Trabajo de Final de Grado

GRADO DE INGENIERÍA INFORMÁTICA

Facultad de Matemáticas
Universidad de Barcelona

MMO de Navegador en Tiempo Real con Node.js y WebSockets

Alejandro Castrelo Cid

Director: Eloi Puertas Prats
Realitzat a: Departament de Matemàtica Aplicada i Anàlisi. UB.

Barcelona, 20 de juny de 2014

Contenido

1 Introducción

1.1 Ámbito del proyecto y motivación

No es nada nuevo entrar en alguna página de internet y encontrarse publicidad de juegos de "Regístrate y juega", que venden su producto con un "No necesitas instalar nada". Y es que, últimamente el mercado de estos juegos se ha incrementado bastante. Suelen ser juegos de estrategia y multijugador, en un mundo lleno de jugadores con los que podemos interactuar.

En mi caso, he jugado durante bastante tiempo a uno de estos juegos. Quizás el más famoso y más jugado de todos. Se llama OGame y está ambientado en batallas galácticas y conquista de planetas. Este es uno de los principales factores que me ha motivado a realizar este proyecto, junto con las ganas de aprender tecnologías nuevas, las cuales mencionaré en fases posteriores de este documento.

Otra motivación personal ha sido que jamás había hecho una aplicación web. Ni siquiera había programado alguna web en HTML. El motivo principal de este proyecto es aprender todo lo posible sobre cosas que desconocía, como programación del servidor en JavaScript, o bases de datos no relacionales.

1.2 Objetivos generales del proyecto

- Crear un juego de estrategia masivo en tiempo real que se pueda jugar a través del navegador web.
- Conocer a fondo nuevas tecnologías para el desarrollo de aplicaciones cliente/servidor, como Node.js, WebSockets y Jade.
- Comprobar el funcionamiento de las bases de datos no relacionales en profundidad, así como su estabilidad.

1.3 Objetivos específicos

- Utilizar únicamente JavaScript como lenguaje de programación, tanto en el cliente como en el servidor.
- Mejorar la interacción entre el cliente y el servidor mediante WebSockets, creando una comunicación bidireccional, *full-dúplex* y en tiempo real.
- Mejorar los conocimientos sobre desarrollo de páginas web como interfaces de aplicaciones (distribución de elementos, claridad de los textos, esquemas de colores, etc.).

1.4 Respecto al diseño de la aplicación

La aplicación web de este proyecto ha sido ejecutada en el *localhost* para realizar las pruebas, ya que aparte de ser la forma más cómoda de probar la aplicación mientras estamos en fase de desarrollo, todavía no existen muchas plataformas gratis para hospedar aplicaciones con Node.js.

El diseño de la página no está finalizado al 100%, ya que no me he centrado en exceso en este apartado. Además, necesitaría a una persona que me hiciera las imágenes para la web, de manera que me cediera sus derechos. Es bastante difícil encontrar un set de imágenes *open source* que estén, por así decirlo, conjuntadas.

1.5 Estructura de la memoria

En este apartado se explica brevemente el contenido de este documento, detallando los diferentes apartados que lo componen y dando un rápido vistazo a su contenido.

- *Capítulo 1 – Introducción:* Se realiza un resumen del estado actual de los juegos MMO de navegador, así como las motivaciones que han llevado a crear este proyecto. También se

enumeran los objetivos, tanto específicos como generales, que se han perseguido a lo largo del mismo.

- *Capítulo 2 – Análisis y diseño:* En el segundo capítulo se habla del análisis de especificaciones del proyecto, de las tecnologías utilizadas a partir de dichas decisiones, y el diseño de la aplicación en todos sus componentes.
- *Capítulo 3 – implementación:* En este capítulo se hablará de la implementación del código de la aplicación, y de cómo hemos utilizado las tecnologías para obtener los resultados requeridos en la fase de análisis.
- *Capítulo 4 – Resultados y posibles mejoras:* En el cuarto capítulo haremos un breve tour por el juego, explicaremos las posibles mejoras que se podrían implementar, y hablaremos sobre las opciones más viables para colocar nuestra aplicación en un servidor.

2 Análisis y diseño

En este capítulo empezaremos explicando el estado actual del mercado de los juegos de navegador MMO-*Realtime*, luego haré unas definiciones básicas para entender el resto de la memoria, y acabaremos en los apartados de análisis y diseño de la aplicación.

2.1 Estado del mercado

Hoy en día, los juegos masivos en tiempo real por navegador son muy frecuentes por la red. Uno de los más conocidos y jugados, tanto por su calidad como por su antigüedad, es el conocido con el nombre de *OGame*. También existen otros juegos, como *Let's Farm* o *BiteFight*.

Aunque las temáticas pueden ser totalmente distintas entre juegos, todas comparten unos puntos clave en común: Son multiplayer, online, y en tiempo real. Además, solo necesitas un navegador para poder jugar a estos juegos.

Cabe decir que la intención de este proyecto no es competir con estos juegos, ya que para ello necesitaría un equipo especializado de diseñadores y desarrolladores. En vez de eso, se espera mejorar ciertos aspectos técnicos de los juegos ya existentes, aparte de, como ya se ha mencionado con anterioridad, aprender a utilizar las nuevas tecnologías.

Observando el funcionamiento de estos juegos, me he fijado que podrían ser más dinámicos si se aplicaran las tecnologías adecuadas. En el caso de OGame, se basan en una tecnología HTTP no asíncrona, y todo apunta a que utilizan PHP en la parte del servidor. Yo utilizaré JavaScript en el servidor mediante Node.js, e implementaré un protocolo de comunicación basado en los nuevos WebSockets, de manera que tanto el cliente como el servidor puedan realizar una petición asíncrona cuando quieran.

2.2 Conceptos sobre el proyecto y los juegos MMO de navegador.

En este apartado se explican algunos de los términos que se van a utilizar durante el resto de la documentación.

MMO-Realtime

MMO proviene de *Massive Multiplayer Online*. De forma genérica, un MMO es un juego multijugador en línea donde una gran cantidad de jugadores concurren en una misma partida, generalmente en un mundo abierto donde pueden interactuar para competir o cooperar para realizar objetivos.

Normalmente se utiliza para referirse de modo abreviado a los MMORPG, que sigue siendo MMO, pero además con las características de un RPG (*Role Play Game*). En nuestro caso, el juego se considera simplemente un MMO.

Por otra parte, en cuanto a la extensión *Realtime* (Tiempo real), se refiere a un tipo de juego en el cual el tiempo transcurre de forma continua para los jugadores, sin que haya ningún tipo de pausa entre las acciones del jugador y lo que sucede en el juego.

Como podemos ver, juntando las definiciones de MMO y Tiempo real, obtenemos un juego masivo online en el cual el tiempo es común para todos, y tanto si estoy realizando acciones o, simplemente, no estoy conectado, el tiempo sigue pasando igualmente para mí como para el resto de jugadores.

Gestión de recursos

Un juego basado en la gestión de recursos, es un juego en el cual empezamos con ciertos recursos básicos, y tenemos que invertirlos en mejorar en varios aspectos del juego.

Una regla de oro en estos juegos, es que siempre hay algún generador de recursos, el cual si queremos ir mejorándolo tendremos que gastar de nuestros recursos. Se puede decir, en cierto modo, que es un juego basado un poco en la inversión. Pagamos un coste fijo a cambio de generar un poco más de recursos en función del tiempo. El ejemplo perfecto para mencionar un gestor de recursos es el simple, pero adictivo juego con el nombre de *cookieclicker*. Invertir galletas en una mayor producción de galletas por segundo.

2.3 Análisis

En este apartado se van a detallar la arquitectura que hemos decidido utilizar en este proyecto, así como las elecciones importantes de cada uno de las componentes claves del mismo.

2.3.1 Arquitectura del proyecto

Para realizar nuestra aplicación web, nos hemos basado en la típica estructura cliente-servidor, ya que es la que se adapta a nuestras necesidades: un servidor que recibe peticiones y se las sirve al cliente mediante su navegador web.

Además, hemos escogido una arquitectura MVC (Modelo-Vista-Controlador). Como se puede apreciar en la *Ilustración 1*, el navegador envía una petición a nuestro servidor (controlador), que a su vez se comunica con la base de datos (modelo). Este devuelve la información al servidor, que genera las plantillas adecuadas (vista) para mostrar en el navegador.

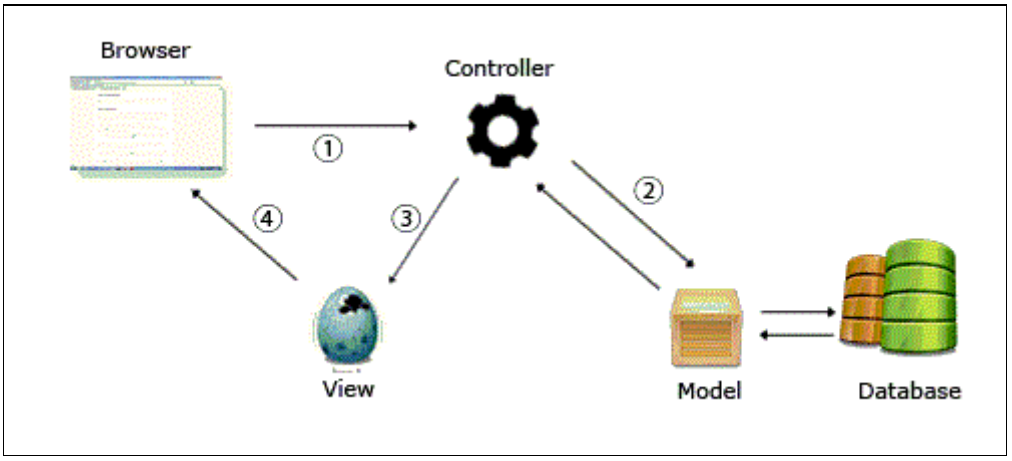

Ilustración 1 – Esquema del flujo de la arquitectura MVC

2.3.2 Elección de lenguaje para el servidor

Esta elección fue la más importante del proyecto ya que, al ser el lenguaje del servidor el elemento fundamental del mismo, ha condicionado las elecciones realizadas posteriormente.

Normalmente, un servidor está programado mediante PHP, ya que es el lenguaje más conocido y estandarizado de todos los posibles. No obstante, nosotros nos fijamos en una opción que probablemente sería mejor para un proyecto como el nuestro: una aplicación en tiempo real, donde es más importante la programación asíncrona y orientada a eventos.

Aunque JavaScript sea visto más bien como un lenguaje de programación para el navegador web, gracias a Node.js podemos programar el servidor en JavaScript. Fue por este motivo por el cual decidimos arriesgarnos a utilizar Node.js, haciendo que nuestro servidor sea completamente asíncrono y orientado a eventos.

2.3.3 Elección de motor de plantillas

Toda aplicación web necesita generar páginas dinámicamente. La mayoría utilizan el propio PHP del servidor, que permite inserción de código HTML directamente.

Puesto que nosotros no vamos a utilizar PHP en el servidor, necesitamos un motor de plantillas que se adapte a nuestras necesidades, es decir, que sea dinámico y totalmente compatible con la arquitectura de nuestra aplicación.

Estuve buscando durante bastante tiempo, hasta que me di cuenta que la mejor solución era dejar de lado incluso el HTML, y utilizar un motor llamado Jade, el cual genera las plantillas desde el servidor y las envía al cliente en formato HTML.

Jade[1] es un lenguaje de plantillas desarrollado por el creador de Express (framework de Node.js) para simplificar la sintaxis de HTML y acelerar el proceso de desarrollo.

2.3.4 Elección de la Base de Datos

He aquí otra elección poco corriente. De hecho, se podría decir que todo mi proyecto ha sido elaborado con tecnologías poco comunes, pero a la vez eficientes y adaptables.

Hoy en día, la mayoría de aplicaciones utilizan bases de datos relacionales, es decir, que permiten establecer interconexiones entre datos guardados en tablas, y relacionarlos a través de dichas conexiones. Nosotros, sin embargo, hemos querido experimentar la estabilidad de una base de datos no relacional, ya que ofrece ciertas ventajas en situaciones como la nuestra.

Una de las principales ventajas es la rapidez con la que manipula los datos complejos. En nuestro caso, queremos guardar bastante información pero con pocas relaciones; es decir, tenemos un

[1] Explicado de forma completa en el apartado *2.4.4 Jade*

objeto para guardar la información de la cuenta del usuario, y otro para los datos del usuario dentro del juego. Debido a la poca cantidad de relaciones de nuestra base de datos, escoger una estructura no relacional ha supuesto una gran ventaja, ya que, además de todo lo mencionado anteriormente, son bastante más simples de utilizar.

Después de una búsqueda exhaustiva, escogimos *MongoDB*[2], ya que de las bases de datos no relacionales, consideramos que era la que mejor se integraba con Node.js. MongoDB guarda los datos en documentos tipo JSON con un esquema dinámico (MongoDB llama a este formato BSON).

2.3.5 Elección del protocolo de comunicación

Uno de los requisitos que buscábamos para nuestro protocolo de comunicación entre el cliente y el servidor era que fuera, a ser posible, bidireccional.

La mayoría de aplicaciones se basan en el protocolo HTTP (*HyperText Transfer Protocol*), que utiliza una comunicación de petición-respuesta. Es decir, el cliente envía una petición y el servidor responde. Lo malo de éste método es que el servidor no puede enviar nada hasta que el cliente envíe otra petición. Hay varias técnicas para simular una comunicación bidireccional, como por ejemplo el 'long-polling', en la cual el cliente envía una petición y se espera hasta que el servidor tenga algo que decirle. Nosotros hemos encontrado una forma de crear una comunicación completamente bidireccional y que, además, encaja perfectamente en nuestro entorno de trabajo con Node.js: los WebSockets[3].

La idea básica de un WebSocket es mantener una conexión persistente abierta entre un servidor web y un navegador. Esto lo que permite es que tanto el navegador como el servidor envíen datos

[2] Explicado de forma completa en el apartado *2.4.2 MongoDB*
[3] Ecplicado de forma completa en el apartado *2.4.3 WebSockets*

cuando lo deseen. Como la conexión es persistente, el intercambio de datos es muy rápido, de ahí que se use el término 'en tiempo real'.

En nuestro caso, hemos usado la librería JavaScript para aplicaciones web en tiempo real 'socket.io', que permite una implementación sencilla de WebSockets. Consta de dos partes: una para el cliente que corre en el navegador, y otra para el servidor, que es básicamente una librería para Node.js.

2.4 Tecnologías

En este apartado vamos a explicar de forma breve y concisa las tecnologías principales utilizadas en este proyecto. Dichas tecnologías son: Node.js, MongoDB, WebSockets (socket.io) y Jade.

2.4.1 Node.js

Node.js es un entorno de programación en la capa del servidor basado en JavaScript, con I/O de datos en una arquitectura orientada a eventos y basado en el motor V8 de Google. Es una idea completamente revolucionaria ya que, a diferencia de las aplicaciones web tradicionales que usan PHP en el servidor, utiliza JavaScript, un modelo completamente asíncrono.

Uno de los principales motivos por el cual he escogido Node.js, es porque es especialmente útil cuando se van a realizar muchas operaciones simultáneas, sobretodo operaciones I/O. También es realmente bueno para aplicaciones *Realtime*, ya que necesitan una conexión persistente entre el cliente y el servidor.

Además, consta de un gestor de paquetes (NPM o *Node Package Manager*), que facilita la tarea de incluir librerías en nuestro proyecto. Todas las librerías utilizadas en mi proyecto (y que explicaré a continuación) han sido incluidas utilizando este gestor.

Las librerías de Node.js instaladas son: *express* (explicado a continuación), *mongoose*, *connect-mongo*, *jade* y, por último, *socket.io*

Express

Express es un framework de desarrollo de aplicaciones web minimalista y flexible para Node.js. Ofrece, entre otras características, un *router* de URL (*get*, *post*, *put*), el cual vamos a usar para capturar los diferentes eventos que se puedan producir, desde cambiar de página, hasta hacer operaciones con la base de datos.

Además, Express nos ofrece la posibilidad de crear un proyecto predefinido con las funcionalidades básicas y a partir del cual empezar a desarrollar nuestra aplicación siguiendo los fundamentos del *framework*. El proyecto generado tiene la siguiente estructura:

Ilustración 2 – Estructura básica de un proyecto generado con Express.

- **node_modules:** Contiene las librerías instaladas mediante NPM.
- **public:** Contiene todos los recursos que estarán disponibles para la parte del cliente, tales como imágenes, ficheros JavaScript, u hojas de estilo (CSS).
- **routes:** Aquí se encontrarán los métodos que se empezarán la lógica del sistema cuando se capte un evento mediante *get* o *post*. Se puede decir, que es como el intermediario entre la petición del cliente y la ejecución de la misma.

- **views:** Contiene la vista del proyecto, las plantillas que generarán nuestra página web. Como he mencionado en el apartado anterior, en este proyecto vamos a utilizar JADE para poder generar páginas dinámicas.
- **app.js:** El fichero principal de la aplicación, el cual se ejecutará para arrancarla.
- **package.json:** Este fichero incluye las declaraciones de las dependencias de nuestra aplicación.

En cuanto a la creación del servidor para nuestra aplicación web, Express también nos facilita la faena. De todos modos, nosotros hemos creado el servidor utilizando WebSockets, de manera que esa parte fue omitida. Mencionar también que muchas de las librerías para node.js (como *socket.io*) son compatibles con *express*, de modo que su uso también se simplifica bastante.

2.4.2 MongoDB

MongoDB es la base de datos NoSQL más utilizada, y que permite crear aplicaciones ágiles y escalables. Desarrollado bajo el concepto de *open source*, utiliza documentos en vez de tablas como lo hacen las bases de datos relacionales. Dichos documentos guardan objetos JSON, lo cual hace que sea mucho más simple crear las estructuras de datos.

Las características más importantes de MongoDB son las siguientes:

- **Indexación:** Cualquier campo en un documento puede ser indexado, al igual que es posible hacer índices secundarios.
- **Consultas Ad hoc:** Soporta la búsqueda por campos, consultas de rangos y expresiones regulares. Las consultas pueden devolver un campo específico del documento, pero también puede ser una función JavaScript definida por el usuario.

- **Replicación:** MongoDB soporta el tipo de replicación maestro-esclavo. El maestro puede ejecutar comandos de lectura y escritura, mientras que el esclavo puede copiar los datos del maestro y realizar operaciones de lectura. Esto es útil para realizar copias de seguridad.
- **Balanceo de carga:** MongoDB se puede escalar de forma horizontal usando el concepto de *shard*, que consiste en guardar la información en varias máquinas, ya que cuando se necesita almacenar muchos datos, a veces con una máquina no tenemos suficiente.

Para utilizar MongoDB en nuestro proyecto, hemos utilizado la librería *mongoose*, que nos facilita ciertas tareas y, en concreto, nos provee de un nuevo objeto, el esquema (*Scheme*), que se utilizará para crear la estructura de los docuemntos que guardaremos en nuestra base de datos.

2.4.3 WebSockets

Como todos sabemos, la web no nació para ser dinámica. Su objetivo era mostrar ficheros de texto de uno en uno. El ciclo web originario partía de un usuario que solicitaba una página web desde el navegador, obtenía una respuesta del servidor, y el mismo navegador mostraba la respuesta que había solicitado.

Poco a poco, los desarrolladores fueron creando tecnologías, tales como el JavaScript, para ampliar lo que una página puede hacer, prácticamente desde sus orígenes.

El paso más cercano a los WebSockets lo realizó AJAX. Su gran aportación fue que permitió a los desarrolladores solicitar datos desde un servidor para un elemento concreto de una página web sin tener que actualizar toda la página. Todo esto, además, lo realiza de forma asíncrona, con lo cual la comunicación es mucho más interactiva y fluida. El problema de AJAX es que la comunicación es unidireccional, es decir, solo va del cliente al servidor. Si queremos una comunicación completamente bidireccional, tendremos que hacer uso de los WebSockets, que nos permiten tener una comunicación bidireccional y en tiempo real.

WebSocket es una tecnología que proporciona un canal de comunicación full-dúplex (bidireccional) sobre un único socket TCP. Está diseñada para ser implementada en navegadores y servidores web, pero puede utilizarse en cualquier aplicación cliente/servidor. Como las conexiones TCP ordinarias sobre puertos distintos al 80 normalmente son bloqueadas por los administradores de redes, el uso de esta tecnología proporciona una solución a este tipo de limitaciones, ya que provee una funcionalidad similar a la apertura de varias conexiones en distintos puertos, pero multiplexando diferentes servicios WebSocket sobre un único puerto TCP.

2.4.4 Jade

Jade es un lenguaje de plantillas desarrollado por el creador de Express para simplificar la sintaxis de HTML y acelerar el proceso de desarrollo.

Este lenguaje intercambia tener que cerrar etiquetas HTML por la indentación, es decir, todo bloque de texto que esté hacia la derecha de la etiqueta que abre, significa que va dentro.

También elimina los símbolos "<" y ">", y los parámetros de las etiquetas se pasan entre paréntesis como si fueran de una función de cualquier lenguaje de programación.

Un ejemplo comparativo entre HTML y Jade:

```
h1(id="title") Welcome to Jade
button(class="btn", data-action="bea").
Be Awesome
```

Equivaldría a:

```
<h1 id="title">Welcome to Jade</h1>
<button data-action="bea" class="btn">Be Awesome</button>
```

Para añadir clases o IDs a los diferentes elementos, podemos utilizar el "." o la "#" en vez de pasarlo parámetro. Por ejemplo:

```
table.class1
table#id1
```

Equivale a:

```
table(class=”class1”)
table(id=”id1”)
```

También cabe mencionar, que si el elemento en cuestión es un *div*, se puede omitir. Por lo tanto:

```
.class1
```

Equivale a:

```
div.class1
```

Otra de las funcionalidades fundamentales por la cual he escogido Jade, es que este soporta JavaScript incrustado en el documento. De este modo, podemos añadir variables corrientes, listas, e incluso iterar con la sentencia *foreach*.

2.5 Diseño de la aplicación

En este apartado explicaremos el diseño final que ha tenido la aplicación. Primero se explicará el mapa web de la misma, y después se detallarán las decisiones más importantes que se han tomado en todos los componentes que forman el proyecto. Para acabar, se explicará de forma muy resumida qué herramientas hemos utilizado para la realización del proyecto.

2.5.1 Mapa web de la aplicación

A continuación podemos ver el mapa web de nuestra aplicación.

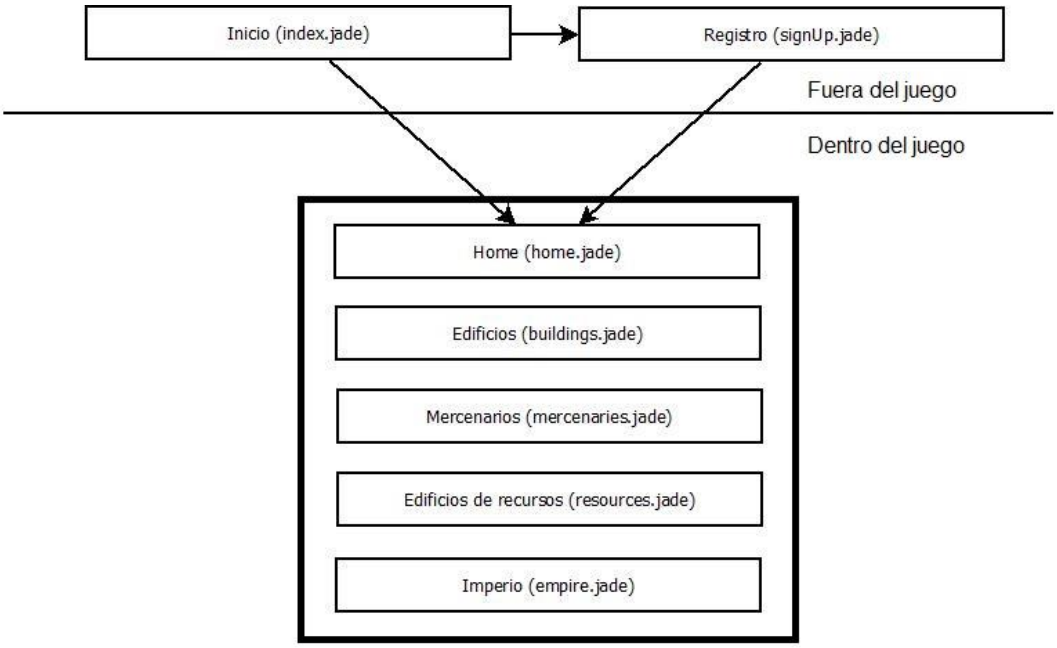

Ilustración 3. Mapa web de la aplicación. Tan solo dos páginas para el exterior del juego y 5 para el interior.

Como podemos ver, la estructura es bastante simple. Tenemos una página de inicio, donde está el formulario para identificarnos en el juego, el cual nos redirige a la página principal del juego (home.jade). También hay una zona para las futuras noticias importantes sobre el juego de cara a los jugadores. Pulsando el botón de "*Play for free*", o accediendo a la opción del menú "*Sign Up*", nos vamos a la página de registro, donde, después de crear una cuenta, se nos redireccionará al interior del juego. Como podemos ver, tanto desde el formulario de registro como en el de identificación se nos lleva a la página principal dentro del juego. Una vez estamos dentro del juego, tenemos un menú superior que nos permite navegar por todas las páginas del mismo.

2.5.2 Diseño del servidor

En este apartado voy a explicar cómo funciona el servidor internamente, es decir, como actúa en función de las peticiones que recibe, y como maneja las sesiones o las validaciones de datos.

Acciones

Empezaremos explicando de manera resumida las acciones que realizará el servidor cuando reciba las peticiones del cliente. Aclarar que las peticiones se realizan mediante *GET* y *POST*, y que no explicaré las acciones básicas, como la de moverse por los menús del juego, que constan simplemente de una pequeña redirección, enviando ciertos datos de la base de datos.

- *Registro:* Esta operación se ejecutará cuando el usuario quiera entrar al juego por primera vez. Se tomarán sus datos de usuario (nombre, contraseña y correo), y después de verificarlos, se guardarán en la base de datos. Acto seguido se redireccionará al usuario dentro del juego, con su cuenta ya creada y con su sesión inicializada.
- *Login:* Se ejecutará cada vez que un usuario previamente registrado quiera entrar al juego con su cuenta. Para ello, introducirá el usuario y la contraseña en la página principal del juego. En el servidor se comprobará que el usuario exista, y que la contraseña esté bien introducida y, de ser así, redireccionarlo dentro del juego.
- *Mejorar:* Por mejorar entendemos construir o subir de nivel un edificio. Se realizará cada vez que el jugador, dentro de uno de los menús donde hayan edificios, pulse el botón *upgrade*. El servidor recogerá la petición, y a través del id del edificio comprobará el coste actual de la acción. Si el usuario tiene suficientes recursos, procederá a subir de nivel el edificio y restar dichos recursos al jugador.
- *Contratar mercenarios:* Desde el menú de mercenarios, cada vez que el usuario quiera contratar uno o varios mercenarios, deberá introducir la cantidad deseada en el campo correspondiente y pulsar el botón *Hire*. El servidor, al igual que con los edificios, comprobará si el usuario dispone de recursos suficientes para realizar la operación y, en caso afirmativo, descuenta los recursos del jugador y añade a su lista de mercenarios los que ha contratado.
- *Atacar:* Desde el menú imperio, el jugador puede atacar al usuario que desee pulsando en la espada a la derecha del usuario. El servidor recuperará los mercenarios disponibles de los

dos jugadores, y se realizará una lógica de combate (por el momento sencilla), que determinará un ganador y un perdedor. El perdedor perderá todas sus tropas, mientras que el ganador las conservará todas.

- *Salir (logout):* Se cierra la sesión actual del jugador y devuelve a la página de inicio.

Gestión de sesiones

La gestión de sesiones se efectúa mediante el módulo de node.js llamado *connect-mongo*. Su funcionamiento es simple: almacenamos la sesión en la base de datos (MongoDB), de manera que conseguimos la persistencia de la sesión sin necesidad de usar PHP ni nada por el estilo (recordamos que no estamos usando PHP en absoluto).

Este módulo se encarga de encriptar automáticamente las sesiones, de modo que es prácticamente imposible que alguien pueda hacerse pasar por algún jugador. Cuando inicializamos nuestro servidor, de hecho, definimos una clave secreta que se utilizará para encriptar dichas sesiones.

La sesión se almacena en el *request*, con lo cual cada vez que el cliente realiza una petición podemos saber de qué jugador proviene. Además, para enviar los datos del jugador en cada petición, se hace una consulta a la base de datos utilizando la sesión para identificarlo.

2.5.3 Diseño de la Base de Datos

Como he mencionado en el apartado "*2.3.1 Elección de la Base de Datos*", no hemos escogido una base de datos relacional, con lo que nuestra información se guarda en forma de objetos. A continuación expongo las estructuras básicas de los modelos de información implementados en el sistema.

Usuario:

Contiene la información personal del usuario, es decir, no relacionada con el juego.

```
{
     username:       { type: String },
     usernameLower:  { type: String },
     password:       { type: String },
     mail:           { type: String }
 }
```

Las variables *usrename*, *password*, e *email* son para guardar los datos del usuario al registrarse; y la variable *usernameLower*, que contiene el *username* en minúscula, para facilitar la búsqueda de coincidencias si un usuario se quiere registrar con el mismo nombre que otro, pero con mayúsculas intercambiadas.

Jugador:

Contiene la información del usuario dentro del juego.

```
{
     name:                { type: String },
     score:          { type: Number, default:  0 },
     guild:          { type: String, default: “”},
     resources: {
          wood:           { type: Number, default: 500 },
          stone:     { type: Number, default: 300 },
          iron:           { type: Number, default: 0 },
          cereal:    { type: Number, default: 0 }
     },
     resourcesPerHour: {
          woodPerHour:    { type: Number, default: 40 },
          stonePerHour:   { type: Number, default: 20 },
          ironPerHour:    { type: Number, default: 0 },
     },
     cerealAvailable:     { type: Number, default: 0 },
     res:                 { type: Array, default: [] },
```

```
    buildings:      { type: Array, default: [] },
    mercenaries:    { type: Array, default: [] },
    coords: {
        country:    { type: Number, default: 0 },
        region:     { type: Number, default: 0 },
        village:    { type: Number, default: 0 }
    }
}
```

En cuanto al jugador, guardamos el nombre del usuario para tener una relación *1 a 1* entre usuario y jugador. Como podemos apreciar, hay objetos dentro de otros objetos, como *resources*, que es donde se guardan los recursos que tiene el jugador. En cuanto a *resourcesPerHour*, se guarda la producción cada hora de los tres tipos de recursos básicos. Tanto *res*, *buildings*, como *mercenaries*, guardan una lista, la cual se llena con objetos del tipo construcción (para *res* y *buildings*), y de tipo mercenario (para *mercenaries*). Por último, otro objeto *coords* que guardará las coordenadas donde se encuentra nuestra aldea.

Construcción:

Guarda toda la información de las construcciones o edificios.

```
{
    id:             { type: String  },
    name:           { type: String },
    level:          { type: Number, default: 0 },
    scalingValue:   {type: Number, default: 2 },
    costs: {
        wood:       { type: Number, default:0 },
        stone:      { type: Number, default:0 },
        iron:       {type: Number, default:0 },
        cereal:     {type: Number, default:0 }
    }
}
```

Como podemos ver, tenemos un *id* para identificar el edificio. El *id* se ha escogido sabiamente, de manera que los edificios de recursos comienzan desde 0, los edificios normales desde 100, y los

mercenarios desde 300 (los 200's pertenecen a las investigaciones, no implementadas por el momento). Esto se ha hecho para identificar si un *id* de una petición proviene de un tipo de elemento o de otro. En cuanto a *scalingValue*, es una variable que determina la manera en la que escalan los costes de los recursos. *Costs* encapsula los costes del edificio.

Mercenario:

Guarda la información de un mercenario.

```
}
id:        { type: String },
     name:      { type: String },
     quantity:  { type: Number, default: 0 },
     costs: {
          wood:      { type: Number, default:0 },
          stone:     { type:Number, default:0 },
          iron:      { type:Number, default:0 },
     },
     stats: {
          hp:        { type: Number, default:0 },
          shield:    { type: Number, default:0 },
          attack:    { type: Number, default:0 },
          speed:     { type: Number, default:0 }
     },
     barracksLevel:  { type: Number, default: 0 }
}
```

En este caso, la diferencia principal respecto a los edificios, es que en vez de guardar el nivel, guardamos la cantidad (*quantity*) de mercenarios que tenemos de un tipo en concreto. Además, también guarda un objeto *stats* que contiene las estadísticas de combate del mercenario. Por último, tenemos un atributo *barracksLevel*, que indica el nivel mínimo del edificio *barrack* para poder contratar ese tipo de mercenario.

2.5.4 Diseño del cliente

Como estamos creando el proyecto con Node.js, podemos utilizar las plantillas Jade para generar el contenido de la web. Debido a que este incluye funcionalidades JavaScript, nos hemos ahorrado muchas líneas de código en ficheros JS. El JavaScript ha sido utilizado mayoritariamente para generar consultas AJAX utilizando JQuery.

AJAX (*Asynchronous Javascript And XML)* es una técnica de desarrollo web para crear aplicaciones interactivas, que se ejecutan en el lado del cliente (navegador web). Se caracteriza por mantener una comunicación asíncrona con el servidor en segundo plano. De esta forma se pueden realizar cambios sobre las páginas sin necesidad de recargarlas, mejorando la interactividad, velocidad y usabilidad en las aplicaciones. Como he mencionado, AJAX no es un lenguaje, sino una tecnología basada en JavaScript, que utiliza el objeto *XMLHttpRequest* para realizar las funciones. Dicho objeto se encuentra de forma nativa en todos los navegadores actuales.

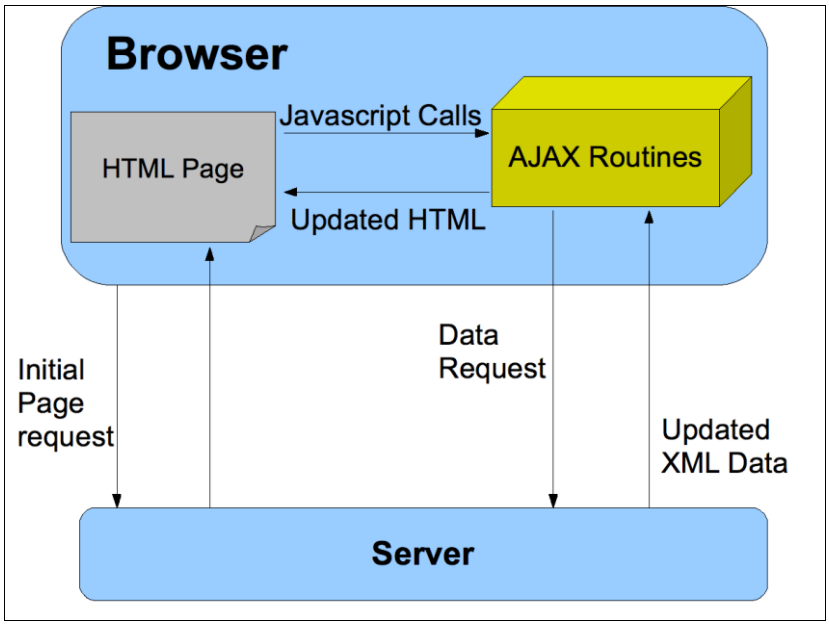

Ilustración 4 – Funcionamiento de AJAX. Una vez el cliente ha solicitado una página al servidor, AJAX es capaz de modificar su contenido dinámicamente sin necesidad de recargar de nuevo la página.

Y en cuanto a JQuery, se trata de una librería JavaScript que simplifica la manera de interactuar con los elementos del DOM, el documento HTML, manejar eventos, desarrollar animaciones (JQuery-UI), y agregar interacción con la técnica AJAX a páginas web.

Respecto al diseño de la página, he utilizado el framework Bootstrap, que incluye los ficheros CSS y JavaScript necesarios para desarrollar rápida y eficazmente una página web. Además, la misma página web ofrece varias plantillas prediseñadas, tales como un formulario para registrarse, o una página con un menú superior (ambos han sido usados en este proyecto). Cabe decir que, aunque la plantilla haya sido cogida de la web de Bootstrap, se han tenido que hacer muchas modificaciones para adaptarla a nuestras necesidades. Una de las modificaciones, sin ir más lejos, ha consistido en cambiar el código de HTML a Jade a mano.

Otra de las modificaciones necesarias ha sido mezclar la plantilla del *login* con la plantilla de la página principal. Como podremos ver más adelante, la página principal contiene un pequeño formulario para identificarse y entrar al juego. Esta integración ha sido posible gracias al sistema de *grid* que trae incorporado Bootstrap, que nos permite organizar los elementos en filas y columnas de tamaño variable, lo cual hace que todas las plantillas creadas con este framework sean compatibles a la hora de integrarlas.

2.5.5 Herramientas utilizadas

A continuación se comentan brevemente las herramientas que se han usado para el desarrollo de este proyecto.

- *Eclipse:* Eclipse es un entorno de desarrollo integrado (IDE) multiplataforma y multilenguaje, que permite la instalación de plugins para añadir funcionalidades. La extensión más importante para este proyecto es la de *nodeclipse*, que nos permite desarrollar JavaScript orientado a Node.js. Con esta aplicación se ha escrito el código tanto del cliente como del servidor.

- *GitHub:* GitHub es una forja para alojar proyectos utilizando el sistema de control de versiones Git. Ya que se ha utilizado un repositorio público, se pueden revisar todas las versiones del proyecto en mi perfil de GitHub *@Neoares*.
- *Navegadores web:* Para testear el código del servidor se ha utilizado el navegador Firefox. Eventualmente se ha probado en Google Chrome, para entrar con diferentes sesiones al juego.
- *Firebug:* Esta aplicación es una extensión para Mozilla Firefox, que provee diferentes herramientas de desarrollo web y debug de JavaScript. Entre las herramientas se encuentra un inspector de código HTML y un visualizador de cookies.
- *Paint.*net: Para la edición sencilla de fotos, como el redimensionamiento o el recorte, se ha utilizado este software, que provee de herramientas básicas pero más potentes que las que nos ofrece el editor por defecto de Windows, Paint.

3 Implementación

En el tercer capítulo vamos a hablar de los detalles de la implementación de nuestra aplicación. Nos centraremos sobre todo en los aspectos importantes mencionados en la fase de diseño.

3.1 Código de la aplicación

En este bloque voy a explicar la implementación de los puntos fundamentales de la aplicación. En ellos se intentará no incluir nada de código, a no ser que sea estrictamente necesario para la comprensión del mismo.

3.1.1 Inicialización del servidor

Para inicializar la escucha de la aplicación en nuestro servidor, ejecutamos la función *listen* sobre el objeto, previamente inicializado, *server*. Esta función (y la mayoría de funciones de las librerías de Node.js) permite pasar como parámetro una función de *callback*, de manera que podamos tratar si ha habido errores o no. En nuestro caso, simplemente imprimiremos por consola que el servidor se ha iniciado y está escuchando en el puerto que le hemos asignado (el 3000).

3.1.2 Gestión de peticiones

En este apartado voy a explicar cómo se han gestionado las peticiones que envía el cliente al servidor.

Express nos proporciona una manera sencilla de captar las peticiones del cliente y redirigirlas a la función que nosotros queramos. De este modo, lo único que tenemos que hacer es, con una línea de código, decir qué URL queremos que lance un evento en concreto. En nuestro fichero principal se encuentran dichas líneas de código, donde redireccionamos las peticiones a funciones de ficheros JavaScript situadas en el directorio *routes*. Veamos un ejemplo de código sencillo:

```
app.get('/game/mercenaries', game.mercenaries);
```

En el directorio *routes* se encuentran los ficheros *user.js*, *player.js* y *game.js*. Cada uno de estos ficheros se encarga de enrutar las peticiones a los métodos adecuados. Si la petición no está relacionada con los usuarios (*user.js* y *player.js*) o con redireccionamiento web a páginas del juego (*game.js*), se llamarán a funciones de ficheros dentro del directorio */db*, ya que significa que se van a realizar operaciones con la base de datos, pero no vamos a redireccionar a ninguna página web.

En cuanto a los métodos internos que modifican la base de datos, se encuentran los de contratar mercenarios, mejorar edificios, y atacar a otro usuario. En los siguientes apartados se explicarán cómo se gestionan algunas de las peticionas mencionadas anteriormente.

3.1.3 Gestión de usuarios

En cuanto a los usuarios, tenemos dos acciones principales, registrarse o identificarse. En este apartado vamos a explicar cómo se gestionan estas dos peticiones.

Registrarse

Esta petición se realiza vía *post*, ya que se envían datos personales que no queremos mostrar por URL. La función ejecutada está situada en *routes* y se llama *user.js*. Lo primero que hacemos es coger del *body* el nombre de usuario del formulario enviado y pasarlo a minúsculas a través de la función *toLowerCase*. Después utilizamos la función *findOne* de *mongoose* para encontrar al jugador con dicho nombre. Como he mencionado en el diseño de la base de datos, vamos a utilizar el campo de nombre de usuario en minúscula para efectuar la búsqueda, de manera que evitamos que si existe un usuario 'user1' se pueda crear otro con nombre 'User1', por ejemplo.

Una vez comprobamos que el método de búsqueda no nos devuelve ni un error ni un documento encontrado (lo cual significa que no existe un usuario con ese nombre), creamos un nuevo objeto

de usuario (esquema de *mongoose*), y le añadimos la información adecuada, como el nombre, la contraseña y el correo electrónico.

Acto seguido vamos a guardar la información mediante la función *save* y, a través de la función de callback, comprobamos si ha habido errores al crear el documento. Si todo ha ido bien, guardamos el nombre de usuario en la sesión (dentro del *request*) y redireccionaremos a la página principal del juego.

Identificarse

La petición para identificarse también se realiza mediante *post*, porque, de igual manera que en el caso de registrarse, no queremos mostrar ciertos datos en la URL. Primero extraemos la información del nombre de usuario y la contraseña del *body*, y buscamos en nuestra base de datos si existe un usuario con dicho nombre (siempre utilizando los nombres en minúscula para realizar la búsqueda). En caso de que exista, se comparará la contraseña introducida por el usuario con la contraseña del documento que nos ha retornado el método *findOne* de *mongoose*.

Si todo está bien, se guardará el usuario en la sesión y se redireccionará a la página principal del juego.

3.1.4 Lógica de juego

En este apartado vamos a explicar cómo se han implementado las funcionalidades básicas dentro del juego, tales como subir de nivel algún edificio, contratar algún mercenario, o atacar a algún jugador. Dichas funcionalidades han sido llamadas mediante AJAX, de manera que se ejecutan asíncronamente en segundo plano.

Antes de empezar me gustaría aclarar, para la mayor comprensión del documento, que todos los métodos de soporte mencionados en este apartado se explican exhaustivamente en el siguiente.

Subir de nivel un edificio productor de recursos

Si hemos seleccionado la opción de subir de nivel un edificio del apartado de recursos, se ejecutará una función que hace lo siguiente:

Primero, obtiene el documento del usuario que ha enviado la petición. Esto se obtiene, como ya hemos mencionado anteriormente, utilizando el nombre de la sesión y buscando en la base de datos mediante el método *findOne*. Una vez tenemos el documento, identificamos el edificio que vamos a mejorar a través de su id. Como mencioné en el apartado de diseño de la base de datos, el id de los edificios de recursos empiezan por 0 y se van incrementando de uno en uno (1, 2, 3…), de manera que obtener la posición del edificio en la lista de edificios es trivial: para el id 0, 0; para el 1, 1, y así sucesivamente.

Una vez sabemos qué edificio hay que mejorar, comprobamos si disponemos de recursos suficientes para realizar la operación a través de la función de soporte *verifyResources*. Si disponemos de los recursos suficientes, incrementaremos el nivel del edificio y calcularemos el coste de recursos para mejorar el edificio al siguiente nivel, ya que esta información también la guardamos en el mismo documento. Dicho cálculo se realiza mediante otra función de soporte llamada *calcCosts*.

Acto seguido, calcularemos la nueva producción por hora del recurso en cuestión (madera, piedra o acero) mediante el método de soporte *calcProduction, y* restaremos del cereal disponible la cantidad que requiere el edificio. Si se trata del edificio que produce cereal, simplemente calcularemos la nueva producción de cereal, y la incrementaremos al instante a la cantidad de cereal disponible, ya que como sabemos, el cereal se incrementa de manera fija, mientras que el resto de recursos se va incrementando en función de nuestra producción por hora.

Una vez efectuados todos los cambios, simplemente guardamos las modificaciones en el documento con el método *save* de *mongoose*.

Subir de nivel un edificio normal

Este caso es como el anterior pero mucho más simple ya que, una vez comprobamos que tenemos los recursos suficientes para realizar la operación, simplemente subimos de nivel el edificio y calculamos los costes para el siguiente nivel. Una vez realizados los cálculos, se guarda el documento con la información actualizada.

Contratar mercenarios

Para contratar mercenarios, disponemos de un campo numérico para insertar el número deseado del mercenario en concreto de que queremos contratar. Una vez pulsamos el botón *Hire* (contratar) se enviará la información del *id* del elemento, y la cantidad que el usuario ha especificado. Una vez explicado qué sucede en el cliente, veamos que pasa en el servidor.

Como siempre, obtenemos los datos enviados desde el cliente a través del *body* (que viene dentro del *request*) y buscamos el documento del usuario obteniendo su nombre a través de la sesión. Ahora necesitamos saber qué elemento de la lista de mercenarios queremos actualizar. Para ello, cogemos el *id* y, sabiendo que el *id* de los mercenarios empieza por 300, hacemos *id%100*, y obtenemos la posición de la lista de mercenarios (0 para el 300, 1 para el 301, etc.). En este caso también utilizaremos el método de soporte *verifyResources* para comprobar si tenemos suficientes recursos, con la diferencia de que en uno de los parámetros, en vez de enviarle el coste de recursos del mercenario, le pasaremos dicho coste multiplicado por la cantidad de mercenarios que el usuario ha solicitado.

Si disponemos de recursos suficientes, sumaremos a nuestro arsenal dicha cantidad de mercenarios y guardaremos el documento en la base de datos.

Atacar

En el menú de Imperio, si el usuario ha clicado en la espada (símbolo de atacar), se procederá a ejecutar una lógica de combate sencilla (explicada en el siguiente apartado). Aunque esta funcionalidad no está finalizada al 100%, se puede apreciar que tanto la interfaz como la lógica interna están preparadas para darle uso en un futuro.

El único parámetro que se envía del cliente al servidor en este caso es el nombre del objetivo del ataque, es decir, el nombre de un jugador. Lo que realiza nuestra función del servidor es obtener los documentos tanto del usuario como del objetivo, y ejecuta el método de soporte, que como ya he mencionado anteriormente, ejecuta una lógica de combate sencilla. Una vez tenemos el resultado, en caso de que no hayan empatado, procederemos a destruir las tropas del jugador que haya perdido. Esto se realiza recorriendo todos los mercenarios y modificando el parámetro *quantity* de cada uno a 0.

3.1.5 Métodos de soporte del servidor

En este apartado vamos a explicar los métodos de soporte que utiliza el servidor para realizar las tareas explicadas en el apartado anterior.

calcCerealCost (db/upgrades.js)

```
/**
•    Calculates the cereal costs of the main resource-producer buildings
 *
•    @param id  the id of the entity
•    @param lvl the level of the next upgrade for the entity with id ‘id’
•    @returns {Number}    the cereal cost of the next upgrade
 */
```

Este método simplemente devuelve el coste de cereal que tiene un edificio generador de recursos con una cierta id, y a un nivel en concreto. Para ello, se usa una fórmula distinta para cada tipo de edificio.

calcCosts (db/upgrades.js)

```
/**
•    Calculates the costs of the buildings, based on its level.
 *
•    @param id            the id of the entity
•    @param lvl           the level of the next upgrade for the entity
•    @param scalingValue  the scaling value of the costs
•    @param costs              the actual cost of the entity
•    @returns {Object}         the costs of the next upgrade
 */
```

Esta función devuelve un objeto con los costes de madera, piedra y acero para un edificio dado su id, su nivel, y su factor de crecimiento de coste o *scalingValue*. Si se trata de un edificio generador de recursos, se calculará a través de una fórmula compleja. Sin embargo, si se trata de un edificio normal, se multiplicarán los costes del nivel actual por el factor de crecimiento de coste. Los edificios normales por defecto tienen un *scalingValue* de 2, es decir, cada nivel su coste se irá duplicando.

calcProduction (db/upgrades.js)

```
/**
•    Calculates the production of the main resource-producer buildings
 *
•    @param resource      the name of the resource that is being upgraded
•    @param lvl           the level of the resource producer
•    @returns {Number}         the new amount of production
 */
```

Este método se llamará cuando hayamos subido de nivel un edificio productor de recursos. El parámetro *resource* es un String que representa el nombre del recurso en cuestión. Sus valores posibles son: *wood*, *stone*, *iron*, *cereal*.

En función del recurso del cual se trate, y del nivel del edificio, devolverá la producción por hora usando una fórmula predefinida.

applyProdFactor (db/upgrades.js)

```
/**
•    Applies the efficiency factor (based on the cereal balance) to the
resource production
 *
•    @param rph      resource production (Resources per Hour)
•    @param c        total amount of cereal
•    @param cA       cereal available
•    @returns        the new production with the efficiency factor 'f'
 */
```

Esta función se encarga de modificar la producción de recursos básicos en función del balance de cereales disponibles. Por ejemplo, si en total disponemos de 2000 unidades de cereal, pero nuestros productores de recursos consumen 4000, dichos productores funcionarán al 50% de rendimiento.

Vamos a calcular el factor de producción dividiendo la cantidad de cereal total que nuestro molino produce por la diferencia entre el cereal total y el cereal disponible (*c/c-cA*). Para prevenir la división entre 0, ejecutamos este código solo si *c-cA!=0*. En caso de que sea 0, el factor se deja a 1, ya que significa que no estamos utilizando cereal.

Una vez tengamos el factor, si es menor que 1, multiplicaremos nuestra producción por dicho factor y la devovleremos. En caso contrario simplemente retornaremos la producción tal cual estaba, ya que significa que tenemos un balance de cereales positivo.

verifyResources (db/upgrades.js)

```
/**
•    Verifies if it's possible to unlock buildings/mercenaries.
•    If it's possible, this method automatically reduces the available
amount of resources.
 *
•    @param ava            resources available
•    @param cost           resources that the building/mercenary costs
•    @returns {Boolean}    true if player can upgrade.
 */
```

Este método recibe por parámetro dos objetos. Uno con los recursos disponibles por el usuario, y otro con el coste de recursos de cierto elemento del juego. Simplemente realiza una comprobación recurso a recurso, y si dispones de una cantidad mayor al coste en cada uno de ellos, resta dicho coste a tus recursos, aumenta tus puntos en una cantidad de recursos consumidos dividido entre 100, y devuelve *true*.

En caso de no haberse cumplido la condición, es decir, de no tener recursos suficientes, la función simplemente devolverá *false*, para que el método que la ha invocado sepa que no puede continuar con el procedimiento que esté llevando a cabo.

3.1.6 Estructura del cliente

El cliente consta de dos partes principales: una dentro del juego y otra fuera. Por lo tanto, los ficheros de la vista que formen parte del exterior del juego (*index* y *signUp*) se encontrarán en la raíz de la carpeta *views*. Los ficheros que formen parte del interior del juego se encontrarán dentro de la carpeta *game* que a su vez está dentro de *views*.

Debido a que el menú superior de cada parte es común para todos los ficheros de la misma, se ha realizado un fichero *layout* para que el resto de ficheros puedan heredar de ellos. Para el interior del juego tenemos *layout.jade*, mientras que para el exterior tenemos *indexLayout.jade*. Con esto

conseguimos ahorrar líneas de código, pero no optimizamos los tiempos de carga, ya que cada vez que cambiamos de página estamos cargando tanto el menú como el contenido. Para optimizarlo, deberíamos haber usado AJAX para generar bloques de código asíncronamente[4].

En cuanto a los WebSockets, disponen de un método *on()*, que nos permite definir el comportamiento de la aplicación en función de varios eventos que se puedan producir. Dichos eventos son, básicamente, *connect* y *disconnect*. El primero se ejecuta cada vez que el usuario hace una petición al cliente, y el segundo antes de lanzar otra petición.

Un ejemplo básico que hemos implementado con WebSockets es un reloj, que nos devuelve la hora cada segundo. Para hacerlo, cuando definimos el evento con *on()*, le decimos el 'identificador' que va a llevar la información. Esto se hace para que el cliente sea capaz de coger la información correcta, en lugar de coger información que no debería. Una vez definido el evento con su identificador, llamamos a la función que envía la hora cada segundo y le pasamos el socket por parámetro. De esta forma, mediante el método *socket.emit()* podemos enviar la información al cliente. Veamos el ejemplo con código para entenderlo más fácilmente:

```
io.of('/date').on('connection', function(socket){
     updateHour(socket);
});
```

Vemos como con 'of' definimos el identificador que llevará la información de nuestro socket, '*connection*' sería el evento lanzado, y recibimos una función de callback con nuestro socket, que pasaremos por parámetro al método que actualiza la hora cada segundo. Para enviar la información utilizamos, como he dicho anteriormente, el método '*emit*':

```
function updateHour(socket){
     setInterval(function(){
        socket.emit('date', {'date': new Date()});}, 1000);
}
```

[4] Mejora explicada en el apartado *4.2 Posibles mejoras*.

4 Resultados y posibles mejoras

En este bloque vamos a mostrar un ejemplo de uso de nuestra aplicación para comprobar los resultados, mencionaremos unas cuantas mejoras que se podrían aplicar al proyecto, y acabaremos hablando de posibilidades para alojar nuestra aplicación en un servidor, ya que de momento solo está disponible mediante el *localhost*.

4.1 Resultados obtenidos

En este apartado vamos a mostrar algunas capturas de la aplicación mientras explico cómo se utilizaría la interfaz de juego. Para empezar, nos crearemos una cuenta, haremos logout, y entraremos con la cuenta previamente creada. Acto seguido pasaremos a subir de nivel los edificios generadores de recursos, y contrataremos unos cuantos mercenarios. Para finalizar, utilizaremos el menú de imperio para atacar a algún enemigo.

Lo primero es dirigirnos a la URL del juego (nosotros hemos utilizado el localhost, pero podría estar alojada en un dominio). Para registrarnos, vamos a la pestaña '*Sign Up*' del menú superior, o clicamos en el botón *'Play for free'*, lo cual nos redireccionará a la siguiente página.

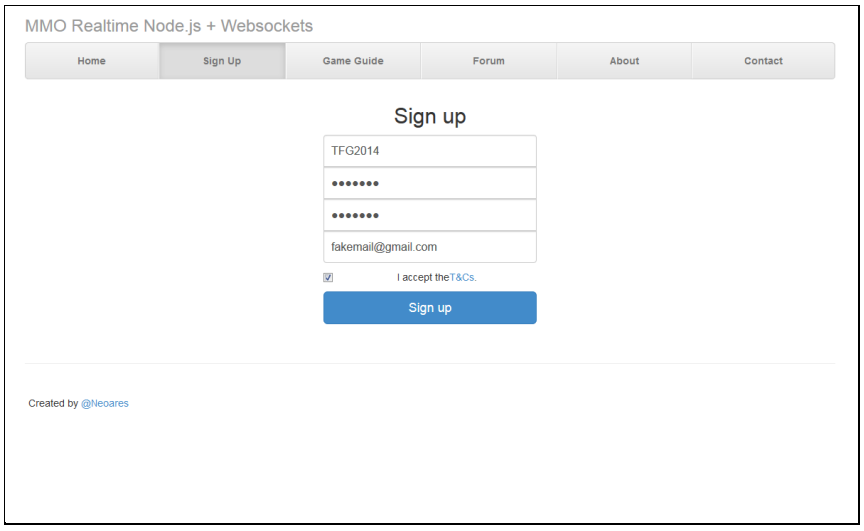

Ilustración 5 – Página de registro del juego.

A continuación se nos redirigirá al interior del juego, para comprobar que el login también funciona, saldremos mediante el botón “Logout” situado al lado de nuestro Nombre en el menú superior.

Ilustración 6 – Botón de *logout* al lado del mensaje de bienvenida del juego.

Una vez en la página principal, ingresaremos nuestros credenciales en el formulario de login y así comprobar si todo funciona correctamente.

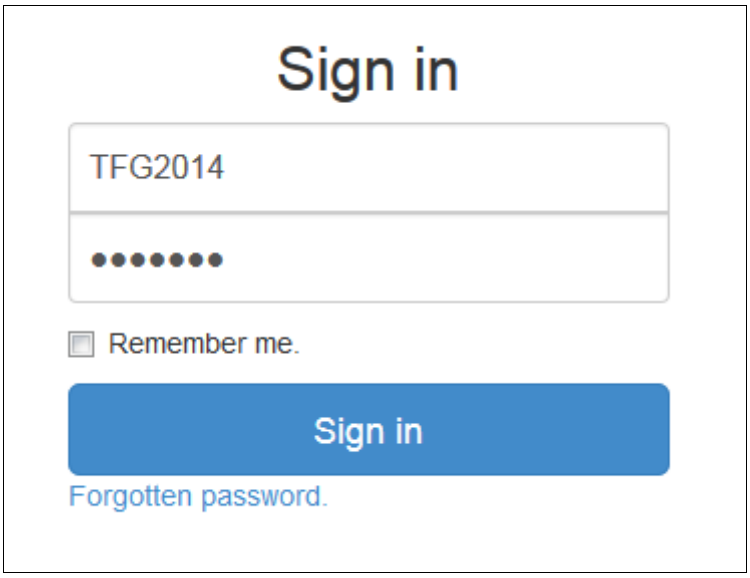

Ilustración 7 – Formulario para identificarse. Se encuentra en la página principal del juego.

Una vez introducidos los datos, pulsamos en “Sign in”, y vemos que se nos lleva a la página principal del juego.

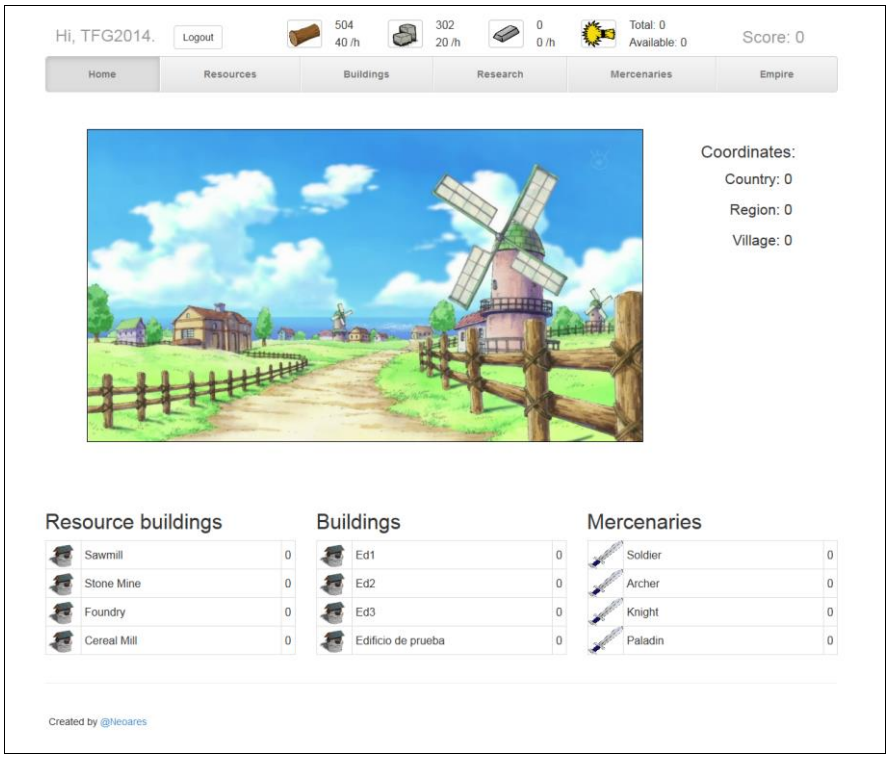

Ilustración 8 – Página principal del juego una vez estamos dentro.

Ahora vamos a subir al nivel 1 el aserradero (*Sawmill*) y la mina de piedra (*Stone mine*), y al nivel 2 el molino de cereales (*Cereal mil*). Una vez lo hemos hecho, comprobamos que nuestra producción ha aumentado y los recursos, sin embargo, han disminuido. De esta forma comprobamos que la producción de recursos funciona correctamente, ya que, de hecho, en esta última imagen se puede apreciar como tenemos 504 de madera, es decir, desde que nos hemos creado la cuenta hasta que se ha realizado la captura, se han generado 4 unidades de madera.

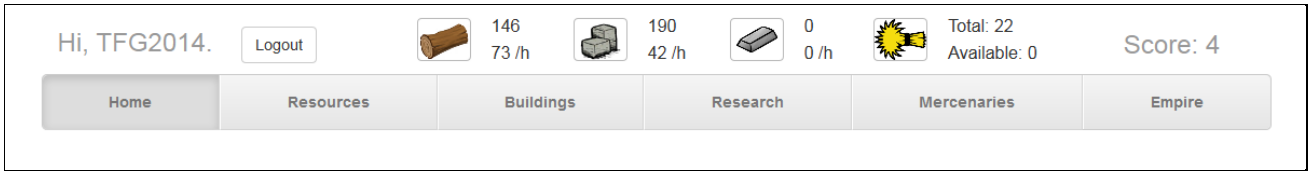

Ilustración 9 – Menú superior del juego.

Ahora accedemos al menú de mercenarios y contratamos 3 soldados. Una vez hecho esto, vamos al menú Imperio y atacamos a un jugador al azar. Nos saldrá una notificación de si hemos ganado, empatado o perdido.

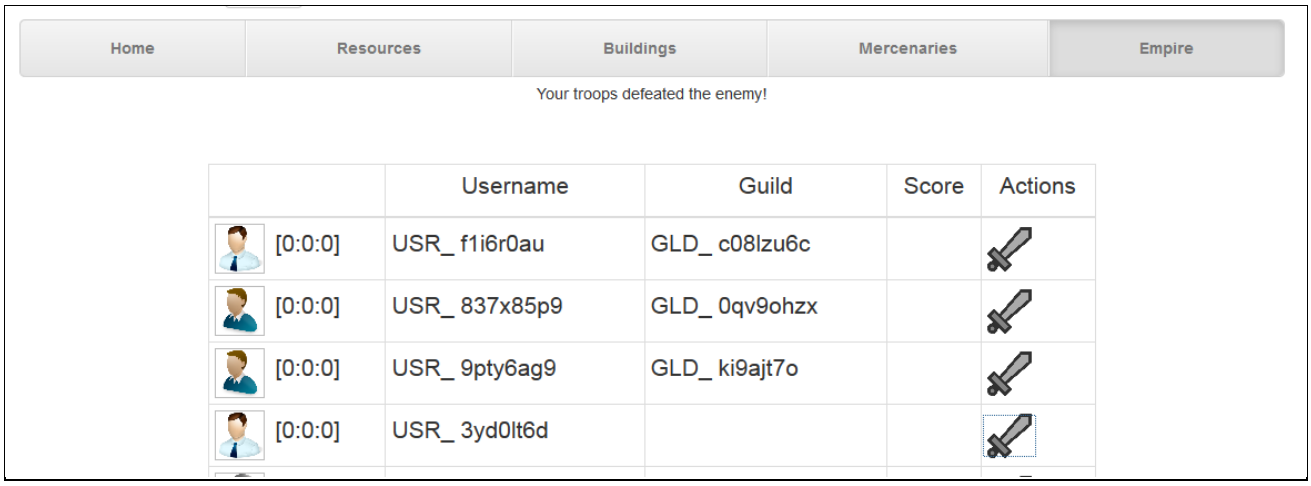

Ilustración 10 – Página *Empire*, donde podemos ver al resto de jugadores y atacarles.

Si perdemos, podremos volver al menú de mercenarios y observar como todos han desaparecido. Esto es debido a que el perdedor de la batalla pierde todos sus mercenarios.

4.2 Posibles mejoras

A continuación voy a nombrar ciertas mejoras que se podrían aplicar al juego, pero no se han aplicado por falta de tiempo, o simplemente porque eran inviables por una situación u otra.

- *Contratar mercenarios:* Se podría añadir la lógica
- *Tema artístico:* El diseño de la página web es quizás lo más importante en un juego de este tipo. Debido a que en este proyecto se han abarcado muchas materias, no me he querido centrar exclusivamente en el diseño artístico. La parte más importante que habría que mejorar es el tema de las imágenes. Debido a que no se pueden utilizar imágenes con copyright, es muy difícil encontrar tantas imágenes que tengan relación entre sí, es decir, que queden bien en conjunto. Una de las soluciones posibles sería contar con la ayuda de un diseñador gráfico, que pudiera elaborar las imágenes para el juego. Desafortunadamente, no conozco ninguno en persona, así que he tenido que poner imágenes de prueba en el lugar en el cual deberían ir las buenas.

- *Aumentar el número de construcciones:* Muchos edificios son de prueba y realmente aún no tienen ninguna utilidad. Se pueden construir correctamente, escalan de coste correctamente, pero realmente les falta algo de "juego". Además, estaría bien añadir una breve descripción de cada edificio, para aumentar la inmersión en el juego. Dicha descripción no contendría información técnica de la estructura, sino más bien una descripción superficial del edificio.
- *Web generada dinámicamente:* Aunque la web tiene elementos dinámicos, una mejora prácticamente obligatoria sería hacer que la web *ingame* fuera una sola, y que los distintos bloques de código que componen todas las páginas, se añadan de forma asíncrona, de manera que solo se produzca un evento de conexión con WebScoekts. Uno podría pensar que esto debe realizarse utilizando AJAX, pero, de hecho, podríamos utilizar los propios WebSockets para enviar el código asíncronamente del servidor al cliente.

4.3 Hosting

Como ya he mencionado, nuestra aplicación actualmente está en fase de pruebas y corre sobre nuestro propio ordenador, en el *localhost*. Sin embargo, he analizado diferentes posibilidades para, en un futuro, poder entrar en fase de producción y hospedar nuestro juego en un servidor. A continuación mencionaré las dos opciones más viables que he encontrado: hosting para aplicaciones Node.js, y servidor cloud.

Hosting Node.js: Actualmente, hay empresas que se dedican solo a ofrecer servicios para node.js, mientras que otras ofrecen el servicio como parte de su *pack*. Quizás las más apropiadas para este caso son las que ofrecen un servicio exclusivamente para aplicaciones Node.js, ya que pueden ser mucho más económicas, y ofrecen servicios más focalizados para nuestros intereses. En nuestro caso, una opción viable podría ser *nodejitsu*, que permite hacer pruebas gratuitas y utilizar GitHub como repositorio para colgar la aplicación en la web. Además, provee de unos servicios variables en función de tus necesidades y presupuesto. Desde 9€ al mes, hasta 1500€.

Servidor cloud: Antes de ver opciones de servidores cloud y planes económicos, me gustaría dar una pequeña introducción a lo que es un servidor cloud. Para ello, haré una pequeña comparativa entre servidor compartido, servidor cloud y servidor dedicado.

Para empezar, un servidor compartido es aquel donde podemos mantener nuestras páginas web y archivos junto con la de otros clientes en distintas particiones de su capacidad. La ventaja de estos servidores es que son más económicos, perfectos para empresas pequeñas o con poco tráfico. La desventaja es que no se tiene acceso al sistema operativo, por lo que no puedes instalar módulos o programas. Es decir, que antes de contratar este servicio tienes que saber bien qué prestaciones tiene.

Un servidor dedicado, sin embargo, tiene todos los recursos para un único cliente en una sola máquina, siendo este el administrador de sus recursos y sin ser afectado por el consumo de ancoh de banda de otros clientes. Como podemos ver, todo son ventajas. Su única desventaja es el precio, ya que suelen ser más caros y orientados a empresas medianas y en adelante.

El servidor cloud, que podría considerarse un híbrido entre los dos tipos de servidor anteriores, se caracteriza por disponer de sus recursos en varias particiones de varias máquinas, lo que hace que sea más escalable y configurable a medida. Su principal semejanza con el servidor dedicado es que en este caso sí tenemos acceso al sistema operativo, por lo que podemos instalar módulos sin problema. La gran ventaja de este tipo de servidores es que cuando necesitemos más potencia de servidor, se puede reconfigurar, obteniendo más memoria RAM, más CPUs, o más GB de almacenamiento masivo.

Arsys es una empresa que ofrece planes para servidores cloud, entre otros. Permite instalar una gran variedad de sistemas operativos, y escoger de forma completamente personalizada los recursos físicos de los cuales queremos disponer. Partiendo de la base de que queremos un servidor con Ubuntu Server 12.04, tendremos una cuota mensual de 35€, con una CPU de 1 núcleo,

0.5GB de RAM, y 50GB de disco duro. A partir de aquí, por cada núcleo adicional de procesador que queramos, nos costará 10€ más al mes (hasta un máximo de 8 núcleos); por cada GB de RAM adicional, 10€ más (hasta un máximo de 128GB); y por cada 50GB extras de almacenamiento masivo, otros 10€ mensuales.

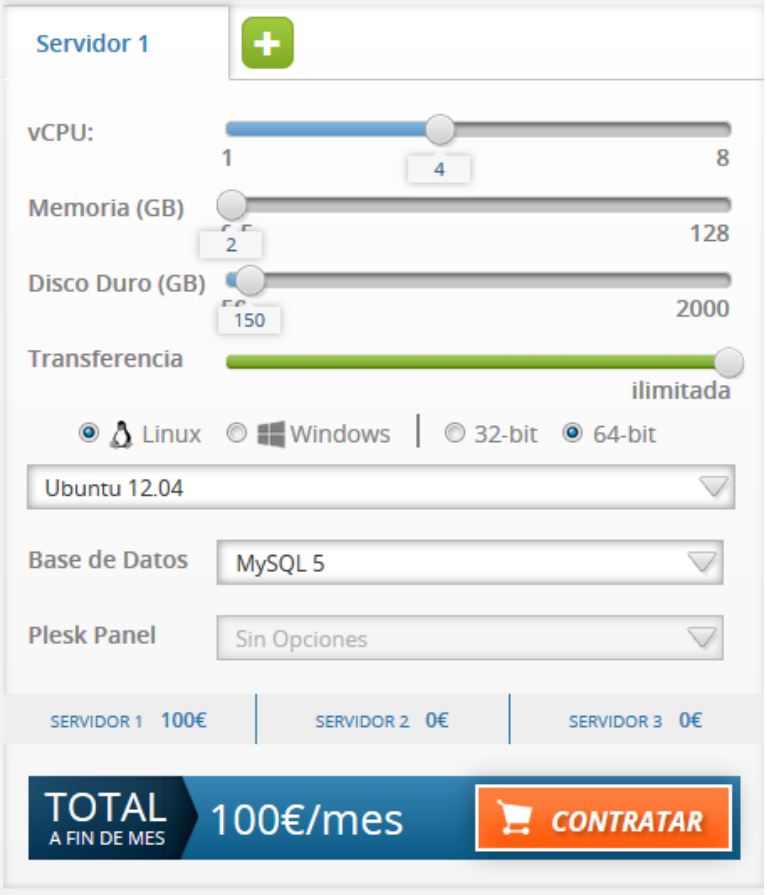

Ilustración 11 – Ejemplo de servidor personalizado con Arsys. Los precios van de 35€/mes en adelante.

Lo lógico sería empezar con un servidor mínimo y, si vemos que poco a poco se va quedando corto, ampliar los recursos físicos del sistema. Para poder realizar estos cambios en caliente, es decir, sin necesidad de reiniciar el servidor, se necesita contratar el servicio Premium, por el cual pagaremos un poco más al mes, pero desbloquearemos también la opción de aumentar los recursos físicos al máximo. Por ejemplo, sin el sistema Premium solo podemos tener hasta 8GB de RAM, mientras que siendo Premium podemos contratar hasta 128GB.

5 Conclusiones

El objetivo principal de este proyecto era aprender a utilizar tecnologías nuevas relacionadas con las aplicaciones web, como Node.js y WebSockets. En este aspecto podría dar los objetivos como cumplidos, tal y como refleja esta documentación.

En cuanto al aprendizaje, hay que remarcar que he tenido que aprender prácticamente de cero todo lo que he utilizado en este proyecto. De hecho, cuando empecé el desarrollo en Febrero, no había realizado nunca una aplicación web. Afortunadamente, la documentación que he encontrado para empezar a trabajar con Node.js es bastante buena, así como las documentaciones de las librerías o frameworks utilizados.

No obstante, me habría gustado haber podido acabar el proyecto al 100%. Aunque esa era mi idea al principio, poco a poco me di cuenta que me había embarcado en un proyecto sumamente extenso, en el cual debería trabajar un equipo especializado. De todas formas no puedo quejarme de los resultados, y mucho menos de todo lo que he aprendido desde que empecé.

Aunque, por ahora, Node.js no es la tecnología más usada para programar servidores, no me cabe la menor duda de que en un futuro no muy lejano podría llegar a predominar el mercado. Y digo esto porque he podido comprobar en primera persona el potencial que tiene. No solo yo, sino una comunidad entera de desarrolladores se ha volcado mucho con esta tecnología, creando librerías para mejorar su uso, desde conectores entre servidor a base de datos, hasta nuevos protocolos de comunicación, como lo son los WebSockets.

6 Anexo

6.1 Contenido de la entrega

La entrega se ha realizado a través del campus virtual y se compone de dos partes: la memoria y el código fuente.

La memoria se encuentra en la raíz, y se llama memoria.pdf. En cuanto al código, está comprimido en un fichero llamado src.zip. Si descomprimimos el fichero nos encontraremos con la carpeta principal del proyecto, y dentro de esta se encuentran todos los ficheros y carpetas del mismo. No he añadido los componentes necesarios para correr la aplicación, puesto que quedaría obsoleto en pocas semanas. En su lugar, en el siguiente apartado pondré la URL de donde podremos descargar lo necesario.

6.2 Instalación y ejecución de la aplicación

Para poder ejecutar nuestra aplicación en el *localhost*, debemos seguir los siguientes pasos.

Primero, descargamos node.js de http://www.nodejs.org/download/. Escogeremos la versión para nuestra plataforma y arquitectura, y la instalaremos como si se tratara de un paquete normal y corriente.

El otro requisito es la base de datos (MongoDB en nuestro caso). Para descargarla seguiremos el siguiente enlace: http://www.mongodb.org/downloads. Podemos descargarnos el instalador o la versión comprimida. En cualquier caso, encontramos una guía de instalación en el siguiente enlace: http://docs.mongodb.org/manual/installation/.

Una vez tenemos Node.js instalado y MongoDB descargado, tenemos que iniciar la base de datos. Para ello, vamos a la carpeta de instalación de MongoDB y ejecutamos el fichero llamado “mongod”. Según la guía también se puede ejecutar el comando “mongo” en sistemas operativos

basados en Linux y OS X, pero en mi caso he utilizado Windows, así que simplemente ejecuto el fichero nombrado anteriormente. De todas formas pongo el enlace a la fuente de esta información: http://docs.mongodb.org/manual/tutorial/getting-started/.

Una vez nuestro servidor está iniciado, simplemente tenemos que dirigirnos al directorio principal de nuestro proyecto con la terminal, y ejecutar el siguiente comando:

```
node app.js
```

Y eso es todo. Nuestra aplicación debería estar corriendo correctamente. Para comprobarlo, nos dirigimos a nuestro navegador web por defecto (evitad usar internet Explorer), y nos vamos a la siguiente ruta:

```
http://localhost:3000
```

Si hemos seguido los pasos al pie de la letra, deberíamos aparecer en la página principal del juego. Una vez aquí podemos seguir los pasos del apartado *4.1. Resultados obtenidos* para echarle un vistazo.

7 Bibliografía

En este apartado se dan los enlaces consultados durante el desarrollo del proyecto (o los más importantes). En lo referente a las fuentes, cabe destacar que no se han realizado búsquedas en libros, ya que, aparte de ser mucho más lento para lo que nosotros buscamos, se quedan desfasados rápidamente.

http://www.nodejs.org/

http://www.nodebeginner.org/index-es.html

http://expressjs.com/guide.html

http://www.codemag.com/Article/1210041

http://mongoosejs.com/docs/

http://docs.mongodb.org/manual/

http://getbootstrap.com/

http://www.nodehispano.com/2011/12/publicando-tu-aplicacion-nodejs/

http://blog.comalis.com/201201-conoce-la-diferencia-entre-un-servidor-compartido-cloud-o-dedicado/

http://stackoverflow.com/

http://opengameart.org/

http://www.clker.com/

http://davidwalsh.name/websocket

http://danielnill.com/nodejs-tutorial-with-socketio/

https://www.youtube.com/user/0utKast/