Node.js y Websockets

Sumario

Node.js

Introducción a Node.js

Node.js es un entorno **JavaScript de lado de servidor** que utiliza un **modelo asíncrono** y **dirigido por eventos**.

Node usa el motor de JavaScript V8 de Google: una VM tremendamente rápida y de gran calidad escrita por gente como Lars Bak, uno de los mejores ingenieros del mundo especializados en VMs.

No olvidemos que V8 es actualizado constantemente y es uno de los intérpretes más rápidos que puedan existir en la actualidad para cualquier lenguaje dinámico. Además las capacidades de Node para I/O (Entrada/Salida) son realmente ligeras y potentes, dando al desarrollador la posibilidad de utilizar a tope la I/O del sistema. Node soporta protocolos TCP, DNS y HTTP.

¿Qué diferencias tiene Node.js respecto a Apache u otros servidores web?

- Apache crea un nuevo hilo por cada conexión cliente-servidor. Esto funciona bien para pocas conexiones, pero crear nuevos hilos tiene un coste, así como la realización de cambios de contexto. A partir de 400 conexiones simultáneas, el número de segundos para atender las peticiones crece considerablemente. Podemos decir que Apache funciona bien pero no es el mejor servidor para lograr máxima concurrencia (lograr tener el número mayor de conexiones abiertas posibles).
- Uno de los puntos fuertes de Node es su capacidad de mantener muchas conexiones abiertas y esperando. En Apache por ejemplo el parámetro **MaxClients por defecto es 256**. Este valor puede ser aumentado para servir contenido estático, sin embargo si se sirven aplicaciones web dinámicas en PHP u otro lenguaje es probable que al poner un valor alto el servidor se quede bloqueado ante muchas conexiones -esto dependerá del trabajo que la aplicación web de al servidor y de su capacidad hardware-.
- Una aplicación para Node se programa sobre un solo hilo. Si en la aplicación existe una operación bloqueante (entrada/salida por ejemplo), Node creará entonces otro hilo en segundo plano, pero no lo hará sistemáticamente por cada conexión como haría Apache.
- En teoría Node puede mantener tantas conexiones como número máximo de archivos descriptores (sockets) soportados por el sistema. En un sistema UNIX este límite puede rondar por las **65.000 conexiones**, un número muy alto. Sin embargo en la realidad la cifra depende de muchos factores, como la cantidad de información que esté la aplicación distribuyendo a los clientes.
- Una aplicación con actividad normal podría mantener **20.000-25.000** clientes a la vez sin haber apenas retardo en las respuestas. Un inconveniente de Node es que debido a su arquitectura (de usar sólo un hilo) también sólo podrá usar una CPU. Un método para usar múltiples núcleos sería iniciar múltiples instancias de Node en el servidor y poner un balanceador de carga delante de ellos.

Node.js no es el único servidor de este tipo, hay otros proyectos como Tornado (Python), Twisted(Python),Apache Mina(Java), Jetty(Java), etc.

Otra buena alternativa puede ser el lenguaje Erlang el cuál permite crear sistemas en tiempo real con alta escalabilidad y alta disponibilidad.

Ejemplo de un código de node.js

```
var http = require('http');
var s = http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
});
s.listen(8000);
console.log('Server running!');
```

JavaScript es muy buen lenguaje para programar asíncronamente, ya que fue diseñado especialmente para ser usado en programación orientada a eventos. Esto le hace especialmente atractivo para realizar aplicaciones no bloqueantes y de alta concurrencia.

Otro ejemplo de código para node.js :

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

fs.readFile('report.txt', function(data) {
    console.log('Read: ' + data);
});

fs.writeFile('message.txt', 'Hello World!', function() {
    console.log('File saved!');
});
```

Cuando se ejecuta este código Node va a mandar leer un fichero y mandará también escribir en otro (siempre en el lado del servidor), y luego pasará a un estado de espera.

Cuando las operaciones se terminen se ejecutarán los callbacks asociados a cada tarea (las funciones anónimas que tenemos programadas). Debido a la peculiaridad de Javascript, no hay nada que garantice el orden en el que van a ser mostrados los mensajes de los callbacks (teóricamente el que termine antes se mostrará en primer lugar).

Esta manera de ejecutar las entradas y salidas (sin estar a la espera de que se terminen de forma secuencial, antes de pasar a la siguiente tarea), asegura que el hilo principal de programa siempre va a estar en ejecución realizando nuevas llamadas a tareas que hayan sido programadas.

Cuando usar Node.js

Node es especialmente bueno en aplicaciones web que necesiten una conexión persistente con el navegador del cliente, ya que su consumo de recursos es mínimo y el número de clientes simultáneos soportados es altísimo.

Lista de aplicaciones para las que Node encajaría perfectamente:

- Juegos online.
- Gestores de correo online: de esta manera teniendo el navegador abierto podríamos ver notificaciones en tiempo real de nuevos correos recibidos.
- Herramientas de colaboración.
- Chats.
- Redes sociales: por ejemplo para actualizar automáticamente tu muro de novedades.
- Herramientas de traducción en tiempo real.

Cuando no usar Node.js

Node tiene muchas ventajas, pero como en todo no hay herramientas ni peores ni mejores, sino unas que se ajustan mejor a unos casos de uso que otras.

- ¿Son importantes tiempos de respuesta bajos y alta concurrencia?
- ¿El número de usuarios activos va a ser considerable?
- ¿El proyecto es pequeño? Si es grande, ¿se disponen de las librerías/drivers de DB necesarias para trabajar desde JavaScript?

Si en dichas preguntas has respondido sí, entonces Node se adapta perfectamente como solución a tus problemas. Si has respondido no a alguna pregunta entonces posiblemente Node no sea la mejor solución para ese software.

Librerías interesantes para Node.js

- **Socket.IO**: Socket.IO proporciona la posibilidad de crear aplicaciones que se comuniquen en tiempo real con un navegador y dispositivo móvil, sin tener en cuenta los diferentes mecanismos de comunicación. Socket.IO usa diferentes métodos de conexión para establecer una conexión Websocket, mediante AJAX Long polling, etc.. permitiendo la creación de aplicaciones que usen tiempo real de una forma muy rápida.
- **Express**: un framework web sobre Node.
- **node.dbslayer.js**: "ofrece soporte MySQL para Node"
- **Handlebars.js**: "para crear plantillas fácilmente"
- **MongoDB driver o Mongoose**.

Información sobre Node.js extraída de http://www.rmunoz.net/introduccion-a-node-js.html

Instalacion de Node.js y npm (node packet manager) en Debian

Veamos como realizar la instalación básica de un servidor Debian y cómo instalar Node.js y el gestor de paquetes de Node npm.

Instalacion basica de una maquina Debian

```
# Instalamos desde el CD-ROM con las opciones por defecto.
# Configuración de interfaz de red:
nano /etc/network/interfaces

auto lo eth0
iface etho inet static
address 10.21.1.x
netmask 255.255.0.0
gateway 10.21.0.254

# Configuramos el DNS
nano /etc/resolv.conf

nameserver 10.0.4.1
domain sanclemente.local
search sanclemente.local

# Una vez instalado configuramos el /etc/apt/sources.list
nano /etc/apt/sources.list

# Comentamos la linea del cd-rom
# deb cdrom
```

```
# Actualizamos el sistema:
apt-get update
apt-get dist-upgrade

#############################################################################
#Diferencias entre php como módulo y como fastcgi
http://blog.layershift.com/which-php-mode-apache-vs-cgi-vs-fastcgi/
#############################################################################

# Instalamos Apache y PHP (como módulo):
# http://www.howtoforge.com/ubuntu_debian_lamp_server

apt-get install apache2 php5 libapache2-mod-php5

nano /var/www/html/test.php
<?php phpinfo(); ?>

# Instalación de MySQL
apt-get install mysql-server mysql-client php5-mysql

# Configuración de MySQL para escuchar en todos los interfaces (accesos externos):

nano /etc/mysql/my.cnf
bind-address = 0.0.0.0

# Reiniciamos el MySQL
service mysql restart

# Instalación de PHPMyAdmin
apt-get install phpmyadmin

# Reiniciamos el servicio Apache:
service apache2 restart

# Crear usuarios y bases de datos.


#############################################################################
http://www.howtoforge.com/how-to-set-up-apache2-with-mod_fcgid-and-php5-on-debian-squeeze
```

Instalación de Node.js en Debian

Para instalar la última versión de Nodejs haremos lo siguiente:

```
curl -sL https://deb.nodesource.com/setup_13.x | sudo bash -

# Instalamos a continuación nodejs y npm
apt-get install -y nodejs

# Chequeamos la versión instalada:
node -v

v13.2.0
```

Comandos básicos en el uso de Node.js

Desde la línea de comandos de node podemos teclear instrucciones para ejecutar aplicaciones o bien escribir una aplicación. En general lo que se suele hacer es crear un fichero .js con las instrucciones de la aplicación que va a ejecutar Node.

El entorno de trabajo al que se accede desde la línea de comandos se denomina REPL.

```
// Para acceder a la línea de comandos REPL de node:
node

// Accederemos a la consola de trabajo indicada con el símbolo >

// Para consultar ayuda de node:
> .help
```

```
// Para salir de node:
> .exit
// O bien CTRL+C CTRL+C

// Ejemplo de código de javascript en el entorno REPL de node:
>a=4;
4


// Ejemplo de pequeño programa en linea de comandos >:
> a=[1,2,3];
> a.forEach(function(valor){
..... console.log(valor);
..... });

// Producirá como resultado
1
2
3
```

Ejemplo de video con comandos de prueba en el entorno REPL de Node:

Introducción a las notificaciones en tiempo real

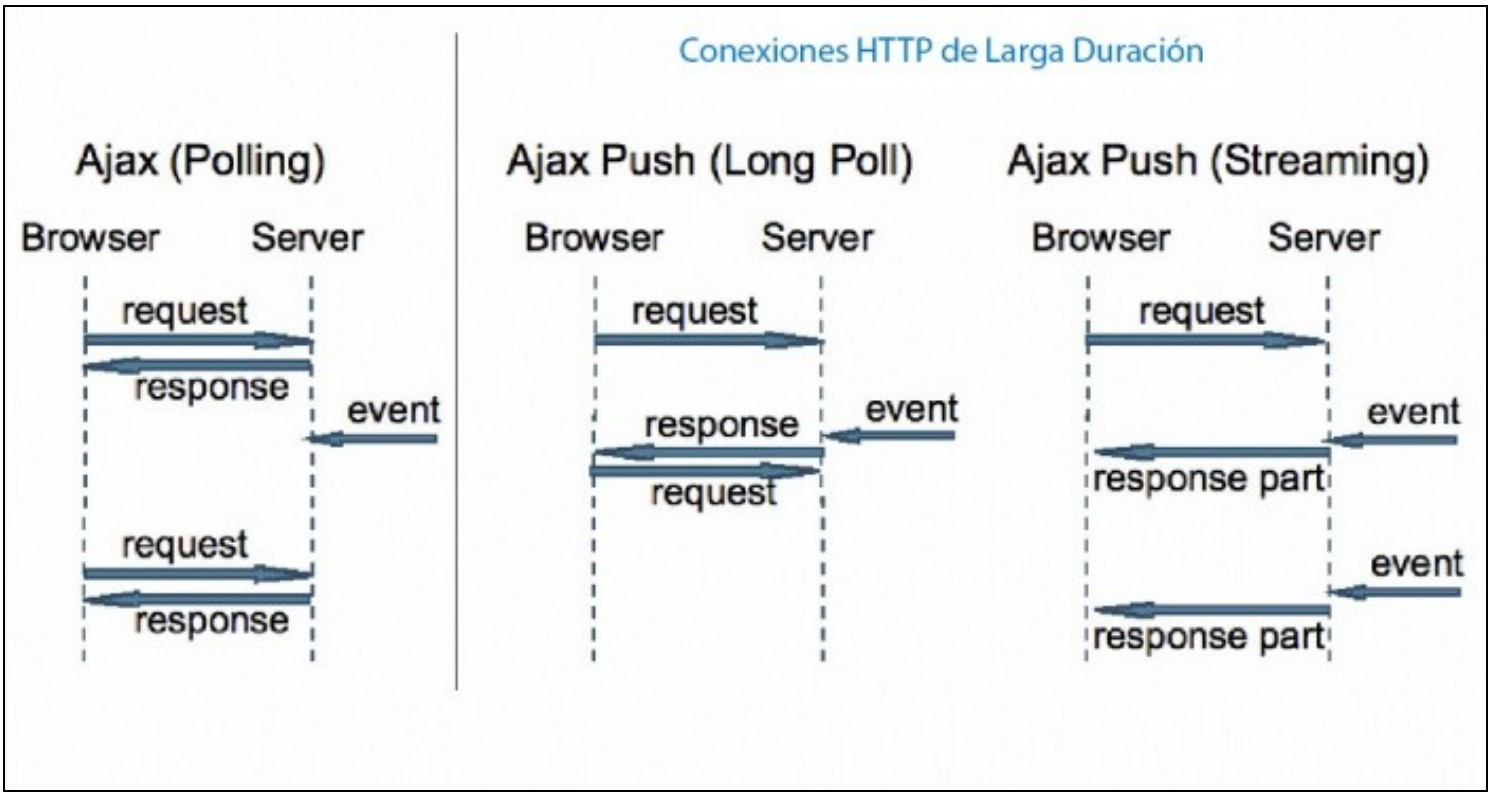
Cuando queremos desarrollar aplicaciones que reciban notificaciones en tiempo real, podremos hacerlo utilizando diferentes técnicas. La mayor parte de ellas consisten en mantener una conexión con el servidor abierta el mayor tiempo posible, de tal manera que se puedan recibir en el lado del cliente las notificaciones del servidor.

Ejemplos de aplicaciones web con información en tiempo real:

- http://demo.kaazing.com/forex/
- http://rumpetroll.com/
- http://www.websocket.org/demos.html

En el desarrollo web se ha creado un nuevo término denominado **Comet**, que se utiliza para describir un modelo de aplicación web en el que una petición HTTP que se mantiene abierta permite enviar datos a un cliente mediante tecnología Push, sin que el cliente lo solicite explícitamente. En realidad **Comet** son múltiples técnicas que consiguen esta interacción.

Aquí vemos un gráfico en el que se muestran las diferentes técnicas.



Polling

La técnica de **Polling** consiste en realizar peticiones ajax desde el cliente al servidor web cada X tiempo, a la espera de obtener nuevos datos desde el servidor.

PROS:

- Facilidad de implementación.

CONTRAS:

- Hay que hacer una conexión cada vez que queremos obtener datos
- Muchas veces no se obtiene ningún resultado en la petición (si no hay datos nuevos en el servidor.)
- Sobrecarga del servidor es proporcional al número de clientes conectados
- Para simular tiempo real habría que aumentar la frecuencia de las peticiones, lo que implicaría más sobrecarga del servidor.

Long Polling

La técnica de **Long Polling** o Polling extendido consiste en realizar peticiones Ajax desde el cliente al servidor web cada vez que queramos obtener datos actualizados. La diferencia con respecto al "Polling" es que en este caso la conexión se mantiene abierta hasta el tiempo máximo permitido de conexión, y si durante ese tiempo recibimos una notificación con datos de respuesta del servidor, los mostraremos en el lado del cliente, cerraremos esa conexión y abriremos una nueva a la espera de recibir nuevos datos.

PROS:

- Facilidad de implementación.
- Sobrecarga de trabajo menor en el servidor, ya que las conexiones permanecen abiertas durante más tiempo.
- Puede resolver problemas de escalabilidad asociados a la técnica de polling.
- Recomendable cuando nuestra aplicación necesita actualizaciones cada 30 segundos o más.

CONTRAS:

- Hay que hacer una conexión cada vez que queremos obtener datos.

- Al tener las conexiones abiertas durante mucho más tiempo, puede ocurrir que no se estén enviando datos, pero se está consumiendo una conexión en el servidor, lo que podría provocar denegaciones de servicio si tenemos muchos clientes conectados.
- Cuando está la conexión abierta, sólo podremos recibir datos desde el servidor en esa conexión. Si queremos enviar datos, tendríamos que abrir una nueva conexión. No es por lo tanto bidireccional.

Http Streaming

La técnica de **Http Streaming** es similar a la técnica de "Long Polling" excepto en que la conexión nunca se cierra, incluso después de que el servidor haya enviado respuestas al cliente.

El cliente enviará una petición Ajax al servidor y recibirá las respuestas a medida que se vayan originando en el servidor, re-utilizando la misma conexión para siempre. Esta técnica reduce significativamente la latencia en la red, ya que no es necesario abrir y cerrar conexiones con el servidor. Por ejemplo Gmail utiliza o utilizaba esta técnica para actualizar su interface de correo en tiempo real.

PROS:

- Complejidad de implementación.
- Mejora de la sensación de tiempo real.
- Recomendable cuando nuestra aplicación necesita actualizaciones muy frecuentes en intervalos cortos de tiempo.

CONTRAS:

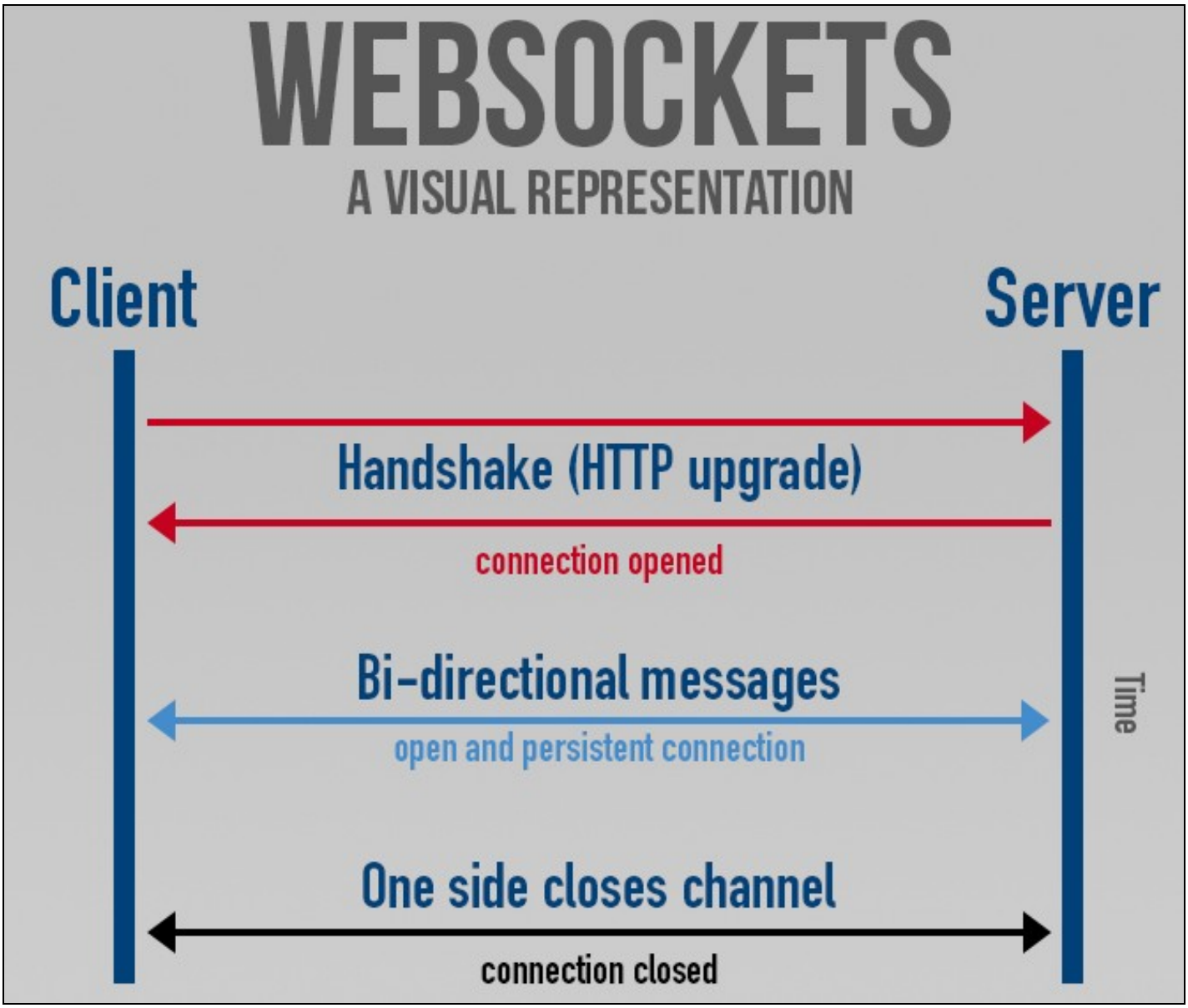
- Los puristas consideran que esta técnica hace abuso del protocolo HTTP.
- Si el servidor envía datos de forma muy continuada, puede afectar al rendimiento de la red y de las aplicaciones Ajax. Puede ocurrir que nuestra aplicación no sea capaz de renderizar la página tan rápido como recibe datos.
- Si tenemos muchos clientes conectados, el servidor puede verse afectado al enviar tantos datos simultáneos en tan poco tiempo.
- Sigue siendo uni-direccional. El cliente sólo puede recibir respuestas del servidor, no puede hacer nuevas peticiones en esa conexión HTTP-streaming.
- Si nuestra aplicación envía datos cada 5 minutos, se recomienda usar long polling, ya que quizás el precio de re-abrir conexiones sea menor que el de mantener una conexión abierta durante largo rato sin recibir datos.
- La especificación HTTP 1.1. sugiere un límite de 2 conexiones simultáneas desde un cliente a un servidor. Esto quiere decir que si nuestra aplicación utiliza long polling o http streaming al mismo servidor, cualquier otra pestaña abierta en el navegador con el mismo servidor daría como resultado bloqueo en la conexión por parte de nuestro navegador.

Introduccion a Websockets

WebSocket es una tecnología que proporciona un canal de comunicación **bidireccional** y **full-duplex** sobre un único socket TCP.

Antes de la llegada de los **websockets en HTML5** las comunicaciones entre clientes web y servidores recaía en el protocolo HTTP.

A partir de ahora, la transmisión de datos puede fluir libremente sobre una **conexión websocket** que es **persistente** (siempre conectada), **full duplex** (bi-direccional de forma simultánea) y **extremadamente rápida**.


WEBSOCKETS
A VISUAL REPRESENTATION
Client
Server
Handshake (HTTP upgrade)
connection opened
Bi-directional messages
open and persistent connection
Time
One side closes channel
connection closed

Está diseñada para ser implementada en navegadores y servidores web, pero puede utilizarse en cualquier tipo de aplicación cliente/servidor.

Como las conexiones TCP ordinarias sobre puertos diferentes al 80 son habitualmente bloqueadas por los administradores de redes, el uso de esta tecnología proporcionaría una solución a este tipo de limitaciones proveyendo una funcionalidad similar a la apertura de varias conexiones en distintos puertos, pero multiplexando diferentes servicios WebSocket sobre un único puerto TCP (a costa de una pequeña sobrecarga del protocolo).

En el lado del cliente, WebSocket está ya implementado en Mozilla Firefox 8, Google Chrome 4 y Safari 5, así como la versión móvil de Safari en el iOS 4.2.1.

En el lado del servidor existen diferentes implementaciones, pero nosotros nos vamos a centrar en su implementación con servidor Node.js utilizando un módulo adicional llamado **socket.io**

Instalacion de Socket.io

Socket.io es módulo-librería adicional para **node.js** que permite la realización de aplicaciones en tiempo real en cualquier navegador o dispositivo móvil, eliminando las diferencias que puedan surgir entre los distintos mecanismos de transporte.

Está programada 100% en JavaScript.

Mediante esta librería podemos montar un servidor que emplee websockets para enviar datos en tiempo real a todos sus clientes conectados.

Más información: http://socket.io/

```
# Necesitaremos previamente tener instalado nodejs y npm.
npm install socket.io
```

Aplicaciones con node.js

Servidor web básico con node.js

Vamos a crear un servidor web básico con Node.js. Para ello haremos una carpeta, por ejemplo y crearemos un fichero con extensión .js Lo editamos e introducimos el siguiente código:

```
// Creamos un fichero servidor1.js con el siguiente contenido:

var http=require("http");

http.createServer(function(req,res)
{
    res.writeHead(200,{"Content-type":"text/plain"});
    res.write("Practicando con node.js \n");
    res.end("parece que funciona. \n");
}).listen(3000,"10.0.1.6");

console.log('Servidor escuchando en puerto 3000 de la IP 10.0.1.6');

/// Ejecutamos ese archivo con el comando: node servidor1.js

////////////////////////////////////////////////////////////////////////////

///// Otro ejemplo prácticamente como el anterior pero enviando html.
// Creamos un fichero servidor2.js con el siguiente contenido:

var http=require("http");

http.createServer(function(req,res)
{
        res.writeHead(200,{"Content-type":"text/html"});
        res.write("<h2>Servidor web con node.js </h2>");
        res.end("...y parece que <strong>funciona!</strong>.");
}).listen(3000);

console.log('Servidor escuchando en puerto 3000 de todas las interfaces.');

/// Ejecutamos ese archivo con el comando: node servidor2.js
```

Servidor web básico con node.js y express

Express es un framework de node que nos permite realizar tareas de forma más rápida.

Primero tendremos que instalar el framework, y eso lo haremos con el gestor de paquetes de node (npm). Lo editamos e introducimos el siguiente código:

```
// Crearemos una carpeta nueva y desde la línea de comandos
// vamos a esa carpeta y ejecutamos desde dentro el siguiente comando:
npm install express

// Una vez instalado aparecerá una carpeta node_modules que contendrá todos los módulos descargados.

// Creamos un fichero servidor3.js con el siguiente contenido:

var http=require("http");

http.createServer(function(req,res)
{
    res.writeHead(200,{"Content-type":"text/plain"});
    res.write("Practicando con node.js \n");
    res.end("parece que funciona. \n");
}).listen(8080,"10.0.1.6");

console.log('Servidor escuchando en puerto 8080 de la IP 10.0.1.6');

/// Ejecutamos ese archivo con el comando: node servidor1.js
// Ahora podremos conectarnos a ese equipo con un navegador a http://ip:8080

//////////////////////////////////////////////////////////////////////////

///// Otro ejemplo prácticamente como el anterior pero enviando html.
// Creamos un fichero servidor2.js con el siguiente contenido:

var http=require("http");

http.createServer(function(req,res)
{
        res.writeHead(200,{"Content-type":"text/html"});
        res.write("<h2>Servidor web con node.js </h2>");
        res.end("...y parece que <strong>funciona!</strong>.");
}).listen(8080);

console.log('Servidor escuchando en puerto 8080 de todas las interfaces.');

/// Ejecutamos ese archivo con el comando: node servidor2.js

// Ahora podremos conectarnos a ese equipo con un navegador a http://ip:8080
```

Configuración de paquetes necesarios con package.json

Si vamos a crear una aplicación un poco más compleja, puede que sea necesario instalar dependencidas en nuestro proyecto. Para ello se puede hacer manualmente, o bien realizarlo de forma automática empleando el fichero **package.json**

Para poder instalar las dependencias de forma automática tendremos que crear primero la carpeta dónde vamos a ejecutar nuestra aplicación de node.js. A continuación crearemos un paquete con el nombre **package.json**. En dicho paquete configuraremos las dependencias necesarias para nuestra aplicación.

Ejemplo de creación de un paquete package.json:

Más información sobre el contenido que se puede poner en package.json: https://www.npmjs.org/doc/json.html

```
{
        "name":"chat",
        "version":"0.0.1",
        "private":"true",
        "dependencies":{
                "socket.io":"0.9.16",
```

```
                "express":"4.0.0"
        }
}

# El hacerlo así tiene mayores ventajas además de las de indicar las versiones
```

Cómo averiguar las versiones de socket.io y express

A la hora de cubrir el paquete package.json necesitaremos indicar que versiones queremos instalar en las dependencias, para nuestra aplicación. Para ello una forma de averiguar las últimas versiones disponibles es a través de:

```
npm info socket.io version

npm info express version
```

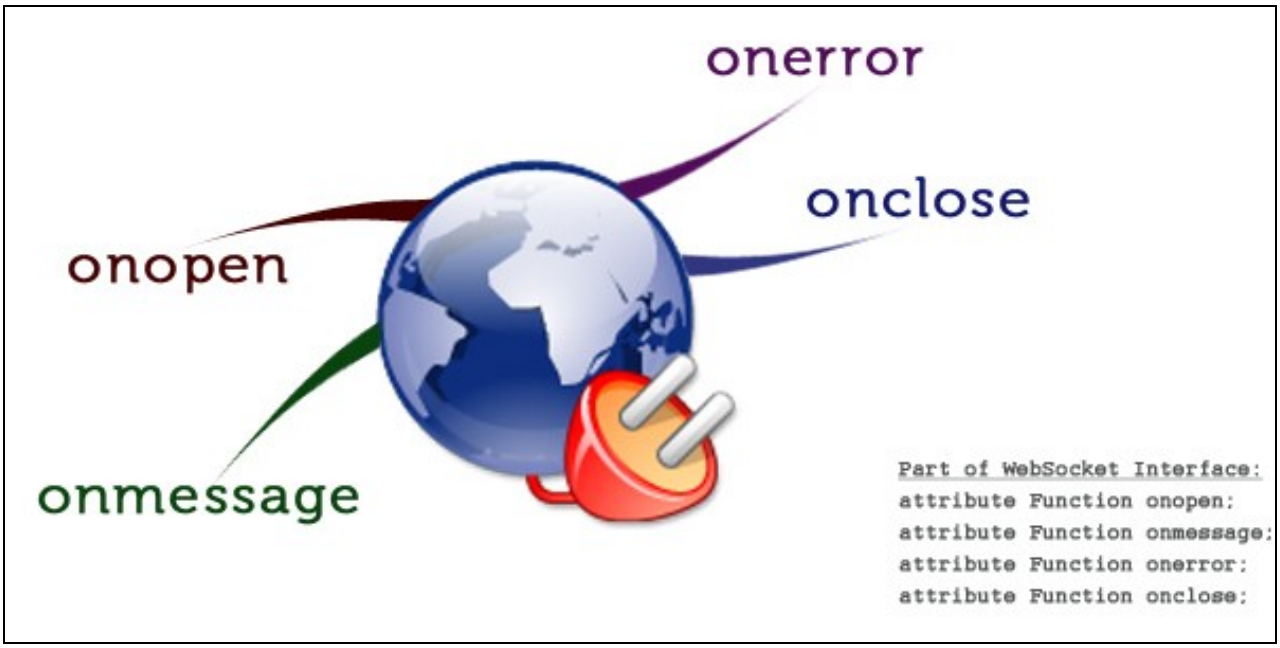
Instalación de los paquetes requeridos en la aplicación

Una vez creado el paquete **package.json** lo único que tendremos que hacer para realizar la instalación automática de los paquetes requeridos en la aplicación es: **npm install**

```
# Ejecutar este comando dentro de la carpeta dónde tenemos el paquete '''package.json''':

npm install
```

Aplicación de chat con websockets utilizando Node.js, express y socket.io

Eventos Websocket



Fichero package.json

Este fichero se utiliza para instalar las dependencias del servidor de Node.js.

Guardar este fichero en la carpeta dónde se encuentra **server.js** y una vez guardado, ejecutar desde la línea de comandos: **npm install**

```
{
        "name":"wschat",
        "version":"1.1.0",
        "private":"true",
        "dependencies":{
                "socket.io":"0.9.x",
                "express":"4.0.x"
        }
}
```

Fichero server.js

```
// Antes de ejecutar este código tendremos que instalar sus dependencias con:
// npm install
// Atención: asegurarse de tener el fichero package.json para que sepa que módulos tiene que instalar el npm.

// Puerto en el que escuchará nuestro servidor:
var puerto = 8080;

// Cargamos el módulo de express en un objeto al que llamamos moduloexpress.

var moduloexpress = require('express');

// Creamos una nueva aplicación sobre la variable app, llamando a () sobre el objeto moduloexpress

var app = moduloexpress();

// Se puede hacer todo en una única línea con:

//var app = require('express')();

// Gracias al framework express podremos más adelante gestionar las peticiones http

// Para instalar socket.io es necesario que tengamos instanciado previamente un servidor http.
// Cargamos el módulo http y creamos un servidor con createServer()
// Como parámetro se le pasa una función con 2 parámetros (req,resp) o en este caso se le pasa
// una instancia de express, que es la forma sencilla de procesar las peticiones que lleguen al servidor.

var servidor = require('http').createServer(app);

// Por último queda cargar el módulo de socket.io y pasarle el servidor como parámetro.

var io = require('socket.io').listen(servidor);

// Comenzamos a aceptar peticiones en el puerto e ip especificadas:   .listen(puerto,[IP])
// Si no se indica IP, lo hará en todas las interfaces.

servidor.listen(puerto);

// Mostramos mensaje en la consola de Node.js:

console.log('Servidor de websockets escuchando en puerto: ' + puerto);

// Mediante express gestionamos las rutas de las peticiones http al servidor.
// Cuando  alguien haga una petición a la raiz del servidor web, le enviamos un texto de información.
// También se le podría enviar un fichero con  res.sendfile(__dirname + '/index.html');

app.get('/', function(req, res) {
    res.end('Servidor de websockets para DWEC IES San Clemente.');
});


////////////////////////////////////////////////////////////////////////////
```

```
////////////  PROGRAMACION DE LOS EVENTOS QUE GESTIONA EL SERVIDOR  /////////////////
//  https://github.com/LearnBoost/socket.io/wiki/Exposed-events
/////////////////////////////////////////////////////////////////////////////////

// Vamos a programar las respuestas a los eventos que se pueden producir en el servidor.
//
// Primero tendremos que gestionar si se está produciendo una nueva conexión de websockets al servidor:
// lo haremos programando el evento 'connection' sobre io.sockets
// El argumento 'socket' de la función callback será utilizado en las futuras conexiones con el cliente.
// OS recuerdo que un socket en términos de Internet, queda definido por un par de direcciones IP
// (local y remota), un protocolo de transporte y números de puerto local y remoto.

// Eventos en servidor: 'connection','anything','disconnect','message')
// Ver la url https://github.com/LearnBoost/socket.io/wiki/Exposed-events
// para información detallada.

io.sockets.on('connection', function(socket)
{
    // Cada vez que se conecta un cliente mostramos un mensaje en la consola de Node.
    console.log('++++ Nuevo cliente conectado ++++');

    // A través del objeto socket que escribimos en la función de callback
    // gestionamos el intercambio de mensajes con el cliente.
    //
    // Nombres de eventos reservados para ese socket:'message', 'disconnect', 'mispropios-eventos'

    socket.on('message', function(datosrecibidos)
    {
        // Al recibir un mensaje lo retransmitimos al resto de usuarios. Ver opciones en:
        // https://github.com/LearnBoost/socket.io/wiki/How-do-I-send-a-response-to-all-clients-except-sender%3F
        io.sockets.emit('message', datosrecibidos);
    });

    socket.on('disconnect', function()
    {
        console.log('>>>> Se ha desconectado un cliente.');
    });

    // A parte de esos eventos reservados podemos programar nuestros propios eventos con el nombre
    // que deseemos. Por ejemplo socket.on('teletienda',function()..

    socket.on('teletienda', function(datosrecibidos) {

        // Usamos io.sockets.emit() para retransmitir datosrecibidos a todos
        // los que escuchan el evento 'teletienda'
        io.sockets.emit('teletienda', datosrecibidos);
        // Si queremos retransmitir a todos los que escuchan el evento 'teletienda'
        // excepto al remitente del mensaje usaremos socket.broadcast.emit()
        // socket.broadcast.emit('teletienda', datosrecibidos);
        // Más info en: https://github.com/LearnBoost/socket.io/wiki/How-do-I-send-a-response-to-all-clients-except-sender%3F
    });

    socket.on('avisos', function(datosrecibidos)
    {
        io.sockets.emit('avisos', datosrecibidos);
    });

});


// Para ejecutar el servidor de websockets con Node.js escribir desde el terminal:
node server.js

// Si realizamos cambios en el código fuente del servidor de Node, tendremos que parar y volver a arrancar el servidor.
// Una forma de que se haga de forma automática cuando detecta el cambio es usar nodemon
// Instalamos Node Monitor con:
npm install nodemon -g

// Para arrancar el servidor lo haremos con:
nodemon server.js

// A partir de ahora el servidor de Node cuando detecte un cambio en el código fuente se reiniciará de nuevo.
// Para pararlo lo haremos con CTRL+C
```

Fichero index.html

Este fichero contiene el código de una página cliente que se conecta al servidor de websockets.

Da por supuesto que el servidor de websockets está en nuestro propio servidor localhost.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8" />
    <title>Chat con Websockets versión 1.1</title>
    <style>
        #chat{
            height:500px;
            width:600px;
            border:solid black;
            border-style:groove;
            float:left;
            overflow:scroll;
            padding:2px;
        }

        #formulario{
            clear:both;
        }

        #notificaciones{
            width:500px;
            vertical-align: bottom;
            -moz-border-radius: 15px;
            border-radius: 15px;
            text-align: center;
            color:white;
            display:none;
            float:left;
            padding: 2px 0;
        }
        img{
            width:580px;
        }
    </style>
    <script src="jquery.js"></script>
    <script src="http://localhost:8080/socket.io/socket.io.js"></script>

    <script>
    $(document).ready(function()
    {
        nick = '';

$("#notificaciones").html("Intentando conexión al servidor de websockets. Por favor espere....").css("background", "orange").fadeIn(3
        var socket = io.connect('http://localhost:8080');

        // Programamos eventos sobre el objeto socket.
        // Eventos que se pueden programar sobre el socket cliente.
        // https://github.com/LearnBoost/socket.io/wiki/Exposed-events
        socket.on('connect', function()
        {
            notificar("Conectado correctamente al servidor de websockets.", "normal");
        });

        socket.on('reconnecting', function(datos)
        {
            notificar("Error: conexión perdida con servidor, intentando reconexión....", "error");
        });

        socket.on('reconnect_failed', function(datos)
        {
            notificar("Error: !! No se puede conectar al servidor de websockets. !!", "error");
        });

        // Los mensajes enviados con socket.send() van por defecto al evento 'message' del socket.
        // Por eso programamos socket.on('message',...
```

```
// socket.on('message', function(datos)
socket.on('teletienda', function(datos)
{

    // Evitamos XSS en los datos recibidos tanto de usuario como de texto.
    datos.usuario=datos.usuario.replace(/<script/g, '');
    datos.texto=datos.texto.replace(/<script/g, '');

    patron = /jpg|jpeg|gif|png/g;
    if (datos.texto.match(patron))// se ha mandado una imagen
        $("#chat").append('<strong>' + datos.usuario + '</strong> envió esta imagen:<br/><img src="' + datos.texto + '"/> <br
    else
        $("#chat").append('<strong>' + datos.usuario + '</strong>: ' + datos.texto + '<br/>');

    // $("#chat").append(datos + '<br/>');
    // Si hemos enviado un objeto con socket.emit() lo gestionaríamos así.
    // $("#chat").append('<strong>'+ datos.usuario + '</strong>: ' + datos.texto + '<br/>');

    // Para conseguir que haga auto-scroll
    // $("#chat").prop("scrollHeight") contiene la altura de la ventana de scroll, no sólo lo que se ve.
    // $("#chat").height() contiene la altura de la ventana de chat.
    // scrollTop contiene la posición Y del scroll (parte superior)
    $("#chat").animate({scrollTop: $("#chat").prop("scrollHeight")}, 200);
});

// Programamos eventos sobre el formulario.
$("#formulario").submit(function(evento)
{
    evento.preventDefault();

    if (nick != '')
    {

        ////////////////////// MUY IMPORTANTE ////////////////////////////
        // Enviamos un mensaje al servidor con socket.send(texto);
        // socket.send() se usa para enviar textos o strings JSON  (hay que hacerles el encode previamente)
        // única y exclusivamente al evento 'message'
        // Se gestionará lo recibido en socket.on('message',...)
        // Por ejemplo:
        /*
         $("#formulario").submit(function(evento)
         {
         evento.preventDefault();
         socket.send('<strong>' + nick + '</strong>: ' + $("#casillatexto").val());
         $("#casillatexto").val('');
         $("#casillatexto").focus();
         });
         */
        // Si queremos enviar objetos la forma más cómoda es con socket.emit('evento',objeto)
        // Con socket.emit hay que indicar obligatoriamente el evento al que enviamos  datos.
        // Ejemplo:
        // socket.emit('teletienda',{usuario: nombre, texto: $("#casillatexto").val()} );
        // En este caso enviamos al evento 'teletienda' aunque también se podría enviar a 'message'
        // Si lo enviamos a 'message' podemos gestionar su recepción con socket.on('message'
        // en otro caso lo haremos con socket.on('nombre-del-evento'),por ejemplo: socket.on('teletienda',...

        // Enviamos al evento/canal específico 'teletienda' un objeto compuesto de usuario y texto.
        if ($("#casillatexto").val() != "")
        {
            socket.emit('teletienda', {usuario: nick, texto: $("#casillatexto").val()});
        }


        $("#casillatexto").val('');
        $("#casillatexto").focus();

    }
    else
    {
        do
        {
            nick = prompt("Introduzca su nick de usuario", "Anónimo");
        } while (nick == '');
```

```
                $("#usuario").html(nick);
                socket.emit('teletienda', {usuario: nick, texto: "<strong>Acaba de entrar a la sala de chat.</strong>"});
                $("#casillatexto").val('');
                $("#casillatexto").focus();
            }
        });


        function notificar(texto, tipo)
        {
            if (tipo == "normal")
            {
                $("#notificaciones").html(texto).css("background", "green").
                        fadeIn(300).delay(1000).fadeOut(1000);
            }

            if (tipo == "error")
            {
                $("#notificaciones").hide().html('').html(texto).css("background", "red").fadeIn(300);
            }
        }


    });
    </script>
</head>
<body>
    <h1>Multi-user chat con websockets mejorado con eventos específicos.</h1>
    <h3>Servidor en VM Debian con express + socket.io + Node.js</h3>
    <h5>Se permite el envío de imágenes al chat. Copie y pegue la URL de la imagen en la casilla "Enviar".</h5>
    <div id="chat"></div>
    <form id="formulario">
        <input size="50" id="casillatexto"/>
        <input type="submit" value='Enviar'/><br/>
        Conectado con el nick:<strong> <span id='usuario'></span></strong>
    </form>

    <div id="notificaciones"></div>
</body>
</html>
```

Cambios en el módulo Express en la nueva versión 4.0

http://scotch.io/bar-talk/expressjs-4-0-new-features-and-upgrading-from-3-0

Cómo hospedar tu proyecto de Node.js en Cloud Node

Cuando tenemos listo nuestro servidor de Node.js tenemos que pensar dónde hospedar dicho servicio.

Si disponemos de un servidor virtual podemos ejecutar allí nuestro servidor de Node.js pero en otro caso tenemos que buscar algún sistema que nos permita tener en ejecución nuestro servidor. Tenemos varias opciones, por ejemplo Openshift.com o también [1]

Os voy a comentar como hacerlo en **Cloud Node** ya que nos ofrece de forma gratuita la posibilidad de subir nuestro propyecto en Node y tenerlo en ejecución de forma indefinida.

Cloudno.de
Lo primero que tenemos que hacer es registrarnos en su web Cloudno.de

Si queremos acelerar la activación nos recomienda publicar un tweet a su canal de Twitter. Generalmente en 2 días o así nos facilitan el acceso.

Requerimientos previos de software

Antes de poder trabajar con Cloudno.de tendremos que instalar en nuestro equipo un sistema de control de versiones (Git) y Node.js

- Control de versiones Git: http://git-scm.com/downloads
- Node.js: http://nodejs.org/download/

Configuración de nuestra clave pública SSH en cloudno.de

Para poder enviar el código fuente de nuestras aplicaciones necesitaremos enviar una clave pública SSH que generaremos en nuestro equipo a nuestra cuenta de cloudno.de

Cómo generar la clave pública SSH

- **Desde Windows y LINUX**

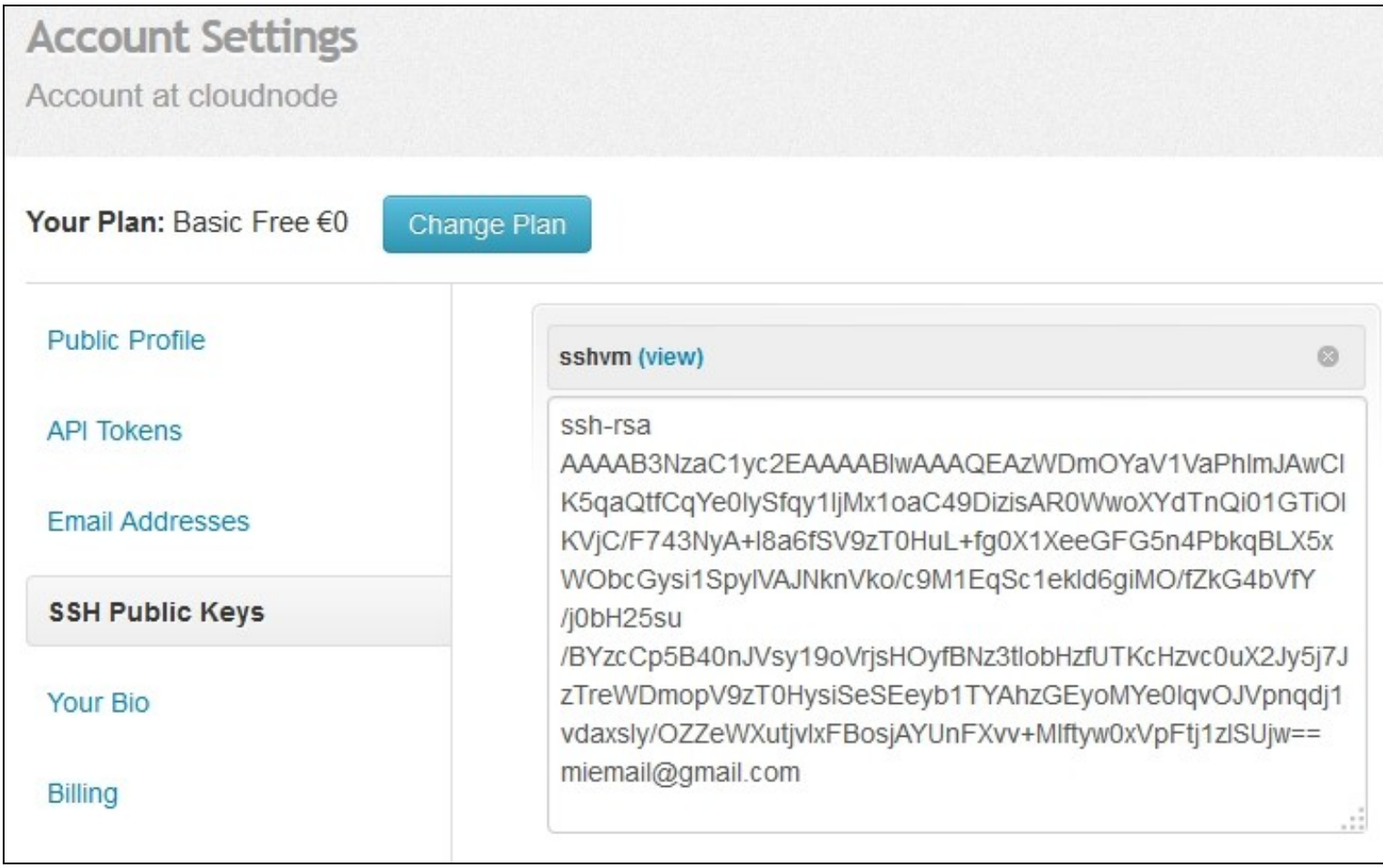
Abrimos una SHELL de GIT con el botón derecho del ratón encima de cualquier carpeta **Git Bash Here** y desde la línea de comandos teclearemos lo siguiente:

```
# Desde la línea de comandos teclearemos:
ssh-keygen

# Y pulsaremos ENTER 3 veces. (La contraseña que pide es opcional.)
# Ya tenemos en la carpeta que nos indica el fichero id_rsa.pub con
# nuestra clave pública SSH.
```

Éste es el resultado de la ejecución dentro de la shell de Git:

- Ahora tendremos que **subir la clave pública a la página web de cloudno.de**.
- Para ello entraremos en **Account** -> **SSH Public Keys** y allí pulsaremos en **Add your SSH Public Key** o en **View** si ya tenemos una clave para cambiarla.
- **- ATENCIÓN IMPORTANTE -** copiar el contenido del fichero **id_rsa.pub** abriendo dicho fichero con el bloc de notas. **No copiar el último salto de línea**.
- **Pegaremos** en ese área de texto el contenido del fichero **id_rsa.pub** y **muy importe sustituir el e-mail** que aparece en la clave SSH y poner el que utilizais en vuestra cuenta de cloudno.de.
- Si cuando lo pegamos el **Key fingerprint** es exactamente igual al que nos dió al generar la clave pública (con el ssh-keygen), entonces hemos pegado correctamente en cloudno.de nuestra clave.

Ejemplo de contenido del fichero id_rsa.pub con el e-mail corregido y que pegaríamos dentro de la casilla de texto en cloudno.de:



Configuración de cloudno.de en nuestro sistema

Una vez instalados los requerimientos anteriores, tendremos que instalar **cloudnode**. Para ello usaremos el gestor de paquetes de node (npm). Esta aplicación es necesaria para poder registrarnos y trabajar desde la línea de comandos con cloudno.de

```
# Desde la línea de comandos teclearemos
# -g es para que lo instale globalmente, o localmente si no ponemos esa opción

npm install cloudnode-cli -g
```

Una vez instalado cloudnode tenemos que **configurar la cuenta de usuario y contraseña** que se utilizarán para subir cosas al servidor. Esta parte solamente hay que realizarla una única vez. Para el resto de aplicaciones que hagamos ya quedará configurado.

```
# Desde el shell del SISTEMA (Ejecutar->CMD en Windows) teclearemos:

cloudnode user setup <usuario> <password>

# El usuario se corresponde con el usuario de cloudno.de
# Password es la API Token que se puede consultar en "Tu cuenta -> API Tokens"
# Por ejemplo:

C:\Users\rafa>cloudnode user setup profesorveiga ls0TdTdsRo

# SI NOS DA ERROR EN WINDOWS AL EJECUTAR CLOUDNODE TECLEAREMOS:
set home=%userprofile%

# Y a continuación repetimos el comando: cloudnode user setup...
# Y obtendremos un mensaje como:

C:\Users>cloudnode user setup profesorveiga ls0TdTdsRo
cloudnode info verifying credentials
cloudnode info user verified..
cloudnode info writing user data to config

# Una vez hecho ésto si queremos ver todas las opciones de cloudnode,
```

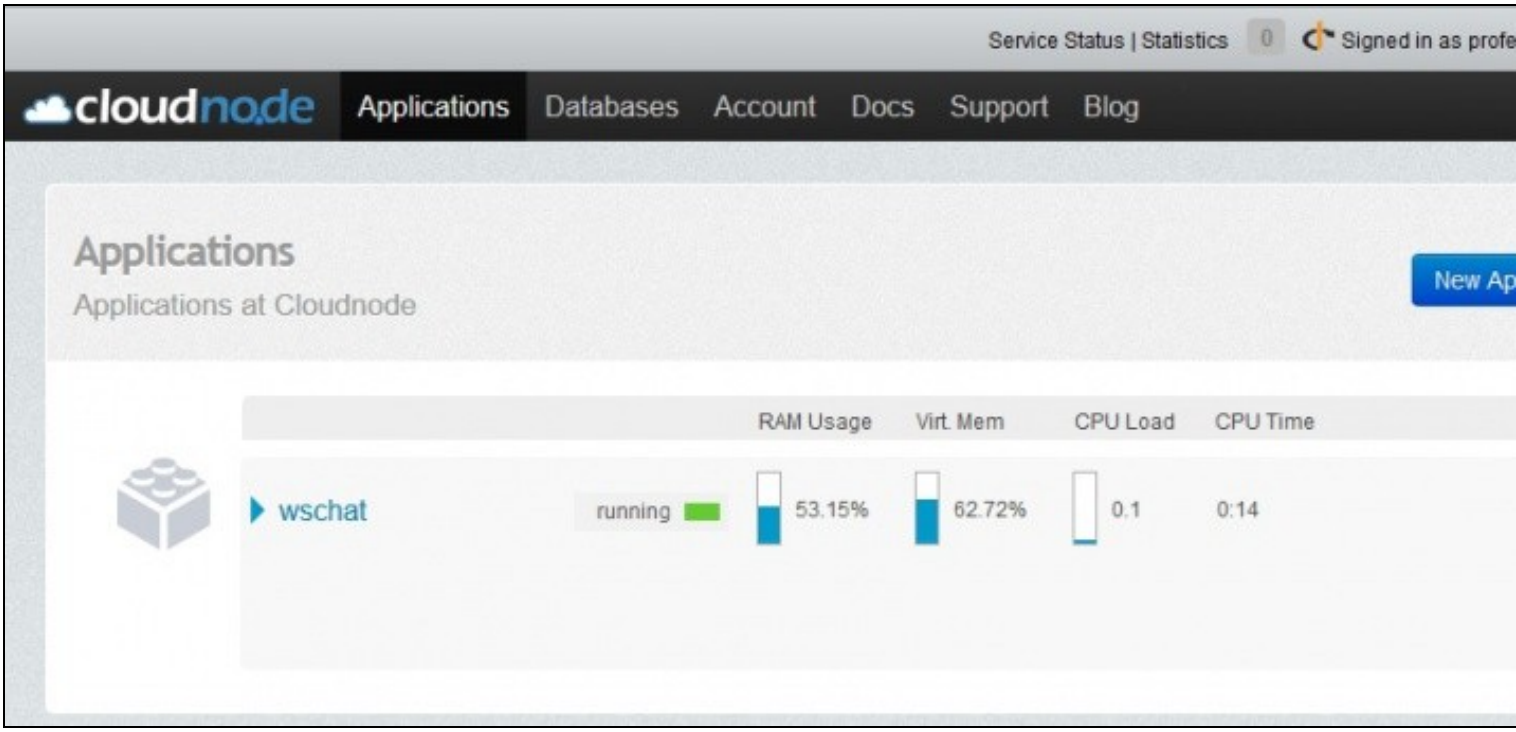
```
# teclearemos:

cloudnode
```

Creación de una aplicación en servidor cloudno.de

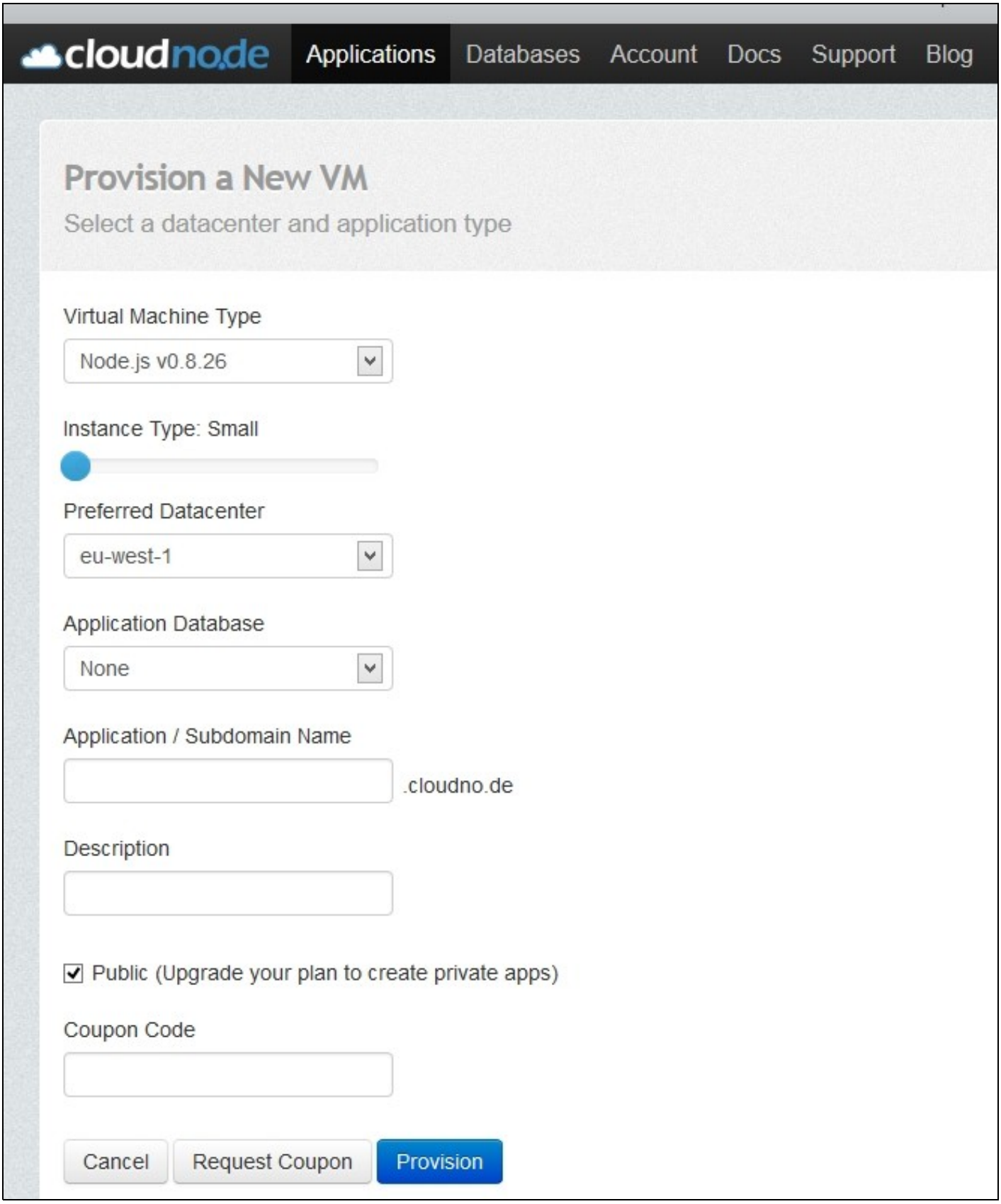
- Tendremos que entrar en la página http://cloudno.de y crear una aplicación:

- Accederemos usando el sistema con el que hayamos creado nuestra cuenta:

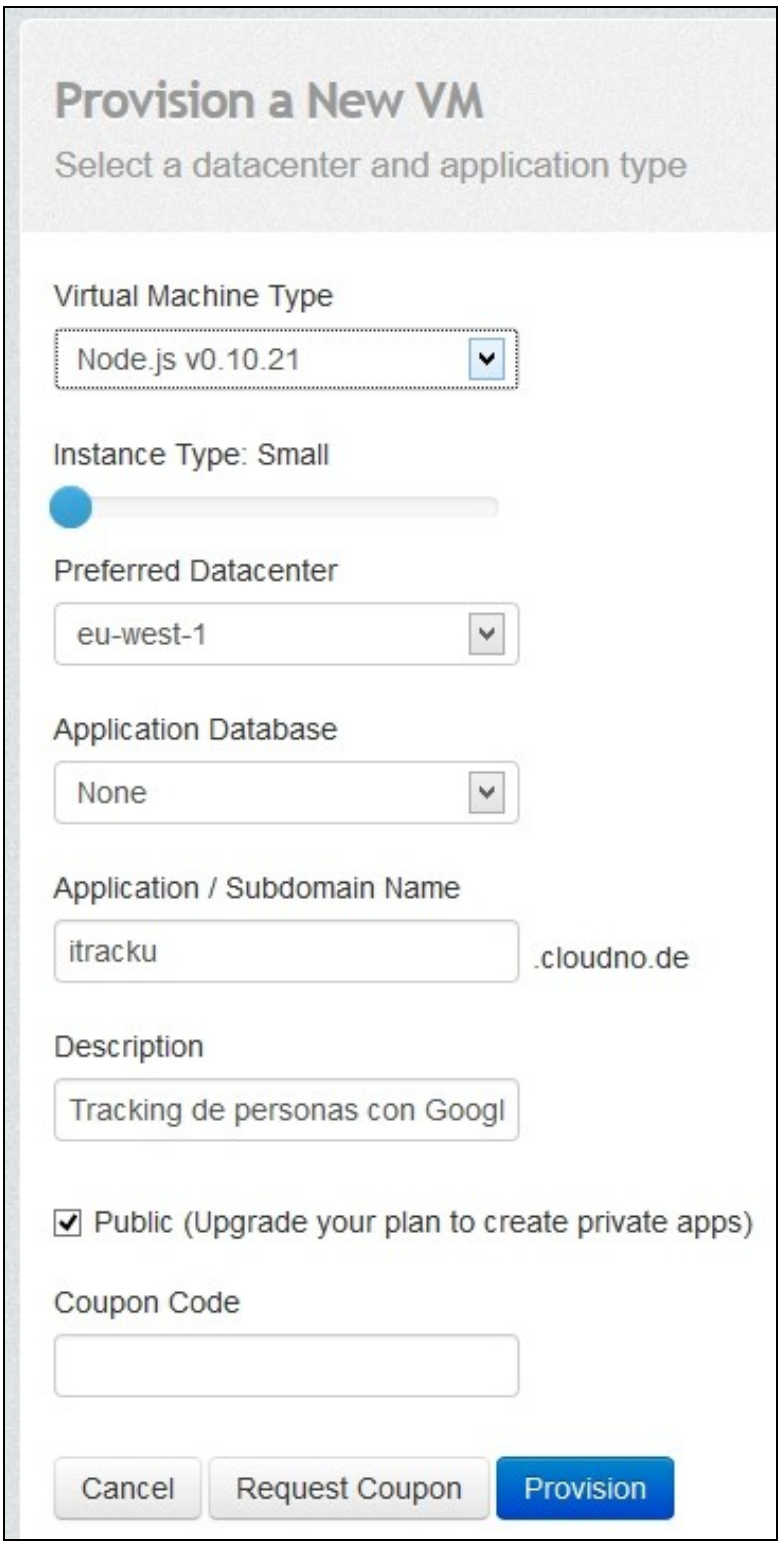
- Una vez dentro podremos ver las aplicaciones que tenemos instaladas, en este caso vamos a crear una.



- Creamos la aplicación pulsando en el botón azul "**New Application**" :

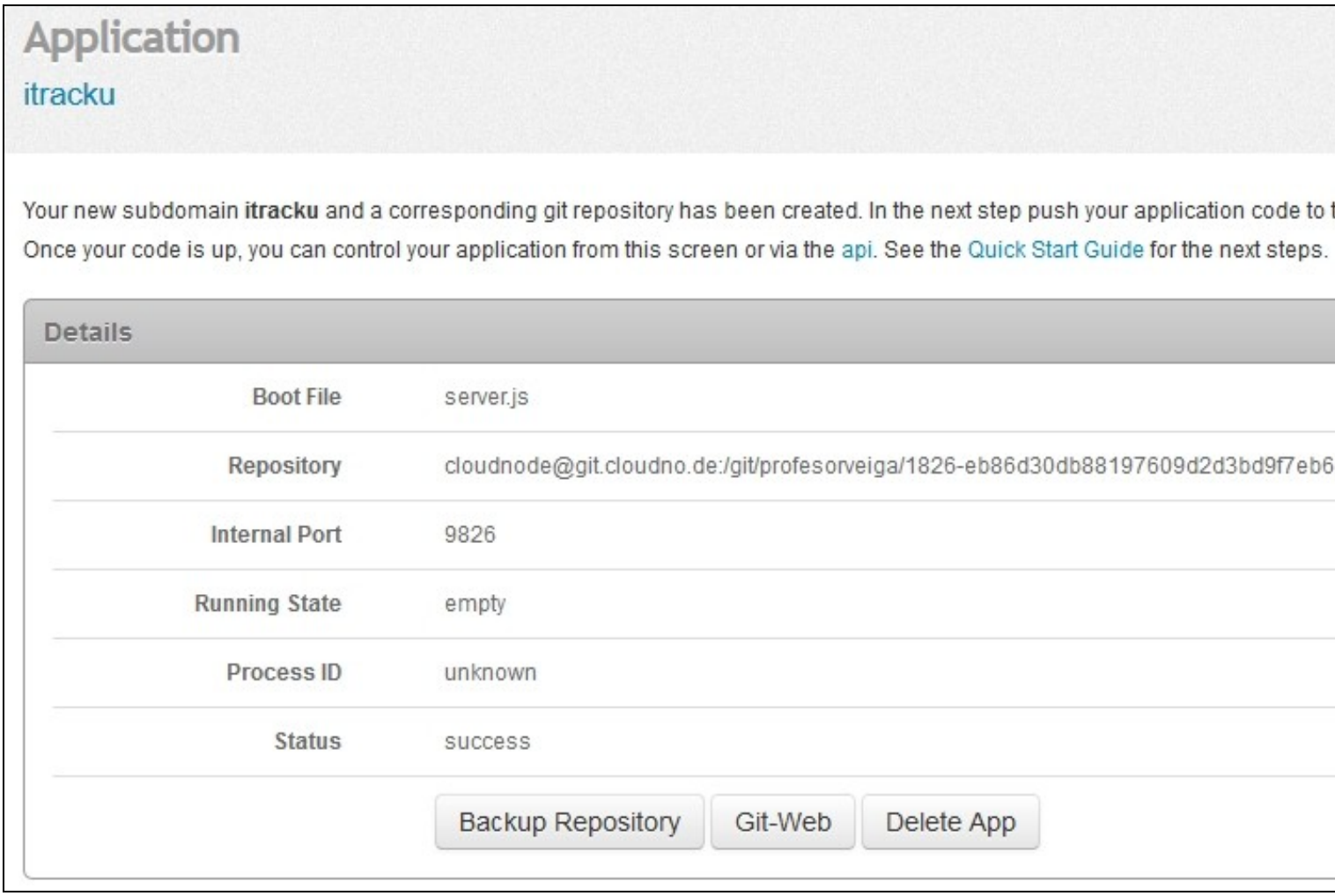


- Cubrimos los datos de nuestra aplicación, así como el dominio deseado.



- Pulsamos el botón **Request Coupon** y a continuación en **Provision**.

- Se nos mostrará la información de la aplicación recién creada:



Creación de la aplicación Node.js en local

Pasos a seguir para inicializar y crear nuestra aplicación en local.

```
# En LINUX:
cloudnode app init nombreaplicacion

# Ese comando va a crear una carpeta llamada nombreaplicacion y realizar
# todos los pasos necesarios

# En WINDOWS:
# El comando anterior no funciona así que los pasos iniciales hay que hacerlos uno a uno:

# 1.- Crear el subdirectorio para nuestra aplicación de Node.js

# 2.- Entramos dentro de esa carpeta e inicializamos un repositorio GIT:
git init

# 3.- Seguimos dentro de la misma carpeta y clonamos el repositorio remoto
#  y la url remota (tenemos esos datos en la aplicacion en Details->Repository):
# Usamos la instrucción git clone para clonar lo que hay en cloudno.de:
git clone cloudnode@git.cloudno.de:/git/profesorveiga/1826-eb86d30db8819763242d3bd9f23eb6e46.git

# 4.- Entramos en la carpeta que nos ha aparecido al hacer el clone
# y allí dentro es donde creamos un fichero server.js con el contenido de nuestro servidor de Node.

# Si estamos haciendo una aplicación de websockets el PUERTO DE NUESTRO SERVIDOR
# tendremos que configurarlo al que se nos indica en Internal Port
# dentro de los detalles de la aplicación en cloudno.de
# Aunque teóricamente no es necesario ya que cloudno.de sobreescribirá los puertos que hemos configurado.

# 5.- Instalamos los módulos uno a uno que sean necesarios en nuestro
# proyecto con npm install socket.io, etc.. .. o bien
# creando el package.json y ejecutando npm install

# 6.- Por último nos quedaría configurar los datos por defecto para los commit:
git config --global user.email "miemail@gmail.com"
git config --global user.name  "Antonio Martinez"
```

Envío de la aplicación a cloudnode

Si queremos enviar nuestra aplicación al servidor de cloudno.de tendremos que hacerlo usando git, mediante un commit.

```
# 7.- Añadimos las modificaciones que hemos hecho a git:
git add .

# 8.- Confirmamos los cambios haciendo un commit:
git commit -am "Envio inicial"

# 9.- Por último sólo nos falta enviar los cambios con git push a cloudno.de:
git push origin master

# Cada vez que hagamos modificaciones en nuestro código fuente en local
# tendremos que realizar de nuevo los pasos: 7,8 y 9 para enviar los cambios a cloudno.de

# Una vez enviados los cambios la aplicación ya estará funcionando en cloudno.de en la URL de la aplicación.
# En nuestro ejemplo la URL dónde está funcionando nuestro servidor es:
http://itracku.cloudno.de/
```

Cambios a realizar en código HTML para adaptarlo al servidor en cloudno.de

Cuando tenemos la aplicación funcionando en cloudno.de, lo único que nos falta por hacer es modificar el código HTML para que hagan uso de ese servidor.

Lo más recomendable también sería que en el servidor de node a nivel interno usáramos los puertos que se nos indican en los detalles de cada aplicación en la web en la sección de **Internal Port**.

Ejemplo de cambios:

```
# Tendremos que modificar la dirección del servidor de websockets en nuestras páginas
# para que apunte al nuevo servidor, en este ejemplo sería:
http://itracku.cloudno.de/

# Ejemplo de código que cargaría el módulo de socket.io en nuestra página:
<script src="http://itracku.cloudno.de/socket.io/socket.io.js"></script>

# Ejemplo de conexión al servidor de websockets:
var socket = io.connect('http://itracku.cloudno.de');
```

-- Realizado por: Veiga (discusión) 19:32 30 mar 2017 (CEST)