:

> db.usuarios.find({usuario:'juanpablo'},{_id:0,posts:[]});

{ "posts" : [{ "title" : "PRIMERPOST", "post" : "hola bienvenidos a mi primer post!!!", "_id" : ObjectId("514a035b6188ddf417000003"), "date" : ISODate("2013-03-20T18:43:39.431Z"), "visitas" : 0 }] }

En el ejemplo anterior solo se utiliza $push para agregar un elemento a un array pero se puede utilizar cualquier operador $inc, $rename, $set, $unset, $pop, $push, $pull, etc. También se pueden utilizar varios de los operadores al mismo tiempo. Para hacer un update múltiple se realiza de la misma forma que en la consola, es decir, agregando { multi:true } luego del documento update.

Y por último para eliminar documentos de una colección se utilizar el método remove de la clase.

```
modelos.Usuario.remove({usuario:'ejemplo'},function(err,numAffectedRows){
    if (err) console.log(err);
    console.log("Cantidad de filas afectadas por remove:
"+numAffectedRows);
});
```

El api de Mongoose incluye otros metodos como findById, findOne, findOneAndUpdate, findByIdAndUpdate, findOneAndRemove, findByIdAndRemove, etcétera. Todas estas funciones son muy simples de usar y solo con el nombre se entiende cuál es su funcionalidad, para más explicaciones sobre estas funciones se pueden dirigir a la página web de mongoose en la parte de documentación de la api.

MYSQL

Decidí incluir como se realiza la conexión con MySql y node por los siguientes motivos

- Hay modelos de negocio que requieren obligatoriamente bases de datos relacionales.
- El gran uso que tiene MySql en el mundo web.
- Su rendimiento no es malo.

- MongoDB NOSQL es un paradigma totalmente distinto y para la gente que ha trabajado siempre con bases de datos relacionales es muy difícil que salgan de ese paradigma.
- El tiempo que requiere aprender mongo.
- Múltiples pedidos de los programadores que han leído esta guía.

Para instalar el conector de MySql mediante npm se utiliza el siguiente comando:

npm install -g mysql

Propongo un ejemplo simple para explicar la conexión a la base MySql, tenemos la base test con la tabla personas(id, nombre, apellido). En la página de inicio se mostrara una lista con los datos de cada persona.

```
var mysql=require('mysql');


var conn=mysql.createConnection({
    user:'root',
    host:'localhost',
    port:'3306',
    database:'test'
});

conn.connect();

exports.index = function(req, res){
    conn.query("select * from personas",function(err,rows){
        if (err) console.log(err);
        else res.render('index', { title: 'Express' , personas:rows });
    });
};
```

Como se ve es muy simple y la función callback recibe dos parámetros primero el error y luego un JSON el resultado del query. Si ocurrio un error lo mostramos por la consola sino renderizamos el index y le pasamos el resultado del query (variable rows). En jade procesamos los datos de la siguiente manera:

Archivo indes.jade

```
extends layout

block content
    h1= title
    p Welcome to #{title}
    ul
    if personas.length>0
        each p in personas
            li ID: #{p.id} NOMBRE: #{p.nombre} APELLIDO: #{p.apellido}
```

```
    else
        li No se encontraron personas
```

Resultado:

Welcome to Express
- ID: 1 NOMBRE: Lucas APELLIDO: Amieva
- ID: 2 NOMBRE: Martina APELLIDO: Peralta
- ID: 3 NOMBRE: Boanerges APELLIDO: Royo
- ID: 4 NOMBRE: Soledad APELLIDO: Vidal Cornejo
- ID: 5 NOMBRE: Belinda APELLIDO: Torino Solá

Cuando se realiza una operación de insert, update o delete, el callback tendrá dos parámetros, el error y luego el informe.

```
conn.query(
    "insert into personas(nombre,apellido) values('Lucas', 'Amieva')",

    function(err,info){
        if (err) console.log(err);
        console.log(info);
    }
);
```

Si vemos el console.log(info) tendrá la siguiente información en cualquier caso (insert, update, delete)

{ fieldCount: 0,

affectedRows: 1,

insertId: 37,

serverStatus: 2,

warningCount: 0,

message: '',

changedRows: 0 }

Contando con la posibilidad de crear páginas de forma dinámica, conectarse a una base de datos y manejar sesiones, el cielo es el límite ahora, ya que con estos conocimientos se puede realizar prácticamente cualquier aplicación web

Socket.io

Cuando una computadora solicita una página web el proceso que se realiza es el siguiente:

- Cliente hace una petición.
- Servidor procesa la petición.
- Servidor envía los datos (en este caso html) y cierra conexión.
- Cliente recibe los datos y cierra conexión.
- Cliente muestra los datos en el navegador.

Este es el procedimiento común que se realiza cuando un usuario entra a una página, es decir, la petición se procesa de forma síncrona. Pero que sucede si el cliente o el servidor quieren realizar una petición asíncrona, es decir, luego de cargar una página o luego de cerrar la conexión, no se podía realizar esto. Por ejemplo: pensemos en un sistema de ventas, tengo por un lado el comprador1 y por otro lado el vendedor1. Primer paso vendedor1 abre la página web y la deja abierta esperando que lleguen ventas. Segundo paso un comprador1 le compra el producto1 al vendedor1 el servidor recibe la compra y la guarda en una base de datos, el servidor no puede comunicarse con el vendedor1 para avisarle que hay una compra nueva, es decir, el servidor no le puede decir al vendedor1: “vendedor1 comprador1 te compro el producto1”. La única solución posible a esto es que el vendedor1 compruebe cada 5 segundos si entraron compras nuevas (mediante AJAX), es decir, vendedor1 le pregunta al servidor cada 5 segundos si hay nuevas ventas, para que el servidor pueda responder esta consulta se tiene que conectar a una base de datos, seleccionar las nuevas compras y responderle si hay o no. Los problemas que presenta este método son los siguientes:

- Se harán muchas peticiones que simplemente responderán que no hay nuevas ventas. Esto produce un desgaste innecesario de la memoria y la banda ancha, tanto del cliente como del servidor.
- Esperar 5 segundos para que llegue una notificación es un tiempo muy grande. Por ejemplo: si fuera un chat, esperar 5 segundos que llegue un mensaje sería realmente el chat más lento que exista.
- Bueno no se quiere esperar 5 segundos entonces hago la petición cada 250 milisegundos (cuatro veces por segundo). Pensemos que nuestra página es bastante grande y generalmente se conectan mil vendedores al mismo tiempo. Entonces un solo vendedor en un minuto haría 240 peticiones y esa cantidad multiplicada por los mil usuarios que se conectan al mismo tiempo son 240.000 peticiones en un minuto. Recordando que cada petición incluye el procesamiento del pedido, la búsqueda en la base de datos y la respuesta. Tendríamos que tener un servidor con una cantidad de memoria gigante, un motor de bases de datos que soporte mucha concurrencia de transacciones y una gran cantidad de ancho de banda solo para responder esas peticiones.

- Qué pasaría si en el sistema se decide agregar mensajería instantánea y un sistema de comentarios en cada producto publicado, ambos con notificaciones asíncronas, el número de peticiones crecería de forma exponencial.

- Qué pasaría si el sitio web crece en la cantidad de usuarios y fueran cien mil usuarios (que no es un número tan grande para un sitio web) ningún tipo de tecnología podría soportar tal concurrencia de peticiones.

Para solucionar este motivo se han creado los sockets. Si seguimos con el ejemplo anterior, ahora el servidor podrá decirle al vendedor: “vendedor1 comprador1 te compro el producto1”, eso soluciona el problema de las peticiones y solo se limita a realizar comunicaciones cuando es necesario. Un término simple que visto en un tutorial para entender los sockets es “AJAX bidireccional” el cual es correcto porque ahora el servidor si se puede comunicar con un cliente. Cuando un socket se conecta a un servidor la conexión entre estos queda “abierta”.

Socket.io trabaja con eventos que se pueden emitir desde el cliente y recibirlos en el servidor o emitirlos en el servidor y recibirlos en el cliente. Para instalar socket.io mediante npm se utiliza el siguiente comando: npm install -g socket.io recordar que cuando sea utilizado con express se tiene que agregar en las dependencias en el archivo package.json y luego ejecutar npm install en la carpeta del proyecto. Una vez hecho eso tendremos que buscar en la carpeta node_modules el archivo js que nos permitirá conectarnos con socket.io la ruta del archivo es la siguiente:

nombreProyecto\node_modules\socket.io\node_modules\socket.io-client\dist\socket.io.min.js

y a ese archivo lo copiamos en la carpeta public\javascripts de express para poder incluirlo como un script común en la vista o archivo html.

Vamos a explicar esto con un ejemplo simple: tendremos una página de inicio que nos pedirá nuestro nombre y luego nos llevara a una página que nos mostrara todos los usuarios que se van conectando y los usuarios que se van desconectando.

Previo a empezar con los sockets voy a explicar la estructura del proyecto.

El ruteo será definido de la siguiente manera:

```
app.get('/', routes.index);
app.post('/',routes.saveUser);
//hacemos el require del archivo donde iniciaremos los sockets.
var sockets_servidor=require('./socket-servidor.js');
```

El archivo index.js tendrá las siguientes funciones exportadas:

```
exports.index = function(req, res){
    res.render('index', { title: 'Express' ,nombre:req.session.nombre });
};

exports.saveUser=function(req,res){
```

```
    req.session.nombre=req.body.nombreIngreso;
    console.log("Usuario guardado: "+req.body.nombreIngreso);
    res.redirect("/");// redireccionamos al inicio.
};
```

Y por último el archivo index.jade tendrá la siguiente estructura.

```
extends layout

block content
    script(src='/javascripts/jquery.min.js')
    script(src='/javascripts/socket.io.min.js')
    script(src='/javascripts/socket-cliente.js')
    h1= title
    p Welcome to #{title}
    if (nombre)
        input(type='hidden',name='nombre',value= nombre)
        .usuarios
    else
        form(method='post',action='/')
            input(type='text',name='nombreIngreso',
                        placeholder='tu nombre aqui')
```

Para comprender bien los sockets hay que saber en todo momento si estamos trabajando del lado del cliente o del lado del servidor, es fundamental tener bien en claro esto para evitar confusiones, por eso los archivos se llaman socket-cliente.js y socket-servidor.js.

El proceso para crear un servidor de sockets es muy simple y parecido a la creación de un servidor web en node. Se debe requerir el modulo socket.io y hacerlo escuchar en algún puerto.

```
var io = require('socket.io').listen(2020);
```

Los sockets siempre se manejan con eventos y el método que utilizaremos para agregar un escuchador de eventos a un socket es el método on(eventName,callbackFn) lo eventos más importante disponibles del lado del servidor son los siguientes: connection, message, disconnect y todos los eventos que nosotros mismos definamos. Para disparar un evento en el cliente utilizaremos la función emit(eventName,data). Archivo socket-server.js

```
io.sockets.on('connection', function (socket) {  /* 1 */

    socket.on('servidor-ingreso', function (data) { /* 2 */
        if (data.usuario) {
            io.sockets.emit('cliente-nuevoUsuario', data); /* 3 */
            socket.usuario = data.usuario; /* 4 */
        }
    });

    socket.on('disconnect', function () { /* 5 */
        if (socket.usuario) {
            var data = {usuario: socket.usuario};
            io.sockets.emit('cliente-disconnectedUser', data); /* 6 */
        }
```

```
    })
});
```

Explicación del código:

/* 1 */ El evento **'connection'** se agrega a todos los sockets y la función recibe como parámetro el nuevo socket.

/* 2 */ Agregamos el escuchador del evento **'servidor-ingreso'** y la función recibe como parámetro los datos provenientes del cliente, en este caso un documento JSON con el formato: {nombre:**'nombreDeUsuario'**}.

/* 3 */ io.sockets representa todos los sockets conectados. Entonces disparamos el evento **'cliente-nuevoUsuario'** para todos los sockets.

/* 4 */ Guardamos en el socket el nombre del usuario.

/* 5 */ El evento **'disconnect'** se disparara cuando el socket se desconecte.

/* 6 */ Disparamos el evento **'cliente-disconnectedUser'** en cliente para todos los sockets y le pasamos el nombre del usuario desconectado.

Ahora vamos al lado del cliente. Aquí tendremos un poco más de trabajo porque tendremos que mostrarle al usuario la información proveniente de los eventos y aparte disparar el evento **'servidor-ingreso'** cuando detectemos que existe el input hidden con el atributo value con nuestro nombre de usuario (si no se acuerdan de donde viene esto vea más arriba el archivo index.jade). Los eventos disponibles más importantes del lado del cliente son los siguientes: connect, connecting, disconnect, connect_failed, error, reconnect, reconnecting, reconnect_failed. Archivo socket-cliente.js

```
$(document).ready(initSocket);  /* 1 */

function initSocket(){

    if ($("input[name='nombre']").val()!='undefined'){   /* 2 */

        var websocket=io.connect("http://localhost:2020"); /* 3 */

        var nombreUsuario=$("input[name='nombre']").val();
        websocket.emit('servidor-ingreso',{usuario:nombreUsuario}); /* 4 */

        websocket.on("cliente-nuevoUsuario",function(data){ /* 5 */
            var p = $("<P>" + data.usuario + "</P>"); /* 6 */
            p.addClass("newUser");                    /* 6 */
            $(".usuarios").append(p);                 /* 6 */
        }); //end on
        websocket.on("cliente-disconnectedUser",function(data){  /* 7 */
```

```
            var p = $("<P>" + data.usuario + "</P>");     /* 8 */
            p.addClass("disconnectedUser");               /* 8 */
            $(".usuarios").append(p);                     /* 8 */
        }); //end on
    }
}
```

Explicación del código:

/* 1 */ Cuando el documento esté se ejecutara la función encargada de crear el socket.

/* 2 */ Cuando el valor del input con el name nombre tenga contenido se creara el socket, de lo contrario no se hará ninguna acción.

/* 3 */ Se realiza la conexión con la variable global io que proviene del archivo socket.io.min.js. es importante que este archivo se ejecute antes de crear una conexión sino se producirá un error.

/* 4 */ Disparamos el evento **'servidor-ingreso'** con los datos necesarios, ósea el nombre de usuario.

/* 5 */ Agregamos el escuchador de evento **"cliente-nuevoUsuario"** y la función recibe los datos desde el servidor.

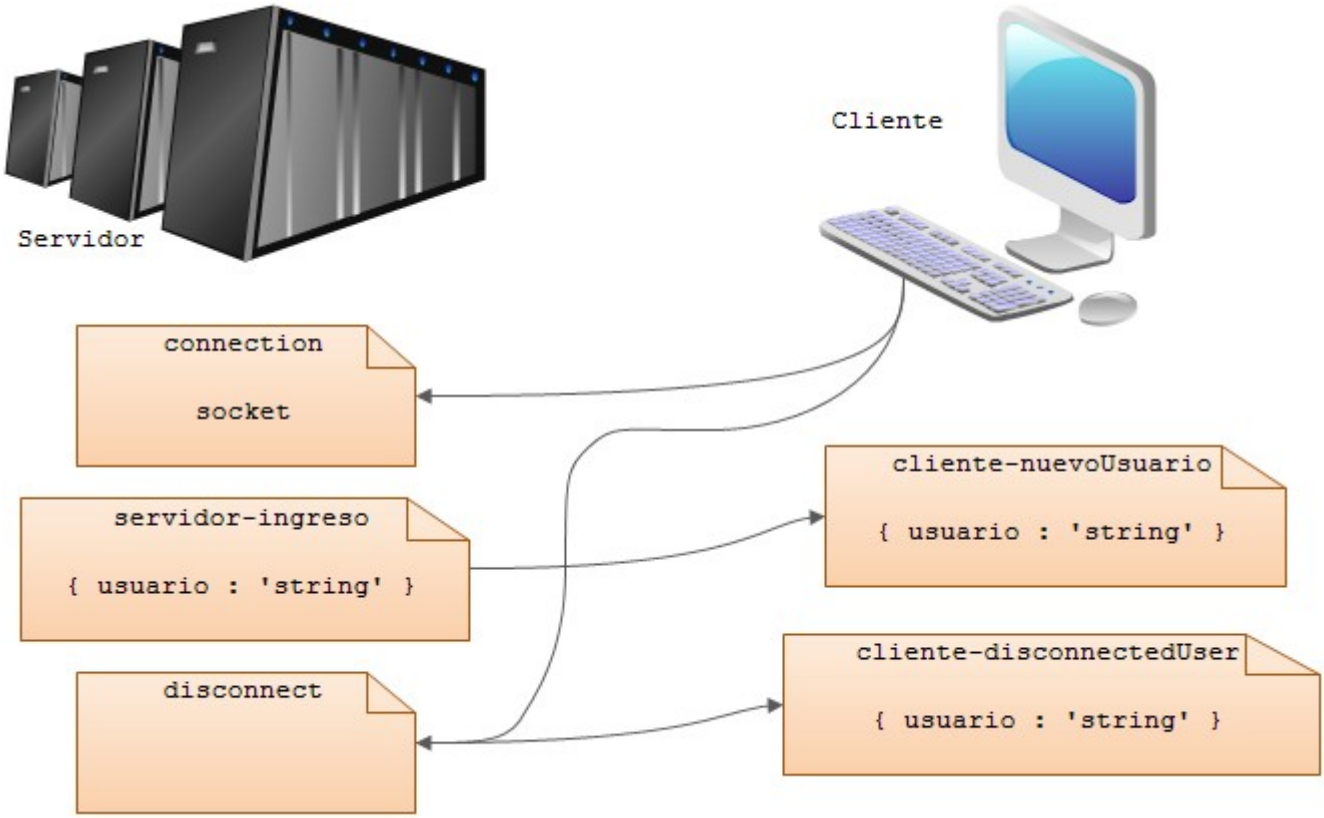
/* 6 */ Mostramos en un párrafo el nombre del nuevo usuario y le agregamos la clase newUser para que se vea verde el nombre.

/* 7 */ Agregamos el escuchador de evento **"cliente-disconnectedUser"** y la función recibe los datos desde el servidor.

/* 8 */ Mostramos en un párrafo el usuario desconectado y le agregamos la clase disconnectedUser para que se vea rojo el nombre.

Algo que te puede ayudar a no confundirte con los sockets es un diagrama. Si bien los diagramas son buenos para la programación estructurada, se vuelve un programa cuando se quiere diagramar programación orientada a eventos ya que no existe un flujo fijo. Así que he inventado una especie de diagrama para comprender el caso anterior.

Cada cuadro representa un evento y abajo se describe que datos se reciben. De cada lado se representa que eventos escucha y las flechas a los eventos del otro lado indican cuales son los eventos que dispara.



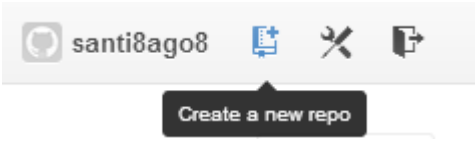
Como se ve el cliente implícitamente dispara el evento connection en el servidor cuando crea la conexión y también dispara implícitamente el evento disconnect cuando cierra la página web.

GIT y GITHUB

Git es un software de control de versiones diseñado por Linus Torvalds, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando estas tienen un gran número de archivos de código fuente (WikipediA). Git por sí solo no es tan bueno porque no dispone de una interfaz gráfica, pero hay una herramienta web, GitHub, que lo complementa perfectamente y le da un entorno visual muy amigable y fácil de utilizar.

Para instalar Git simplemente hay que bajar un instalador y no requiere ninguna configuración inicial. Para crear un cuenta en GitHub es como cualquier proceso de registro en una página.

Una cuenta en GitHub puede contener varios repositorios, para explicarlo de forma simple un repositorio es como una carpeta con archivos. Para organizarnos mejor vamos a tener un repositorio para cada proyecto que tengamos. Y para crear un repositorio hay que hacer clic en un botón que se encuentra en la parte superior derecha de la página.



No pedirá un nombre para el repositorio, descripción, público (gratis) o privado (pago), por ultimo le damos aceptar y nos creara el repositorio.

Para comenzar un proyecto con git te debes dirigir con la consola a la carpeta del proyecto y escribir git init y luego hacer el primer commit git commit –m “primer commit”, un commit es la confirmación del cambio de código.

Para actualizar el código en GitHub hay que escribir en la consola git push origin master y se nos pedirá el usuario y la contraseña para poder hacerlo.

A medida que vamos realizando cambios en el proyecto como por ejemplo agregar y quitar archivos o carpetas, previo a hacer un commit tendremos que agregarlas o eliminarlas del repositorio:

- Para agregar archivos utilizamos el comando add, ejemplo: git add * (agrega todo lo nuevo), git add “nombreArchivo”, git add “nombreCarpeta”
- Para eliminar archivos del proyecto se utiliza el comando rm. No hacer nunca git rm * porque eliminara todos los archivos del directorio, ejemplo: git rm “carpeta” –r (el –r se utiliza para eliminar todo lo que hay dentro de la carpeta), git rm “archivo”
- Para renombrar mover archivos del proyecto se utiliza el comando mv, ejemplos: git mv “archivoOrigen” “archivoDestino”

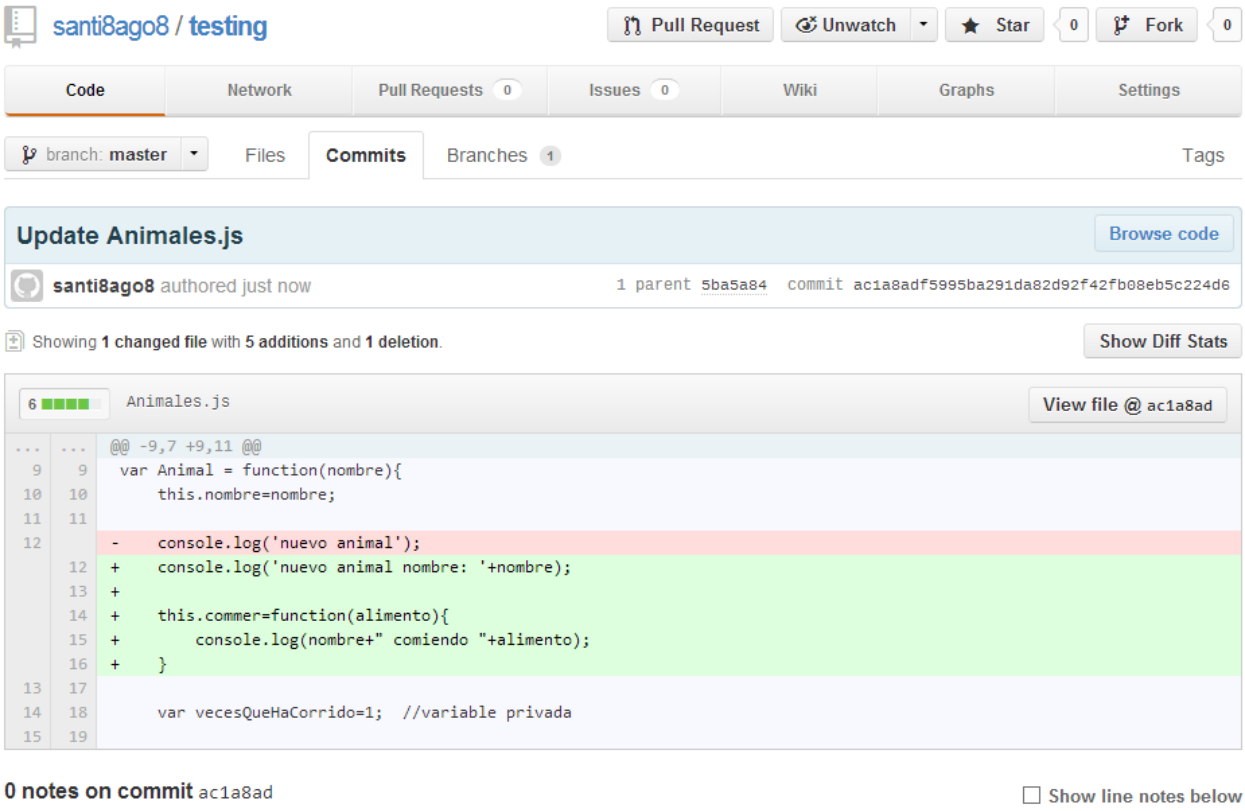
Sino recuerdas que archivos has creado o eliminado, solo basta con intentar hacer un commit y saldrá un error con una lista de los nuevos archivos y los archivos borrados. Luego de agregar y borrar los archivos realizas el commit y puedes sincronizar de nuevo con GitHub (git push origin master).

Si han llegado hasta este párrafo se preguntaran para que sirve todo esto. La verdad es muy útil por muchos varios motivos:

- Control del código por varias personas que no estén en el mismo lugar.
- Copia del código.
- Registro de todas las modificaciones que se efectuó en el código.
- En caso de pérdida del código no se tendría que volver a empezar de nuevo el desarrollo (es recomendable hacer commits y subir el código a GitHub varias veces al día)

- Puntos de restauración: si hemos cometido un error importante con un archivo y tenemos que volver a como estaba antes simplemente se busca en GitHub y se reemplaza.

GitHub nos brinda un excelente servicio web para controla el código, si nos dirigimos al repositorio en la pestaña commits podremos ver una lista completa de todos los commit realizados en el proyecto y si entramos a alguno podremos ver en detalle que se ha modificado.



En ese caso vemos un commit que ha modificado el archivo Animales.js

Otros comandos útiles:

- Pull (git pull): descarga las modificaciones de GitHub y actualiza los archivos en el disco local.
- Clone (git clone repositorio): clona el proyecto en el disco local. Ejemplo: git clone git://github.com/santi8ago8/Devter.git creara una carpeta con el nombre del repositorio y copiara todos los archivos del repositorio a la carpeta creada.

Node Hosting

Para subir una a página a internet no hay muchos servicios que tengan compatibilidad con node pero hay algunos que son gratis y muy buenos. Mi preferencia es AppFog (Nodester) pero de todas formas doy nombre de otros servicios: Cloud Foundry, Cloudnode, DotCloud, Heroku, Modulus, MangoRaft, no.de, NodeSocket, Nodejitsu, JSApp.US, Node Ninja(JP), NAE(CN), OpenShift, Pogoapp.

Modulo complementario de Node

Hay un módulo de node llamado util con métodos como format que explicare más adelante y otros métodos como isArray, isRegExp, isDate, idError que no explicare porque es obvio su funcionamiento.

Método format, retorna un string formateado usando el primer argumento como formato.

```
var util = require('util');

console.log(util.format("nombre: %s","juan"));
console.log(util.format("nombre: %s, %s","juan","perez"));
console.log(util.format("nombre: %s, %s edad: %d","juan","perez",20));
console.log(util.format("nombre: %j",{nombre:'juan',apellido:'perez'}));
```

La salida por consola de ese archivo es la siguiente:

nombre: juan

nombre: juan, perez

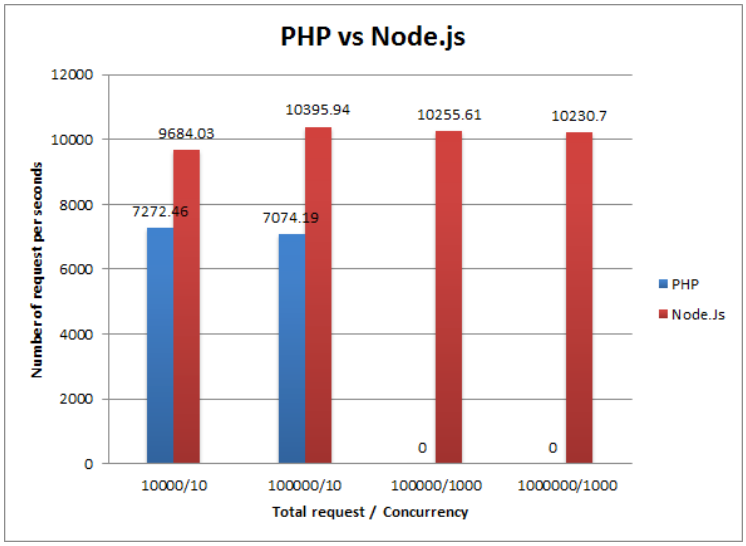
nombre: juan, perez edad: 20

nombre: {"nombre":"juan","apellido":"perez"}

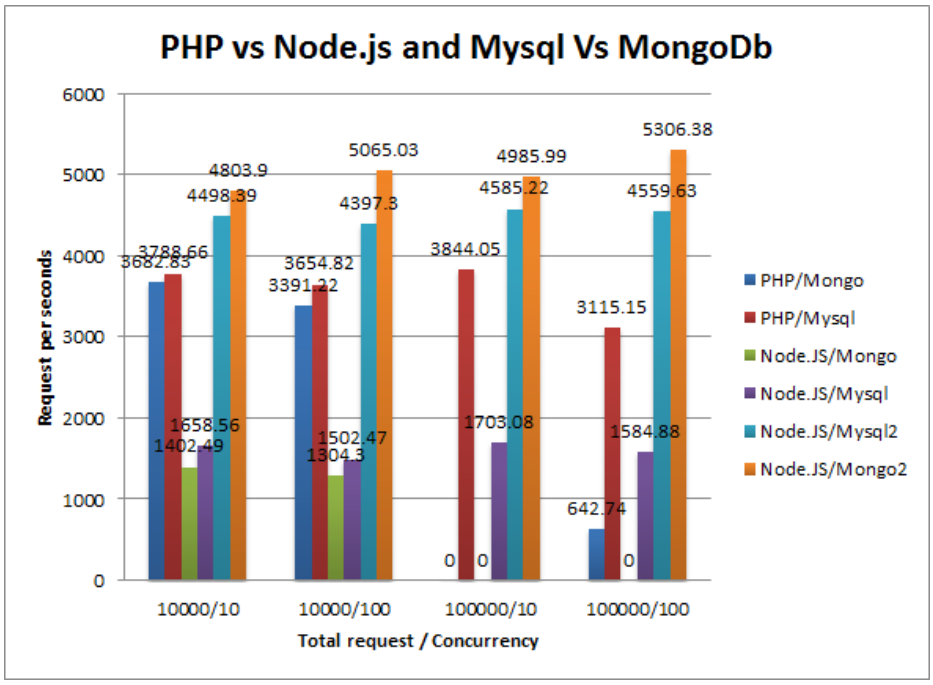
Resumiendo. %s representa un string, %d representa un número, %j representa un documento JSON.

Gráficos de comparación

Comparación entre Php y Node, cantidad de solicitudes procesadas por segundo:



Comparación entre Php, Node, MySql y MongoDB:



Fuente: http://zzarbi.tumblr.com/post/16870870471/php-nodejs-mysql-and-mongo

Comparación entre MySql y MongoDB en relación al espacio de almacenamiento (3,65 millones de registros):

MongoDB	MySQL InnoDB uncompressed	MySQL InnoDB 4KB block size compression
Data: 306MB	*Data:* 251MB	*Data:* 113MB
Index: 83MB	*Index:* 59MB	*Index:* 22MB
Total: 439MB	*Total:* 310MB	*Total:* 135MB
100%	71%	31%

Fuente: http://blog.trackerbird.com/content/mysql-vs-mongodb-disk-space-usage/

Conclusión

Si bien Node.js tiene como gran desventaja que es un poco complicado de comprender al principio, por lo que puede resultar bastante frustrante comenzar con ello, esta tecnología a mi parecer tiene mucho futuro. Todavía es un poco inmaduro, pero eso no quita que sea mucho más potente que otras tecnologías habituales.

Con respecto a MongoDB, es muy efectivo para entornos que se requiere velocidad extrema, sin importar el espacio de almacenamiento (que hoy en día no es un problema). Como cada vez son más escasos los programas de escritorio, es muy importante que se enseñen otros tipos de bases de datos, además de las tradicionales bases de datos relacionales.

Independiente del lenguaje que se elija para programar en web, con una conexión a cualquier base de datos, la posibilidad de devolver HTML y un poco de imaginación, es suficiente para realizar una gran aplicación web.

Anexos

Figura 1:

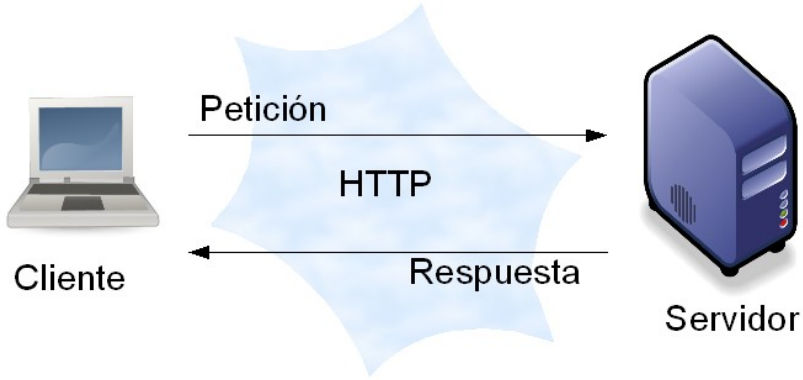


Figura 2:

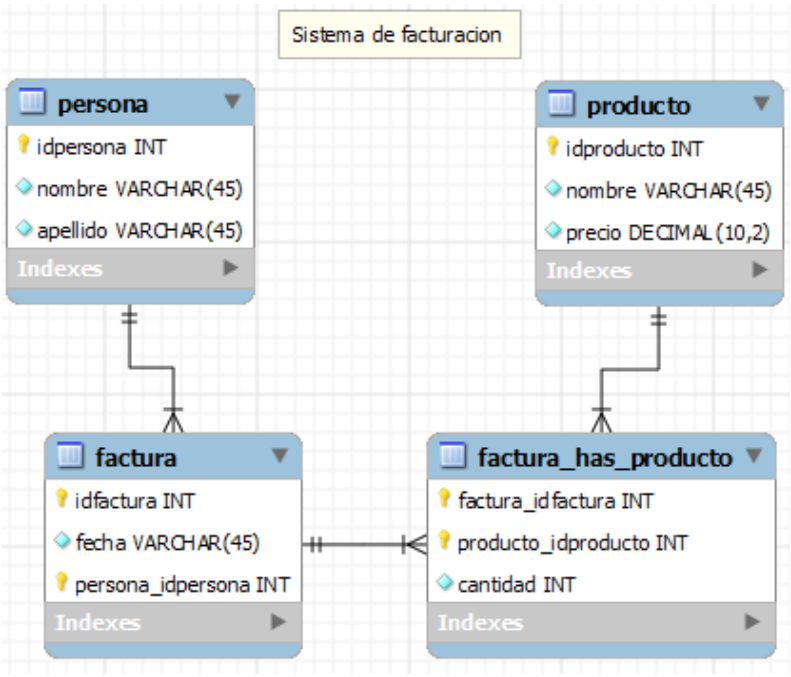
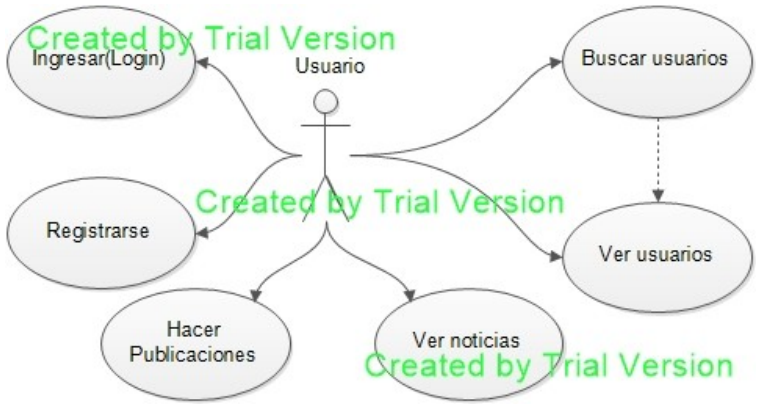


Figura 3


Created by Trial Version
Ingresar(Login)
Usuario
Buscar usuarios
Registrarse
Created by Trial Version
Ver usuarios
Hacer Publicaciones
Ver noticias
Created by Trial Version