ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

Titulación:

INGENIERO EN INFORMÁTICA

Título del proyecto:

Herramienta de gestión y planificación de robots web basado en la arquitectura MEAN

Carlos Martínez González

Tutor: José Ramón González de Mendívil

Pamplona, 24 de Junio de 2015

Resumen

Durante este proyecto se va diseñar y desarrollar una herramienta para la gestión de robots web. Para el desarrollo de la misma se ha optado por usar una plataforma relativamente nueva que no hemos aprendido durante la carrera. Esta plataforma conocida como MEAN está formada por los siguientes componentes: MongoDB, Express, AngularJS, y NodeJS.

El objetivo de la herramienta, es proporcionar un sistema que nos permita gestionar distintas configuraciones de robots web, y planificarlos para su posterior ejecución. La configuración de los robots y de sus planificaciones tiene que ser lo más sencilla y simple posible.

Para la gestión de las configuraciones de los robots y de las planificaciones de los mismos se ha desarrollado una interfaz web. Esta interfaz permite al usuario crear nuevas configuraciones, planificarlas de distintas formas, y visualizar los logs de las ejecuciones de los robots web.

Debido a que la escalabilidad era uno de los objetivos más importantes de la herramienta, se ha separado la parte de gestión de robots de la parte de ejecución de robots, creando así un sistema distribuido. Para que este sistema funcione, se ha diseñado un protocolo que permite la comunicación entre las distintas partes de la herramienta. De esta manera, la tarea de los robots (que consume muchos más recursos que las tareas de gestión) queda totalmente separada. Además, la parte de gestión de robots puede conectarse a cualquier número de servidores en los que se encuentran los robots, permitiendo así repartir el trabajo de los robots entre varias máquinas.

Además, en la parte de ejecución de robots se ha desarrollado un sistema que permite desplegar de forma muy sencilla robots en JavaScript. Como JavaScript nos permite ejecutar comandos del sistema en modo seguro, con un programa en JavaScript se podría lanzar un robot escrito en cualquier lenguaje.

De forma adicional a la herramienta, se ha desarrollado una pequeña aplicación demo para enseñar y demostrar el funcionamiento de la plataforma MEAN, y se ha probado el rendimiento de NodeJS como servidor.

Índice

Motivación

A lo largo de estos años hemos desarrollado, dentro del Grupo de sistemas distribuidos, diferentes sistemas de captura automática de datos procedentes de páginas web. Estos sistemas se han basado en la construcción de Robots web: programas que acceden de forma automática a diferentes páginas web para recoger su información con el objetivo de su almacenamiento, clasificación y procesamiento.

En un primer trabajo [1] se estudiaron las posibles técnicas para la construcción de este tipo de programas. Estas técnicas se basaban tanto en sistemas de web-testing (Selenium [2], Watir [3]) como en primitivas de más bajo nivel, como las ofrecidas por diferentes lenguajes de programación orientados a la web (Curl de Php [4], Url de Python [5]). Una de las cuestiones que aprendimos de estos primeros trabajos es que en algunas ocasiones es muy conveniente combinar diferentes métodos para abordar la extracción de datos sobre algunas páginas web, que podríamos clasificar como complejas, debido a la aparición de elementos ejecutados a través de javascript en sus diferentes facetas (ajax, jquery, etc).

A medida que fue creciendo la necesidad de construcción de un mayor número de robots web, también surgió la necesidad de tener un mayor control sobre la ejecución de dichos robots. Dentro de uno de los proyectos en los que participamos, se requería controlar la ejecución de aproximadamente 50 x 10 robots web que extraían datos relativos a los supermercados de España, almacenando los datos en simples ficheros csv para su posterior tratamiento en una base de datos. Estos tratamientos correspondían a procesos de limpieza, catalogación e identificación de productos idénticos entre distintos supermercados, para facilitar la comparación de precios (en trabajo de Aitor [6] se presenta una aproximación a la catalogación semiautomática y en trabajo de Ion [7] se estudia el problema de la localización de productos idénticos y sus limitaciones).

Así, en un segundo trabajo [8] desarrollamos un sistema de planificación y control de robots web basado en una planificación mediante las utilidades propias del sistema operativo (cron) extraída de una definición almacenada en una Base de datos MySQL. En esta misma base de datos se depositaban las informaciones relativas a la ejecución: parámetros de configuración de los robots, tiempos de ejecución, resultados de los mismos, logs de error, etc.

Por otro lado, para ampliar el conjunto de técnicas para la construcción de robots se estudió la construcción de robots mediante la librería Scrappy de Python [9]. Esta librería permite controlar el número de peticiones concurrentes que actúan a la vez sobre un mismo sitio web. Por otra parte, también se han estudiado e implementado soluciones al problema de la sobrecarga que puede resultar en un sitio web por la utilización continuada de un robot web actuando sobre dicho sitio. Estas soluciones basadas principalmente en el uso de “proxies” permiten repartir la carga de peticiones de un robot sobre un mismo sitio web.

A medida que hemos avanzado en estas líneas de trabajo también han surgido en los últimos años nuevos paradigmas de programación que permiten abordar la

definición de una “fábrica de robots” como un servicio web. Esta aproximación nos ofrece la ventaja de facilitar el despliegue del sistema de robots en PaaS [10] específicas, eliminando la necesidad de administración propia de máquinas en el laboratorio y garantizando un servicio de alta disponibilidad (esto último depende del SLA contratado, aunque la fiabilidad de ciertos sistemas es muy alta).

En los últimos años, la combinación de bases de datos NoSQL (Not Only SQL) y sistemas servidores con tecnologías no-bloqueantes ha recibido gran atención por parte de la comunidad de desarrolladores web; debido principalmente a sus buenas propiedades de tiempo de respuesta y escalabilidad horizontal de los datos persistentes. Entre estas nuevas tecnologías encontramos de forma destacada la pila “MEAN” (the MEAN stack)[11]: MongoDB, Express, AngularJS, NodeJS. La ventaja de esta aproximación es la utilización de un único lenguaje, Javascript, ejecutándose en cualquier lugar de las capas de la aplicación web. Por otro lado, como hemos indicado alrededor de estas tecnologías han surgido PaaS que facilitan en gran medida el despliegue de este tipo de aplicaciones en la Nube. Una de las plataformas de despliegue que facilitan la labor con MEAN es Heroku. Heroku ofrece un despliegue de la aplicación NodeJS de forma directa desde el entorno GIT [12]. Una de las promesas de Heroku es que la aplicación escala y tiene buen tiempo de respuesta, siempre y cuando sigas su manifiesto de 12 factores (The twelve factor [13]). Recientemente, nuestro grupo está participando en el desarrollo de una PaaS, Kumori [14], construida alrededor de NodeJS con un sistema de comunicación basado en 0MQ (en el trabajo de Raul[15] se puede encontrar una aplicación distribuida basada en los patrones de comunicación ofrecidos por 0MQ).

Volviendo al tema concreto de este Proyecto Fin de Carrera expondré los objetivos concretos que abarca el mismo:

- Desarrollar una plataforma de control y ejecución de robots web implementada con la tecnología tipo MEAN: especificación de la funcionalidad, diseño e implementación de la plataforma.
- Estudiar el despliegue de la plataforma en un sistema PaaS tipo Heroku.
- Realizar las correspondientes pruebas de funcionamiento que ratifiquen la viabilidad de la aproximación ofrecida.

El desarrollo de la memoria del proyecto consiste en los siguientes capítulos:

En el Capítulo 1 se realizará una aproximación sencilla a los conceptos que se encuentran detrás de MEAN. En concreto las facilidades de MongoDB para el almacenamiento persistente; y la filosofía que se encuentra en la aproximación de NodeJS. También se presenta la filosofía de “12 factores” y los conceptos asociados al despliegue en la nube (Heroku).

En el Capítulo 2 se presenta la funcionalidad implementada, a modo de manual de uso, de la “fábrica de robots” denominada The Webby Scrappy.

En el Capítulo 3 se presenta la arquitectura software, sus componentes, funcionalidad, interacción, y los detalles de la implementación de Webby Scrappy.

En el Capítulo 4 se mostrarán los resultados obtenidos al usar la herramienta. De forma adicional se realizarán pruebas de la capacidad de NodeJS como servidor.

En el Capítulo 5 se explicarán las conclusiones y experiencias obtenidas en el uso de la plataforma MEAN, ya que esta plataforma es totalmente distinta a las plataformas estudiadas durante la carrera. Además, se explicarán distintas líneas por las que se podría seguir avanzando el proyecto, así como posibles mejoras y cambios de enfoque del mismo.

Finalmente, en el capítulo 6 se listarán las fuentes de información usadas para la realización del proyecto y de la memoria.

Capítulo 1. MEAN y PaaS

En este primer capítulo explicaremos la plataforma MEAN, las ventajas y desventajas que presenta ante otros sistemas, y cómo usarlo para desarrollar una pequeña aplicación web. También explicaremos que es un Paas, y las ventajas que obtenemos al usarlo. Además también explicaremos unos principios fundamentales para el desarrollo de aplicaciones escalables.

1.1. MEAN

Una de las primeras plataformas de código abierto para el desarrollo de aplicaciones web que ganó fama fue la plataforma LAMP [16]: Linux como sistema operativo, Apache como servidor web, MySQL para la base de datos y Perl, Python o PHP como lenguaje de programación para la lógica de la aplicación.

El problema con estas tecnologías es que fueron desarrolladas por separado, de forma independiente, y no estaban pensadas para funcionar juntas. Fue más adelante, cuando un ingeniero informático decidió unir estas tecnologías y formar este tipo de plataformas.

Actualmente está surgiendo con mucha fuerza una nueva plataforma para el desarrollo de aplicaciones web. Esta nueva plataforma es conocida como MEAN, y consta de MongoDB, Express, AngulaJS y NodeJS. La plataforma MEAN nos permite usar el mismo lenguaje de programación en todas las partes de la aplicación, desde la parte del cliente, hasta la parte de servidor. El lenguaje usado es JavaScript [17], un lenguaje de programación de alto nivel que nos permite el desarrollo rápido de aplicaciones.

Si comparamos la plataforma LAMP con la plataforma MEAN podemos observar varias diferencias. La primera y la más importante, es que en la plataforma MEAN se ha sustituido el sistema operativo por NodeJS, un intérprete de JavaScript. Existen versiones de este intérprete para casi todos los sistemas operativos, facilitando así la portabilidad de la aplicación de un sistema operativo a otro.

Otra diferencia es que el servidor usado, Express, es un módulo programado en NodeJS. Esto hace que NodeJs y Express se fusionen formando un único paquete que se encarga de realizar todas las tareas relacionadas con la lógica de la aplicación. Estas dos partes de MEAN sustituirían al servidor y al lenguaje de programación usado (PHP, Perl o Python).

Respecto a la base de datos, se pasa de un sistema SQL relacional, a un sistema noSQL orientado a objetos. MongoDB almacena todos sus datos en formato JSON, el formato nativo de JavaScript. Esto simplifica en gran medida las interacciones de la aplicación con la base de datos, permitiendo un desarrollo rápido de las mismas.

Para finalizar, la plataforma MEAN consta de AngularJS, un framework en JavaScript ideado para la creación de aplicaciones web dinámicas. Este framework sigue el patrón

de diseño MVC (modelo vista controlador), facilitando así la ordenación y estructuración del código.

Como se puede observar, esta nueva plataforma pone a nuestra disposición una serie de herramientas pensadas para trabajar las unas con las otras, facilitando y ayudando el desarrollo rápido de aplicaciones web. Además, permite el uso del mismo lenguaje de programación tanto en la parte del cliente, como en la parte del servidor, reduciendo así el tiempo de aprendizaje.

Para poder entender mejor los distintos conceptos y como se comunican las distintas partes de la arquitectura MEAN, se va a desarrollar y explicar una aplicación web sencilla, que en este caso va a ser un chat. Este desarrollo comenzará en el apartado de Express, ya que NodeJS es la base sobre la que se apoya.

1.1.1. NodeJS

NodeJS [18] es una plataforma basada en el *runtime* de JavaScript de Google Chrome (motor V8 de Google), para el desarrollo rápido de aplicaciones de red escalables. NodeJS sigue un modelo no bloqueante basado en eventos, lo que lo convierte en una plataforma ligera y eficiente, perfecta para aplicaciones en "tiempo real"[1] en sistemas distribuidos.

Haciendo algunas cuentas simples, observamos las ventajas que ofrece NodeJS frente a otros lenguajes usados en el lado del servidor, como por ejemplo Java o PHP. Cualquiera de estos dos lenguajes, al recibir una petición, crea un nuevo hilo para responder a la misma. Cada hilo puede necesitar fácilmente 2 MB de memoria RAM. Si tenemos un servidor con 16 GB de memoria RAM estaríamos limitados a 8000 hilos. Por otro lado, NodeJS, en vez de crear un nuevo hilo de ejecución, lo que hace es lanzar un evento dentro del proceso que está atendiendo la petición. De esta forma, se reduce el coste computacional y de recursos del sistema permitiendo mantener más conexiones simultáneas con los mismos recursos. En el capítulo 4 se mostrarán resultados acerca del rendimiento de un servidor en NodeJS.

Otra de las ventajas que ofrece, es que su arquitectura es no bloqueante, por lo que podemos realizar operaciones I/O de forma asíncrona. Esto nos permite realizar tareas de forma simultánea en un único proceso. De hecho, estas son las características que permiten hacer que los sistemas desarrollados en NodeJS sean escalables, aunque esto dificulte en algunos casos el diseño de las aplicaciones.

Respecto a JavaScript, el lenguaje usado para desarrollar aplicaciones en NodeJS, es un lenguaje interpretado, orientado a objetos, basado en prototipos, débilmente tipado y muy dinámico. Estas características lo convierten en el lenguaje perfecto para desarrollos rápidos.

[1] Para ser más precisos, deberíamos decir "en un tiempo de respuesta bajo", pero en el argot de los sistemas basados en eventos se tiende a abusar del término "en tiempo real" como una característica más.

Además de todo esto, NodeJS pone a nuestra disposición un sistema de módulos que nos permite organizar y mantener el código de forma más sencilla y eficiente. Para importar un módulo únicamente hay que llamar a la función de importación. Una de las ventajas que presenta este sistema de importación frente a otros sistemas es que permite importar módulos de forma dinámica, permitiendo importar módulos durante la ejecución del programa en función de las condiciones.

Para agilizar aún más el desarrollo de aplicaciones con NodeJS, la propia plataforma tiene ya programados una gran cantidad de módulos. Entre ellos podemos encontrar el módulo HTTP, que nos permite realizar peticiones HTTP y crear servidores, del cual hablaremos más adelante. También tenemos el módulo Cluster, que nos permite crear programas multiproceso para aprovechar al máximo los recursos de la máquina, entre otros muchos módulos.

Además, también tiene un gestor de paquetes llamado npm (node package manager) [19], que nos permite declarar las dependencias de la aplicación e instalarlas de forma rápida y sencilla. Esta dependencias se declaran en un fichero llamado "packaje.json" y se descargan en el proyecto llamando simplemente al comando "npm install". Una vez descargadas se pueden referenciar mediante el sistema de importación de módulos que proporciona NodeJS.

Modulo HTTP

Una de las grandes ventajas que ofrece NodeJS es la simplicidad y velocidad con la que se puede crear un servidor. Para esta tarea, la plataforma pone nuestra disposición el módulo HTTP y el módulo HTTPS (para conexiones seguras).

Para crear un servidor únicamente hay que llamar al método "createServer" del módulo HTTP. En cuatro líneas se puede crear un servidor que responda un "Hello World" a las peticiones web que le lleguen.

```
require('http').createServer(function(req, res){
   res.writeHead(200); // Write response code
   res.end('Hello World!\n'); // Write response body
}).listen(8080);
```

Como se puede ver en este ejemplo, importamos el módulo "http" usando la función "require". De ese módulo llamamos a la función "createServer" que devuelve una instancia de Server. Como parámetro de la función de creación pasamos la función anónima que se llamará cada vez que el servidor reciba una petición. En este caso la función anónima simplemente escribe la cabecera y el mensaje de la respuesta. Para terminar, al objeto que devuelve la función "createServer", le especificamos en que puerto y URL debe escuchar.

En el objeto "req", que es una instancia de http.IncomingMessage tenemos toda la información sobre la petición realizada al servidor, como por ejemplo los parámetros, los headers, etc... Por otro lado, en el objeto "res", que es una instancia de http.ServerResponse, tenemos los métodos necesarios para poder responder la petición.

Asincronía

NodeJS, por defecto, se ejecuta en un único proceso, aunque esto no implica que únicamente puede realizar una tarea al mismo tiempo. Para poder atender varias tareas al mismo tiempo, NodeJS utiliza un sistema de eventos. Durante la ejecución del hilo principal los eventos que ocurran se van almacenando en una lista. Cuando la tarea que se está ejecutando en hilo principal sale del mismo, se atienden los demás eventos y tareas.

En el ejemplo de la sección anterior, en el que se programaba un servidor que respondía un hola mundo, lo que estamos haciendo es crear un *listener* a un evento, y diciéndole lo que tiene que hacer cuando el evento se dispare. Una vez creado el *listener* el programa deja libre el hilo principal, permitiendo así responder a las peticiones que lleguen. Al dispararse un evento, este entra en la cola de eventos y al estar libre el hilo principal, éste pasa a ejecutarse inmediatamente.

Es importante tener en cuenta que si una tarea nunca abandona el hilo principal, los eventos que no se hayan atendido ya nunca serán atendidos. Para provocar este problema basta con atender un "while(true)" o una tarea muy larga en cualquier parte del ejemplo de la sección anterior. Por ejemplo, si lo añadimos después de responder:

```
require('http').createServer(function(req, res){
   res.writeHead(200);
   res.end('Hello World!\n');
   while(true);
}).listen(8080);
```

En el ejemplo anterior, una vez respondida la primera petición, la tarea se queda en el hilo principal indefinidamente, impidiendo que el servidor responda a más peticiones.

Como se puede observar, en un sistema basado en eventos como NodeJS, es muy importante no realizar tareas largas de forma bloqueante, ya que esto haría que el programa completo se bloqueara. Para evitar esto, las tareas largas deben ejecutarse de forma asíncrona, aprovechando los tiempos en los que el hilo principal está desocupado para ejecutarse, y dejándolo libre cuando sea necesario atender otra tarea.

Cluster

Como se ha explicado anteriormente, NodeJS por defecto se ejecuta en un único proceso, pero, ¿qué pasa si queremos aprovechar la totalidad de los procesadores del equipo en el que está ejecutándose?

Para realizar tareas multiproceso NodeJS pone a nuestra un módulo que se llama "Cluster", que nos permite crear nuevos procesos de forma similar los creados en Unix. Siguiendo con el ejemplo del servidor, un servidor base multiproceso quedaría de la siguiente manera:

```
var cluster = require('cluster');
var http = require('http');
var numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
   for (var i = 0; i < numCPUs; i++) {
      cluster.fork(); //Create new precess
   }
}
else {
   http.createServer(function(req, res) {
      res.writeHead(200); //Write header code
      res.end("Hello World!\n"); //Write body
   }).listen(8080);
}
```

Lo primero que hacemos es importar los módulos “cluster” y “http”, y utilizar el módulo “os” para obtener el número de procesadores de nuestra máquina. A continuación, si es el proceso padre (master), creamos un proceso hijo por cada procesador disponible usando el método “fork()” del módulo “cluster”. Por otro lado, si es un proceso hijo, creamos un *listener* como ya se explicó en la sección anterior.

El *cluster* de NodeJS ya incorpora un balanceador para repartir las peticiones que llegan a los procesos hijos, evitando así que varios atiendan la misma petición, simplificando de esta manera aún más la creación del servidor.

Otra cosa a tener en cuenta es que, el *cluster* de NodeJS, proporciona un sistema de comunicación por paso de mensajes entre los procesos hijos y el proceso padre. Además no existe la posibilidad de crear variables ni espacios de memoria compartida, por lo que las únicas opciones para compartir datos son usar una base de datos o el sistema de ficheros del propio sistema, con el coste que ello conlleva.

1.1.2. Express

Express [20] es un framework mínimo y flexible para el desarrollo de aplicaciones web con NodeJS. Permite crear de forma rápida y sencilla un servidor HTTP preparado para empezar a recibir peticiones. Por un lado, nos proporciona un generador aplicaciones que nos permite crear mediante un único comando un servidor multiproceso completo con toda la arquitectura de ficheros y un sistema de *routing*. Además nos permite añadir un sistema de *templating* para el desarrollo de la aplicación. Por otra parte, el módulo de Express también nos proporciona una serie de métodos para facilitar y simplificar la forma de atender las peticiones web que recibe el servidor.

Para crear una aplicación Express, lo primero que tenemos que hacer es instalar el generador de aplicaciones de Express. Para ello ejecutamos lo siguiente en la consola de comandos:

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

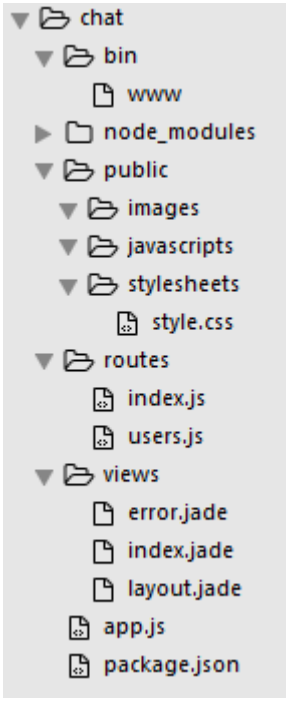
A continuación creamos nuestra aplicación llamando al comando de Express de la siguiente manera:

```
$ express chat
```

Al ejecutar el comando, nos creará una nueva carpeta en la que se encuentra el esqueleto de la aplicación Express, con todos los archivos y directorios necesarios para empezar a trabajar. Lo único que nos quedaría para tener la aplicación Express funcionando es instalar las dependencias del proyecto. Para ello entramos en el directorio de nuestra aplicación y ejecutamos el instalador de npm.

```
$ cd chat
$ npm install
```

Tras esto, la jerarquía de archivos y directorios de nuestro proyecto tendrá la siguiente forma:



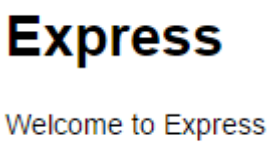
A continuación se explicará que hay en cada uno de los ficheros y de los directorios creados por Express, y los cambios necesarios para hacer funcionar el chat demo.

bin

En esta carpeta se encuentra el fichero que ejecuta nuestra aplicación. Para lanzar el servidor únicamente hay que ejecutarlo usando NodeJS.

```
$ node bin\www
```

A continuación, si escribimos en el navegador la URL de nuestro servidor, éste nos responderá con la página de inicio de la aplicación. En nuestro caso la URL sería http://localhost:3000/. La respuesta a la petición sería:



Para que nuestro servidor escuche en el puerto 80 tenemos que buscar la línea en la que pone:

```
var port = normalizePort(process.env.PORT || '3000');
```

Y sustituirla por:

```
var port = normalizePort(process.env.PORT || 80);
```

node_modules

En este directorio se encuentran todos los módulos instalados mediante npm. Al añadir un nuevo módulo al fichero "package.json" y ejecutar "npm install" se crea esta carpeta con todos los módulos declarados como dependencias del proyecto. De momento únicamente están los módulos necesarios para que Express funcione.

En caso de estar usando GIT como sistema de control de versiones, conviene añadir esta carpeta al completo al fichero ".gitignore". En este fichero se configuran los archivos que no queremos que se copien al control de versiones. No es necesario mantener los módulos en el control de versiones ya que puedes descargarlos de forma sencilla mediante npm.

public

Esta carpeta es de acceso público, por lo que todos los archivos que se coloquen en ella serán accesibles a través de peticiones http al servidor. En esta carpeta colocaremos las hojas de estilos CSS, el JavaScript que se va a ejecutar en la parte del cliente, y las imágenes del proyecto.

Para nuestro proyecto, en esta carpeta hemos incluido BootStrap, JQuery y AngularJS (que se explicará más adelante).

Además, hemos creado una hoja de estilos propia, que se llama style.css:

```
body,
html,
.height-100{
    max-height: 100%;
}

textarea{
```

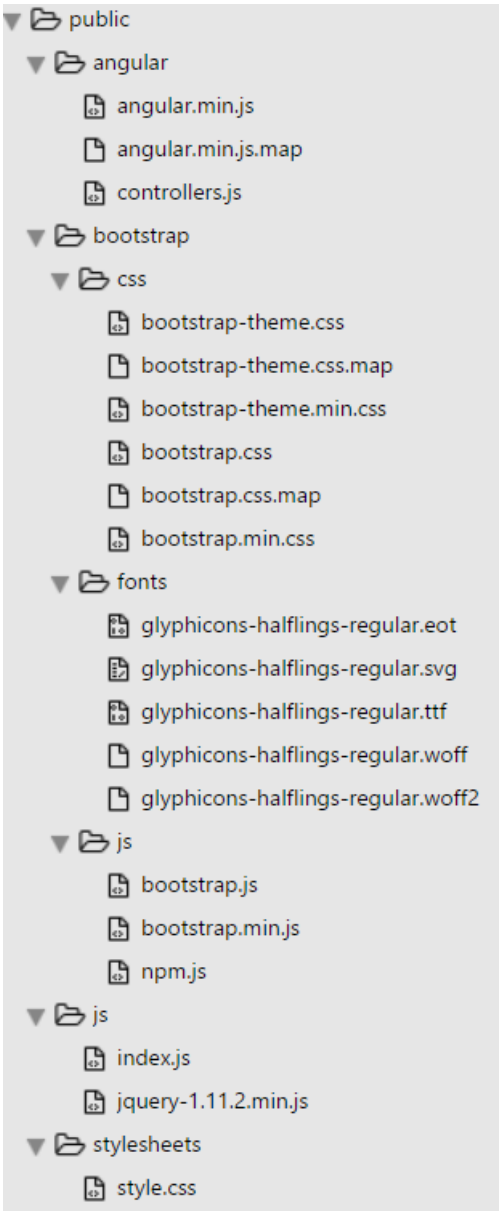
```
    resize: vertical;
}

#page_content{
    margin-top: 20px;
}

#col_form{
    border-right: 1px solid #ddd;
    padding: 10px;
}

#col_msgs{
    padding: 10px;
    padding-left: 20px;
}

#row_msgs{
    margin-bottom: 10px;
    padding-top: 5px;
    padding-bottom: 5px;
    background-color: #eee;
    border: 1px solid #ddd;
    border-radius: 5px;
}

.user{
    font-weight: bold;
}
```

Y también hemos creado un script en JavaScript para la interacción del usuario con la página que se llama index.js:

```
$("#btn_send").click(function(){
    //Get params values
    var data = {
            user: $("#input_user").val() || "Guest",
            msg: $("#input_msg").val()
    }

    if(!data.msg){
            printError("You can not send an empty message");
            return false;
    }

    $.ajax({
            method: "GET",
            url: "comment/new",
            data: data
    })
    .done(function(res){
```

```
                try{
                        var resp = JSON.parse(res);
                        if(resp.code != 200)
                                printError(resp.data)
                }
                catch(e){
                        printError("Error sending the message");
                }
        })
        .error(function(){
                printError("Error sending the message");
        });

        return false;
});

function printError(msg){
        $("#notify_msg").remove();
        $("#btn_send").before('<div id="notify_msg" class="alert alert-danger alert-dismissible" role="alert"><button type="button" class="close" data-dismiss="alert" aria-label="Close"><span aria-hidden="true">&times;</span></button><strong>Error: </strong>' + msg + '</div>');
}
```

Como resultado, la carpeta “public” tiene los componentes que se muestran a continuación:



routes

Este es el directorio en el que dejaremos todos los ficheros JavaScript encargados de tratar las peticiones HTTP. La aplicación de ejemplo generada por Express genera dos ficheros: “index.js” y “users.js”.

El primero de ellos, simplemente se limita a renderizar y servir la página principal de la aplicación. Para ello utiliza el sistema de templating conocido como JADE. Para nuestra aplicación, vamos a cambiar el título de “Express” a “Chat demo”. El fichero “index.js” quedaría de la siguiente manera:

```
var express = require('express');
var router = express.Router();

router.get('/', function(req, res) {
  res.render('index', { title: 'Chat demo' });
});
```

```
module.exports = router;
```

La función “get” del objeto “router” nos permite establecer a que URL va a responder el API. En este caso, cuando la URL sea http://localhost/ el servidor responderá con la página correspondiente a al template “index.jade”. Además nos da acceso al objeto “req”, donde se encuentra toda la información de la petición, y al objeto “res”, que utilizaremos para responder a la misma.

Por otro lado, vamos a renombrar “users.js” a “comment.js”. La idea del API es que tenga dos métodos, uno para obtener todos los mensajes, y otro para crear uno nuevo. Para ello creamos dos funciones, quedando el esqueleto del API de la siguiente manera:

```
var express = require('express');
var router = express.Router();


router.get('/', function(req, res) {
    //TODO GET FROM DB
});

router.get('/new', function(req, res) {
    var message = {
                    user: req.param("user"),
                    msg: req.param("msg"),
                    date: Date.now()
    }

    if(!message.msg && !message.user)
            return res.send(genRes(400,"Message and user are necesary"));

    //TODO INSERT IN DB
});
function genRes(code, data){
    var res = {
                    code: code,
                    data: data
    };
    return JSON.stringify(res);
}

module.exports = router;
```

Como se puede observar, además de generar las dos funciones del API, hemos generado una tercera función que nos simplifica y unifica el método en el que el API va a responder a las peticiones.

Además, también podemos ver que el “/new” admite dos parámetros, “user” y “msg”. Estos parámetros se obtienen del objeto “req” (request) usando el método

param(“tag_param”). Tras obtenerlos, comprobamos que sean correctos, y, en caso de no serlos, respondemos con un código y un mensaje de error.

Views

En esta carpeta se almacenan los ficheros que el sistema de templating usa para “renderizar” la página. Tanto en la aplicación creada por Express por defecto, como en la aplicación de ejemplo creada, se utiliza Jade como sistema de templating.

Jade [21] nos permite definir de forma sencilla el HTLM de la página, y además de permitir la inclusión de contenido dinámico. Esto se consigue pasándole como argumentos parámetros que luego se usarán al “renderizar” la página. Esto ya lo hemos hecho cuando hemos llamado a la función:

```
res.render('index', { title: 'Chat demo' });
```

En esa llamada, le estamos pasando como argumento al render un objeto en el que hemos almacenado el título de la página. Para acceder a este parámetro, únicamente hay que utilizar el símbolo “=”. Así pues, si queremos que el título de la página sea el que hemos pasado por parámetro, el código en Jade quedaría de la siguiente manera.

```
title= title
```

Esto nos generaría el siguiente HTML:

```
<title>Chat demo</title>
```

Además, en Jade, la inclusión de un elemento HTML dentro de otro está basada en el número de espacios en blanco antes del propio elemento, ahorrándonos así las etiquetas de apertura y de cierre propias de HTML. Si tenemos el siguiente código en HTML:

```
<p>
  <strong>Hola</strong>
</p>
```

Su equivalente en Jade sería:

```
p
  strong Hola
```

Otra funcionalidad muy interesante de Jade es que nos permite segmentar es partes una página, y componerla después. En nuestro proyecto, tanto “error.jade” como “index.jade” usan “layout.jade” como base. De esta forma, podemos crear plantillas y añadir el contenido que queramos a la misma.

Si observamos “layout.jade”, veremos que hay una línea en la que pone “block content”. Si después vamos a “index.jade”, en la primera línea nos dice “extends layout”, y a continuación “block content”. Esto quiere decir que “index.jade” extiende de

“layout.jade”, y sustituye “block content” por lo que se define del “block content” de “index.jade”.

Para la aplicación de ejemplo vamos a modificar los ficheros “index.jade” y “layout.jade”.

En el fichero “layout.jade” incorporamos todas las dependencias de JavaScript y de estilos CSS. De esta forma también quedarán incorporados en “index.jade”. El resultado sería el siguiente:

```
doctype html
html(ng-app='chat_demo')
  head
    meta(charset='utf-8')
    meta(http-equiv='X-UA-Compatible' content='IE=edge')
    meta(name='viewport' content='width=device-width, initial-scale=1')
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
    link(rel='stylesheet', href='/bootstrap/css/bootstrap.min.css')
  body(ng-controller='msg_list_controller')
    block content
    script(src='/js/jquery-1.11.2.min.js')
    script(src='/bootstrap/js/bootstrap.min.js')
    script(src='/js/index.js')
    script(src='/angular/angular.min.js')
    script(src='/angular/controllers.js')
```

En el fichero “index.js” definimos todos los componentes de la página, en nuestro caso un formulario y un lugar donde mostrar los mensajes del chat. El resultado sería el siguiente:

```
extends layout

block content
  div(class='container height-100')
    div(class='header clearfix')
      nav
        h3(class='text-muted')= title
    div#page_content(class='row height-100')
      div#col_form(class='col-md-4')
        form
          div(class='form-group')
            label(for='input_user') User name
            input#input_user(type='text' class='form-control' placeholder='Enter user name')
          div(class='form-group')
            label(for='input_msg') Message
            textarea#input_msg(class='form-control' placeholder='Enter a message' rows='3')
          div(class='form-group text-right')
            button#btn_send(class='btn btn-primary')
```

```
            span(class='glyphicon glyphicon-send')
            span  Send

        div#col_msgs(class='col-md-8 height-100')
          div#row_msgs(ng-repeat='msg in msgs' class='row')
            div(class='col-sm-2 user') {{msg.user}}
            div(class='col-sm-10 msg') {{msg.msg}}
```

app.js

En este fichero se realiza la mayor parte de la configuración de la aplicación. En él se define el sistema de templating, cómo tratar algunos errores, y definir los distintos routers que utiliza la aplicación.

Para nuestro ejemplo, tenemos que sustituir estas dos líneas:

```
var users = require('./routes/users');
app.use('/users', users);
```

Por estas dos:

```
var comment = require('./routes/comment');
app.use('/comment', comment);
```

En estas líneas le estamos importando y especificando qué routers van a atender qué URLs. Cuando la URL sea “http://localhost/”, la atenderá “index.js”, y cuando sea http://localhost/comment” lo atenderá “comment.js”

package.json

El “package.json” contiene información general de la aplicación, como por ejemplo, nombre, versión, etc... Así como la información relativa a las dependencias de la aplicación.

Para la aplicación demo que estamos realizando necesitamos añadir el driver de la base de datos, por lo que hay que añadir la línea “mongoskin: ~1.4.13” a las dependencias. El resultado final del archivo sería el siguiente:

```
{
 "name": "chat_demo",
 "version": "0.0.1",
 "private": true,
 "scripts": {
  "start": "node ./bin/www"
 },
 "dependencies": {
  "express": "~4.9.0",
  "body-parser": "~1.8.1",
  "cookie-parser": "~1.3.3",
  "morgan": "~1.3.0",
  "serve-favicon": "~2.1.3",
  "debug": "~2.0.0",
```

```
    "jade": "~1.6.0",
    "mongoskin": "~1.4.13"
  }
}
```

1.1.3. AngularJS

AngularJS [22] es un framework escrito en JavaScript cuya finalidad es dotar de dinamismo e inteligencia a las aplicaciones web. Este framework facilita la programación de SPAs (*single page aplications*) [23], ya que permite usar el patrón MVC (modelo vista controlador) [24] sobre aplicaciones de una única página. Es un proyecto de código abierto mantenido por Google, cuya primera versión se publicó en 2009, aunque cuando se ha hecho realmente popular ha sido a partir del 2012.

Este framework surge de la necesidad de segmentar y organizar el código que se ejecuta en la parte del cliente de una aplicación web. Anteriormente, disponíamos de JQuery [25] y librerías similares para la gestión y manipulación del DOM, y para dotar de cierto dinamismo a la aplicación. El problema de estas librerías es que no existe un patrón a seguir, dando como resultado un código difícil de comprender y poco manejable.

Los objetivos de AngularJS son separar la gestión y manipulación del DOM de la lógica de la aplicación, separar el lado del cliente del lado de la interacción con el servidor, y dar una serie de pautas a los desarrolladores para que sigan un patrón estructurado. Estas separaciones facilitan la realización de pruebas y la corrección de errores.

AngularJS pone a nuestra disposición una serie de *tags*, directivas y atributos que nos permiten extender la funcionalidad del HTML. De esta forma mantenemos la semántica y dejamos de depender de librerías externas como JQuery o AJAX.

En nuestra aplicación vamos a utilizar AngularJS para realizar el listado de los mensajes del chat. Para ello, es necesario definir mediante tags que es una aplicación de AngularJS. Para ello, primero indicamos que va a ser una aplicación AngularJS añadiendo el tag “ng-app” al elemento “html” del HTML:

```
html(ng-app='chat_demo')
```

Y continuación definimos el *controller* en el elemento “body”:

```
body(ng-controller='msg_list_controller')
```

Para finalizar añadimos la directiva “ng-repeat” en el “div” creado para mostrar los mensajes:

```
div#row_msgs(ng-repeat='msg in msgs' class='row')
  div(class='col-sm-2 user') {{msg.user}}
  div(class='col-sm-10 msg') {{msg.msg}}
```

En estas últimas líneas de código, lo que le estamos diciendo a AngularJS es que para cada mensaje en la lista de mensajes, repita la misma estructura. Además, le indicamos que de cada mensaje, escriba el usuario y el texto del mismo.

Para finalizar, tenemos que definir el controlador que carga los datos en el *scope*, de tal forma que estés accesibles para la directiva "ng-repeat" y pueda "renderizarlos". Para ello, en el script "controllers.js" escribimos lo siguiente:

```
var phonecatApp = angular.module('chat_demo', []);

phonecatApp.controller('msg_list_controller', function ($scope) {
    getMessages(function(messages){
            $scope.msgs = messages;
            $scope.$apply();
    });

    setInterval(function(){
            getMessages(function(messages){
                    $scope.msgs = messages;
                    $scope.$apply();
            });
    },200);
});

function getMessages(callback){
    $.ajax({
            method: "GET",
            url: "comment/",
    })
    .done(function(res){
            try{
                    var resp = JSON.parse(res);
                    if(resp.code == 200)
                            callback(resp.data)
                    else
                            callback([])
            }
            catch(e){
                    callback([]);
            }
    })
    .error(function(){
            callback([]);
    });
}
```

Lo que hace el código básicamente es pedir los datos al API cada 200 milisegundos, y actualizar la vista de AngularJS. Para obtener los datos se ha programado la función "getMessages", que utilizar AJAX para realizar la petición. Esta función es llamada

periódicamente (setInterval()), y, cuando tiene respuesta, actualiza el scope y refresca la vista llamando a "$scope.$apply()".

1.1.4. MongoDB

MongoDB es una base de datos NoSQL orientada a documentos. MongoDB utiliza el formato JSON para guardar estructuras dinámicas, permitiendo un desarrollo rápido y cambiante de las aplicaciones. Permite que algunos de estos campos sean listas y objetos embebidos, dando aún más versatilidad y posibilidades a la creación de la estructura de datos.

La nomenclatura, respecto a las bases de datos SQL, cambia. Lo que en una base de datos SQL era una tabla, en MongoDB es una colección, y lo que sería una fila en SQL se correspondería con un documento. Así pues, una base de datos MongoDB está compuesta por colecciones, que a su vez están compuestas por documentos.

Otra ventaja de usar MongoDB en la plataforma MEAN, es que la consola de la base de datos puede ejecutar JavaScript. Esto nos permite escribir scripts en este lenguaje para realizar consultas complejas, o tareas de mantenimiento del servidor. Además, mediante el uso de JavaScript nos permite definir funciones, que pueden ser llamadas posteriormente ahorrando tiempo y líneas de código.

A la hora de realizar consultas, MongoDB tiene su propio sistema de querys. Utilizando este sistema nos permite realizar búsquedas por campos, rangos, o incluso realizar búsquedas utilizando expresiones regulares. Además, nos permite definir índices de búsqueda propios, permitiendo así aumentar el rendimiento notablemente en consultas que se vayan a repetir frecuentemente.

Unos de los puntos más fuertes de MongoDB son su escalabilidad y alta disponibilidad. MongoDB nos permite tanto replicar los datos (replica sets) para la disponibilidad, como balancear la carga (sharding) para la escalabilidad, o incluso combinar ambas. Esto nos permite ir añadiendo y eliminando nodos (shards) conforme se vaya necesitando, permitiendo que el sistema escale horizontalmente.

Para nuestra aplicación demo, vamos a usar el driver de MongoDB MongoSkin, que nos proporciona un conjunto de funciones muy similares a las proporcionadas por la Shell de MongoDB.

En el fichero "comment.js" de la carpeta "routes" importamos el módulo de MongoSkin y realizamos la conexión con la base de datos al comienzo del script:

```
var mongo = require('mongoskin');
var db = mongo.db("mongodb://localhost:27017/chat_demo", {native_parser:true});
```

En primer lugar, vamos a usar el módulo para realizar una consulta que nos devuelva todos los mensajes que se han escrito en el chat. Para ello, usaremos el método "db.Collection.find()". Este método devuelve un cursor, por lo que tendremos que

llamar al método “toArray()” del mismo para obtener los elementos de la base de datos. Sustituimos el comentario “//TODO GET FROM DB” por lo siguiente:

```
db.collection("messages").find().toArray(function(err, messages){
    if(err) return res.send(genRes(400, "Error getting the messages."));
    res.send(genRes(200, messages));
});
```

A continuación, seguimos con el método que insertará un nuevo mensaje en la base de datos. Para ello, usaremos la función “insert()” del módulo MongoSkin. Sustituimos el comentario “//TODO INSERT IN DB” por lo siguiente:

```
db.collection("messages").insert(message, function(err){
    if(err) return res.send(genRes(400, "Error saving the message"));
    res.send(genRes(200, "OK"))
});
```

Tras esta última modificación la aplicación demo está terminada y lista para funcionar. El código de la aplicación creada se puede encontrar en el siguiente enlace:

- https://bitbucket.org/cmartineztdl/demo-chat

La aplicación está funcionando en la plataforma Heroku en el siguiente enlace:

- https://demochatcarlos.herokuapp.com/

1.2. PaaS (Plataform as a Service)

En la actualidad, debido a los altos costes de instalación y mantenimiento de los servidores, cada vez más las empresas están migrando sus datos y sus servicios a la nube. Los servicios más comunes son IaaS (Infrastructure as a Service), SaaS (Software as a Service) y PaaS (Plataform as a Service). Éste último lo explicaremos en más detalle a continuación.

Los PaaS son un tipo de servicio de computación en la nube que proporciona al cliente una plataforma sobre la que desarrollar y desplegar aplicaciones casi de cualquier tipo, eliminando la complejidad de construir y mantener la infraestructura típica para este tipo de aplicaciones. Ésta es una de las mayores ventajas que ofrece, ya que permite a un programador sin apenas conocimientos de sistemas desarrollar la aplicación olvidándose completamente de la parte de sistemas. Este tipo de plataformas son usadas sobre todo para el despliegue de páginas web y de APIs (Application Programming Interface).

A la hora de contratar un servicio en la nube hay que tener muy en cuenta el SLA (Service Level Agreement). El SLA o acuerdo de nivel de servicio, es un documento en el que el proveedor de servicio especifica el servicio mínimo que se compromete a proporcionar. Es importante elegir un proveedor que nos proporcione una alta disponibilidad.

Otro aspecto importante a la hora de contratar un servicio en la nube es el hecho de que nos dé la posibilidad de monitorizar el rendimiento del servicio contratado.

Uno de los proveedores de este tipo de servicios que está teniendo una mayor repercusión es Heroku [26], del que hablaremos más adelante. También tenemos AppEngine [27], desarrollado por google, Engine Yard, Red Hat OpenShift, entre otros. Actualmente se está desarrollando Kumori Systems, con el que se está colaborando de forma directa, aunque todavía el desarrollo del mismo no está totalmente completado.

1.2.1. Heroku

Heroku es un PaaS que nos permite desarrollar nuestras aplicaciones en lenguajes como Ruby, NodeJS, Python, Java y PHP, y desplegarlos de forma rápida y sencilla usando GIT. Esta forma de trabajar nos permite probar sobre la plataforma final el funcionamiento de la aplicación.

Además, pone a nuestra disposición una serie de paquetes para añadir funcionalidad a nuestra aplicación. Entre ellos podemos encontrar paquetes para mejorar el rendimiento, bases de datos, monitores del sistema, etc…

Heroku también nos da unos consejos, una filosofía de programación para el desarrollo de aplicaciones escalables. Estos consejos son los 12 factores, los cuales explicaremos más adelante.

Creación de una aplicación usando Heroku

Crear una aplicación en Heroku es muy sencillo. Para ello tenemos que descargar e instalar el Toolbelt de Heroku (disponible en la página web). A continuación nos autenticamos con el siguiente comando:

```
$ heroku login
```

A continuación descargamos el esqueleto de la aplicación de node usando GIT:

```
$ git clone https://github.com/heroku/node-js-getting-started.git
$ cd node-js-getting-started
```

Para finalizar creamos la aplicación en Heroku, subimos el contenido de nuestra aplicación usando GIT y configuramos que tenga al menos una instancia.

```
$ heroku créate
$ git push heroku master
$ heroku ps:scale web=1
```

Para abrir la página y ver el contenido de la misma escribimos el siguiente comando:

```
$ heroku open
```

1.2.2. Los 12 factores

En la actualidad, las aplicaciones web están tomando cada vez un papel más importante. Los 12 factores [13] es una metodología para el desarrollo de aplicaciones que:

- Utilizan formatos declarativos para minimizar el tiempo y coste de la unión de nuevos desarrolladores al equipo.
- Ofrecen máxima independencia con el sistema operativo ofreciendo máxima portabilidad entre plataformas.
- Son adecuadas para aplicaciones en la nube, eliminando así las necesidades de servidores y administración de sistemas.
- Minimizan la divergencia entre el entorno de desarrollo y el de producción, permitiendo despliegues rápidos y continuos para maximizar la velocidad del desarrollo.
- Pueden escalar de forma sencilla sin demasiados cambios de herramientas, arquitectura o desarrollo.

Los 12 factores que componen la metodología son los siguientes:

Codebase

Una aplicación que sigue la metodología de los 12 factores necesita un control de versiones como por ejemplo Git, Mercurial, Subversion, etc… Para una única aplicación tiene que existir únicamente un repositorio. Además la relación entre el repositorio y los despliegues tiene que seguir una relación 1 a 1.

Si para una aplicación existen varios repositorios con un *codebase* diferente, tenemos es un sistema distribuido, pudiendo seguir la metodología de los 12 factores en cada una de las partes de forma independiente. Por otro lado, si tenemos varias aplicaciones compartiendo el mismo codebase se considera una violación de la metodología de los 12 factores. La solución en estos casos consiste en extraer la parte compartida por varias aplicaciones en módulos e importarlos en cada una de las aplicaciones.

Dependencias

La mayoría de los lenguajes de programación proporcionan un sistema para la instalación de dependencias. Una aplicación que siga la metodología de los 12 factores tiene que declarar todas las dependencias mediante un manifiesto de declaración de dependencias.

Durante la ejecución de la aplicación, el sistema elegido comprobará si se disponen de todas las dependencias recogidas en el manifiesto, instalándolas en caso contrario. Este sistema de resolución de dependencias se utiliza tanto en el entorno de producción como en el entorno de desarrollo.

Usando NodeJS, el sistema más cómodo y usado para la gestión e instalación de dependencias es npm.

Configuración

Se considera "configuración" a todos aquellos parámetros que pueden variar entre despliegues. Normalmente se incluye información acerca de acceso a la base de datos, caché de memoria, servicios en segundo plano, credenciales, valores iniciales, etc…

Algunas aplicaciones guardan estos valores como constantes en el propio código. En una aplicación que siga esta metodología tiene que existir una distinción clara entre el código y la configuración. Esta configuración puede variar mucho entre despliegues, pero el código debe de ser el mismo.

Para una aplicación desarrollada en NodeJS, lo más cómodo es crear un módulo con toda la configuración. De esta forma, cuando queramos acceder a esos datos únicamente hay que importar el módulo. Un módulo de configuración tiene el siguiente aspecto:

```
var conf = {
   conf1: val1,
   conf2: val2
}
module.exports = conf;
```

Para accede a esa información desde otro script lo importamos usando la función require:

```
var conf = require("./nombre_modulo_configuracion.js");
console.log(conf.conf1); //Mostrará val1
console.log(conf.conf2); //Mostrará val2
```

Backing Services

Se considera *backing service* a cualquier servicio que la aplicación utilice a través de la red de forma independiente a las operaciones que realiza la aplicación de forma normal. En estos servicios se incluyen acceso a base de datos, mensajería, servicios SMTP, etc… Algunos de estos servicios, como por ejemplo el de la base de datos, tradicionalmente eran gestionados por el mismo sistema donde se desplegaba la aplicación. Además de estos servicios gestionados localmente también existen servicios gestionados por terceros.

Una aplicación que siga el patrón de los 12 factores no debe hacer distinción entre los servicios tratados localmente y los servicios de terceros. Ambos tipos deben ser tratados como recursos adjuntos, accediendo a ellos mediante una URL u otro sistema.

Construcción, despliegue, ejecución

En una aplicación que siga este patrón, el despliegue a producción se realiza en tres pasos:

1 – Construcción: en este paso el código del repositorio se transforma en programa ejecutable, con todas sus dependencias instaladas y compiladas.
2 – Despliegue: en esta etapa el ejecutable creado en el paso anterior se combina con el archivo de configuración de la versión anterior ya desplegada. De esta forma el programa está preparado para la ejecución de forma inmediata.
3 – Ejecución: los procesos necesarios para la ejecución se lanzan. Esto ocurre en el entorno de ejecución.

Procesos

Las aplicaciones ejecutadas en el entorno de ejecución pueden tener uno o más procesos. En el caso más simple, el código está en único script, el entorno de ejecución es el propio equipo del desarrollador, y el proceso se lanza mediante línea de comandos. Por otro lado, también tenemos el caso contrario, en el que la aplicación tiene varios tipos de procesos, los cuales pueden estar ejecutándose o no.

Los procesos basados en los 12 factores no deben tener estado y los procesos no deben compartir nada entre ellos. Cualquier tipo de información que se necesite y sea persistente en el tiempo, se almacenará usando un *backing service* como por ejemplo, un servicio de base de datos. Tanto la memoria como el sistema de ficheros pueden ser usados únicamente de forma breve para transacciones simples a modo de caché.

Port binding

Algunas veces las aplicaciones web se ejecutan en un contenedor de un servidor de aplicaciones web. Por ejemplo, una aplicación desarrollada en PHP podría ejecutarse en forma de módulo dentro de un servidor Apache HTTPD.

Una aplicación que siga los 12 principios estará completamente contenida en sí misma y no delegará en la creación de un servidor para realizar un servicio de *web-facing* durante la ejecución. Esto quiere decir que la propia aplicación tiene que encargarse de responder las peticiones en los puertos configurados en la misma sin que haya un intermediario que realice una redirección, como por ejemplo Nginx en Linux.

Concurrencia

Cualquier programa, una vez está en ejecución, está representado mediante uno o más procesos. En una aplicación que siga los 12 factores, los procesos deben tomar una forma muy similar al del modelo de procesos de Unix.

Usando ese modelo, el desarrollador puede diseñar la aplicación para atender a distintas cargas de trabajo, asignándole a cada proceso un tipo. Por ejemplo, las peticiones HTTP serán atendidas por un proceso web, y los procesos largos serán atendidos por procesos *workers*.

Esto no excluye que cada proceso individual pueda tener su propio sistema multihilo o multiproceso a nivel interno, como por ejemplo, el sistema de *theads* de una VM

(*virtual machine*) de Java. El problema del uso de una única VM es que el crecimiento vertical (mejoras de hardware del equipo) está muy limitado. Este problema hace que sea indispensable crear un sistema de procesos independientes que se puedan repartir entre distintas máquinas, permitiendo así el crecimiento horizontal (añadir más máquinas a la infraestructura) de la aplicación.

Desechabilidad

Los procesos de una aplicación que siga esta metodología deben ser desechables. Esto significa que los procesos tienen que poder pararse e iniciarse en el momento que se dé la instrucción de parada o de inicio. Esto permite una mayor flexibilidad a la hora de escalar, despliegues rápidos de código o de configuraciones, y una mayor robustez en el entorno de producción.

Para un proceso web que responda peticiones HTTP, al recibir la señal de parada debe parar de atender peticiones y finalizar cuando haya atendido a la última petición que hubiera recibido. Para un proceso que realice procesos largos, lo ideal sería mandar la tarea a la cola de tareas a realizar y finalizar. Esto asegura una finalización rápida del proceso.

Paridad entre desarrollo y producción

Anteriormente, había muchas diferencias entre el despliegue en los entornos de desarrollo y producción. Las personas que desarrollaban no eran las mismas que desplegaban en el entono de producción, las herramientas tampoco eran las mismas, y a esto hay que añadir el tiempo que pasaba entre despliegue y despliegue.

Las aplicaciones que siguen los 12 principios están diseñadas para realizar despliegues continuos, haciendo que la diferencia entre los despliegues de desarrollo y producción sea mínima. Para conseguir esto los desarrolladores tienen que hacer despliegues cada poco tiempo, y las herramientas y *backing services* que se usen tanto en desarrollo como en producción deben ser las mismas.

Logs

Los *logs* proporcionan visibilidad sobre el comportamiento de la aplicación. En los servidores normalmente estos *logs* suelen ser escritos en un fichero en disco, aunque éste no es el único formato de salida.

Una aplicación que siga estos factores nunca se encargará ella misma de escribir y mantener su propio log. En vez de esto, cada proceso en ejecución escribirá en un *stream*, sin *buffer*, a *stdout*. De esta forma, durante las pruebas en local el desarrollador podrá ver en tiempo de ejecución los logs que está generando la aplicación. En producción, este *stream* será leído junto con el resto de los *streams*, y traspasado a su destino final dónde se pueda visualizar.

Procesos de administración

La arquitectura de procesos es el conjunto de procesos que se ejecutan de forma regular en una aplicación, como por ejemplo un proceso para atender peticiones HTTP.

A parte de esto, a los desarrolladores les puede interesar poder llevar a cabo tareas de administración o de mantenimiento, como por ejemplo, migraciones de bases de datos, ejecutar scripts o tener acceso a una consola.

Estos procesos de administración deben ejecutarse en un entorno similar al de los procesos que llevan a cabo tareas largas. El código de administración debe ir junto al código de la aplicación para evitar problemas de sincronización.

Capítulo 2. The Webby Scrappy, funcionalidad y manual de uso

En esta sección se explicará en qué consiste la herramienta, sus características y funcionalidades, y ser realizará un manual de uso explicando cómo se despliega el centro de control y el servidor de robots. También se explicará cómo desplegar un robot en el servidor de robots desarrollado.

2.1. ¿Qué es The Webby Scrappy?

The Webby Scrappy es una herramienta ideada para la gestión y automatización de procesos de scrapping, aunque también se podría utilizar para cualquier tipo de proceso. Anteriormente, en otro proyecto [8], se diseñó un sistema con este mismo propósito, pero presentaba algunos inconvenientes.

Uno de los mayores inconvenientes era la complejidad para acceder al planificador y a los resultados de los robots. Para añadir un nuevo robot al planificador o para ver los logs de los mismos era necesario conectarse con la base de datos, y realizar inserciones y consultas en las mismas mediante queries de SQL. En la nueva plataforma desarrollada disponemos de una interfaz gráfica web que nos permite tanto configurar los robots, como ver los resultados de forma clara y sencilla.

Por otro lado, el lenguaje elegido (JavaScript, NodeJS) nos permite un desarrollo rápido de nuevas características y funcionalidades. Además se ha elegido como sistema de base de datos MongoDB, que nos da una gran flexibilidad a la hora de almacenar los datos, facilitando aún más si cabe el desarrollo de nuevas funcionalidades.

Otra característica importante del sistema es que es un sistema distribuido. Anteriormente, se disponía de un servidor en el que se encontraba la información de la planificación y los robots para ese servidor. Era un único bloque que se encargaba de realizar todas las tareas. En el nuevo sistema se han planteado los robots como un servicio al que conectarse. De esta forma se puede separar la tarea del scrapping de la tarea de gestión y planificación.

Para lograr esta independencia entre la gestión y los procesos de scrapping se ha definido una arquitectura y un protocolo que asegura la compatibilidad de la herramienta de gestión con los servidores en los que se encuentran los robots. A la herramienta de gestión se le ha llamado “Centro de control”, y a los servidores en los que se encuentran los robots “Servidor de robots”.

Uno de los objetivos que se pretendía con el desarrollo de este sistema es la posibilidad de escalar horizontalmente. Con el sistema planteado, un único centro de control puede conectarse con un número cualquiera de servidores de robots. De esta forma, para escalar de forma horizontal lo único que hay que hacer es crear un nuevo servidor de robots.

Además, como ya se ha mencionado anteriormente, los robots se plantean como un servicio. Esto permite que terceros creen nuevos robots y los pongan a disposición de

otras personas, por lo que se podría generar un negocio únicamente vendiendo el servicio para la herramienta. Además, como lo que se ha diseñado es el protocolo y la herramienta, los servidores de robots pueden estar programados en cualquier lenguaje siempre y cuando cumplan con el protocolo. Esto dota de aún más flexibilidad al sistema.

En este proyecto se ha desarrollado el módulo de "Centro de control" y un "Servidor de robots" demo para comprobar y demostrar su funcionamiento. Ambos módulos se han implementado usando NodeJS y MongoDB.

2.1.1. Centro de control

Este módulo es el centro neurálgico de la herramienta. Es un servidor Express programado en JavaScript (NodeJS). Como ya se ha mencionado anteriormente usa como sistema de almacenaje una base de datos MonogoDB.

Para acceder a las configuraciones, las planificaciones y los resultados de los robots pone a nuestra disposición una interfaz web. Mediante ella se pueden añadir, editar y eliminar las configuraciones de los robots, planificarlos tanto de forma automática como de forma manual, y ver los resultados de los mismos.

Por otro lado, es el encargado de, según la planificación configurada, conectarse con los servidores de robots y recibir las respuestas de los mismos. Un único centro de control puede conectarse con cualquier número de servidores de robots. Una vez lanzada la petición, el servidor espera el resultado.

2.1.2. Servidor de robots

Este módulo es el encargado de dar el servicio de los robots. A él se le conectan los centros de control y le realizan las peticiones. Una vez que el proceso del robot ha terminado, se conecta con el centro de control que le ha realizado la petición y le devuelve el resultado.

Lo más importante de este módulo es que cumpla el protocolo establecido para la herramienta. El lenguaje en el que esté desarrollado y el sistema de la base de datos que se use no son relevantes siempre y cuando se cumpla el protocolo. De esta forma nos aseguramos una gran flexibilidad de cara a dar el servicio al centro de control.

2.2. Funcionalidades y características

2.2.1. Funcionalidades

La herramienta nos ofrece, desde la interfaz gráfica del centro de control, las siguientes funcionalidades:

Gestión de configuraciones de robots

Se permite la creación, edición y borrado de las configuraciones de los robots. Estas configuraciones contienen dos partes diferenciadas: la configuración para el centro de control, y la configuración para el servidor de robots.

La configuración para el centro se compone del nombre interno que usaremos para identificarlo, una breve descripción, la dirección url a la que el centro de control tiene que hacer la petición para iniciar el proceso del robot, y la dirección url para forzar la parada del proceso del robot.

Por otro lado, la configuración del servidor de robots depende de cada servidor de robots. Para organizar esta configuración se han creado dos campos que tiene como formato el formato JSON. Una de las áreas es para la autenticación en el centro de robots, y la otra para configurar la ejecución del robot.

Gestión de planificaciones

Para la gestión de las planificaciones tenemos dos formas distintas que funcionan de forma independiente. Por un lado, podemos establecer un intervalo de tiempo entre petición y petición. Al hacerlo, se añadirá automáticamente a la lista de planificaciones una nueva planificación cada "x" tiempo, donde "x" es el tiempo establecido en el intervalo.

Por otro lado, se permite la creación de planificaciones de forma manual indicando la fecha y hora a la que queremos que se ejecute. Tanto estas planificaciones creadas manualmente, como las creadas de forma automática al establecer un intervalo, pueden ser modificadas, eliminadas y desactivas.

Visualización de logs

Se ha creado una sección en la que se permite visualizar el resultado de los distintos robots de forma rápida. Además, también se dispone de una vista detallada del log.

2.2.2. Características

Simplicidad

La simplicidad y facilidad de uso era uno de los hitos importantes, ya que era uno de los mayores problemas encontrados en el anterior sistema de gestión de robots. Para garantizar esta simplicidad se ha creado una interfaz web intuitiva que te permite configurar de forma rápida y sencilla los robots, sus planificaciones, y visualizar sus logs.

Escalabilidad

Uno de los objetivos que se ha buscado es dotar al sistema de una gran capacidad para escalar. Para ello se ha creado y definido un sistema escalable en el que los componentes pueden variar tanto en cantidad como en arquitectura interna de los mimos.

Portabilidad

La solución del proyecto se ha desarrollado en NodeJS y MongoDB, por lo que el servidor podría funcionar en cualquier máquina que de soporte para ellos. Actualmente

Linux, Windows y Mac dan soporte para ambos, por lo que tenemos un servidor que puede ejecutarse en casi cualquier equipo.

Compatibilidad

Para el desarrollo de la interfaz se ha decidido utilizar una plataforma web. De esta forma, una vez funcionando el servidor, se podría usar con cualquier sistema que disponga de un navegador, evitando así la necesidad de instalar nuevo software para la gestión del sistema.

Remoto

La gestión de los robots mediante la página web se basa en una arquitectura cliente servidor, por lo que no es necesario estar en la máquina en la que está el gestor para poder acceder a ella. Esto nos permite acceder a nuestro sistema desde cualquier parte y realizar las modificaciones que sean necesarias.

2.3. Manual de uso

A pesar de que se ha intentado realizar un diseño lo más sencillo e intuitivo posible, se ha considerado necesario realizar un pequeño manual explicando las distintas partes de la herramienta.

Durante el manual, se irán explicando las distintas secciones en el orden en el que se visitarían de forma natural. Empezaremos por el despliegue, continuaremos con la página de inicio, en la que se presenta la aplicación, seguiremos con la autenticación, y terminaremos por el dashboard.

2.3.1. Despliegue de la herramienta

Como ya se ha mencionado anteriormente, uno de los motivos por los que se ha elegido NodeJS es por su facilidad de despliegue. Tanto para el centro de control, como para el servidor de robots, lo único que necesitamos es tener instalado NodeJS y MongoDB.

La única diferencia entre el despliegue del centro de control y el despliegue del servidor de robots es el script que hay que ejecutar para iniciar la herramienta. En el caso del centro de control habría que ejecutar “bin/www”. Por otro lado, para iniciar el servidor de robots habría que ejecutar el script “startServer.js”.

Es importante tener en cuenta que antes de ejecutar el script de inicio de la herramienta hay que instalar las dependencias usando npm, como ya se ha explicado anteriormente.

Además, para poder acceder al centro de control necesitamos tener al menos un usuario en la base de datos. Para ello tenemos que hacer una inserción de un documento

en la colección “users”. Este documento tiene que tener al menos el campo “name” y el campo “pass”.

Por otro lado, si vamos a utilizar el servidor de robots que se ha programado para la demo tenemos que añadir un usuario en la base de datos del servidor de robots. La estructura que sigue es la misma que la del centro de control.

2.3.2. Despliegue de robots

Para desplegar un robot en el servidor de robots lo único que hay que hacer copiar el código del mismo en el directorio robots. El robot tiene que ser un módulo en JavaScript que exporte una función. Esta función tiene que admitir como parámetros la configuración del robot, y la función que se llamará cuando termine el proceso. Esta función devolverá como resultado el log de la ejecución. Un ejemplo mínimo de una tarea para el servidor de robots sería:

```
module.exports = function(conf, callback){
   callback(null, conf);
}
```

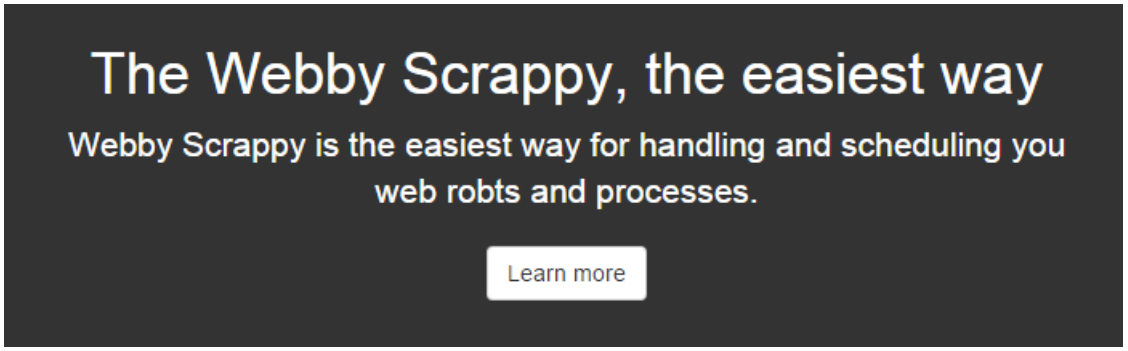
Lo único que hace esta función es pasar como log lo recibido como configuración.

2.3.3. Página de inicio

La idea de esta página es explicar de forma breve en que consiste la herramienta. Para ello se han credo tres secciones en las que se explican distintos aspectos de la aplicación. Además, también tenemos un enlace que nos lleva directamente al login del dashboard.

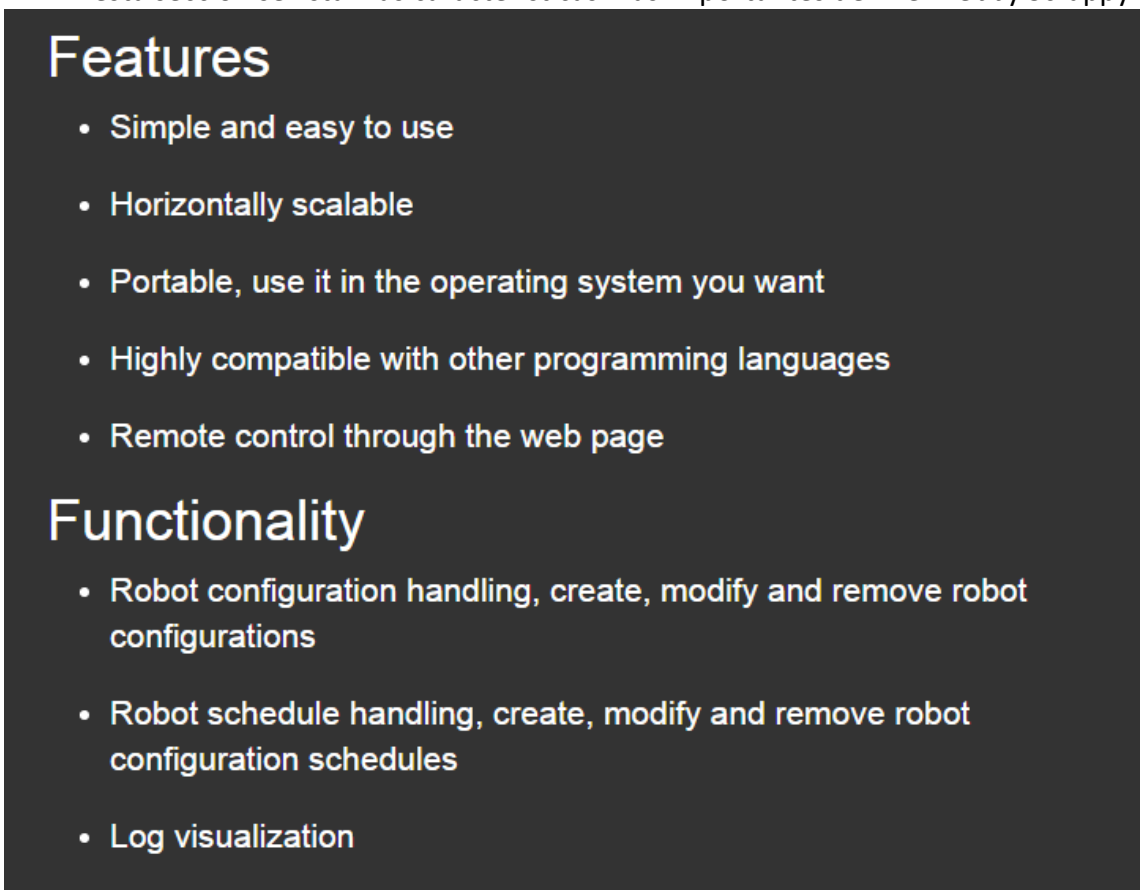
Home

En esta sección se explica de forma muy breve en una frase qué es la herramienta y para qué sirve. El botón “Learn more” nos lleva directamente a la siguiente sección en la que se explican las características de la herramienta.



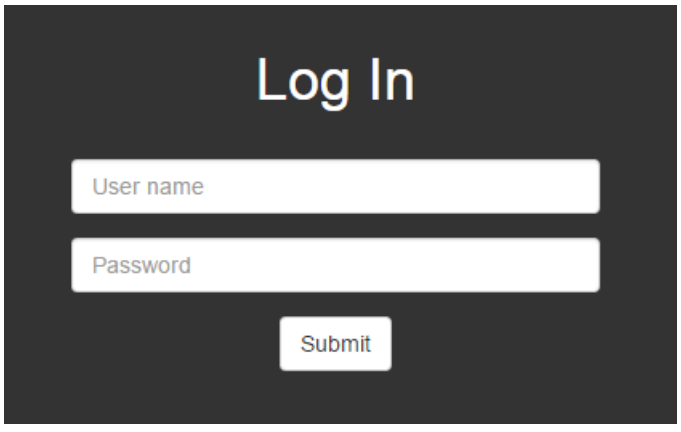
Features

En esta sección se listan las características más importantes de The Webby Scrappy.



2.3.4. Autentificación

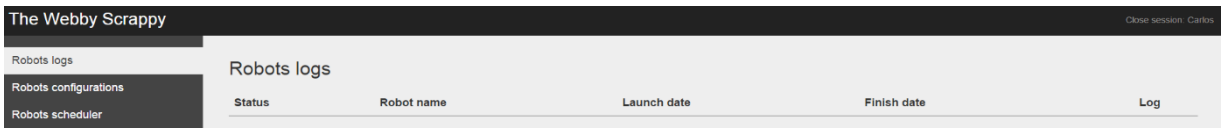
Para acceder al dashboard tenemos que acceder a "/admin". Al acceder a esta página, nos preguntará nuestro nombre de usuario y contraseña, y, tras verificar que existe en la base de datos de usuarios, nos permitirá acceder.



2.3.5. Dashboard

Tras autentificarnos, la herramienta se abre directamente la sección del log de las ejecuciones de los robots. Todas las secciones de la herramienta comparten una barra

en la parte lateral izquierda que nos permite navegar de forma rápida por el dashboard, y el cabecero de la página, en el que tenemos la opción de cerrar sesión.

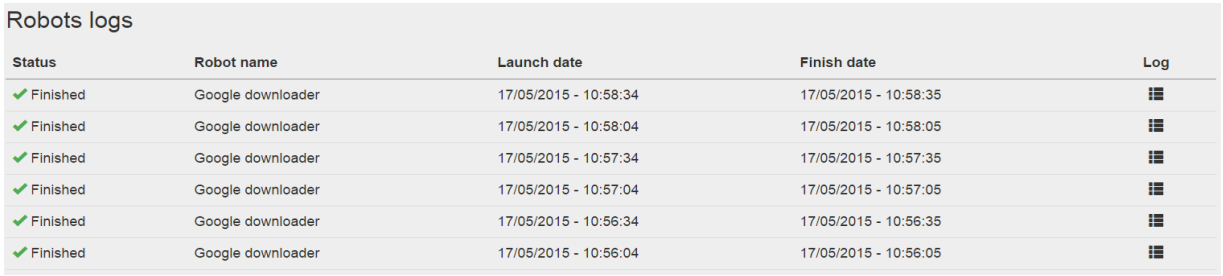


Logs

La sección de los logs de los robots es la primera que se muestra. En ella podemos comprobar los resultados de los robots que tenemos en ejecución, o que ya hayan terminado de ejecutarse.

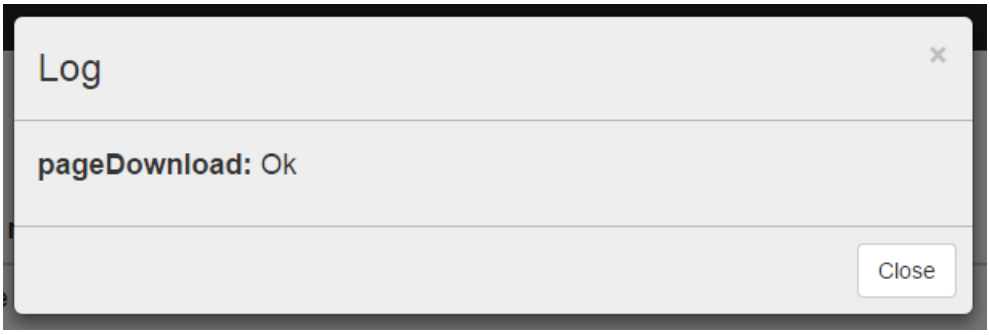
Para esta visualización disponemos de una tabla que contiene la siguiente información:

- **Status**: nos muestra el estado de los robots. Estos estados pueden ser:
  - **Running**: el robot está aún en ejecución.
  - **Stop**: el robot se ha parado manualmente.
  - **Error**: ha ocurrido un error durante la ejecución.
  - **Finished**: la ejecución ha terminado correctamente.
- **Robot name**: nombre del robot al que pertenece el log.
- **Launch date**: fecha y hora en la que se ha lanzado el robot.
- **Finish date**: fecha en la que ha terminado la ejecución del robot.
- **Log**: icono indicando si el robot ha dejado log o no. En caso de que no haya log para la ejecución el icono de log aparece con marca de agua y no se puede hacer click sobre él.


Robots logs

Status	Robot name	Launch date	Finish date	Log
✔ Finished	Google downloader	17/05/2015 - 10:58:34	17/05/2015 - 10:58:35	
✔ Finished	Google downloader	17/05/2015 - 10:58:04	17/05/2015 - 10:58:05	
✔ Finished	Google downloader	17/05/2015 - 10:57:34	17/05/2015 - 10:57:35	
✔ Finished	Google downloader	17/05/2015 - 10:57:04	17/05/2015 - 10:57:05	
✔ Finished	Google downloader	17/05/2015 - 10:56:34	17/05/2015 - 10:56:35	
✔ Finished	Google downloader	17/05/2015 - 10:56:04	17/05/2015 - 10:56:05	

En caso de que haya un log de la ejecución el icono de la columna de log estará sin marca de agua y nos permitirá hacer click en él. Al hacer click, nos abrirá una venta mostrándonos el log de la ejecución. La información de este log es la que nos devuelve el robot, a no ser que haya habido un error interno la hacer la petición de la ejecución del mismo.

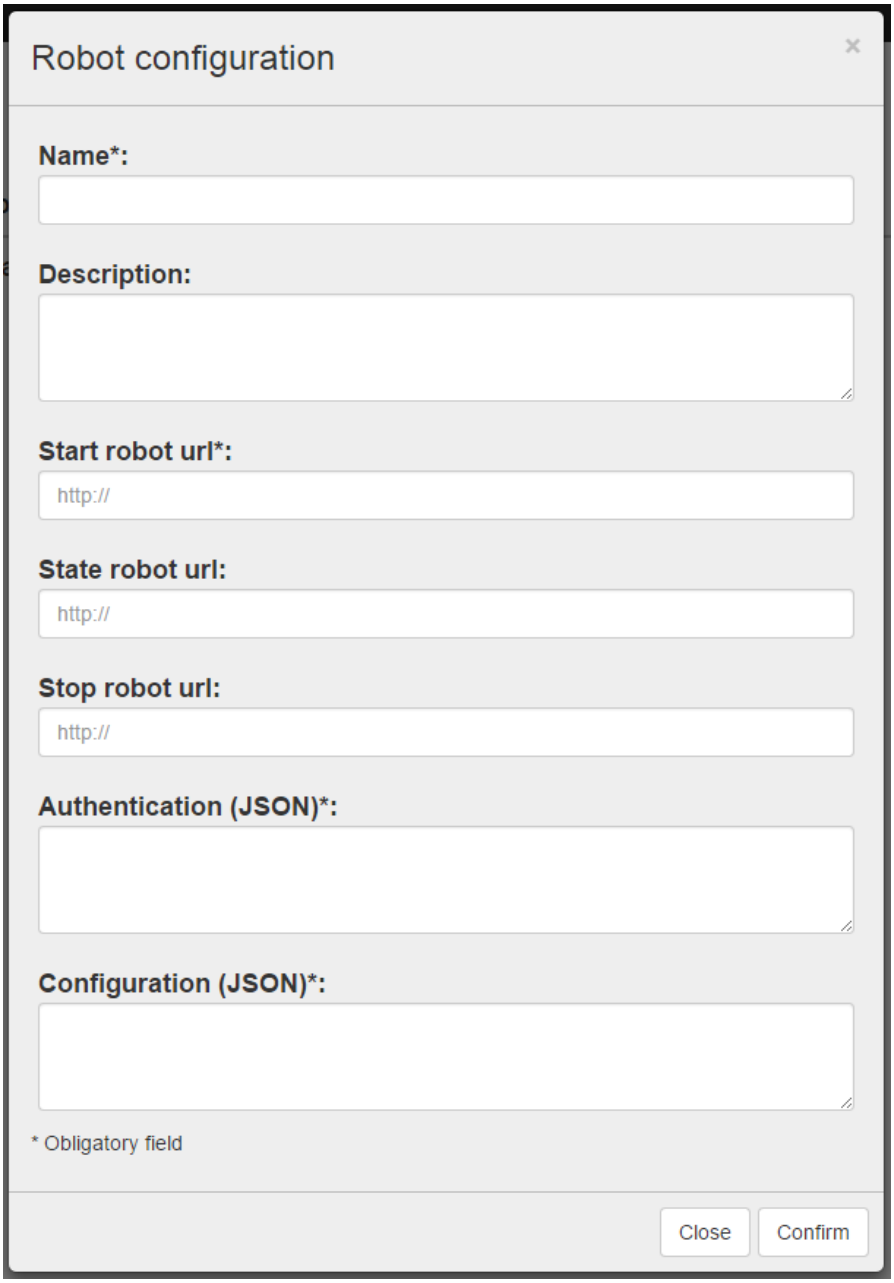


Configuraciones de los robots

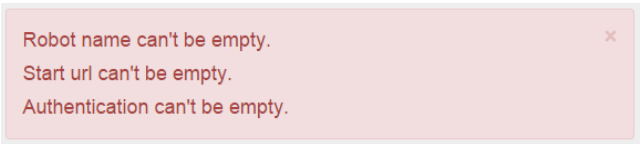
Este es el apartado nos permite por una lado visualizar los robots ya configurados, y por otro configurar tanto los robots en sí, como los intervalos de los mismos. Para acceder a esta sección hay que hacer click en “Robots configurations” en la barra lateral izquierda del dashboard.

Para crear un nuevo robot, tenemos hacer click sobre el icono “más” que se encuentra a la parte derecha del título de la sección. Al hacerlo nos aparecerá una ventana con un formulario para configurarlo. Los campos de configuración son los siguientes:

- **Name**: nombre para identificar el robot en la aplicación.
- **Description**: descripción breve del robot.
- **Start robot url**: ulr a la que hay que mandar la configuración y la autenticación para iniciar el robot.
- **Stop robot url**: url a la que hay que mandar la señal de parada del robot para pararlo durante la ejecución.
- **Authentication**: JSON con la información requerida para la autenticación por el servidor de robots. Esta información se manda al servidor de robots.
- **Configuration**: JSON con configuración que se manda al servidor de robots. Esta información puede variar en función del servidor.



En esta nueva ventana se nos indica mediante asteriscos los campos obligatorios del formulario. En caso de haber cometido algún error rellenando el mismo, se nos notificará en la parte inferior.



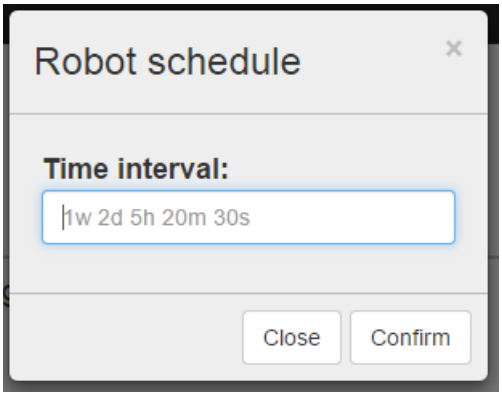
A simple vista tenemos una lista con un resumen de la configuración de los robots. En ella se muestra la siguiente información:

- **Name**: nombre que se ha elegido para identificar al robot.
- **Description**: Breve descripción del robot.

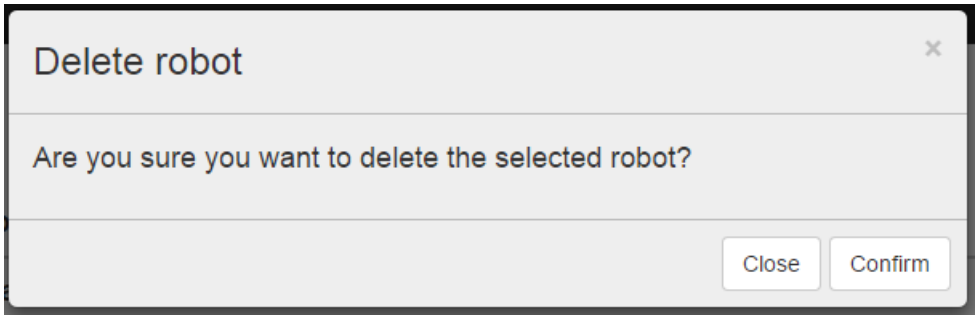
- **Properties**: Propiedades del robot. Indica si el robot puede iniciarse (todos deben poder iniciarse), y el segundo indica si se puede parar durante la ejecución. Al igual que en el log de la sección anterior, si el icono tiene un tono oscuro es que sí, y si tiene marca de agua es que no.
- **Scheduled**: Indica que si el robot tiene una planificación programada. Si el icono está oscuro es que si, y si el icono tiene marca de agua es que no.

Además, la última columna de la lista nos permite llevar a cabo acciones sobre el robot. Las acciones que se pueden realizar son las siguientes:

- **Schedule**: nos permite indicar un intervalo de tiempo para que el robot se ejecute de forma periódica. Para ello tenemos un campo de texto que nos permite introducir el tiempo en el siguiente formato: "1w 1d 1h 1m 1s". Como se puede observar, hay que escribir la unidad seguida de la cantidad sin espacio, y con espacios entre las distintas unidades. Así pues, en el ejemplo anterior el robot se ejecutaría cada vez que pasé la suma de una semana, un día, una hora, un minuto y un segundo.



- **Edit**: nos permite editar la información del robot. Al abrirla nos precargará la información del robot a configurar, y nos permitirá editarla. La ventana que nos abre es la misma que se usa para la creación de una nueva configuración.
- **Delete**: nos permite eliminar del sistema una configuración. Tras hacer click nos aparecerá una ventana de confirmación.

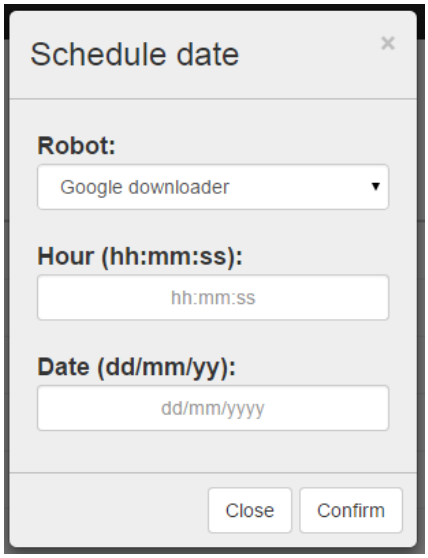


Planificador de robots

Esta sección del dashboard también tiene una doble funcionalidad. Por un lado nos permite visualizar las planificaciones ya realizadas, y por otro añadir manualmente nuevas o editar las ya existentes.

Esta lista se rellena automáticamente si hay algún robot configurado con un intervalo de tiempo. En la imagen de ejemplo hay un robot que se va a ejecutar cada 30 segundos.

Por otro lado, nos permite introducir manualmente una ejecución de un robot. Para ello hay que hacer click en el icono "más" que se encuentra a la derecha del título de la sección. Al hacerlo, nos abrirá una ventana en la que tendremos que elegir el robot, la hora y la fecha de la ejecución



Para la visualización de las planificaciones tenemos una lista de las próximas ejecuciones ordenadas en función del tiempo que falta hasta la ejecución. En la lista se nos muestra la siguiente información:

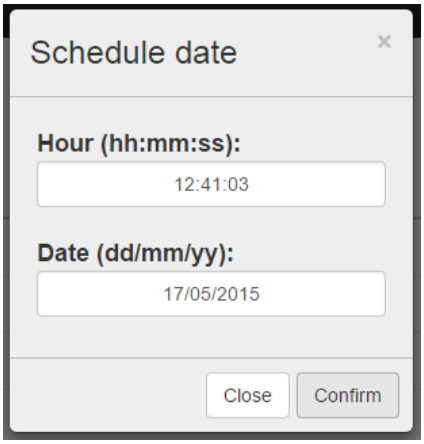
- **Robot name**: nombre que se le ha asignado al robot para identificarlo.
- **Launch Date**: fecha y hora prevista para iniciar la ejecución
- **Type**: tipo de planificación. Puede ser automática (creada por intervalo de tiempo), o manual (introducida por el usuario).
- **Active**: checkbox que indica si la planificación esta activa. Haciendo click permite cambiar el estado de activa a inactiva. Si la planificación está inactiva, el robot no se lanzará cuando llegue la "Lunch Date".

Robots schedules +

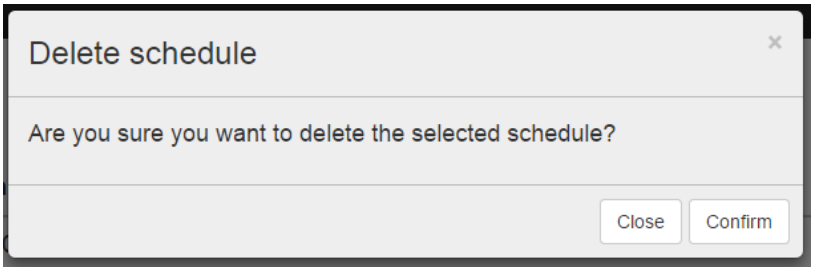
Robot name	Launch Date	Type	Active	Actions
Google downloader	17/05/2015 - 12:40:03	Auto		
Google downloader	17/05/2015 - 12:40:33	Auto		
Google downloader	17/05/2015 - 12:41:03	Auto		
Google downloader	17/05/2015 - 12:41:33	Auto		
Google downloader	17/05/2015 - 12:42:03	Auto		

Además, en la última columna nos permite realizar las siguientes acciones sobre las planificaciones:

- **Edit**: permite editar la hora y fecha de la ejecución. Al presionar sobre el icono se abrirá una ventana igual que la usada para crear una planificación, pero en éste caso no tendremos la opción de elegir robot.



- **Delete**: nos permite borrar una planificación. Al hacer click nos aparecerá una ventana para confirmar el borrado.



Capítulo 3. Especificación e implementación

A lo largo de este capítulo se explicará en mayor detalle el funcionamiento de la herramienta desarrollada. Para ello, primero se explicará la arquitectura general de la solución. A continuación se explicará el protocolo usado por los dos componentes para comunicarse entre ellos. Finalmente se comentarán los aspectos más interesantes de la implementación de cada uno de los dos componentes.

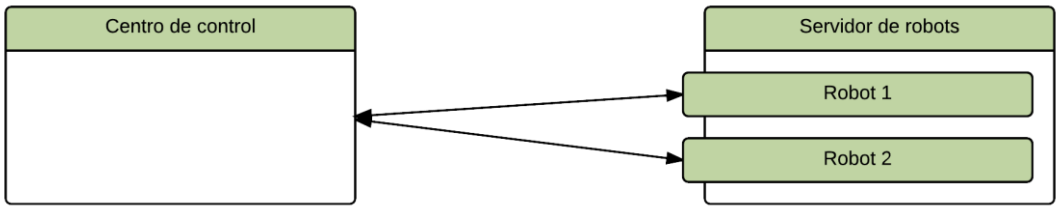
3.1. Arquitectura general

Como ya se ha mencionado en el apartado 2.1, la solución implementada es un sistema distribuido, en el que un centro de control se conecta a uno o varios servidores de robots. A su vez, un servidor de robots puede recibir peticiones de uno o varios centros de control.

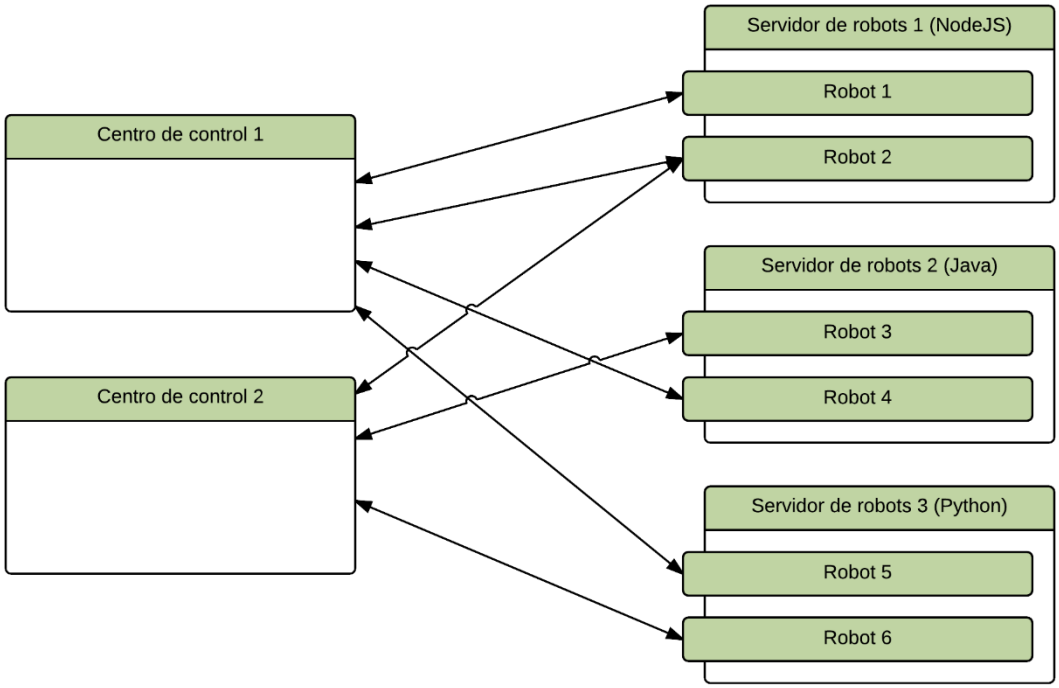
Esta solución nos permite distribuir los distintos robots entre distintas máquinas, gestionando todos ellos desde una única interfaz De esta forma la carga de trabajo se distribuye, obteniendo así una mayor cuota de entrada y salida de datos.

Por otro lado, al separar el centro de control de los robots, nos permite ofrecer los robots como servicio a través de un API. De esta forma terceros pueden desarrollar robots y ponerlos a disposición de sus clientes.

Así pues, tenemos dos componentes totalmente diferenciados. El centro de control, y el servidor de robots. Para hacer funcional el sistema se necesita al menos un centro de control y un servidor de robots como mínimo.



En un sistema ideal, un único centro de control se conectaría con distintos servidores de robots. A su vez a un servidor de robots se conectaría varios centros de control. De esta forma la carga de los robots web quedaría distribuida entre distintas máquinas. Además, como para la comunicación entre los distintos componentes lo único que importa es el protocolo, los servidores de robots podrían estar desarrollados en cualquier lenguaje siempre y cuando lo cumplan.



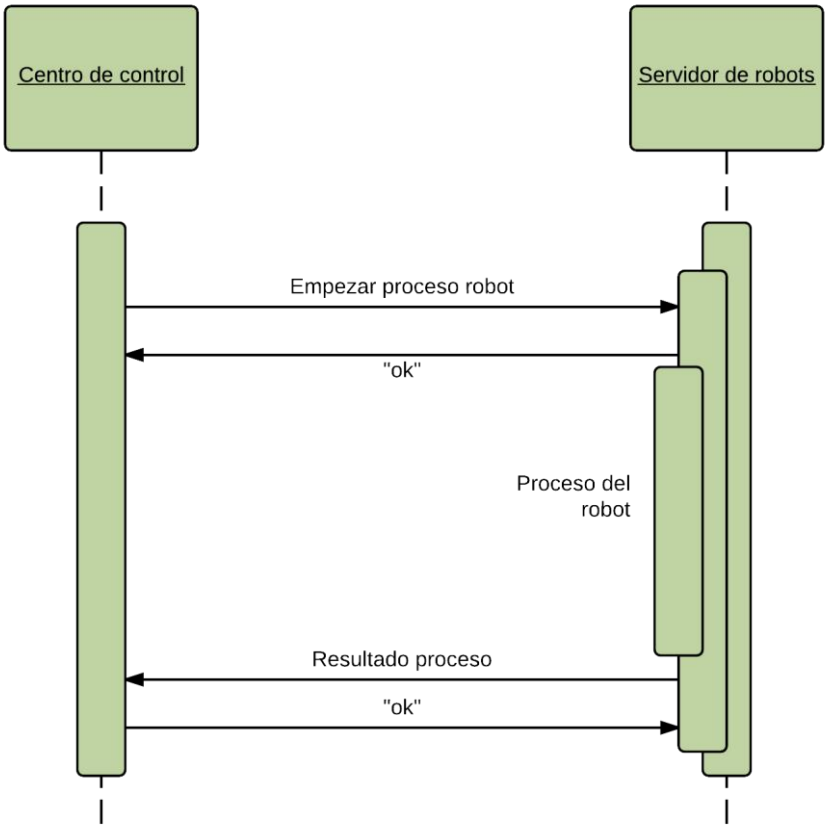
3.2. Protocolo

El protocolo es la parte más importante de toda la herramienta, ya que permite la comunicación del centro de controlo con los servidores de robots. Este protocolo tiene que cumplirse de forma estricta si se quiere crear un nuevo componente para el sistema. Si se cumple este protocolo se pueden crear tanto nuevos centros de control, como nuevos servidores de robots.

Entre el centro de control y el servidor de robots se dan dos procesos de comunicación. Por un lado tenemos el proceso de solicitar el servicio de un nuevo robot, y por otro el proceso de interrumpir el proceso.

3.2.1. Solicitar nuevo robot

El proceso de solicitar un nuevo robot se realiza en dos pasos. Primero, el centro de control realiza una petición web al servidor de robots. Éste, si los parámetros son los correctos, responderá un “ok” y comenzará el proceso del robot. Una vez el robot haya terminado su ejecución, el servidor de robots realizará una petición al centro de control devolviéndole los resultados del proceso.



En la imagen de ejemplo se pueden observar de forma clara los dos pasos que se dan cundo se solicita un nuevo robot. Ha sido necesario realizar esta tarea en dos peticiones ya que el proceso de ejecución de un robot puede ir desde unos segundos hasta varios días. Si se esperase a terminar la ejecución del proceso de un robot, seguramente la conexión daría un “timeout”, ya que el tiempo antes de que esto suceda suele rondar los 20 segundos.

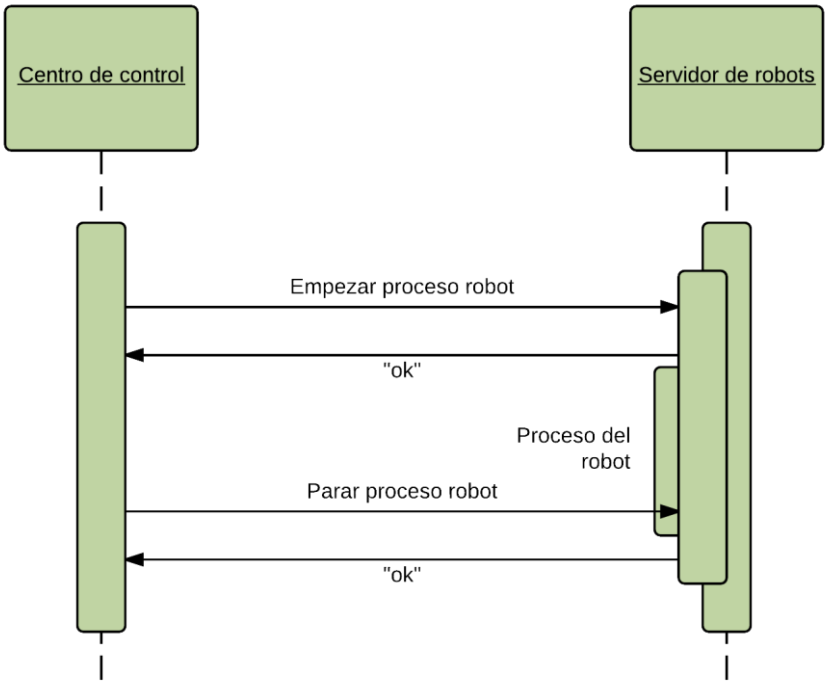
En la solicitud de un nuevo robot, se manda un identificador único generado por el centro de control que, posteriormente, permite al servidor de robots indicar a que petición está respondiendo. Además de este indicador, el centro de control le especifica al servidor de robots a que URL tiene que mandar el resultado. Junto con esto se manda toda la configuración introducida a través de la interfaz web (configuración y autenticación). El servidor de robots, en la respuesta, le envía un identificador interno del proceso para que el centro de control pueda referenciarlo y mandar parar le proceso.

Cuando el proceso del robot ha terminado, el servidor de robots, utilizando el identificador enviado por centro de control, le envía el resultado del proceso. Este resultado incluye como ha finalizado (éxito, error, etc…), un log de la ejecución (obligatorio), y los datos obtenidos (si es que lo hay).

Para un mayor detalle de cómo tienen que ser las peticiones y las respuestas del proceso consultar el API en las siguientes secciones de este mismo capítulo.

3.2.2. Parar proceso robot

Para realizar la parada de un robot que está ejecutándose en un servidor de robots se debe realizar una petición desde el centro de control usando el identificador proporcionado por el servidor de robots al iniciar el proceso del robot. Una vez solicitada la parada, el servidor de robots responderá con un "ok" y parará la ejecución del mismo, cancelando el envío de los datos al centro de control. No es obligatorio implementar la posibilidad de parar la ejecución de robots en el servidor de robots. Si un robot ha sido parado desde el centro de control, se ignorará la respuesta del servidor de robots de todos modos.



3.3. Centro de control

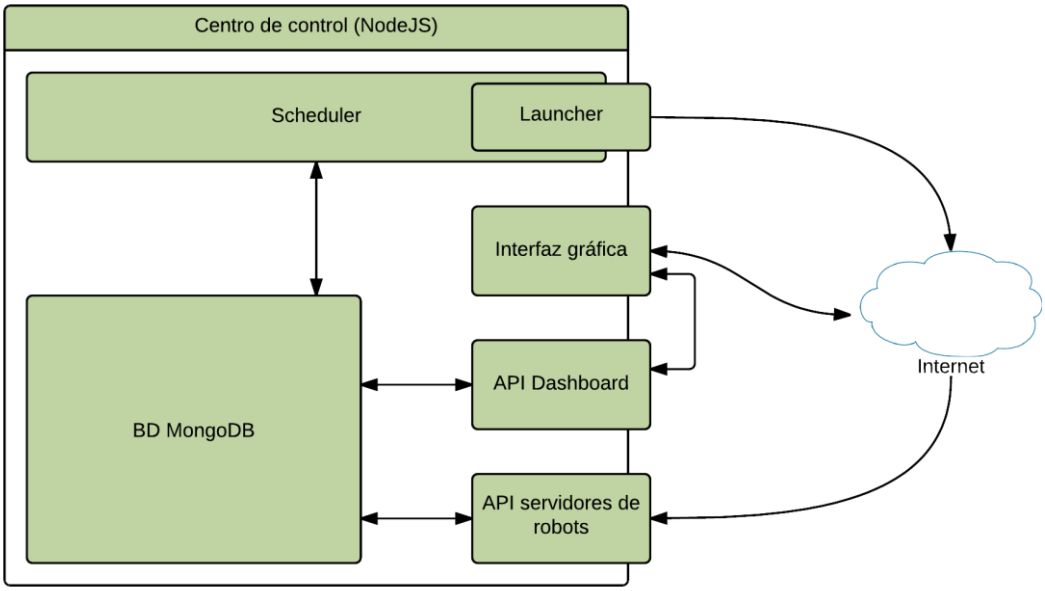
El centro de control es la pieza más importante de toda la solución. Sobre ella se configuran y planifican los robots, se encarga de realizar las peticiones a los servidores de robots, y de recibir y mostrar los resultados de los mismos.

Para el desarrollo de toda esta funcionalidad se va programar un servidor en NodeJS usando Express (que nos facilita la creación del servidor), MongoDB como sistema de bases de datos, y Jade como sistema de templatig.

3.3.1. Arquitectura

Como se ha podido observar, el centro de control lleva a cabo una gran cantidad de tareas. Esto hace que sea imprescindible dividirlo en varias partes independientes para poder analizarlas y desarrollarlas por separado. Las distintas partes de las que se compone el centro de control son las siguientes:

- Base de datos: base de datos MongoDB en la que se almacenan todos los datos necesarios para el funcionamiento del centro de control.
- Scheduler: encargado de planificar y realizar las peticiones a los servidores de robots.
- Interfaz gráfica: interfaz que se muestra al usuario para la gestión de los robots.
- API Dashboard: API usada por el Dashboard.
- API Servidor de robots: API usada por los servidores de robots.



En las próximas secciones se analizarán en más detalle cada una de las distintas partes del centro de control.

3.3.2. Base de datos

Para mantener todos los datos usados por el centro de control se ha elegido MongoDB como sistema gestor de base de datos. Como la estructura de los documentos en MongoDB es libre, para asegurar que todos mantengan una estructura de datos similar, se ha usado el driver de Mongoose [28].

Mongoose

Mongoose es un driver de MongoDB para NodeJS. Mongoose nos permite definir el esquema de los documentos de las colecciones como objetos. Además nos permite añadir métodos a estos objetos, facilitando así la separación de las interacciones con la base de datos del resto del código.

Un ejemplo de un esquema sencillo sería el siguiente:

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

//Declaramos el esquema
var schema_user = mongoose.Schema({
    name: String,
    pass: String
});

//Le asignamos una función para combrobar si el usuario y la contraseña es correcta
schema_user.statics.isValidUser = function(username, userpass, callback){
    //Consulta
    var query = User.where({"name": username, "pass": userpass});
    query.findOne(function(err, user){
        if(err){
            console.error(err);
            callback(false);
            return false;
        }
        callback(user != null);
    });
};

//Cremos el modelo
var User = mongoose.model('User', schema_user);

//Exportamos el modelo
exports.User = User;
```

En el ejemplo anterior, se declara un esquema para los usuarios de la aplicación. Primero definimos el esquema de usuario, que tendrá un nombre y una contraseña. A continuación creamos un método estático que nos permite comprobar si existe un usuario con la contraseña introducida. Finalmente, para poder usar el esquema, creamos un modelo del mismo y lo exportamos.

Para usar el modelo, simplemente lo importamos. Para comprobar si el usuario existe llamamos al método declarado anteriormente como estático:

```
var User = require('../models/User').User;
User.isValidUser(user_name, user_pass, function(isValid){

});
```

Además, al crear un modelo, automáticamente te genera un constructor del esquema definido. Cuando instancias un modelo, Mongoose proporciona una serie de métodos que permiten la interacción de forma simplificada con la base de datos.

Colecciones

Las colecciones usadas en el centro de control son las siguientes:

Launches

En esta colección se almacena toda la información relativa a las peticiones realizadas a los servidores de robots. En ella se almacena tanto el estado de la petición, como el resultado de la misma una vez que la ejecución ha acabado. Los documentos de esta colección tienen los siguientes campos:

- _id: clave única del documento que identifica la ejecución.
- robot_id: identificador del robot al que pertenece la ejecución. Hace referencia al "_id" de la colección "robots".
- lauch_date: fecha y hora a la que ha iniciado la ejecución del robot.

- status: indica el estado en el que se encuentra la ejecución.
- __v: versión de la aplicación.
- server_id: identificador del robot en el servidor de robots.
- finish_date: fecha y hora a la que ha terminado la ejecución del robot.
- hasData: Booleano que indica si se han generado datos a parte del log.
- log: JSON con toda la información del log de ejecución.

Un ejemplo de un documento de esta colección sería:


```
{
    "_id" : ObjectId("54ef3b177aa48208129530ca"),
    "robot_id" : "54ef350424cf96a41f4c0542",
    "launch_date" : ISODate("2015-02-26T15:26:15.280Z"),
    "status" : "Finished",
    "__v" : 0,
    "server_id" : "54ef3b177771b94c0b84870d",
    "finish_date" : ISODate("2015-02-26T15:26:15.921Z"),
    "hasData" : true,
    "log" : "{\"pageDownload\":\"Ok\"}"
}
```


Robots

En esta colección se guardan las configuraciones de los robots introducidas por los usuarios a través del Dashboard. Los campos de los documentos de esta colección son los siguientes:

- _id: identificador único del document.
- name: nombre de la configuración robot.
- description: descripción breve de la configuración robot
- start_url: URL a la que hay que hacer la petición para iniciar el proceso del robot.
- stop_url: URL a la que hay que hacer la petición para parar el proceso del robot.
- authentication: JSON con la información para autenticarse en el servidor de robots.
- configuration: JSON con la configuración mandada al servidor de robots.
- __v: versión de la aplicación.
- interval_num: número de milisegundo entre ejecución y ejecución del robot.
- interval_str: intervalo de ejecución en formato de texto.

Un ejemplo de un documento de esta colección sería:


```
{
    "_id" : ObjectId("54ef350424cf96a41f4c0542"),
    "name" : "Google downloader",
    "description" : "Download the page of google.",
    "start_url" : "http://localhost/startRobot",
    "state_url" : "",
    "stop_url" : "",
    "authentication" : "{\"user\": \"Carlos\", \"pass\": \"1234\"}",
```

```
    "configuration" : "{\"port\": \"80\", \"url\": \"www.google.es\", \"robotName\":
\"pageDownloader\"}",
    "__v" : 0,
    "interval_num" : 30000,
    "interval_str" : "30s"
}
```


Schedules

Esta colección contiene información de las próximas ejecuciones planificadas. La estructura de los documentos es la siguiente:

- _id: identificador único del documento.
- robot_id: identificador del robot de la planificación. Hace referencia al "_id" de la colección "robots".
- launch_date: fecha a la que se lanzará el robot.
- is_manual: indica si el robot se ha planificado automáticamente o de forma manual.
- is_active: indica si la planificación del robot está activada.
- __v: versión de la aplicación

Un ejemplo de un documento de esta colección sería:


```
{
    "_id" : ObjectId("555873a98bd5f8bc0cf43a5c"),
    "robot_id" : "54ef350424cf96a41f4c0542",
    "launch_date" : ISODate("2015-05-17T11:00:33.948Z"),
    "is_manual" : false,
    "is_active" : true,
    "__v" : 0
}
```


Sessions

En esta colección se almacena la información de las sesiones activas en el dashboard. Los campos que contienen los documentos de esta colección son los siguientes:

- _id: identificador único del documento.
- sid: identificador de la sesión.
- expires: fecha y hora en la que la sesión expira.
- lastAccess: última vez que el usuario ha realizado una petición al servidor.
- data: contiene los datos de la sesión. Estos datos son el nombre del usuario, el timestamp del último acceso, y los datos de la cookie de sesión.
- __v: versión de la aplicación.

Un ejemplo de un documento de esta colección sería:


```
{
    "_id" : ObjectId("54ef33de88e2c73fab330031"),
    "sid" : "0qihO-1692EVBk7maDS-pEqtWe4vnQ1O",
    "expires" : ISODate("2015-03-29T01:46:52.774Z"),
    "lastAccess" : ISODate("2015-02-26T15:46:52.775Z"),
```

```
    "data" : {
        "name" : "Carlos",
        "lastAccess" : 1424965612775,
        "cookie" : {
            "path" : "/",
            "httpOnly" : true,
            "expires" : "2015-03-29T01:46:52.774Z",
            "originalMaxAge" : 2627999999
        }
    },
    "__v" : 0
}
```


Users

En esta colección se almacena la información de los usuarios que tiene acceso al dashboard. Los campos del documento son los siguientes:

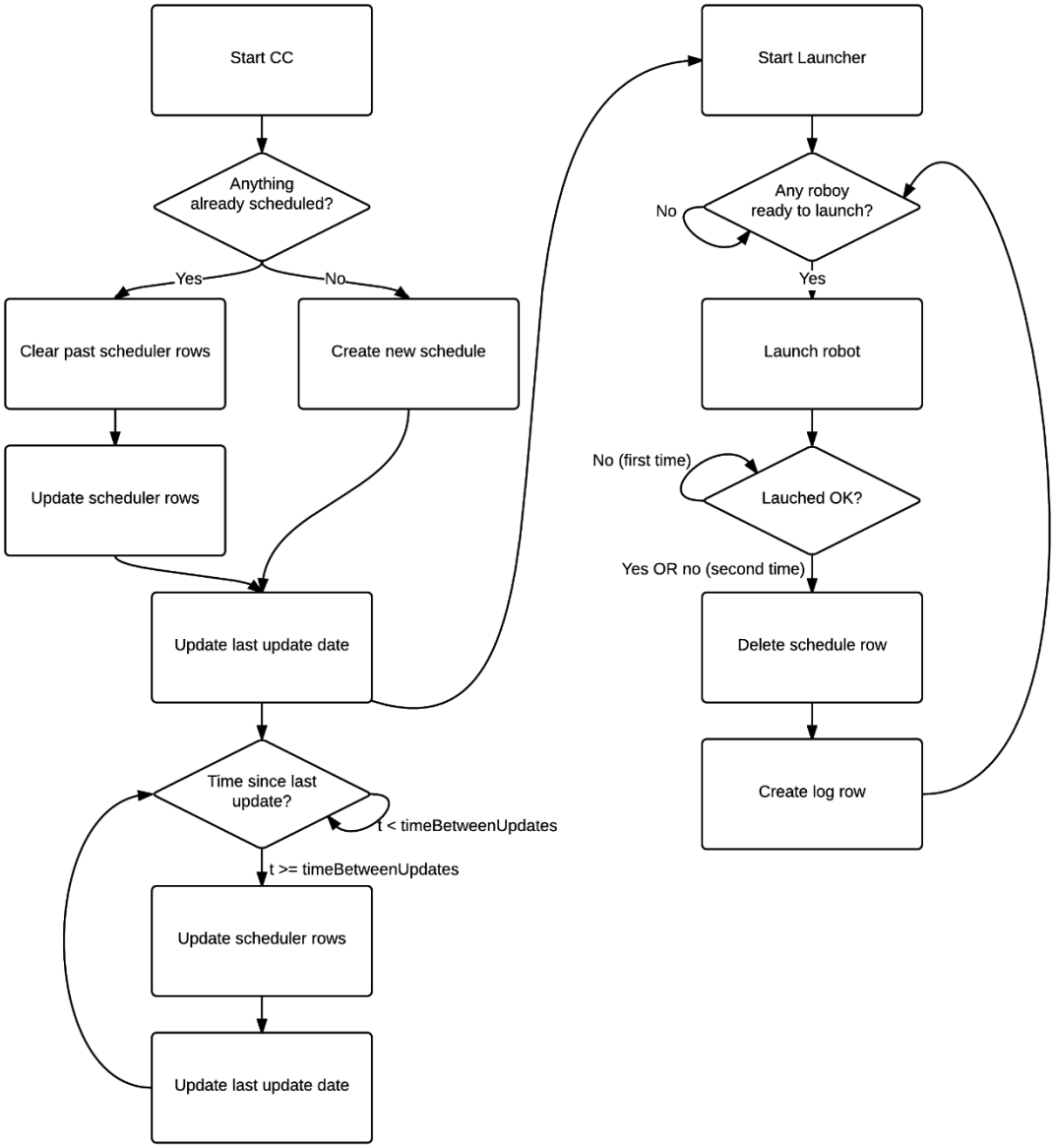
- _id: identificador único del documento.
- name: nombre del usuario.
- pass: contraseña del usuario.

Un ejemplo de un documento de esta colección sería:


```
{
    "_id" : ObjectId("54895e6ca36ff06155d4252a"),
    "name" : "Carlos",
    "pass" : "1234"
}
```


3.3.3. Scheduler

El scheduler o planificador es el encargado de generar las planificaciones automáticas y de lanzar las peticiones al servidor de robots. Para simplificar el algoritmo se ha dividido en dos partes: el planificador como tal, y el lanzador o launcher. En la imagen siguiente, en la que se representa el algoritmo para el scheduler, en la parte izquierda se encuentra el planificador, y en la derecha el lanzador.



Como se observa en la imagen de la parte superior, el proceso del planificador se lanza en cuanto se inicia la aplicación. El proceso es totalmente asíncrono, permitiendo así que el servidor siga funcionando (si no fuese asíncrono bloquearía el servidor impidiendo que respondiese a las peticiones que recibiese).

Cuando la aplicación se inicia, lo primero que hace el planificador es limpiar las planificaciones pasadas, dejando únicamente las que su ejecución estuviera para después del inicio del proceso del planificador. Una vez se ha realizado esta tarea, se planifican todos los robots por el tiempo configurado. Si se tiene configurado que se planifican automáticamente 24 horas, y tenemos un robot cada 6 horas, se planificará 4 veces. Este proceso se lanza cada vez que pasa un periodo de tiempo configurable.

Por otro lado, una vez que se ha creado toda la planificación, el proceso del lanzador empieza. Este proceso se lanza cada vez que pasa un intervalo de tiempo configurable. Lo que le proceso hace es comprobar si hay algún robot para lanzar. De ser así, lo lanza

y lo borra de la planificación. En caso de que haya un error al lanzar el robot, genera un log para informar del error.

3.3.4. Interfaz gráfica

Para dar acceso a las distintas funcionalidades de la herramienta, se ha diseñado y desarrollado una interfaz web. Esta interfaz se ha maquetado usando el sistema de templating Jade. El uso de AngularJS se ha descartado debido a que no se ha visto necesario introducir en el proyecto un framework completo para la realización de listados sencillos. Para ayudar en la maquetación se ha usado Bootstrap [29], y para dotar a la página de un mayor dinamismo se ha usado JQuery [25].

Jade

Para la implementación de toda la interfaz de usuario se ha usado el motor de templating Jade. Se ha elegido esta opción ya que nos permite desarrollar el código de una forma más clara y abreviada, permitiendo ahorrar tiempo en la implementación y facilitando el mantenimiento de la página.

En Jade, se eliminan las etiquetas de cierre de HTML, y la inclusión de un elemento dentro de otro se calcula en función del número de espacios o tabuladores respecto los elementos que se encuentran encima. Además nos permite crear templates, y hacer que otras templates las usen como base, haciendo innecesaria la repetición de las partes comunes de distintas páginas.

Por ejemplo, estas líneas interpretadas por el motor de templating Jade:

```
div
 address
 i
 strong
```

Se convertirían en el siguiente HTML:

```
<div>
 <address></address>
 <i></i>
 <strong></strong>
</div>
```

Como se puede observar el número de caracteres necesarios para maquetar en jade es menos que la mitad que los necesarios en HTML. A parte de esta ventaja, Jade nos ofrece otras muchas ventajas y funcionalidades para la creación de contenido dinámico.

Bootstrap

Para facilitar el diseño de la página se ha usado el framework Bootstrap. Bootstrap es un framwork que pone a nuestra disposición una serie de hojas de estilo CSS y unas funciones en javascript que ayudan a realizar y a agilizar el desarrollo de la página web.

Una de las funcionalidades más interesantes que nos ofrece es el "grid". Este "grid" nos permite estructurar la página definiendo filas y columnas. Cada fila de Bootstrap tiene doce columnas. Además, según la forma en la que las definas, hace que se comporten de una manera u otra en distintas resoluciones, simplificando así el hacer la página adaptable a dispositivos móviles.

Otra de las funcionalidades usadas es el sistema de pestañas. Esto nos permite tener en un único HTML el código de toda la página, mostrando únicamente la sección seleccionada. Al hacerlo de esta forma no hay tiempos de carga al pasar de una sección a otra. Para esta tarea no es necesario implementar nada de código en JavaScript, ya que Bootstrap ofrece la posibilidad de hacerlo usando únicamente tags de HTML.

Para los formularios y ventanas emergentes se han usado las ventanas modales. Además de esto, Bootstrap también pone a nuestra disposición una serie de iconos que ayudan a hacer más visual la herramienta, entre otros muchos elementos y funcionalidades.

JQuery

Para simplificar la manipulación de eventos y del DOM del HTML, y para la realización de las peticiones AJAX al API se ha usado JQuery. JQuery es una librería en JavaScript que facilita y simplifica la manipulación del HTML, el tratamiento de eventos, animaciones, y peticiones AJAX. Además las funciones de JQuery tienen soporte para múltiples navegadores, evitando que el desarrollador tenga que estar comprobando que función llamar en que navegador.

3.3.5. API Dashboard

Para la interacción de la interfaz de la herramienta con el servidor se ha creado un API. Para la creación de API se ha usado el sistema de routing de Express. La interfaz se conecta a este API mediante peticiones AJAX. Este API se encuentra en "/admin". El API definido es el siguiente:

URL	Descripción	Parámetros	Respuesta

/validate_user	Comprueba si el usuario coincide con la contraseña en la base de datos, y, en caso de coincidir, genera una sesión al usuario.	**name (String):** nombre del usuario.
		pass (String): contraseña del usuario.
			data: {
			code (Int): 200 (ok) o 400 (error),
			msg (String): mensaje del error o "Ok" si todo correcto
			}

/close_session	Cierra la sesión activa de un usuario.	--	**data**: {
			code (Int): 200 (ok),
			msg (String): "Ok
			}

/add_robot_conf	Método que permite tanto la creación como modificación de las configuraciones de los robots. Si se manda el parámetro "id" se intenta modificar el robot. Si no se manda el "id" se crea una nueva configuración.	**name (String):** nombre del robot.
		desc (String): breve descripción del robot.
		satartUrl (String): url para empezar el proceso del robot.
			data: {
			code (Int): 200 (ok) o 400 (error),
			msg (String): mensaje del error o "Ok" si todo correcto
			}

		stopUrl (String): url para terminar el proceso del robot. **auth (JSON):** JSON con los datos de la autenticación en el servidor de robots. **conf (JSON):** JSON con la configuración que se mandará al servidor de robots. **id (String):** identificador único de la configuración del robot	

/delete_robot_conf	Elimina del Sistema una configuración de un robot.	**id (String):** identificador único de la configuración del robot	**data**: {
			code (Int): 200 (ok) o 400 (error),
			msg (String): mensaje del error o "Ok" si todo correcto
			}

/manage_robot_scheduler	Permite añadir, modificar o eliminar la configuración del intervalo de tiempo para la planificación automática de los robots. Para eliminar la planificación automática "interval_num" tiene que ser 0.	**id (String):** identificador único de la configuración del robot.
		interval_str (String): texto que representa el intervalo de tiempo.
		interval_num (Int): tiempo entre ejecuciones en milisegundos.
			data: {
			code (Int): 200 (ok) o 400 (error),
			msg (String): mensaje del error o "Ok" si todo correcto
			}

/get_all_robots	Obtiene todas las configuraciones y sus datos.	--	**data**: {
			code (Int): 200 (ok) o 400 (error),
			msg (String): mensaje del error o "Ok" si todo correcto,
			data (JSON): datos solicitados al API
			}

/get_all_schedules	Obtiene todas las planificaciones y sus datos.	--	**data**: {
			code (Int): 200 (ok) o 400 (error),
			msg (String): mensaje del error o "Ok" si todo correcto,
			data (JSON): datos solicitados al API
			}

/manage_schedules	Permite la creación, modificación y eliminación de planificaciones. Mediante el parámetro "action" se especifica la tarea que va a realizar (["create", "actívate", "desactivate", "edit", "trash"]).	**action (String):** indica la acción a realizar por el API.
		rbt_id (String): identificador del robot de la planificación.
		sched_id (String): identificador de la planificación.
		launch_date (Date): fecha a la que se quiere configurar el lanzamiento del robot.
			data: {
			code (Int): 200 (ok) o 400 (error),
			msg (String): mensaje del error o "Ok" si todo correcto
			}

/get_logs	Obtiene todos los logs y sus datos.	--	**data**: { **code (Int):** 200 (ok) o 400 (error), **msg (String):** mensaje del error o "Ok" si todo correcto, **data (JSON):** datos solicitados al API

			}

3.3.6. API Servidor de robots

Este API es la encargada de recibir los datos del servidor de robots. Al igual que en el API para el dashboard, se ha utilizado el sistema de routing de Express. Este API se encuentra en "/data. El API es el siguiente:

URL	Descripción	Parámetros	Respuesta

/	Recibe el resultado de la ejecución de un robot y lo almacena en la base de datos.	**data:** {
		id(String): identificador de la ejecución,
		log (JSON): log de la ejecución,
		status (String): status de la ejecución
		}
			data: {
			code (Int): 200 (ok) o 400 (error),
			msg (String): mensaje del error o "Ok" si todo correcto
			}

3.3.7. Petición al servidor de robots

Cuando el centro de control detecta que se tiene que lanzar un robot hace una petición a la URL del servidor de robots configurado. El servidor de robots tiene que tener un API para poder atender estas peticiones. La petición se realiza con un único parámetro llamado "data" en formato JSON que tiene la siguiente estructura:

```
data (JSON):{
    authentication (JSON): datos de autenticación en el servidor de robots,
    clientData (JSON):{
            host (String): url de centro de control al que mandar la respuesta,
            id (String): identificador de la planificación
    },
    options (JSON): opciones de ejecución introducidas por el usuario
}
```

Para terminar la ejecución del robot se llama a la URL especificada en la documentación mandando como parámetro, al igual que en la petición anterior, un único parámetro llamado "data" en formato JSON con la siguiente estructura:

```
data (JSON):{
    authentication (JSON): datos de autenticación en el servidor de robots,
    id (String): identificador del proceso del robot
}
```

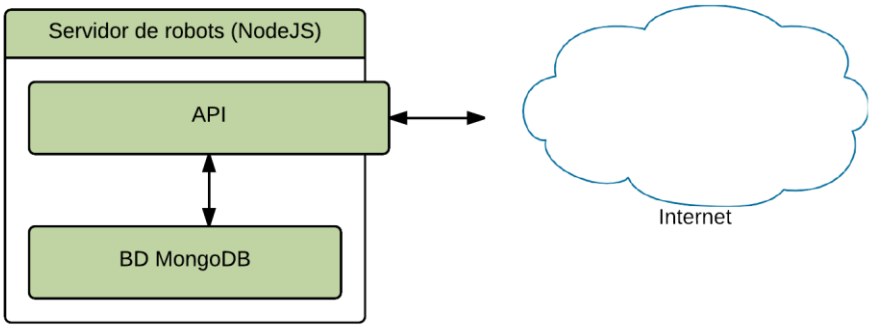
3.4. Servidor de robots

El servidor de robots es el encargado de recibir las peticiones de ejecución de los robots, ejecutar el que corresponda, y devolver el resultado al centro de control. Los servidores de robots pueden estar programados en cualquier lenguaje siempre y cuando cumplan el protocolo de comunicación establecido.

En mi caso, para demostrar y comprobar el funcionamiento del sistema, he desarrollado un servidor de robots usando NodeJS y MongoDB. El resto de secciones explican cómo se ha implementado este servidor de robots.

3.4.1. Arquitectura

En el caso del servidor de robots, la arquitectura es mucho más sencilla que la del centro de control. En este caso únicamente tenemos un API para recibir las peticiones del centro de control, y la base de datos en MongoDB.



3.4.2. Base de datos

Para la base de datos del servidor de robots, al igual que en el centro de control, se ha utilizado MongoDB. El driver de conexión usado ha sido Mongoose.

Mongoose

Al igual que en el centro de control, se ha elegido el driver de Mongoose. Se ha elegido debido a la experiencia adquirida usándolo, y por los motivos ya expuestos en el apartado de Mongoose del centro de control.

Colecciones

Las colecciones definidas en la base de datos del servidor de robots son las siguientes:

Baserobots

En esta colección se guarda la información de las peticiones que recibe de los centros de control. Los campos de los documentos de la colección son los siguientes:

- _id: identificador único del documento en la colección.
- clientData: JSON con la información necesaria para mandar la respuesta al centro de control.
- configuration: configuración introducida por el usuario en la interfaz del centro de control.
- startDate: fecha y hora en la que se ha iniciado el proceso del robot.
- __v: versión de la aplicación.
- log: log de la ejecución del robot.

- finishDate: fecha y hora en la que se ha terminado el proceso del robot.
- status: estado de la ejecución del robot.

Un ejemplo de un documento de esta colección sería:


```
{
    "_id": ObjectId("54ef3c9e5067b8b418d33015"),
    "clientData": "{\"host\":\"http://localhost/data\",\"id\":\"54ef3c9e7aa48208129530e4\"}",
    "configuration": "{\"port\":\"80\",\"url\":\"www.google.es\",\"fileName\":\"google\",\"robotName\":\"pageDownloader\"}",
    "startDate": ISODate("2015-02-26T15:32:46.485Z"),
    "__v": 0,
    "log": "{\"pageDownload\":\"Ok\"}",
    "finishDate": ISODate("2015-02-26T15:32:47.145Z"),
    "status": "Finished"
}
```


Users

Esta colección contiene información acerca de los usuarios que pueden hacer peticiones de robots al servidor. Los campos de los documentos de la colección son los siguientes:

- _id: identificador único del usuario.
- name: nombre del usuario.
- pass: constraseña del usuario.
- __v: versión de la aplicación.

Un ejemplo de un documento de esta colección sería:


```
{
    "_id" : ObjectId("5473067b9f56671c138cde38"),
    "name" : "Carlos",
    "pass" : "1234",
    "__v" : 0
}
```


3.4.3. Routing con NodeJS

Para el servidor de robots se ha creado un sistema de routing propio simplificado para atender a las peticiones. Para ello se han programado tres scripts en JavaScript que llevan a cabo la tarea. A continuación se explicarán los scripts en su formato mínimo para que el sistema funcione.

El primero de ellos, “server.js”, tiene el método que inicia el servidor. El método “init” recibe como parámetros la función de enrutamiento, las funciones encargadas de recibir las peticiones, y el puerto en el que escucha. Este método crea una función que lee los parámetros de la petición y llama a la función del router pasándole como

parámetros los funciones encargadas de atender las peticiones, la URL de la petición, el objeto para la respuesta, y los parámetros recibidos en la petición. El código sería el siguiente:

```
//Imports
var http = require("http");
var url = require("url");

//Funcion que hace de servidor
function init(route, handle, port) {
  function onRequest(request, response) {
    var urlParts = url.parse(request.url, true);
    var pathname = urlParts.pathname;
    var postData = urlParts.query;

    request.setEncoding("utf8");
    route(handle, pathname, response, postData);
  }

  http.createServer(onRequest).listen(port);
  console.log("WEB ROBOTS SERVER listining in port " + port);
}

//Exports
exports.init = init;
```

El router es una función sencilla que lo único que hace es comprobar si existe un método establecido para la url de la petición, y en caso de existir, llama al método. Si no existe devuelve un error “404 Not found”. El código del “router.js” sería el siguiente:

```
function route(handle, pathname, response, postData) {
        if (typeof handle[pathname] === 'function') {
                handle[pathname](response, postData);
        }
        else {
                response.writeHead(404, {"Content-Type": "text/html"});
                response.write("404 Not found");
                response.end();
        }
}

exports.route = route;
```

Finalmente, para hacer funcionar el sistema, hay que iniciar el servidor pasándole los parámetros correctos. Un ejemplo de llamada del servidor sería el siguiente:

```
//Imports
var server = require("./server/server");
var router = require("./server/router");
```

```
var port = 8888;

// handlers
var handle = {};
handle["/"] = function(res, params){
    res.writeHead(200, {"Content-Type": "text/html"});
    res.write("Hello world");
    res.end();
};

//Init server
server.init(router.route, handle, port);
```

Con esto tenemos un servidor con un sistema de routing que responde un "Hello world" en "/". Para añadir más urls únicamente hay que añadir más funciones al objeto "handle" siendo la clave la URL deseada, y la función lo que queremos que el API haga.

3.4.4. Despliegue automático de robots

Dos de los hitos que me propuse a la hora de desarrollar el servidor de robots en NodeJS fueron el poder lanzar robots programados en cualquier lenguaje, y que el despliegue de un robot no requiera ninguna configuración adicional en el servidor de robots.

Robots de cualquier lenguaje

Para la ejecución de robots en cualquier lenguaje, he aprovechado el modulo "child_process". Este módulo tiene un método llamado "exec" que permite ejecutar un comando del sistema en un proceso hijo. De esta forma podemos lanzar cualquier robot que se pudiera lanzar desde la línea de comandos. Un ejemplo de uso de la función seria:

```
var exec = require('child_process').exec;
exec("ls", function (err, stdout, stderr){
    if(err) return console.error(err);
    console.log(stdout, stderr);
});
```

Despliegue sin configuraciones

Para el despliegue de robots sin configuración se han creado dos alternativas válidas. En ambas existe un directorio en el que se despliegan los robots. Estos robots requieren de un fichero en JavaScript que se encargará de llamar al proceso del robot.

La diferencia entre los dos métodos reside en la forma de llamar a este fichero en JavaScript encargado de llamar al proceso del robot. Una de las alternativas, quizás la menos limpia, es utilizar la función "exec" del módulo "child_process". Este método presenta algunas complejidades a la hora de obtener las salidas de los robots.

La otra alternativa es programar los ficheros de JavaScripts encargados de lanzar los procesos de los robots como módulos de JavaScript. Luego, en tiempo de ejecución, se

importan esos ficheros como módulos permitiendo la llamada de los mismos. Usado este método se simplifica el paso de parámetros y resultados.

3.4.5. API Centro de control

Este API es la encargada de recibir las peticiones del centro de control. A diferencia que en el API para el dashboard, se ha utilizado el sistema de routing propio. Este API se encuentra en "/". El API es el siguiente:

URL	Descripción	Parámetros	Respuesta

/startRobot	Recibe los datos de configuración para iniciar el proceso de un robot.	**data (JSON):** {
		authentication (JSON): datos de autenticación en el servidor de robots,
		clientData (JSON): {
		host (String): url de centro de control al que mandar la respuesta,
		id (String): identificador de la planificación
		},
		options (JSON): opciones de ejecución introducidas por el usuario
		}
			data: {
			code (Int): 200 (ok) o 400 (error),
			msg (String): mensaje del error o "Ok" si todo correcto
			}

/stopRobot	Termina el proceso de ejecución de un robot.	**data (JSON):** {
		authentication (JSON): datos de autenticación en el servidor de robots,
		id (String): identificador del proceso del robot
		}
			data: {
			code (Int): 200 (ok) o 400 (error),
			msg (String): mensaje del error o "Ok" si todo correcto
			}

Capítulo 4. Pruebas y resultados obtenidos

En este capítulo mostraremos los resultados obtenidos tanto con la herramienta desarrollada, como con NodeJS como servidor. En la parte de The Webby Scrappy se monstrarán los resultados de uso de la herramienta, y en la parte de NodeJS, el rendimiento de NodeJS como servidor.

4.1. The Webby Scrappy

Para probar la herramienta se han desarrollado tres robots web que descargan la lista de productos de unas páginas y guardan sus datos en formato JSON. Estos tres robots se han desplegado en el servidor de robots cómo se explicó anteriormente en la sección del manual de uso. A continuación se han añadido las configuraciones necesarias en el centro de control para realizar las peticiones a los robots programados.

Para la realización de esta prueba se van a ejecutar los tres robots mencionados cada cinco minutos durante cuatro horas, y se recogerán los datos de las ejecuciones de los mismos. De esta forma podremos comprobar el funcionamiento de la herramienta.

Tras tener la herramienta funcionando durante cuatro horas con la configuración descrita anteriormente, observamos que se han generado 144 ficheros JSON con los datos de los productos de los tres robots. El total de productos descargados es 294.048, los cueles ocupan 285MB.

Para el cálculo de los tiempos medios de las ejecuciones se ha programado una función en MongoDB usando MapReduce [31]. Tras ejecutarla obtenemos los siguientes resultados.

	Tiempo medio (ms)	Tiempo máximo (ms)	Tiempo mínimo (ms)	#Productos por ejecución
Robot 1	501	931	349	150
Robot 2	689	1750	438	237
Robot 3	5913	11019	4435	5739

Como se puede observar, la herramienta funciona correctamente, generando tanto datos como logs de ejecución útiles.

4.2. NodeJS como servidor

Para la realización de las pruebas de cómo se comporta NodeJS como servidor se han realizado dos scripts. Se ha creado tanto un servidor como un programa que realiza peticiones al mismo. Se ha usado el módulo cluster de NodeJs para poder configurar el número de procesos que se van a utilizar en cada tarea. El programa encargado de realizar las peticiones lo llamaré “requester” a partir de ahora.

Como el ordenador desde el que se van a realizar las pruebas dispone de cuatro núcleos (ocho virtuales), el máximo de procesos con el que se van a lanzar los programas

son 8. De esta forma aprovecharemos al cien por cien el rendimiento del ordenador a la hora de hacer pruebas. Se realizarán pruebas usando uno y cuatro núcleos tanto para el servidor como para el programa que lanza las peticiones.

Cada proceso del programa que lanza las peticiones lanza de forma asíncrona 100.000 peticiones. Al terminar muestra por pantalla el número de peticiones realizadas correctamente, el número de peticiones en las que se ha dado un error, y el tiempo requerido para tratar las 100.000 peticiones.

4.2.1. Servidor monoproceso

Para la realización de estas dos mediciones se va a configurar para que únicamente haya un proceso escuchando peticiones.

Requester monoproceso

TEST	Nº reqs ok	Nº reqs error	Tiempo máximo	Reqs/sec	Max RAM (MB)
1	100 000	0	49 605 ms	2 015,93	14 + 60
2	100 000	0	46 632 ms	2 144,45	14 + 60
3	100 000	0	48 681 ms	2 054,19	14 + 60
4	100 000	0	47 933 ms	2 086,25	14 + 60
Media	**100 000**	**0**	**48 213 ms**	**2 075**	**14 + 60**

En este caso, en el que tanto el servidor como el programa que realiza las peticiones tienen un único proceso, podemos observar que el número de peticiones por segundo que soporta el sistema se encuentra alrededor de 2000. Este límite de peticiones por segundo puede ser producido tanto por el programa que realiza las peticiones como por el propio servidor.

Requester con cuatro procesos

TEST	Nº reqs ok	Nº reqs error	Tiempo máximo	Reqs/sec	Max RAM (MB)
1	400 000	0	92 450 ms	4 326,66	14 + 60
2	400 000	0	95 768 ms	4 189,89	14 + 60
3	400 000	0	97 970 ms	4 082,88	14 + 60
4	400 000	0	97 940 ms	4 084,13	14 + 60
Media	**400 000**	**0**	**96 032ms**	**4 170,89**	**14 + 60**

En este caso, en el que tenemos un servidor de un único proceso, y cuatro procesos realizando peticiones, se puede observar la capacidad real del servidor en NodeJS. En estos resultados el número de peticiones por segundo está limitada por el servidor ya que, aumentando el número de procesos que hacen peticiones no aumenta el número de peticiones atendidas por segundo.

4.2.2. Servidor con cuatro procesos

Para la realización de estas dos mediciones se va a configurar para que haya cuatro procesos escuchando peticiones.

Requester monoproceso

TEST	Nº reqs ok	Nº reqs error	Tiempo máximo	Reqs/sec	Max RAM (MB)
1	100 000	0	39 693 ms	2 519,34	14 + 60*4
2	100 000	0	41 700 ms	2 519,08	14 + 60*4
3	100 000	0	40 123 ms	2492,34	14 + 60*4
4	100 000	0	40 822 ms	2449,66	14 + 60*4
Media	**100 000**	**0**	**40 584 ms**	**2 463,99**	**14 + 60*4**

En este caso, en el que el proceso tiene cuatro procesos y el programa que hace las peticiones uno, podemos ver la limitación del programa que lanza las peticiones ya que, como hemos podido observar en un test anterior, el número de peticiones que soporta un servidor monoproceso se encuentra alrededor de 4.100 peticiones por segundo.

Requester con cuatro procesos

TEST	Nº reqs ok	Nº reqs error	Tiempo máximo	Reqs/sec	Max RAM (MB)
1	400 000	0	70 782 ms	5 651,15	14 + 60*4
2	400 000	0	70 301 ms	5 689,82	14 + 60*4
3	400 000	0	71 178 ms	5 619,71	14 + 60*4
4	400 000	0	71 328 ms	5 607,90	14 + 60*4
Media	**400 000**	**0**	**70 897 ms**	**5 641,97**	**14 + 60*4**

En este caso, a pesar de que lo esperado sería tener alrededor de 8.000 peticiones por segundo, la media de peticiones soportadas por el servidor y el programa que realiza las peticiones ronda las 5.600 peticiones por segundo. Esto seguramente se deba a que, a pesar de que el ordenador usado para las pruebas tenga ocho núcleos virtuales, parte de la capacidad de proceso se usa en monitorizar, en otras aplicaciones en ejecución, y en el propio sistema operativo.

Capítulo 5. Conclusiones, experiencias y líneas futuras

En este apartado se explicaran las conclusiones y las experiencias sobre el uso de la plataforma MEAN. Primero se explicará la curva de aprendizaje de NodeJS, ya que supone un cambio importante en la forma de estructurar el código respecto otros lenguajes aprendidos durante la carrera (Java, C, etc…). A continuación se expondrán las experiencias y conclusiones en el uso de JavaScript en la parte del servidor y en el uso de MongoDB. Además, se van a comentar distintas mejoras del proyecto o líneas de trabajo por las que seguir desarrollando la herramienta. Se van a tratar por separado las mejoras del centro de control de las mejoras del servidor de robots.

5.1. Curva de aprendizaje de NodeJS

A lo largo de la carrera se han estudiado varios lenguajes con su metodología de programación correspondiente, como por ejemplo C, Lisp y Java. A pesar de haber estudiado y aprendido distintos lenguajes con su correspondiente forma de programarlos, a la hora de empezar a programar en NodeJS me encontré con que rompía con muchos de los esquemas que tenía preestablecidos.

Por un lado tenemos que NodeJS está orientado a programación asíncrona, en el que todo el programa se ejecuta en un único proceso. Esto no implica que no pueda realizar más de una tarea al mismo tiempo. Por ejemplo, podemos estar leyendo un fichero del disco mientras se escribe en otro. Para realizar varias tareas NodeJS utiliza un sistema de eventos. Al lanzar una tarea asíncrona, esa tarea sale del hilo de ejecución principal, aprovechando cuando éste está libre para ejecutar su tarea. Una vez que la tarea ha finalizado se lanza un evento el cual es añadido a la cola de eventos. Cuando el hilo principal está libre se atienden los eventos que estén encolados.

Para hacer más simple la tarea de ejecutar código al terminar una tarea asíncrona, lo que se hace es pasar a la propia función el código que queremos ejecutar como una función anónima. Este concepto se conoce como “callback”. Esta función pasada como parámetro es llamada como respuesta al evento de finalización. De esta forma permite al desarrollador ahorrarse el tener que programar el “listener” al evento de finalización. Un ejemplo sencillo de esta tarea es la lectura de un fichero de forma asíncrona:

```
var fs = require('fs');
fs.readFile("fichero.txt", function(err, data){
    if(err) return console.error(err);
    console.log("Número de caracteres: " + data.length);
});
console.log("Ahora estamos leyendo el fichero");
```

Como se puede ver en el ejemplo, importamos el módulo “fs” (file system), y ejecutamos la función de lectura asíncrona. A esta función le pasamos dos parámetros: el nombre de fichero y la función que queremos ejecutar cuando termine la lectura. Al acabar la lectura del fichero, la función “readFile” va a llamar la función anónima que hemos pasado como parámetro con dos argumentos. El primero, por convenio, siempre es el de error (err), y el segundo contiene una cadena de texto con el contenido del fichero (data). La función anónima lo único que hace es mostrar el error si es que lo hay,

o mostrar el número de caracteres si se ha leído correctamente. Como la lectura del fichero es asíncrona, la salida será la siguiente:

```
Ahora estamos leyendo el fichero
Número de caracteres: xx
```

Como se puede observar, se ejecuta primero el "console.log" del final ya que la llamada a "fs.readFile" sale del hilo principal.

Entender bien cómo funciona la asincronía en NodeJS es muy importante, y quizás uno de los grandes problemas cuando empiezas. Otro concepto que se me hizo complejo inicialmente fue el de mandar funciones anónimas como parámetros de otras funciones. Esto último hace que el código sea más difícil de seguir y de entender.

5.2. NodeJS, JavaScript en el servidor

Durante la carrera, en algunas de las prácticas, hemos tenido que desarrollar aplicaciones tanto para un servidor en JAVA (Apache Tomcat), como para un para un servidor en PHP (Apache). En estas dos plataformas, el lenguaje JavaScript únicamente se utiliza en la parte de cliente para mejorar la experiencia del usuario dotando a la página web de un mayor dinamismo.

Para la realización de este proyecto, a pesar de que inicialmente se empezó a desarrollar la parte del servidor en JAVA, finalmente decidí desarrollarlo en NodeJS. Este cambio se debe a que, por curiosidad, busqué un tutorial de cómo programar en NodeJS y me puse a seguirlo para aprender. Al cabo de un rato siguiendo el tutorial tenía un servicio más completo que el desarrollado en Java durante varios días. Éste fue el principal motivo por el que reprogramé mi proyecto en JavaScript.

NodeJS permite un desarrollo muy rápido de aplicaciones ya que dispone de métodos de muy alto nivel, y JavaScript es un lenguaje muy flexible. Además, tiene un repositorio de módulos (npm [19]) que nos ayuda a agilizar aún más el desarrollo de las aplicaciones. Esto último, junto con la sencillez del testeo del servidor, ya que no deja de ser un programa ejecutado desde la consola de comandos, hace muy sencillo y rápido el desarrollo y testeo del código programado.

A raíz de haber aprendido NodeJS durante el proyecto, he tenido la oportunidad de participar en un proyecto más grande en una empresa en el cual nos hemos visto obligados a realizar tests de rendimiento para calcular el número de instancias del servidor necesarias para atender un número determinado de peticiones por segundo. Anteriormente, cuando hablábamos de NodeJS en el Capítulo 1, dimos algunos datos del rendimiento y los recursos necesarios para un servidor Apache Tomcat en Java. A continuación, daré algunos resultados obtenidos a la hora de usar NodeJS en la parte de del servidor.

En el proyecto de la empresa tenemos dos servidores con cuatro procesadores y dos gigabytes de RAM cada uno. Para balancear la carga entre ambos tenemos un equipo con HAProxy. Para lanzar las pruebas contra estos servidores tenemos cuatro

ordenadores doble núcleo con cuatro gigabytes de RAM. A la hora de lanzar las pruebas observamos que la infraestructura descrita en la parte superior es capaz de soportar 500 conexiones por segundo sin problemas. Viendo los procesos de las máquinas observamos que cuando los servidores están al cien por cien de capacidad de carga, la mayoría de los recursos están siendo utilizados por el sistema de base de datos local. Los recursos usados por las instancias de NodeJS apenas llegan al 10% de los recursos de la máquina.

Para ver cuantas peticiones es capaz de recibir un servidor NodeJS se ha creado un servidor, y un programa que lanza peticiones (Resultados en el capítulo 4.2). Tras varias ejecuciones del programa obtenemos que un servidor monoproceso es capaz de atender alrededor 4.200 peticiones por segundo, usando apenas 60 megabytes de memoria RAM.

Como se puede observar, NodeJS nos permite el desarrollo rápido de aplicaciones web, además de darnos un rendimiento y uso de recursos más que aceptable. En mi opinión es una de las mejores opciones para desarrollos ágiles que no tengan un coste computacional muy elevado.

5.3. MongoDB, base de datos noSQL

A pesar de que a lo largo de la carrera no se nos ha enseñado ninguna base de datos noSQL, no es la primera base de datos de este tipo que utilizo. El sistema de base de datos noSQL que utilicé anteriormente fue JDO [30], utilizando como PaaS el servicio AppEngine [27] de Google. Gracias a esta experiencia en este sistema de base de datos noSQL, y por la simplicidad de MongoDB el tiempo de aprendizaje fue mínimo.

A la hora de configurar y usar MongoDB no tuve ningún problema debido a que, además de ser muy sencillo e intuitivo, utiliza JavaScript en la consola, lenguaje con el que estoy muy familiarizado. Además, al igual que el resto de los componentes de la plataforma MEAN, está diseñado para el desarrollo rápido de aplicaciones, por lo que las consultas básicas se realizan de manera muy fácil. Al igual que en JavaScript, el formato utilizado es JSON, por lo que no tienes que aprender ningún formato adicional ni realizar conversiones de formato.

Respecto al driver usado, Mongoose, es de gran utilidad cuando estás acostumbrado a un sistema en el que hay que definir los objetos a almacenar previamente, por ejemplo en SQL. Más adelante, tras la realización de las prácticas en la empresa en la que utilizo MongoSkin como driver, me di cuenta de la libertad que pierdes y el trabajo adicional que supone usar Mongoose. Las grandes ventajas que supone el uso de Mongoose es que te ayuda a no cometer errores en las inserciones, ya que los tipos y los campos están definidos previamente, y el hecho de que te ayuda a separar las funciones para acceder a la base de datos del resto del código.

Otra de las cosas que se han podido probar gracias a las prácticas en la empresa es el funcionamiento del Sharding de MongoDB. Este sistema nos permite repartir los datos entre varias instancias de las base de datos. De esta forma se puede mejorar el rendimiento de la base de datos escalando horizontalmente. Al igual que la instalación

de MongoDB, la configuración del Sharding se realiza de forma muy sencilla y está muy bien documentada. El rendimiento de las consultas incrementa notablemente cuando se realiza el Sharding sobre las colecciones de las bases de datos. Otro punto muy interesante es que nos permite elegir el ShardKey (que puede ser simple o compuesta), dejándonos elegir así la forma en la que se van a repartir los datos en los distintos Shards. Esto es de gran utilidad si sabes qué tipo de consultas vas a utilizar ya que te permite optimizar el reparto de los datos para mejorar el rendimiento de las consultas.

5.4. Mejoras del centro de control

Tras enseñar el proyecto a profesores, compañeros y amigos he recibido algunas ideas para mejorar el centro de control. Estas mejoras se centran sobre todo en la interfaz y en la forma de acceder a la herramienta.

5.4.1. Interfaz

La interfaz es uno de los puntos en los que más se puede avanzar y mejorar. A pesar de que la herramienta aún no tiene usuarios como tal, he recibido algunas sugerencias y consejos.

Por un lado debería hacerse un mejor aprovechamiento del espacio. Actualmente los datos se muestran con una gran separación para visualizar de forma más rápida los resultados, aunque esto hace que el volumen de datos que se muestran simultáneamente sea menor. Al ser una herramienta de trabajo habría que buscar un mejor equilibrio entre el volumen de datos mostrados y la separación de los mismos.

Otra sugerencia que se realizó fue la de incluir filtros y la posibilidad de ordenar de los listados mostrados. Actualmente se muestra lo más reciente en la parte superior. Un sistema de filtrado podría ayudar a encontrar rápidamente los errores, o los resultados del robot que nos interese.

5.4.2. Centro de control como servicio

Actualmente cuando se despliega un centro de control todos los usuarios del mismo acceden al mismo dashboard, es decir, un usuario despliega el centro de control para uso propio, o de su equipo. La idea del centro de control como servicio consiste en permitir registros de usuarios, los cuales tendrían acceso a su propio dashboard, sin necesidad de una infraestructura propia. Cada usuario tendría sus configuraciones de robots y sus planificaciones.

5.5. Mejoras del servidor de robots

Aunque el desarrollo del servidor de robots se hizo únicamente para demostrar el funcionamiento del protocolo y del centro de control, se ha construido una base sobre la que poder seguir avanzando y mejorando. Algunas mejoras que se podrían realizar en siguientes iteraciones serían añadir una interfaz al servidor, programarlo como servicio, y mejorar el actual sistema de despliegue.

5.5.1. Interfaz

Una interfaz gráfica sería de gran utilidad para controlar qué está pasando en el servidor de robots, quiénes están realizando peticiones y cómo están yendo las mismas. Como los robots se exponen al exterior como un servicio, tener un panel de control de usuarios ayudaría a gestionar quienes pueden realizar peticiones al servidor. Por otro lado nos interesa poder ver cómo están yendo las ejecuciones de los robots desplegados para detectar cuanto antes los fallos y poder dar un buen servicio. Para esta tarea también se podría crear un sistema de alarmas que avisen a los desarrolladores en caso de que algo vaya mal.

5.5.2. Acceso a los datos

En el sistema actual, los datos obtenidos por el robot se guardan en el servidor de robots en ficheros de texto. Sería muy interesante crear un sistema que nos permita acceder y visualizar estos datos.

5.5.3. Servidor de robots como servicio

El servidor de robots, al igual que el centro de control, está pensado para que sea desplegado por el usuario que va a hacer uso del mismo. El usuario que pone a funcionar la instancia del mismo puede desplegar nuevos robots y configurar el servidor. Lo que se propone es crear un sistema en el que un desarrollador de robots pueda crearse una cuenta en el servidor de robots, desplegarlos allí, y configurar después los accesos a los mismos. De esta forma un desarrollador no necesitaría de una infraestructura en la que desplegar los robots.

5.5.4. Mejora del sistema de despliegue

Una de las grandes dificultades encontradas es el crear un sistema de despliegue de robots en el que la configuración posterior sea mínima. A pesar de que se ha alcanzado una solución óptima para el despliegue de robots en JavaScript, se sigue teniendo problemas con despliegues de otros lenguajes, ya que supone hacer llamadas al sistema pudiendo comprometer la seguridad del mismo.

Capítulo 6. Bibliografía

[1] PFC ingeniería Técnica de informática de gestión: Sistemas de extracción de datos de la web y posibles usos, Asier Galech Zuazu, 2013
[2] Selenium: http://www.seleniumhq.org/
[3] Watir: http://watir.com/
[4] Curl de PHP: http://curl.haxx.se/
[5] URL de Python: https://docs.python.org/2/library/urllib.html#
[6] TFG en ingeniería informática: A MODEL FOR PRODUCT/ CATALOG INTEGRATION: Data Processes Design and Prototype Implementation, González de Mendívil Grau, 2014
[7] TFG en ingeniería informática: ESTUDIO SOBRE LA DETECCION DE DUPLICADOS EN ORIGENES DE DATOS HETEROGENEOS, Ion Gorostizu, 2014
[8] PFC ingeniería informática: Planificación y monitorización de sistemas basados en robots web, Asier Galech Zuazu, 2014
[9] Scrappy de Python: http://scrapy.org/
[10] PaaS: https://en.wikipedia.org/wiki/Platform_as_a_service
[11] MEAN stack: http://mean.io/#!/
[12] GIT: https://git-scm.com/
[13] Twelve factor: http://12factor.net/
[14] Kumori: http://www.kumori.systems/
[15] TFG en ingeniería informática: Replicación Activa en NodeJS con 0MQ, Raúl Oses, 2014
[16] LAMP: http://es.wikipedia.org/wiki/LAMP
[17] JavaScript: http://es.wikipedia.org/wiki/JavaScript
[18] NodeJS: https://nodejs.org/
[19] NPM: https://www.npmjs.com/
[20] Express: http://expressjs.com/es/
[21] JADE: http://jade-lang.com/
[22] AngularJS: https://angularjs.org/
[23] SPA: http://es.wikipedia.org/wiki/Single-page_application
[24] MVC:
http://es.wikipedia.org/wiki/Modelo%E2%80%93vista%E2%80%93controlador
[25] JQuery: https://jquery.com/
[26] Heroku: https://www.heroku.com/
[27] AppEngine: https://cloud.google.com/appengine/docs
[28] Mongoose: http://mongoosejs.com/
[29] Bootstrap: http://getbootstrap.com/
[30] JDO: https://en.wikipedia.org/wiki/Java_Data_Objects
[31] MapReduce: http://docs.mongodb.org/manual/core/map-reduce/

Herramienta de gestión y planificación de robots web basado en la arquitectura MEAN

Carlos Martínez González

Tutor: José Ramón González de Mendívil

Pamplona, 24 de Junio de 2015

upna Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Trabajos anteriores del grupo de sistemas distribuidos

- **Sistemas de extracción de datos de la web y posibles usos**, Asier Galech, 2013
- **A MODEL FOR PRODUCT/ CATALOG INTEGRATION: Data Processes Design and Prototype Implementation**, Aitor González de Mendívil, 2014
- **ESTUDIO SOBRE LA DETECCIÓN DE DUPLICADOS EN ORÍGENES DE DATOS HETEROGÉNEOS,** Ion Gorostizu, 2014
- **Planificación y monitorización de sistemas basados en robots web**, Asier Galech, 2014

The Webby Scrappy, objetivos

- Funcionalidad
  - Gestión de configuraciones de robots
  - Gestión de planificaciones de robots
  - Visualización de logs de ejecución
- Distribuido y escalable
  - Centro de control
  - Servidor de robots
- Nuevos paradigmas de programación
  - MEAN Stack

Nuevos paradigmas de programación

- Plataform as a Service (PaaS)
- Bases de datos noSQL
- Sistemas de servidores con tecnologías no-bloqueantes
- Escalabilidad horizontal

Plataform as a Service (PaaS)

- **Servicio WEB** que proporciona una plataforma en la que desarrollar, desplegar y gestionar aplicaciones web

- Ventajas
  - Elimina la complejidad y costes de construir y mantener la infraestructura
  - Permite centrarse en el desarrollo de la aplicación
  - Facilita el trabajo en equipo

- Heroku
  - Despliegue usando GIT
  - Ruby, Node.js, Python, Java, y PHP
  - Addons para bases de datos, monitorización, sistemas de correo electrónico, …
  - Escalabilidad horizontal y desarrollo ágil, los 12 factores

Bases de datos noSQL

- **Sistema de gestión de bases** de datos que difieren del modelo clásico relacional, y no usan SQL como lenguaje de consultas

- Ventajas
  - Buena respuesta a la escalabilidad horizontal
  - Capacidad de manejo de grandes volúmenes de datos
  - Mayor libertad a la hora de almacenar los datos
  - Desarrollo ágil y evolutivo

- MongoDB
  - Modelo de datos flexible
  - Alta escalabilidad horizontal
  - Alta disponibilidad

Sistemas de servidores con tecnologías no bloqueantes

- Utilizan señales para saber cuando están listos los datos en un socket y de esta forma poder realizar otras tareas mientras no se reciben datos

- Ventajas
  - Uso de menor número de procesos
  - Uso más eficiente de los procesos
  - Menor coste por petición

- NodeJS como servidor
  - Basado en eventos
  - Librería de funciones muy completa
  - Sistema gestor de paquetes (NPM)
  - Gran comunidad

NodeJS, simple y rápido

```
require('http').createServer(function(req, res) {
        res.writeHead(200);
        res.end('Hello world!');
}).listen(8000);
```

Escalabilidad horizontal

- A mayor número de nodos, mayor rendimiento

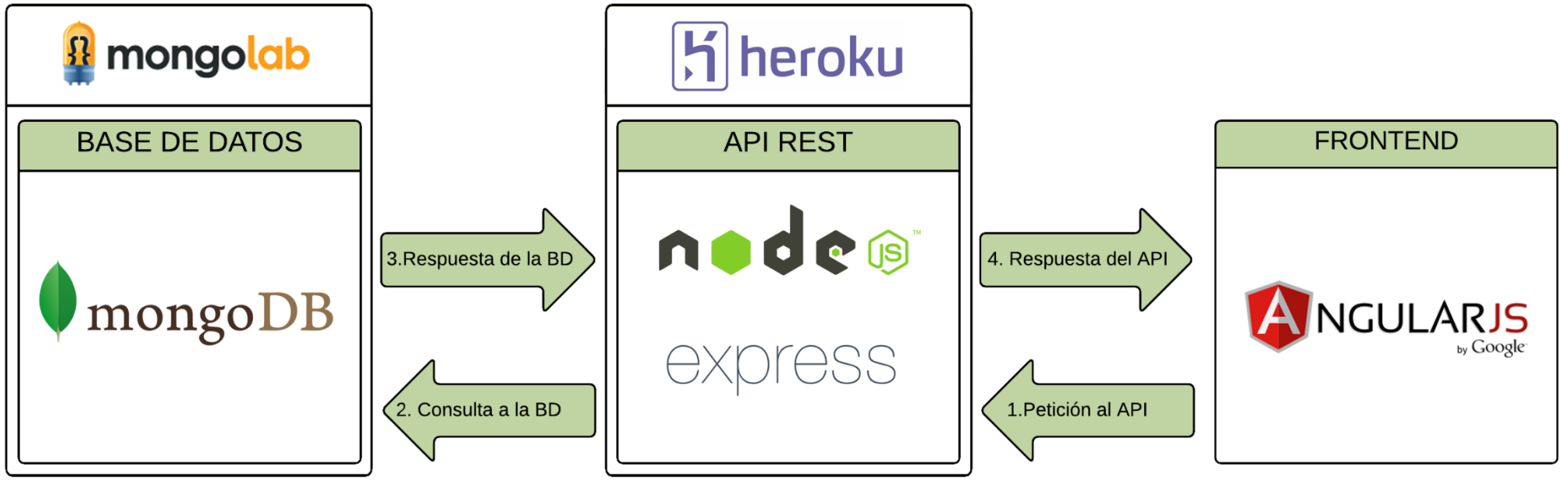
- Ventajas
  - Aumento de coste lineal
  - Más económico
  - Sin límite

- Los 12 factores de Heroku
  - Uso de formatos declarativos
  - Máxima independencia del sistema operativo
  - Minimizar la divergencia entre el entorno de desarrollo y el de producción
  - Creación de servicios independientes

MEAN stack

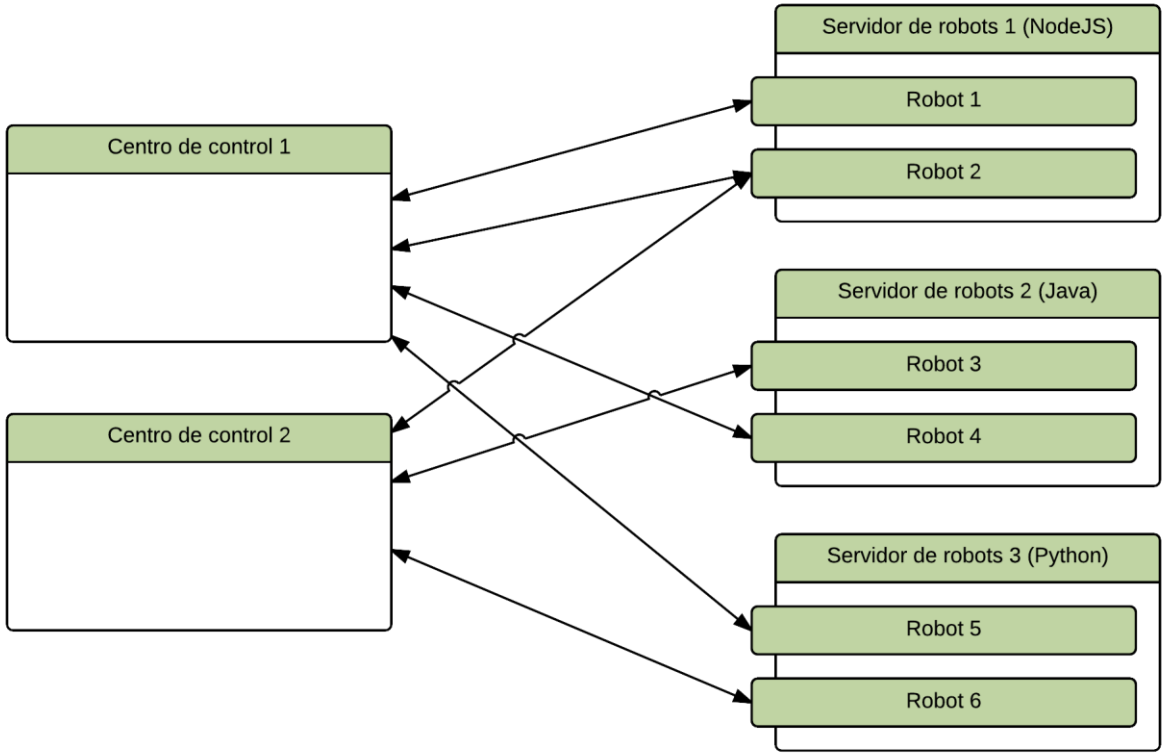
- Componentes
  - MongoDB
  - Express
  - AngularJS
  - NodeJS
- Ventajas
  - Mismo lenguaje en todas las capas de la aplicación: **JavaScript**
  - Desarrollo rápido y ágil
  - Aplicaciones robustas

Mean stack: chat demo

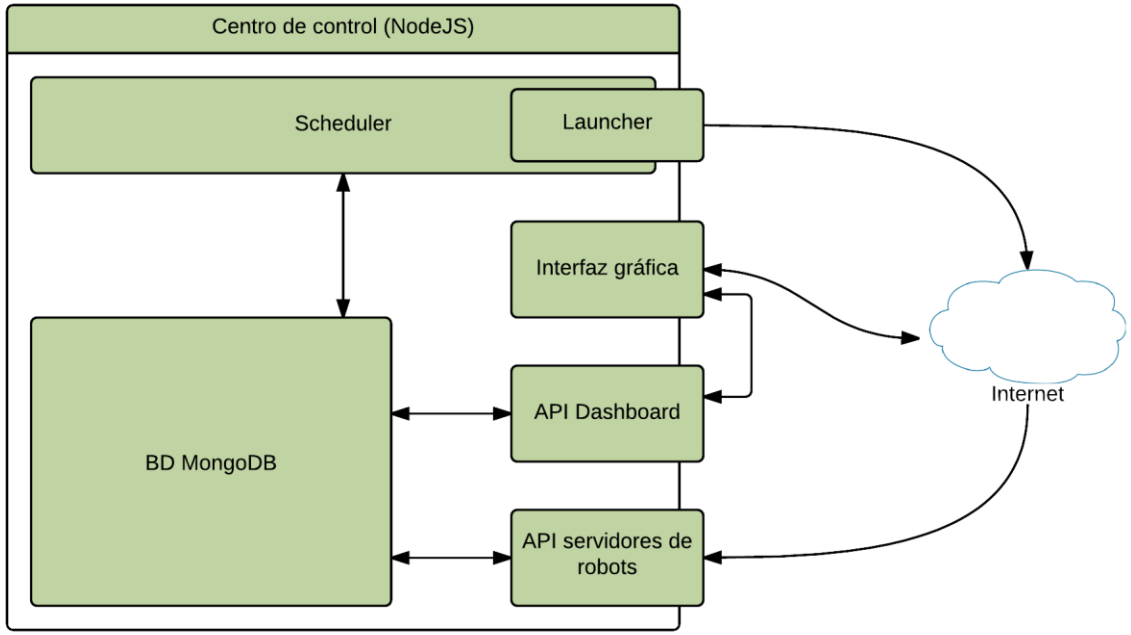


- Link: https://demochatcarlos.herokuapp.com/

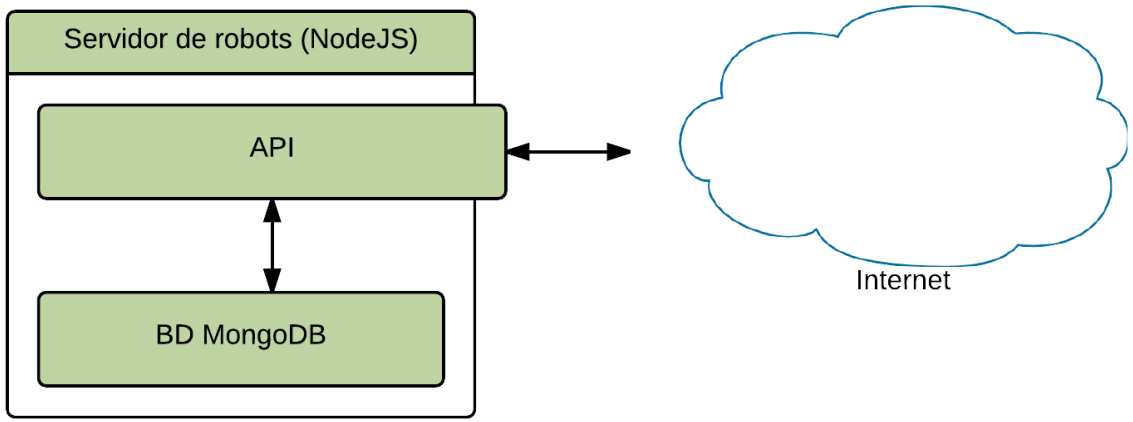
The Webby Scrappy: Arquitectura general



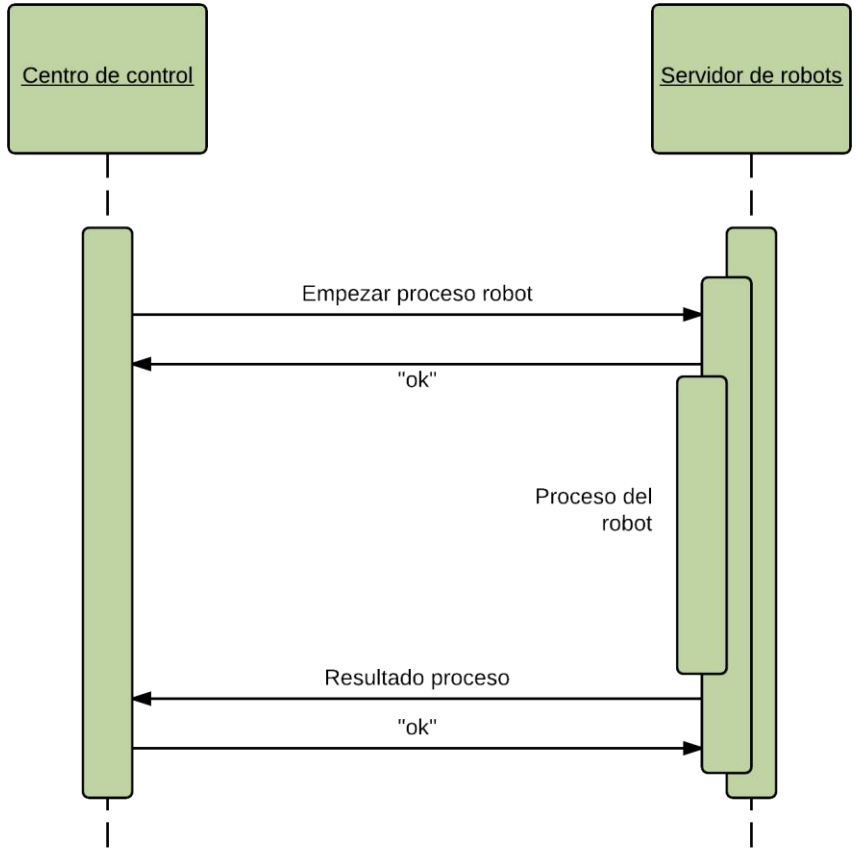
Arquitectura: Centro de control



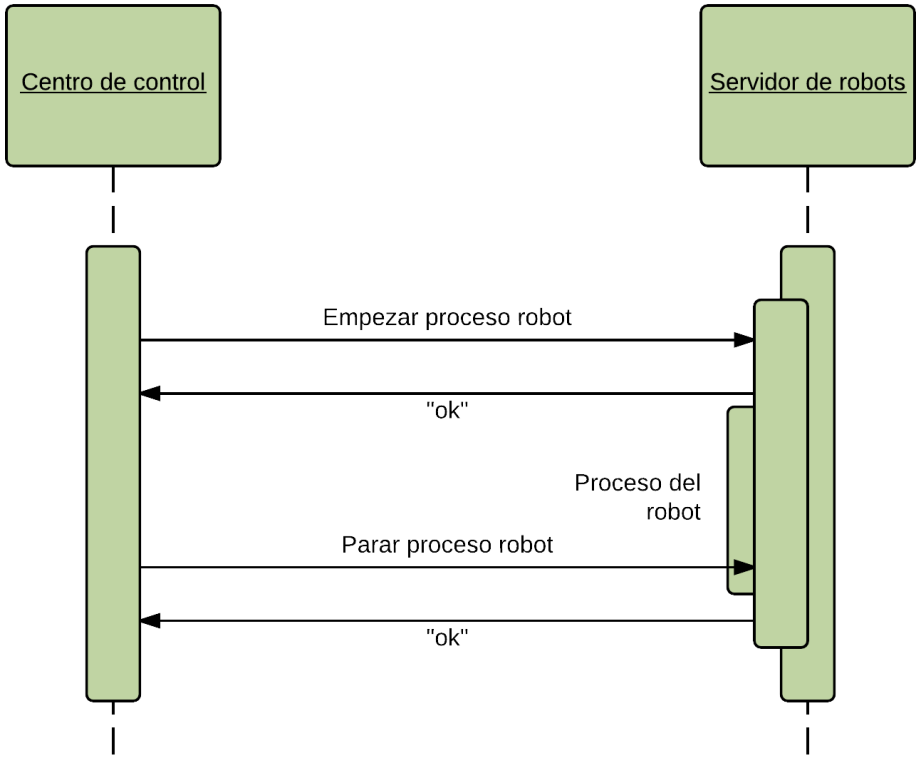
Arquitectura: Servidor de robots



Protocolo: Iniciar robot



Protocolo: Parar robot



Resultados: The Webby Scrappy

- Datos de las pruebas
  - Número de robots: 3
  - Intervalo de ejecución: 5 minutos
  - Duración de la prueba: 4 horas

- Resultados obtenidos
  - Número de ejecuciones: 144
  - Productos descargados: 294.048
  - Espacio en disco ocupados por los datos: 285 MB

	Tiempo medio (ms)	Tiempo máximo (ms)	Tiempo mínimo (ms)	#Productos por ejecución
Robot 1	501	931	349	150
Robot 2	689	1750	438	237
Robot 3	5913	11019	4435	5739

Resultados: NodeJS como servidor

- Servidor: 1-4 procesos; Requester: 1 proceso
  - Límite de peticiones por segundo producido por el requester
- Servidor 4 procesos; Requester: 4 procesos
  - Límite producido por la capacidad del procesador
  - 5 641,97 peticiones por segundo
- Servidor 1 proceso, Requester: 4 procesos

Nº TEST \ resultado	Nº reqs ok	Tiempo máximo	Reqs/sec	Max RAM (MB)
1	400 000	92 450 ms	4 326,66	14 + 60
2	400 000	95 768 ms	4 189,89	14 + 60
3	400 000	97 970 ms	4 082,88	14 + 60
4	400 000	97 940 ms	4 084,13	14 + 60
Media	**400 000**	**96 032ms**	**4 170,89**	**14 + 60**

Conclusiones, experiencias y líneas futuras

CONCLUSIONES Y EXPERIENCIAS

- Prácticas en Biko2
- Curva de aprendizaje de NodeJS
  - Asincronía
  - Callbacks
- NodeJS como servidor
- MongoDB como base de datos

LÍNEAS FUTURAS

- Mejoras del centro de control
  - Interfaz
  - Centro de control como servicio
- Mejoras del servidor de robots
  - Interfaz
  - Acceso a los datos
  - Sistema de despliegue para robots que no estén en JavaScript
- MVP Kumori Systems

```
console.log("Gracias a todos!");
process.exit();
```