ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Trabajo Fin de Grado

DESARROLLO DE UN JUEGO WEB INTERACTIVO MEDIANTE TECNOLOGÍAS REST, AJAX Y WEBSOCKET

AUTOR: Rosa Mª Moreno de la Santa Carretero

DIRECTOR: Juan Ángel Pastor Franco

TITULACIÓN: Grado en Ingeniería Telemática

Mayo / 2016

Autor	Rosa Mª Moreno de la Santa Carretero
E-mail del Autor	rosa_msc_28@hotmail.com
Director	Juan Ángel Pastor Franco
E-mail del Director	*juanangel.pastor@upct.es*

Título del TFG	Desarrollo de un juego web interactivo mediante tecnologías REST, Ajax y WebSocket

Resumen

El proyecto consiste en el desarrollo de un juego web interactivo haciendo uso de tecnologías REST y WebSocket principalmente.

Estas tecnologías están perfectamente diferenciadas a la hora de darle funcionalidad al juego:

- REST: Inicialmente, para el servicio de login de clientes previamente registrados y servicio de signin para nuevos usuarios.
- WebSocket: Una vez iniciado el juego, establece una comunicación asíncrona entre los jugadores (clientes) y servidor (gestor del juego) y a su vez aporta la posibilidad de tener un escenario multijugador en el cual los cambios o movimientos del juego se propaguen y sean visibles por otras sesiones de usuarios establecidas.

Titulación	Grado en Ingeniería Telemática
Departamento	Tecnologías de la Información y las Comunicaciones
Fecha de presentación	Abril 2016

Agradecimientos

A mi tutor por su paciencia, ánimo y dedicación.

A todos aquellos compañeros y amigos que me han ayudado
a lo largo del desarrollo de este proyecto.

Y en especial a mis padres y a Víctor, por las palabras
de aliento durante todos estos años.

Gracias

Índice

Agradecimientos .... **v**

Índice .... **vii**

Índice de Figuras .... **ix**

1. Introducción **1**

1.1 Planteamiento inicial *1*

1.2 Objetivos *1*

1.3 Plan de trabajo *1*

1.4 Recursos y entornos de desarrollo *2*

1.5 Organización de los contenidos *3*

2. Estado de la técnica **5**

2.1 NetBeans *5*

2.2 Maven *6*

2.3 WebSockets vs REST *8*

¿Qué son los WebSockets? 8

¿Qué es REST? 9

¿WebSocket vs REST? 10

2.4 Java *10*

2.5 JavaScript *11*

2.6 CSS3 .... *11*

2.7 HTML5 .... *12*

2.8 Canvas *12*

2.9 Bootstrap *12*

2.10 AJAX *13*

2.11 JQuery *14*

2.12 JSON *14*

2.13 GlassFish *15*

2.14 MongoDB *15*

3. Organización del código y ejecución **17**

3.1 Jerarquía en el panel de proyecto *17*

3.2 Instalación y configuración *21*

3.2 Manual de arranque *22*

3.3 Manual de uso *23*

4. Diseño e implementación **29**

4.1 Partes que intervienen. Cliente y servidor 29
4.2 Login y Signin 29
Cliente 29
Servidor 31
5.3 Juego 32
Cliente 33
Servidor 39
5. Construcción de un Juego mínimo. 43
5.1 Pasos a seguir 43
6. Cambio de vistas 63
7. Conclusiones 69
7.1 Puntos fuertes y objetivos conseguidos 69
7.2 Mejoras y trabajos futuros 69
8. Bibliografía 71
9. Referencias 73

Índice de Figuras

FIGURA 1: ESTRUCTURA DE UN ARCHIVO POM 7
FIGURA 2: COMUNICACIÓN WEBSOCKET 8
FIGURA 3: COMUNICACIÓN REST 9
FIGURA 4: COMPARATIVA MODELO AJAX & MODELO CLÁSICO 14
FIGURA 5: ESTRUCTURA DE UN PROYECTO CON MAVEN 17
FIGURA 6: JERARQUÍA DE CARPETAS GENERAL EN NETBEANS 18
FIGURA 7: JERARQUÍA DE CARPETAS - WEB PAGES 18
FIGURA 8: JERARQUÍA DE CARPETAS - REMOTE FILES 19
FIGURA 9: JERARQUÍA DE CARPETAS - RESTFUL WEB SERVICES 19
FIGURA 10: JERARQUÍA DE CARPETAS - SOURCE PACKAGES 20
FIGURA 11: JERARQUÍA DE CARPETAS - DEPENDENCIAS 20
FIGURA 12: JERARQUÍA DE CARPETAS - PROJECT FILES 21
FIGURA 13: ARRANQUE DEL PROYECTO 23
FIGURA 14: VISTA DEL LOGIN 23
FIGURA 15: VISTA DEL FORMULARIO DE REGISTRO 24
FIGURA 16: VISTA DEL JUEGO 24
FIGURA 17: DOS COMPAÑEROS DE UNA MISMA SESIÓN 26
FIGURA 18: DOS SESIONES DIFERENTES 27
FIGURA 19: COMUNICACIÓN CLIENTE-SERVIDOR WEBSOCKET 33
FIGURA 20: JUEGO MÍNIMO. PRUEBA DE CONEXIÓN 47
FIGURA 21: LOGIN (CAMBIO DE VISTA) 67
FIGURA 23: REGISTRO (CAMBIO DE VISTA) 67
FIGURA 24: JUEGO MINIONS (CAMBIO DE VISTA) 68

1. Introducción

1.1 Planteamiento inicial

En el presente trabajo se desarrolla una aplicación web basada en un juego interactivo multijugador, el cual tiene como principal objetivo hacer un ensayo de funcionamiento de diversas tecnologías web integradas en una misma aplicación, y la obtención de resultados para este entorno.

Haciendo uso de las tecnologías integradas en dispositivos web disponibles, se pretende crear una aplicación que permita a un usuario registrarse en la aplicación para comenzar posteriormente a jugar. Cuando el juego comienza, internamente se transmiten mensajes entre el cliente (jugador) y el servidor, ambos se intercambian información para que los cambios se vayan efectuando de forma transparente al usuario. El proceso de login puede hacerse con el protocolo http, pero la ejecución del juego hace recomendable utilizar un protocolo más flexible que permita al servidor mandar mensajes al cliente sin una petición previa de éste.

Partiendo de un estudio teórico y de posteriores pruebas empíricas, se escogerán las tecnologías más adecuadas para conseguir el resultado deseado, y así poder proceder a programar la aplicación web en sus diferentes fases: login[1], sign-in[2], juego.

1.2 Objetivos

Los objetivos planteados en este proyecto son los siguientes:

- Disponer de una aplicación para enseñar tecnología web
- Estudiar los WebSocket en aplicaciones web interactivas.
- Estudiar la responsividad[3] de un juego web interactivo.
- Desarrollo de prototipos orientados a la resolución de problemas concretos: login, suscripción de observadores, políticas de concurrencia, etc.
- Realización de pruebas funcionales
- Redacción de la memoria

1.3 Plan de trabajo

Para la consecución del proyecto, este se ha dividido en una seria de fases bien diferenciadas:

[1] Login: es el proceso mediante el cual se controla el acceso individual a un sistema informático mediante la identificación del usuario utilizando credenciales provistas por el usuario

[2] Signin: es el proceso por el cual un usuario se registra en un sistema informático para poder acceder posteriormente a él generalmente mediante login.

[3] Responsividad: En informática hace referencia a la velocidad con que una aplicación responde a las acciones del usuario.

1. Ejemplo de uso de servicios web y base de datos MongoDB para el login y signin. Documentación sobre el funcionamiento de la base de datos MongoDB, la representación y almacenamiento de los datos, la conexión con nuestro entorno de desarrollo, y el traspaso de datos entre cliente y MongoDB (REST y AJAX).
2. Extensión con WebSocket. Aprendizaje del desarrollo de aplicaciones con WebSocket. Esto incluye la lectura y estudio de esta tecnología, realizando tutoriales propuestos y considerando los pros y contras de utilizarla con respecto a otras posibles opciones.
3. Extensión con canvas. Profundizar en el desarrollo de aplicaciones utilizando este elemento para dibujar y editar el juego en sí dentro del navegador.
4. Extensión con concurrencia e inteligencia artificial. Se han implementado una serie de mejoras en el servidor, como son el aporte de inteligencia artificial al juego y la concurrencia de varios usuarios dispuestos a utilizar la aplicación al mismo tiempo. Estos aspectos serás explicados en mayor profundidad más adelante.
5. Pruebas. Durante el desarrollo de la aplicación se irán realizando una serie de pruebas con la finalidad de comprobar el correcto funcionamiento así como de depurar posibles fallos o comportamientos no deseados.
6. Redacción de la memoria. En último lugar se procederá a redactar la memoria detallando todo el proceso de desarrollo del proyecto para su posterior defensa.

1.4 Recursos y entornos de desarrollo

Como entorno de desarrollo se ha utilizado el kit de desarrollo de software NetBeans IDE[4] 8.0.2. Se ha considerado el uso de este entorno ya que es capaz de soportar el desarrollo de todo tipo de aplicaciones Java, por su estabilidad, amplio número de herramientas y porque aparte de ser muy sencillo de utilizar contiene un amplio número de plantillas, ejemplos y tutoriales para el desarrollo y aprendizaje en la realización de diferentes proyectos.

Además, en cuanto al proyecto creado en NetBeans se ha considerado oportuno y necesario que fuese *maven*, ya que nos facilita la integración de librerías y dependencias en toda la aplicación web.

Por otra parte, en cuanto a la distinción entre cliente y servidor es fácilmente diferenciable, puesto que la parte del cliente está desarrollada en su totalidad en JavaScript mientras que la parte del servidor está implementado en Java, ambos se comunican mediante webSockets sin problema alguno, utilizando Glassfish como servidor de aplicaciones disponible en NetBeans.

[4] Un entorno de desarrollo integrado **(IDE)** es una aplicación de software que proporciona servicios integrales para los programadores informáticos para el desarrollo de software. Un IDE consiste normalmente en un editor de código fuente, construir automatización de herramientas y un depurador.

1.5 Organización de los contenidos

Para facilitar la lectura de la presente memoria se va a proceder a hacer una breve descripción de los contenidos que se van a describir en cada apartado.

- Estado de la técnica: En este apartado se van a enumerar y explicar las diferentes tecnologías usadas a lo largo del proyecto, además se hace una breve introducción a la función que realiza cada una de ellas dentro de la aplicación web.
- Organización del código y ejecución: En esta parte se muestra la jerarquía que siguen las carpetas creadas dentro del IDE de NetBeans para este proyecto, explicando la finalidad de creación de cada una de ellas y los tipo de archivos que contienen por separado. Además, se exponen las diferentes configuraciones realizadas a lo largo del desarrollo de la aplicación y por último, se enseñan los pasos a seguir para la ejecución del juego y se identifican los diferentes elementos que aparecen en él.
- Diseño e implementación: Se explican las principales funciones que realiza el cliente y el servidor, como se ha implementado cada parte para que cumpla con los objetivos fijados y como gestiona cada uno la comunicación con el contrario. Es en este punto donde se habla en profundidad sobre el cliente y el servidor, las acciones que realiza y las tecnologías usadas para este fin. Este punto tiene como principal objetivo diferenciar cuales son las tareas llevadas a cabo por cliente y por el servidor y como se ha conseguido realizarlas.
- Construcción de un juego mínimo: Puesto que explicar paso a paso cómo y por qué se ha programado cada método o clase de la aplicación sería tedioso y no se conseguiría el fin propuesto para este proyecto, se ha optado por hacer un pequeño guion donde se realiza un juego sencillo con las funcionalidades básicas que aportan los conceptos necesarios a partir de los cuales poder seguir ampliando el juego. Este apartado tiene fines docentes que enseñen la manera de crear un juego basado en tecnologías WebSocket.
- Cambio de vistas: En este espacio se han dado las pautas a seguir para modificar la temática del juego, ya que en pocos pasos se podría cambiar toda su apariencia, obteniendo así otro totalmente diferente en lo que a aspecto se refiere.

2. Estado de la técnica

2.1 NetBeans

Para el desarrollo del trabajo se utiliza NetBeans [1] como entorno de desarrollo integrado IDE. Es una herramienta de código libre que permite escribir, compilar, depurar y ejecutar programas. El IDE NetBeans soporta el desarrollo de todos los tipos de aplicaciones Java (J2SE, web, EJB y aplicaciones móviles) y permite que sean desarrolladas a partir de un conjunto de componentes software llamados módulos. Las aplicaciones construidas a partir de estos módulos pueden ser extendidas agregándole otros nuevos y, debido a que pueden ser desarrollados independientemente, las aplicaciones basadas en la plataforma de NetBeans pueden ser extendidas fácilmente por otros desarrolladores de software.

Netbeans se identifica como el mejor soporte para las últimas versiones de Tecnologías Java, ya que, con sus constantes actualizaciones y mejoras dispone de multitud de características y herramientas avanzadas que lo convierten en una de las mejores opciones para el desarrollo de tecnologías de vanguardia.

- <u>Código de edición rápido e inteligente.</u> Proporciona facilidad al usuario al desarrollar la aplicación ya que proporciona plantillas, aporta sugerencias y contiene generadores de código.
- <u>Gestión de proyectos fácil y eficiente.</u> Proporciona diferentes vistas de los datos, dispone de varias ventanas de proyectos para gestionarlos de forma eficiente y emplea herramientas útiles para la creación de aplicaciones, lo que le permite al usuario profundizar en los datos de forma rápida y sencilla.
- <u>Desarrollo rápido de la interfaz de usuario.</u> Permite la creación de aplicaciones rápidamente y sin problemas mediante el uso de editores y herramientas de drag and drop. Además ofrece apoyo a la edición en cuanto a contexto se refiere. En general la interfaz gráfica de usuario es muy fácil e intuitiva.
- <u>Código libre de errores.</u> Proporciona herramientas de análisis, especialmente con la herramienta FigBugs (ampliamente utilizada) identifica y soluciona problemas comunes en código Java. Por otro lado, NetBeans Profiler proporciona asistencia de expertos para optimizar el uso de la velocidad y la memoria de la aplicación, lo que hace que sea más fácil construir aplicaciones fiables y escalables de Java. Además NetBeans IDE incluye un depurador visual que permite depurar las interfaces de usuario sin mirar el código fuente.
- <u>Plataformas soportadas.</u> NetBeans IDE se puede instalar en todos los sistemas operativos compatibles con Java, como lo son Windows, Linux y sistemas Mac OS X.
- <u>Análisis en vivo.</u> El IDE analiza el código fuente en vivo mientras se escribe. El editor marca errores, destaca ocurrencias, ofrece soluciones rápidas y advertencias.
- <u>Diseño web adaptable a la capacidad.</u> Es compatible con el uso de arquitecturas responsive.
- <u>Vista preliminar de las páginas web.</u> Profunda integración con Chrome y el navegador WebKit Embedded interno que asegura una perfecta conexión entre el código y el diseño de la página

- Apoyo al desarrollo de Node.js. NetBeans permite el desarrollo rápido y fácil con el entorno de ejecución Node.js. A través de asistentes y plantillas, se pueden configurar rápidamente aplicaciones Node.js y utilizar NetBeans IDE para hacer pruebas, depurar el código y ejecutarlo.

En definitiva se ha considerado el uso de NetBeans por su robustez, estabilidad, diversas funcionalidades y herramientas, y el uso fácil e intuitivo del entorno. Para el desarrollo de la aplicación web se ha utilizado la versión 8.0.2.

2.2 Maven

Maven [2] es una herramienta de software utilizada para la gestión y la construcción de proyectos Java. Se basa en un fichero central, Project Object Model (POM), para describir el proyecto de software que se va a construir, las dependencias de otros módulos y componentes externos, y el orden de construcción de los elementos. Maven viene integrado con Netbeans y proporciona una serie de ventajas muy interesantes, que se explican en los siguientes párrafos, y que justifican su empleo en este proyecto.

Con Maven la gestión de dependencias entre módulos y distintas versiones de librerías se hace muy sencilla. En este caso, solo tenemos que indicar los módulos que componen el proyecto, o qué librerías utiliza el software que estamos desarrollando en el fichero de configuración de Maven del proyecto mencionado anteriormente. Además, en el caso de las librerías, no es necesario descargarlas a mano, ya que Maven posee un repositorio remoto (Maven central) donde se encuentran la mayoría de librerías que se utilizan en los desarrollos de software, y que la propia herramienta se descarga cuando es necesario.

Maven nos provee de un repositorio donde alojar, mantener y distribuir datos, permitiéndonos una gestión correcta de nuestras librerías, proyectos y dependencias. El uso de Maven, a día de hoy, es una necesidad en cualquier proyecto Java/Java EE.

Además aporta una semántica común de desarrollo del software, llegando incluso a establecer una estructura común de directorios para todos os proyectos. Por ejemplo el código estará en `${raíz del proyecto}/src/main/java`, los recursos en `${raíz del proyecto}/src/main/resources`. Los tests están en `${raíz del proyecto}/src/test` etc.

En la figura 1 se muestra un ejemplo de POM:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>org.urjc.libromaven.multiproyecto</groupId>
        <artifactId>multiproyecto</artifactId>
        <version>1.0</version>
    </parent>

    <artifactId>simple-weather</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>simple-weather</name>
    <url>http://maven.apache.org</url>

    <licenses>
        <license>
            <name>Apache License, Version 2.0</name>
            <url>http://www.apache.org/licenses/LICENSE-2.0.txt</url>
            <distribution>repo</distribution>
            <comments>A business-friendly OSS license</comments>
        </license>
    </licenses>

    <dependencies>
        <dependency>
            <groupId>dom4J</groupId>
            <artifactId>dom4J</artifactId>
            <version>1.6.1</version>
        </dependency>
        <dependency>
```

Figura 1: Estructura de un archivo POM

Como se puede ver, en el mismo archivo también indicamos aspectos como el nombre del proyecto, licencias, quién desarrolla el código, el sitio web, el repositorio de control de versiones, un identificador único del proyecto, etc. En este ejemplo estamos viendo el fichero más sencillo posible pero la estructura del fichero POM puede llegar a ser muy compleja y puede llegar a depender de otros POM.

Al igual que maven existen otros gestores de paquetes muy conocidos que realizan funciones similares para otros lenguajes, como son Bower [3] y Npm [4]. Son dos herramientas de gestión de dependencias y tienen por tanto como función principal gestionar todos los paquetes instalados en el sistema o en el proyecto, manteniendo su usabilidad. Pero la principal diferencia entre ambos es que Npm se utiliza para la instalación de módulos de node.js[5] y por el contrario Bower se utiliza para controlar los componentes front-end[6] como son html, css, js, etc

[5] Node.js: Es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor (pero no limitándose a ello). Creado con el enfoque de ser útil en la creación de programas de red altamente escalable, como por ejemplo, servidores web.

[6] Font-end: Es la parte del software que interactúa con el o los usuarios (interfaz)

2.3 WebSockets vs REST

¿Qué son los WebSockets?

WebSocket [5] es una tecnología que proporciona un canal de comunicación bidireccional y full-duplex entre el navegador del usuario y un servidor, sobre un único socket TCP7. Con esta API8, se pueden enviar mensajes al servidor y recibir respuestas controladas por eventos sin tener que consultar periódicamente al servidor para obtener una respuesta.

La especificación WebSocket define un API que establece conexiones "socket" entre un navegador web y un servidor. Dicho con otras palabras: existe una conexión persistente e interactiva entre el cliente y el servidor, y ambas partes pueden empezar a enviar datos en cualquier momento.

Debido a que es un protocolo independiente basado en TCP, no se requiere de la intervención de HTTP aparte de en el establecimiento de la conexión (véase Figura 2).

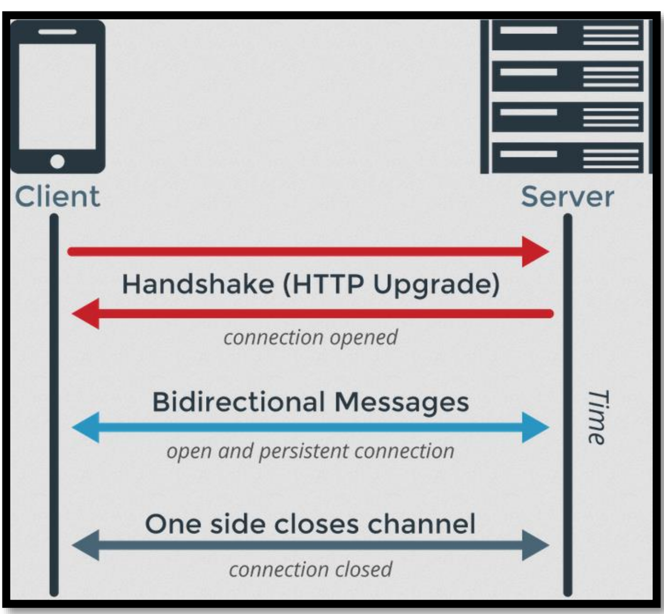


Figura 2: Comunicación WebSocket

A continuación se muestra un ejemplo simple de la creación de una conexión WebSocket que envía y recibe datos del servidor:

[7] TCP: Protocolo de Control de Transmisión

[8] API (Interfaz de Programación de Aplicaciones): Es el conjunto de subrutinas, funciones y procedimientos que ofrece una biblioteca para ser utilizado por otro software como una capa de abstracción.

```
var connection = new WebSocket ('ws://sontan.name/stream/');

//Cuando la conexión está abierta, envía datos al servidor
connection.onopen = function () {
    connection.send('Ping'); //Envía el mensaje 'Ping' al servidor
};
//Registra los mensajes del servidor
connection.onmessage = function (e) {
  console.log('Server: ' + e.data);
};
```

¿Qué es REST?

Es un tipo de arquitectura de desarrollo web que se apoya totalmente en el estándar HTTP. REST [6] nos permite crear servicios y aplicaciones que pueden ser usadas por cualquier dispositivo o cliente que entienda HTTP, por lo que es mucho más simple y convencional que otras alternativas que se han usado en los últimos años.

Tenemos que tener en cuenta que REST nunca guarda el estado en el servidor, por lo que toda la información que se requiere para mostrar, debe ser solicitada en la consulta por parte del cliente. Por lo tanto, el servidor solo mandará información al cliente cuando este se lo solicite (véase Figura 3).

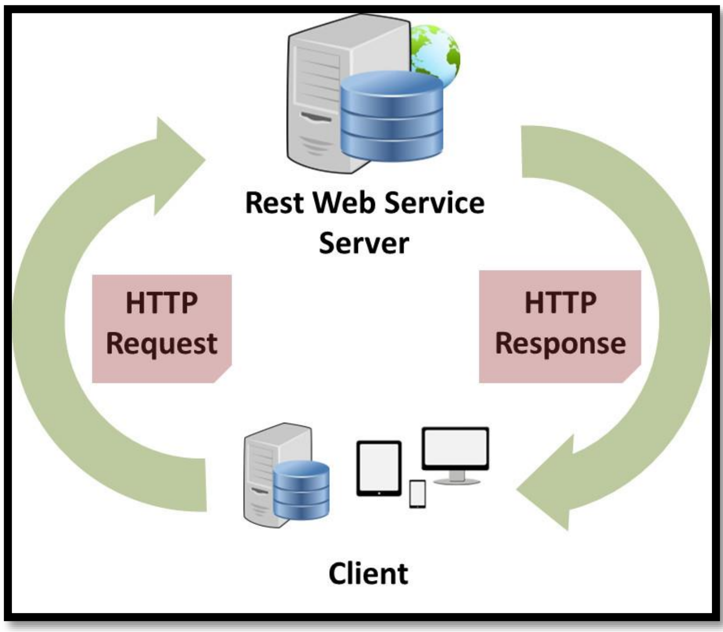


Figura 3: Comunicación REST

REST organiza estas solicitudes de manera predecible, usando tipos de operación HTTP para construir respuestas adecuadas. Las solicitudes se originan en el cliente, y los tipos de operación más comunes son GET, POST, PUT, DELETE, que corresponden generalmente a operaciones de leer, crear, modificar y borrar los datos respectivamente.

¿WebSocket vs REST?

Con HTTP cada vez que hacemos una petición, por ejemplo descargar un html o una imagen, un puerto/socket se abre, los datos se transfieren y luego se cierra, pero la apertura y cierre de la conexión supone una gran sobrecarga en términos de comunicaciones y de computación, y para ciertas aplicaciones, especialmente aquellas que quieren respuestas rápidas o interacciones en tiempo real o corrientes de visualización de datos, esto simplemente es inviable.

En 2011, el WebSocket se normalizó, y esto permitió a la gente a utilizar el protocolo WebSocket, que era muy flexible, para la transferencia de datos hacia y desde los servidores al navegador.

REST es, con mucho, la forma más normalizada de la estructuración de la API para las solicitudes de servicio, pero ya que implica el uso de HTTP también tiene la sobrecarga asociada con ese protocolo. Para algunas aplicaciones, la información sólo necesita ser transferida cuando un usuario realiza una acción. Para una mayor interacción en tiempo real, o la transferencia en tiempo real de los datos, HTTP y REST no son la combinación de protocolos más adecuada.

En conclusión, los WebSocket al ser una tecnología de conexión rápida y bidireccional, es claramente la mejor opción para juegos. Con el uso de los websockets los datos pueden enviarse de forma instantánea tanto desde el servidor al jugador como desde el jugador hacia el servidor de forma simultánea, ya que se está usando un protocolo full-duplex. Además, un beneficio obvio es que ahora el ancho de banda es mayor, ya que no se requieren solicitudes constantes para buscar nuevos datos. En vez de eso, la conexión WebSocket se deja abierta y los datos pasan de manera instantánea al jugador o al servidor en cuanto se necesita. Es perfecto para los juegos de ritmo rápido que requieren una actualización cada pocos milisegundos.

Por todo ello, y tras sopesar las dos posibilidades hemos decidido usar las dos tecnologías pero en partes diferentes del juego. Por un lado se utiliza REST para la comunicación entre cliente y servidor en la parte del login y signin, ya que el servidor tendrá que realizar las tareas únicamente cuando el cliente se lo solicite; y por otro lado se emplean los websocket como forma de comunicación entre cliente y servidor durante el juego en sí.

2.4 Java

Java [7] es un lenguaje de programación de propósito general, concurrente, orientado a objetos que fue diseñado específicamente para tener tan pocas dependencias de implementación como fuera posible. Su intención es permitir que los desarrolladores de aplicaciones escriban el programa una vez y lo ejecuten en cualquier dispositivo, lo que quiere decir que el código que es ejecutado en una plataforma no tiene que ser recompilado para correr en otra.

Java se convierte, a partir de 2012, en uno de los lenguajes de programación más populares en uso, particularmente para aplicaciones de cliente-servidor web.

Es el lenguaje de programación que se ha utilizado en el desarrollo de este proyecto, en el lado del servidor, para procesar la información y las peticiones de los usuarios y enviarles dichos datos como respuesta.

2.5 JavaScript

JavaScript [8] es un lenguaje de programación orientado a objetos utilizado principalmente en el desarrollo de aplicaciones web en el lado del cliente, aunque también es utilizado en algunos casos en el lado del servidor y en otros entornos. Surgieron diferentes versiones similares a JavaScript hasta que finalmente se propuso como estándar, para evitar incompatibilidades según el navegador utilizado. Centrándose en el lado del cliente en páginas web, como afecta a este trabajo, la función principal de estos scripts es la modificación de lo mostrado en la ventana del navegador por un lado, y por otro tanto la creación de WebSocket como el envío y recepción de los datos.

JavaScript es un lenguaje bastante orientado a la gestión de eventos. Sobre algunos elementos o variables, se crea un *listener* o manejador de eventos, por ejemplo se programa el código JavaScript de tal manera que si se realiza un click de ratón sobre cierto elemento se dispara el evento *click* sobre dicho elemento (el lanzamiento del evento está integrado de forma nativa por el navegador) y el manejador captura el evento y realiza la función que se le hubiese encomendado o programado.

2.6 CSS3

CSS (Cascading Style Sheets) [9] u hojas de estilo en cascada, nos permite describir con facilidad estilos de aplicaciones Web a través de una lista de reglas.

El CSS para un documento HTML puede ser incluido dentro de la etiqueta de apertura de un elemento, mediante el atributo *style* indicando como valor el estilo de dicho elemento. Otra opción mejor, consiste en poner el estilo de todo el documento entre las etiquetas de apertura y cierre de un elemento *style* dentro del propio HTML, para una mejor lectura y que el contenido del HTML sea más claro, reduciendo el tamaño de las etiquetas de los elementos y evitando repetirlo para cada uno de los elementos que compartan características. Sin embargo, la mejor opción es poner esa información en un documento externo al HTML, en un documento CSS, y referenciarlo desde la cabecera del documento HTML con un elemento link, con el atributo *rel* con valor *stylesheet* y el atributo *href* con el valor de la URL del fichero CSS.

A menudo, CSS se descuida o se malinterpreta en el desarrollo de juegos, pero es una herramienta muy potente. Con CSS no solo podemos describir estilos sino también comportamientos.

Las cascadas de CSS son un potente sistema que nos permite redefinir con facilidad estilos existentes al proporcionar un selector CSS más preciso. El navegador define estilos predeterminados que podemos anular. Entonces podemos expresar estilos generales y amplios que muestran en cascada muchos elementos y, después, anularlos para cualquier elemento específico, según sea necesario.

CSS también nos permite realizar transiciones animadas, algo que a menudo es una alternativa mejor a la animación de JavaScript porque es acelerada por hardware mediante el motor de renderizado del navegador.

Concretamente para este trabajo, se ha realizado una hoja de estilos sencilla utilizada para centrar los formularios en el navegador (login y signin) y para centrar el tablero del juego y todos los elementos que se ubican dentro de él (botones, texto, distintos jugadores…)

2.7 HTML5

HTML (HyperText Markup Language) [10] es un lenguaje de marcado de hipertexto, es el lenguaje que describe todo el contenido de una página Web. Al igual que XML, es un leguaje de marcado, lo que significa que se estructura como un árbol de etiquetas con contenido. La gama actual de elementos de HTML disponibles es muy amplia. Enlaza y estructura diferentes recursos: imágenes, vídeos, hojas de estilo y scripts de JavaScript. La especificación de HTML que se ha lanzado más recientemente es HTML5, que incluye nuevas etiquetas, nuevas API de JavaScript como los elementos canvas, video, audio, drag & drop, etc.

Al ser una aplicación web necesitamos de este lenguaje para definir el contenido que tendrá cada página web que se quiera mostrar.

2.8 Canvas

La salida visual es uno de los componentes principales de la mayoría de juegos, así que la capacidad para producir y administrar gráficos 2D dentro de un navegador es muy importante.

El canvas de HTML5 [11] (a menudo denominado sólo "canvas") se trata de una API de JavaScript y el elemento HTML correspondiente que permite que los gráficos de mapas de bits se creen, dibujen y se editen dentro del navegador. Los puntos a favor del canvas son su velocidad y su capacidad para producir gráficos de pixeles exactos con facilidad. Los aspectos negativos son que el rendimiento varía entre unas plataformas y otras y que la funcionalidad de las animaciones no está integrada.

Canvas se ha desarrollado en la parte del cliente. Su principal función es dibujar el tablero de juego y todos los demás elementos que se sitúan en él.

2.9 Bootstrap

Bootstrap [12] es un framework o conjunto de herramientas de software libre para desarrollar aplicaciones web, aportando, principalmente, funcionalidades para diseñar el estilo de una página web haciendo uso de HTML, CSS y JavaScript. Fue creado por trabajadores de Twitter (una famosa red social) para lograr una homogeneidad en el estilo de las páginas creadas por la empresa, aunque fue publicada de forma libre y su uso se ha popularizado, siendo actualmente una herramienta muy utilizada. Bootstrap da prioridad a la navegación y visualización de las páginas web, desarrolladas con este framework[9], en dispositivos móviles. Sin

[9] Framework: Es un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar.

embargo, esta herramienta permite desarrollar simultáneamente aplicaciones web con una interfaz adecuada para dispositivos de diversos tamaños de forma sencilla.

Bootstrap se ha utilizado en la parte de login y registro de la aplicación. Véase apartado 5.1.1.

2.10 AJAX

Inicialmente la modificación del árbol DOM[10] con JavaScript no añadía información que no existiese en el documento HTML o en el script de JavaScript en el momento de su descarga o información introducida por el usuario, por ejemplo a través de un formulario. Sin embargo, posteriormente la necesidad de una web más dinámica dio lugar a la aparición de la tecnología AJAX [13] (Asynchronous JavaScript And XML) que permite realizar peticiones al servidor y recibir una respuesta y mostrarla sin necesidad de recargar la página entera, añadiendo con JavaScript en el árbol DOM la información recibida, la cual puede ser previamente procesada.

AJAX no es una tecnología basada exclusivamente en JavaScript, sino que también se basa en HTML y CSS para mostrar el contenido, para servir de interfaz al usuario para solicitar las interacciones AJAX o simplemente para mostrar al usuario que se está procesando una solicitud.

Esta tecnología permite el diseño de páginas web más eficientes al poder solicitar solo la información requerida, aprovechando los datos previamente descargados, y también permite crear aplicaciones más fluidas y dinámicas de cara al usuario final, logrando simular el funcionamiento de programas instalados en un dispositivo a través de páginas web cargadas en un navegador, eliminando la sensación de estar esperando a que la página se descargue, ya que, mientras se descargan y se procesan los datos solicitados, la página sigue mostrándose y puede seguir siendo funcional, ya que las peticiones al servidor se realizan en segundo plano y de forma transparente para el usuario.

Por consiguiente, se ha considerado interesante utilizar AJAX en la parte de login y signin. Al introducir los datos del usuario en los formularios, es una función AJAX la encargada de recogerlos y pasárselos al servidor para que este realice las operaciones de comprobación en la base de datos y le devuelva más tarde unos resultados que dicha función AJAX puede mostrar en el navegador.

La Figura 4 muestra gráficamente el funcionamiento del modelo AJAX frente al modelo tradicional.

---

[10] DOM (Document Object Model): es esencialmente una interfaz de programación de aplicaciones (API) que proporciona un conjunto estándar de objetos para representar documentos HTML, XHTML y XML, un modelo estándar sobre cómo pueden combinarse dichos objetos, y una interfaz estándar para acceder a ellos y manipularlos.

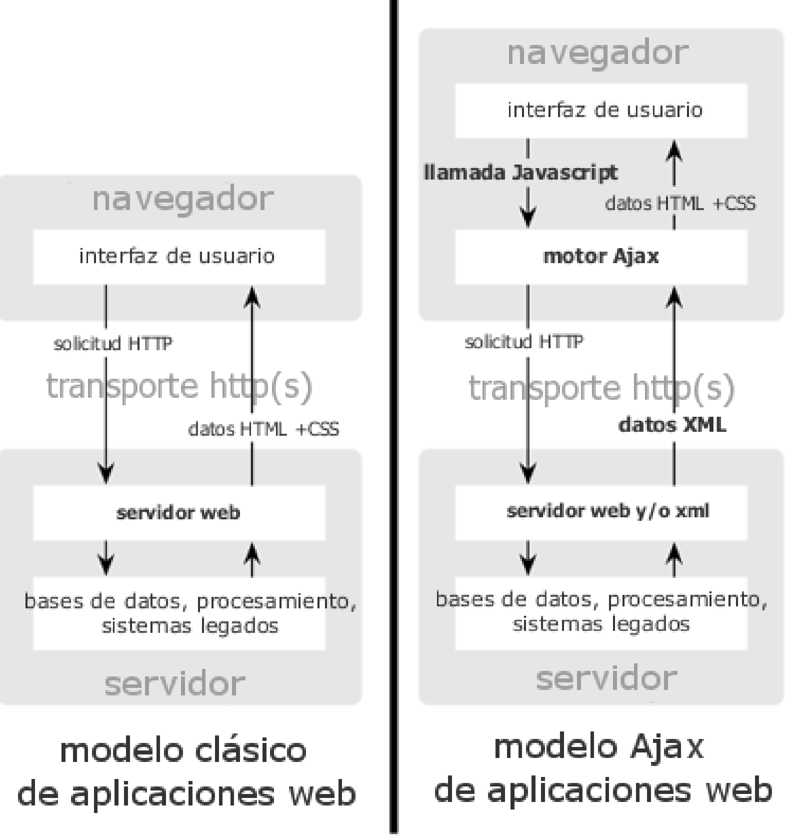


Figura 4: Comparativa modelo Ajax & modelo clásico

2.11 JQuery

Una de las bibliotecas de JavaScript más utilizadas es jQuery [14]. Esta biblioteca facilita en gran medida el acceso y la manipulación del árbol DOM, el manejo de eventos y la modificación de estilos o del CSS, así como la implementación de funciones muy útiles y facilitar la realización y procesado de peticiones AJAX. Una de las causas de su popularidad es la licencia de software libre que permite utilizarla tanto para proyectos libres como proyectos privados.

La función principal de jQuery es `$()`, un alias de `jQuery()`, a la que se le pasa como parámetro una regla CSS o el nombre de una etiqueta HTML, seleccionando todos los elementos que están asociados con esa expresión. Sobre estos elementos seleccionados se puede aplicar una función de jQuery, como puede ser eliminarlos, modificar su estilo o programar un manejador de eventos entre otras opciones.

Ha sido necesaria la utilización de la biblioteca jQuery para el desarrollo de la parte del cliente del login y signin, ya que al realizar ambas páginas con Bootstrap se necesita de esta biblioteca que le aporta funcionalidad a toda la interfaz.

2.12 JSON

JSON o JavaScript Object Notation [15] es un formato ligero para el intercambio de datos. Es fácil de leer y de escribir por humanos, y fácil de generar y leer para un ordenador. Un documento JSON contiene datos que pueden ser convertidos a objetos o variables en distintos

lenguajes de programación, al igual que existen multitud de bibliotecas para realizar la operación inversa. Es un formato muy utilizado en la actualidad para enviar información desde un servidor a un cliente ante una petición AJAX, siendo dicha información interpretada por código JavaScript del cliente, permitiéndole trabajar con esos datos como si se tratase de cualquier otra variable.

En este caso, JSON es el formato utilizado para el intercambio de datos o información desde el cliente al servidor y viceversa.

2.13 GlassFish

GlassFish [16] es un servidor de aplicaciones desarrollado por Sun Microsystems[11] que implementa las tecnologías definidas en la plataforma Java EE y permite ejecutar aplicaciones que siguen esta especificación.

Un servidor de aplicaciones proporciona generalmente gran cantidad de funcionalidades de forma transparente al usuario de manera que no sea necesario escribir código fuente. Estas funcionalidades son posibles ya que los componentes se ejecutan dentro del contenedor en un espacio de ejecución virtual llamado dominio de ejecución. Su función principal es la de interponerse entre las llamadas que se hacen a los métodos y las implementaciones de los mismos, de modo que entre otras cosas puede hacer las comprobaciones para verificar si el usuario que llama al método tiene los permisos adecuados, antes de llamarlo.

En este proyecto se usa el servidor GlassFish 4.1 ya que es el que viene en NetBeans por defecto.

2.14 MongoDB

MongoDB [17] es un sistema de base de datos NoSQL[12] orientado a documentos, desarrollado bajo el concepto de código abierto.

MongoDB forma parte de la nueva familia de sistemas de base de datos NoSQL. En vez de guardar los datos en tablas como se hace en las base de datos relacionales, MongoDB guarda estructuras de datos en documentos tipo JSON con un esquema dinámico (MongoDB llama ese formato BSON), haciendo que la integración de los datos en ciertas aplicaciones sea más fácil y rápida.

---

[11] Sun Microsystems: fue una empresa informática que se dedicaba a vender servidores, componentes informáticos, sistemas operativos y servicios informáticos. Fue adquirida en el año 2010 por Oracle Corporation, y formó parte de los iconos de Silicon Valley, como fabricante de semiconductores y software.

[12] NoSQL: es una amplia clase de sistemas de gestión de bases de datos que difieren del modelo clásico del sistema de gestión de bases de datos relacionales en aspectos importantes, el más destacado es que no usan SQL como el principal lenguaje de consultas. Los datos almacenados no requieren estructuras fijas como tablas, normalmente no soportan operaciones JOIN y habitualmente escalan bien horizontalmente.

Imaginemos que tenemos una colección a la que llamamos Usuarios. Un documento podría almacenarse de la siguiente manera:

```
{
  Nombre: "Pedro",
  Apellidos: "Martínez Campo",
  Edad: 22,
  Aficiones: ["fútbol","tenis","ciclismo"],
  Amigos: [
   {
     Nombre:"María",
     Edad:22
   },
   {
     Nombre:"Luis",
     Edad:28
   }
  ]
}
```

El documento anterior es un clásico documento JSON. Tiene strings, arrays, subdocumentos y números. En la misma colección podríamos guardar un documento como este:

```
{
  Nombre: "Luis",
  Estudios: "Administración y Dirección de Empresas",
  Amigos:12
}
```

Este documento no sigue el mismo esquema que el primero. Tiene menos campos, algún campo nuevo que no existe en el documento anterior e incluso un campo de distinto tipo.

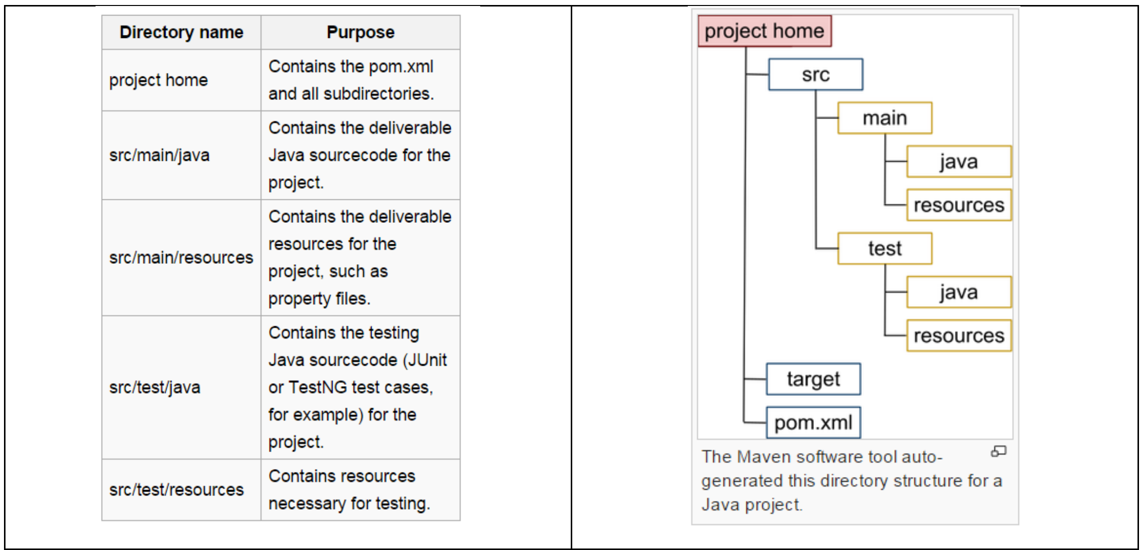
Esto que es algo impensable en una base de datos relacional, es algo totalmente válido en MongoDB, por esta razón ha sido un serio candidato para almacenar los datos de registro de usuario de la aplicación.

3. Organización del código y ejecución

3.1 Jerarquía en el panel de proyecto

Un proyecto en la plataforma NetBeans es un término que se utiliza para referirse a la agrupación de carpetas y archivos que queremos que se consideren como una sola unidad. El tratamiento de las carpetas y los archivos relacionados como una única cosa hace que trabajar con ellos sea más fácil para el usuario final. Para este trabajo se ha creado un proyecto Maven en NetBeans.

Maven, proporciona una estructura por defecto para la organización de los ficheros asociados a un proyecto (código fuentes, librerías, ficheros de configuración, datos, etc.) que se está convirtiendo en un estándar de facto para el desarrollo de aplicaciones en Java. (Figura 5)



Directory name	Purpose
project home	Contains the pom.xml and all subdirectories.
src/main/java	Contains the deliverable Java sourcecode for the project.
src/main/resources	Contains the deliverable resources for the project, such as property files.
src/test/java	Contains the testing Java sourcecode (JUnit or TestNG test cases, for example) for the project.
src/test/resources	Contains resources necessary for testing.

Figura 5: Estructura de un proyecto con Maven

Para la aplicación web hemos creado un proyecto Maven llamado *JuegoWebSocket* y dentro de él se van a encontrar diferentes carpetas que clasifican los diferentes archivos necesarios que hemos creado para nuestro juego. (Véase Figura 6).

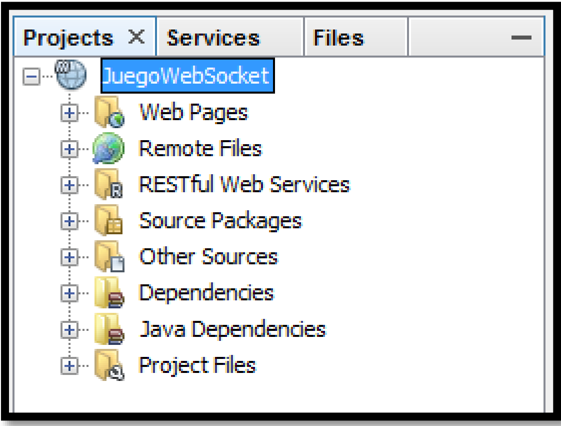


Figura 6: Jerarquía de carpetas general en Netbeans

Como hemos visto, dentro de nuestro proyecto se encuentran diferentes carpetas que contienen todas las fuentes y metadatos del proyecto.

- Web Pages: En esta carpeta se encuentran, por un lado, todos los archivos css, html y javascript que hemos creado a lo largo del desarrollo de la aplicación, y por otro existen dos carpetas más, una llamada WEB-INF creada por defecto al crear el proyecto y que está vacía, y otra llamada *img* creada para alojar todas las imágenes y sonidos de nuestro juego.

  Podemos considerar que en esta carpeta se encuentran todos los archivos de la parte del cliente que se ejecutan en el navegador. (Véase Figura 7).

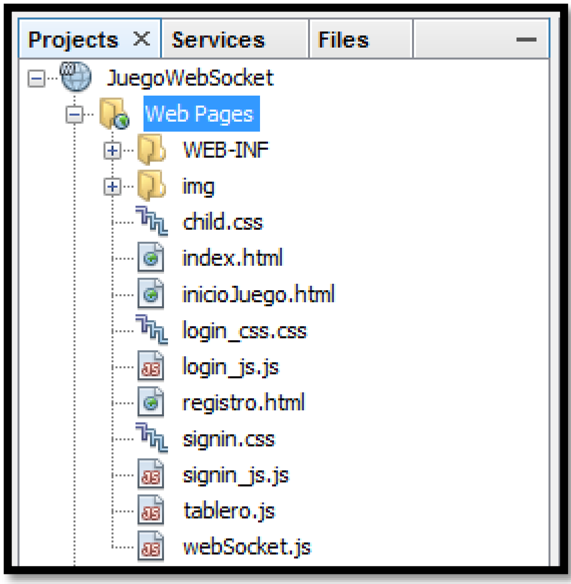


Figura 7: Jerarquía de carpetas - Web Pages

- Remote Files: En esta carpeta se ubican todos los archivos que hemos incluido en la parte del cliente y que apuntan a archivos remotos. NetBeans se encarga de identificarlos y seleccionarlos para almacenarlos en esta carpeta. (Véase Figura 8).

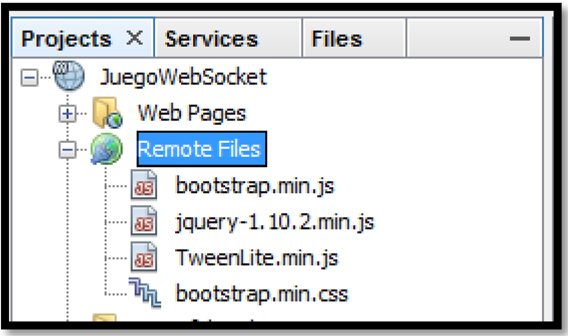


Figura 8: Jerarquía de carpetas - Remote Files

- RESTful Web Services: Esta carpeta es necesario que sea creada, ya que no contamos con ella al crear el proyecto. Aquí se almacenan los archivos de servicios web REST que hayan sido necesarios crear, en nuestro caso han sido dos: LoguinResource y SigninResource. Vemos además que aparecen los métodos que contienen cada uno de ellos. (Véase Figura 9).

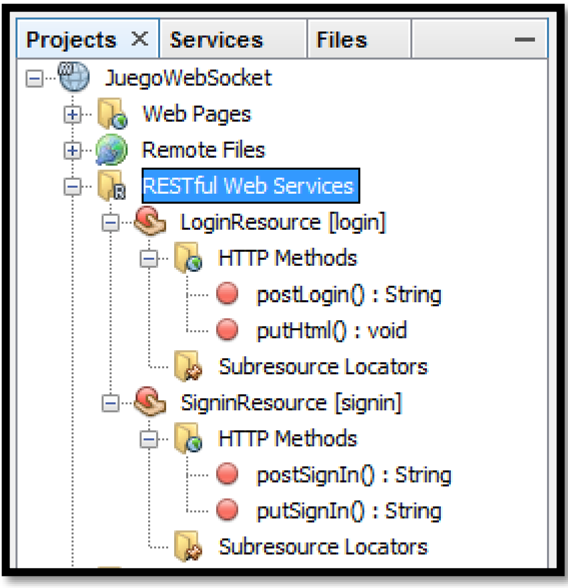


Figura 9: Jerarquía de carpetas - RESTful Web Services

- Source Packages: Podríamos decir que es el paquete principal de nuestro proyecto, aquí están alojados todos los archivos java del juego. Aquí se encuentran todos los archivos de la parte del servidor. (Véase Figura 10).

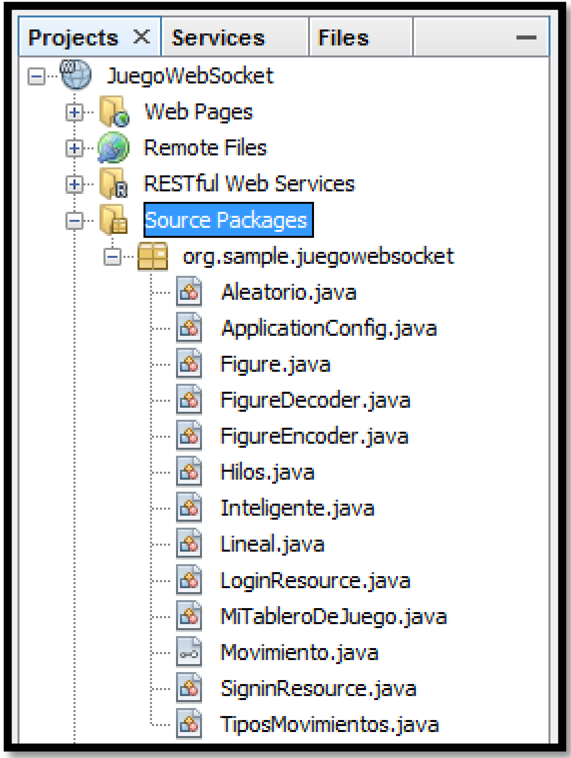


Figura 10: Jerarquía de carpetas - Source Packages

- Dependencies y Java Dependencies: Aquí, como su propio nombre indica, se guardaran todas las dependencias necesarias para nuestro proyecto. Por un lado están las dependencias java que incluye el proyecto por defecto y por otro las dependencias que se van añadiendo según se van necesitando como es en nuestro caso el driver de mongoDB y la librería de JSON. (Véase Figura 11).

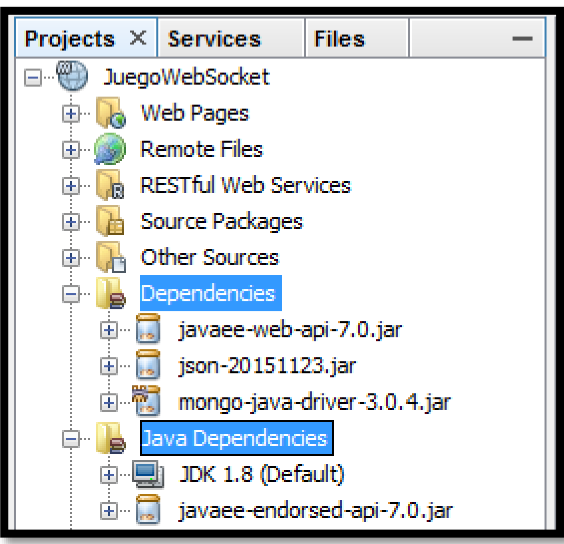


Figura 11: Jerarquía de carpetas - Dependencias

- Project Files: Por último, esta carpeta almacena el archivo POM y el de configuración, ambos xml. (Véase Figura 12).

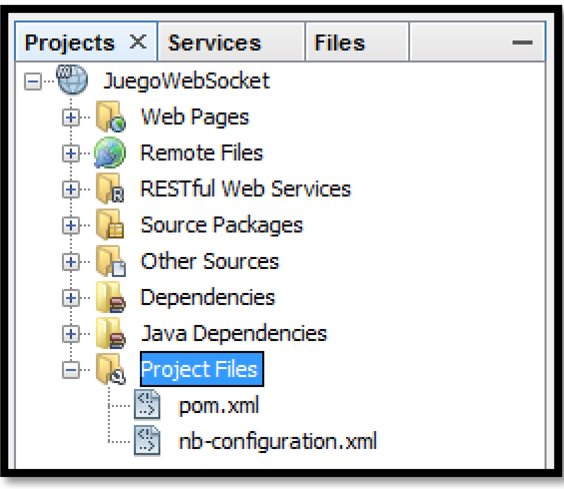


Figura 12: Jerarquía de carpetas - Project Files

3.2 Instalación y configuración

Para el desarrollo y uso de la aplicación web la única instalación que ha sido necesaria hacer es la del entorno de desarrollo NetBeans y la de MongoDB. Este último también teniendo que ser configurado posteriormente.

Para la instalación de NetBeans accedemos a la página oficial y desde ahí haremos la descarga (https://netbeans.org/downloads/), no es necesaria ninguna configuración extra.

Para MongoDB ha sido necesario instalarlo y configúralo como un servicio de Windows. A continuación expongo los pasos a seguir:

1. Descargamos MongoDB para Windows de la siguiente página: https://www.mongodb.org/downloads?_ga=1.198234025.788605313.1444207927#production
2. Una vez instalado abrimos un terminal cmd[13] como administrador y procedemos a la configuración.
3. Creamos los directorios *C:\data\db*, *C:\data\log* y *C:\mongodb*
4. Dentro de la carpeta anteriormente creada (mongodb) tendremos que crear un archivo llamado mongod.cfg y copiamos lo siguiente dentro de él:

```
systemLog:
         destination: file
         path: C:\data\log\mongod.log
storage:
     dbPath: C:\data\db
```

[13] cmd: es el intérprete de comandos de sistemas basados en Windows.

5. Una vez llegados a este punto vamos a instalar MongoDB como un servicio, escribimos la siguiente secuencia de comandos en el terminal abierto en el paso 2.

```
"C:\mongodb\bin\mongod.exe" --config "C:\mongodb\mongod.cfg" –install
```

6. Ya se ha instalado correctamente MongoDB, ahora para iniciar el servicio tendremos que abrir dos terminales cmd como administrador. En uno de ellos escribiremos lo siguiente:

```
Net satart MongoDB
```

7. En el otro terminal que hemos abierto nos posicionamos dentro de C:\mongodb\bin y ponemos:

```
Mongo
```

Una vez iniciado el servicio se puede empezar a crear bases de datos, tablas y datos. En nuestro caso hemos creado una base de datos llamada *webservice* y dentro de esta una tabla *login* en la que se almacenarán los datos de los usuarios registrados en el juego. Todo esto se realiza de la siguiente manera sobre el terminal.

```
use webservice        // crear base de datos
db.login.insert({nom_usu:"rmorcar", edad:"25", password: "147258369”})
//crea una colección (tabla) e inserta un documento

db.login.find() //muestra todos los documentos que hay en la colección
```

3.2 Manual de arranque

En primer lugar debemos tener en cuenta que el servicio MongoDB debe estar iniciado en nuestro terminal (punto 6 del apartado anterior).

Una vez hecho esto, y con el proyecto abierto en NetBeans tendremos que:

1. Pulsar botón derecho sobre el proyecto y pulsar Build with Dependencies.
2. Una vez terminado este proceso y si todo ha ido bien pulsaremos sobre Run para poner en marcha el juego. A continuación la Figura 13 muestra donde se encuentran dichos botones.

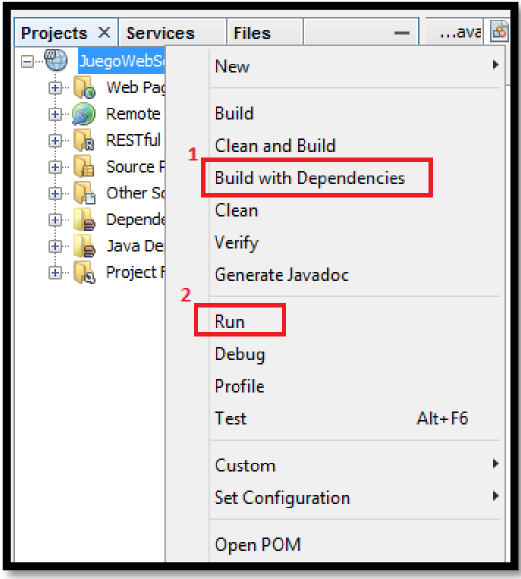


Figura 13: Arranque del proyecto

3.3 Manual de uso

Una vez iniciada la aplicación la primera pantalla que se muestra es la de login, en ella se espera recoger los datos del usuario para hacer una comprobación en la base de datos. Si los datos introducidos corresponden a un usuario de la base de datos entonces se abrirá una nueva página con el juego, y si por el contrario no es así se mostrará un mensaje advirtiendo al usuario que debe registrarse primero para poder empezar a jugar.

La Figura 14 muestra la página inicial de login del juego:

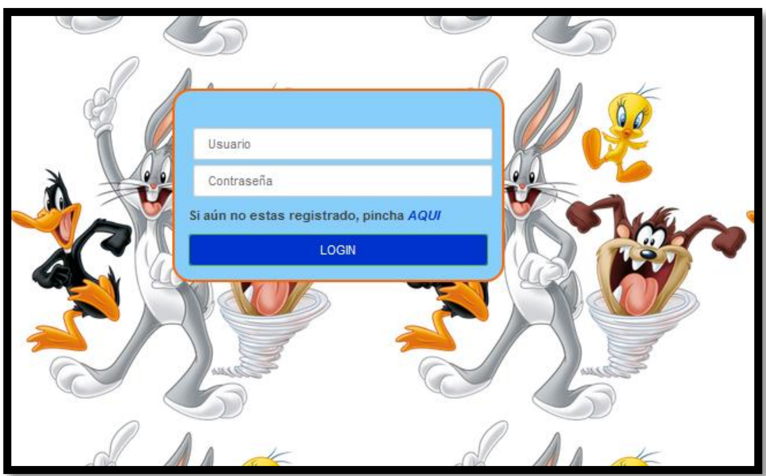


Figura 14: Vista del Login

En caso de que el usuario no esté registrado en la base de datos deberá pinchar en el enlace indicado en la página principal, el cual le lleva a otra página nueva de registro. Aparecerán (como se ve en la Figura 15) una serie campos que será necesario introducir. Si todas las

comprobaciones son correctas (error por duplicado) se almacenarán en la tabla de *login* de la base de datos *webservices.*

Figura 15: Vista del formulario de registro

Es de especial interés explicar que para obtener los datos utilizamos AJAX y mediante un POST le pasamos los datos a la clase propia de REST que se conectará a la base de datos y hará las comprobaciones de las que hemos hablado anteriormente, cuando acaben las comprobaciones le será devuelto a la función AJAX un resultado que será el que muestre en pantalla (véase apartado 5.2)

Tras pasar por el login, y si el usuario y contraseña son correctos, se iniciará el juego. Al abrirse una nueva pantalla nos encontramos con la siguiente interfaz (veáse Figura 16) en la que el usuario (jugador) intentará esquivar al enemigo y evitar que le robe el mayor de vidas posibles.

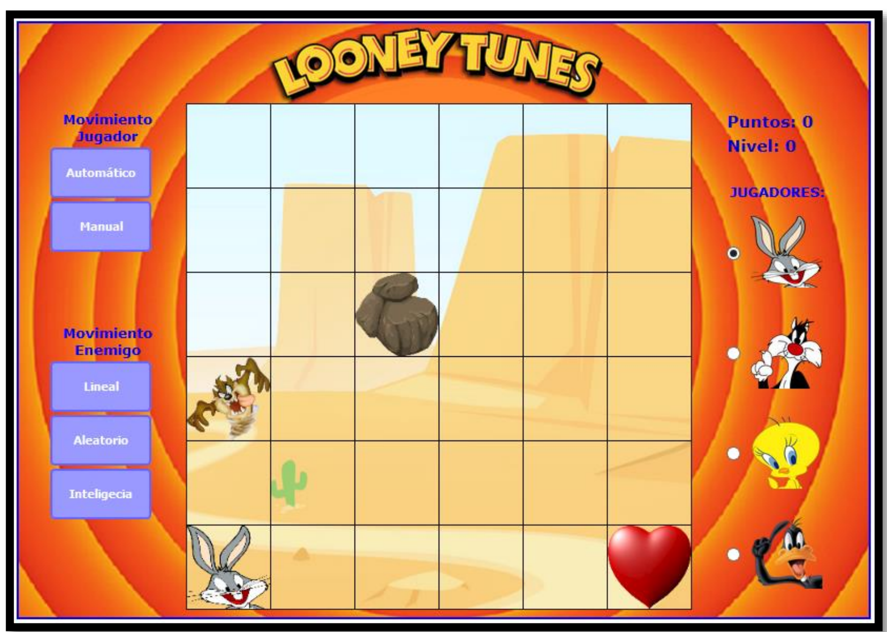


Figura 16: Vista del juego

Como podemos observar el juego está compuesto de cuatro partes:

1. El tablero de juego, en el cual se encuentra el enemigo, el jugador, un obstáculo por el que ninguno de los dos anteriores podrá pasar, y una vida que en caso de ser cogida por el jugador le dará puntos a este e irá subiendo de nivel, y en caso de cogerla el enemigo le irá restando puntos al jugador hasta vencerle en la partida.

2. El movimiento del jugador. Son dos botones que permiten elegir entre un movimiento manual o automático del jugador. En el primer caso, el jugador lleva el control del movimiento del personaje y este solo cambiará de posición si es pulsada alguna de las teclas para el movimiento. En la opción automática, solo se le indicará la dirección en la que se desea que se mueva y el personaje seguirá moviéndose en esa dirección mientras que no se pulse otra tecla que la haga cambiar.

3. El movimiento del enemigo. Existen tres posibilidades en este caso: lineal, aleatorio e inteligente.

    i. Lineal: Cuando seleccionamos este tipo de movimiento el enemigo se desplazará por cada cuadro siguiendo un patrón fijo. Es el tipo de movimiento que hace que el juego tenga menor dificultad.

    ii. Aleatorio: El enemigo se moverá por el tablero de manera random, creando en cada intervalo de tiempo una dirección a la que moverse. Este tipo de movimiento es un poco más difícil que el anterior, ya que el movimiento del enemigo es imprevisible.

    iii. Inteligente: Este tipo de movimiento le añade mayor dificultad al juego, ya que el enemigo de forma inteligente persigue al jugador y corre en busca de las vidas. Determina, en función de la distancia que lo separa del jugador y de la vida, cual es la mejor opción de persecución.

4. Puntuación y nivel. En esta sección se indica la puntuación que tiene el jugador en cada momento y el nivel en el que se encuentra. Cada vez que este obtiene una vida aumentan en 10 sus puntos, pero cada vez que el enemigo se la quita decrementan en 5 unidades. Si el jugador se queda sin puntos pierde la partida.

    Por otro lado, el nivel en el que se encuentra el jugador aumenta cada vez que este coge 3 vidas. Subir de nivel significa añadir dificultad al juego, que se hará aumentando la velocidad de persecución del enemigo. Existen actualmente 18 niveles de dificultad.

5. Diferentes jugadores. El jugador puede elegir el personaje con el cual jugar. Hay 4 opciones disponibles. El personaje se podrá cambiar en cualquier momento del juego.

El desarrollo de la aplicación es posible utilizando WebSocket que comunican cliente con servidor. En la parte del cliente hemos implementado toda la lógica del juego exceptuando la del enemigo que se implementa a través de un hilo[14] en el servidor, y la del control de las

[14] Hilo: También llamado Thread, es un trozo de código de un programa que puede ser ejecutado al mismo tiempo que otro.

diferentes sesiones[15]. Tanto el cliente como el servidor intercambian mensajes informando al otro del estado actual del juego con las modificaciones que ha realizado cada uno.

Como hemos mencionado anteriormente también se hace un seguimiento de las sesiones abiertas, a cada sesión se le pueden añadir "compañeros", es decir, ventanas abiertas que pertenecen a la misma sesión de usuario, las cuales pueden observar la estrategia de juego llevada a cabo por el jugador, e intervenir en el juego en cualquier momento ya que los cambios realizados se propagan a todos los integrantes de una misma sesión.

Es necesario indicar, que para visualizar el funcionamiento de dos sesiones diferentes con sus respectivos compañeros es necesario abrir primero una sesión y añadir compañeros a esta, posteriormente se deberá hacer de forma similar con otra sesión, ya que al probar el juego en local los compañeros se unen a la última sesión abierta. Una vez realizado, se comprueba que los cambios solo se propagan entre los compañeros integrantes de una misma sesión.

En la Figura 17 aparece una misma sesión con dos jugadores unidos a ella. Se puede comprobar lo explicado anteriormente, ambos tienen el mismo tablero de juego y los cambios que realiza uno se propagan a todos.

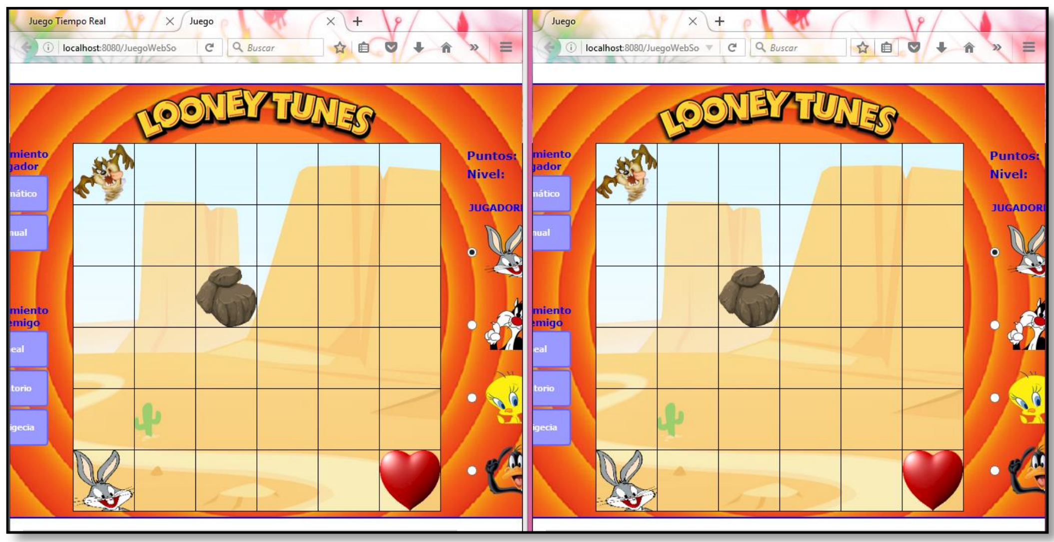


Figura 17: Dos compañeros de una misma sesión

[15] Sesión: una sesión es un intercambio de información interactiva semi-permanente, también conocido como diálogo, una conversación o un encuentro, entre dos o más dispositivos de comunicación. Véase apartado 5.3 Juego >> servidor >> sesión.

Por otra parte, en la Figura 18 se aprecia lo contrario. Cuando dos usuarios juegan en sesiones diferentes, los datos intercambiados solo se propagan a los miembros de una misma sesión. Por lo que dos sesiones distintas abiertas tendrán un tablero de juego distinto.

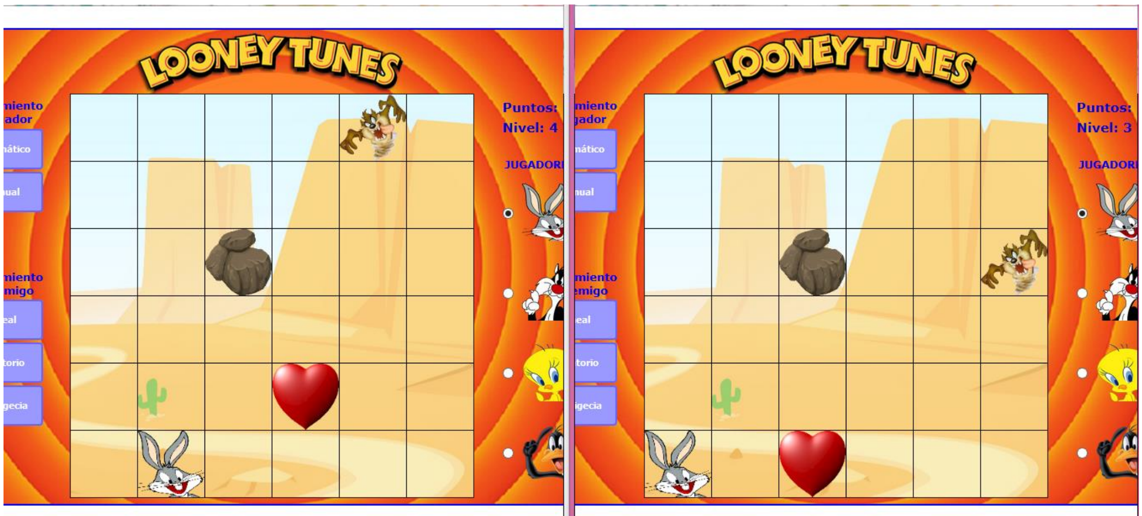


Figura 18: Dos sesiones diferentes

4. Diseño e implementación

A pesar de que el código está adjunto a la memoria correspondiente a la aplicación web, en esta sección se procederá a explicar con detalle cada una de las partes que intervienen en el juego (cliente y servidor). Se presupone al lector un cierto dominio de Java y JavaScript.

4.1 Partes que intervienen. Cliente y servidor

De forma general se puede clasificar el desarrollo de la aplicación web en dos grandes grupos, los formados por el cliente y por el servidor. A continuación explicaremos las funcionalidades que adquieren cada uno de ellos, las tecnologías que usan y cómo funcionan.

Para que la explicación sea más clara separamos la aplicación en dos partes, una la conformada por la parte inicial de login y signin, y otra por el juego en sí. Dentro de estos dos grupos existe un cliente y un servidor a describir a continuación.

4.2 Login y Signin

Al iniciar la aplicación lo primero que se muestra es un navegador en el cual el usuario tiene que identificarse con su nombre de usuario y su contraseña. En caso de que el usuario aún no se haya registrado en el juego y no disponga de estos datos será necesario que se registre en el juego. Para eso deberá hacer clic sobre un enlace indicado en ese navegador, y a través del cual se abrirá otro navegador donde poder introducir en un formulario sus datos y poder posteriormente iniciar el juego.

En este proceso intervendrá tanto un cliente como un servidor:

- La parte del cliente es la que se encarga de mostrar el contenido en el navegador, de la interfaz mostrada al usuario para introducir los datos o para registrarse por primera vez.
- La parte del servidor es la encargada de recoger los datos introducidos por el usuario y operar con ellos en la base de datos, ya sea para verificar los datos del usuario como para introducirlos en ella.

La comunicación cliente-servidor de esta parte se hace mediante la arquitectura REST.

Cliente

Como se acaba de explicar anteriormente podemos considerar que la parte del cliente es todo aquello que el usuario puede visualizar en su navegador al iniciar el proceso de login o de registro. Para crear esta parte hemos usado: HTML, CSS, JavaScript.

Tanto la página de login como de registro (signin) se han realizado de forma similar. En primer lugar se desarrolla la página HTML en la cual se añaden todos los elementos necesarios como CSS, JavaScript.

- LOGIN: En la página HTML de la parte de login (index.html) se añade un formulario centrado en el navegador, en el cual el usuario tendrá que introducir sus datos. Tanto el formulario

como el movimiento de la imagen del fondo del navegador, están realizados con Bootstrap, JavaScript y jQuery. Es necesario añadir los script a la página HTML, en este caso son necesarios los siguientes:

```
<script src="http://mymaplist.com/js/vendor/TweenLite.min.js"></script>
<script src="//code.jquery.com/jquery-1.10.2.min.js"></script>
<script
src="//netdna.bootstrapcdn.com/bootstrap/3.2.0/js/bootstrap.min.js"></script>
<script src="login_js.js" type="text/javascript"></script>
```

Los tres primeros script son precisos para el movimiento de la imagen del fondo, y para el formulario. Estos han sido descargados directamente de internet, y contienen funcionalidades necesarias útiles para el desarrollador de la aplicación, facilitando el trabajo, siento este es el objetivo principal de Bootstrap. Por otro lado, el último script "login_js.js" ha sido necesario crearlo y contiene dos funciones AJAX, una que recoge los datos introducidos por el cliente en el formulario y se los envía al servidor, y otra que detecta el movimiento del ratón sobre el navegador y hace variar la posición de la imagen del fondo.

A continuación vemos la función AJAX que recoge los datos introducidos por el usuario:

```
$("#login-form").submit(function( event ) {
        event.preventDefault();
        $.ajax( {
            type: "POST",
            url: "webresources/login",
            data: $("#login-form").serializeArray(),
            success: function( response ) {
               if (response ==="ok") {
                   $("#noRegistrado").html("");
                }
               else
                   $("#noRegistrado").html(response);
            }
    });
    });
```

Esta función obtiene los datos enviados en la parte del navegador que tiene como identificador `login-form` y los envía mediante un POST a la parte del servidor cuya dirección es `webresources/login`. En el servidor se hacen las comprobaciones pertinentes (con la base de datos) y si la respuesta de este es "ok" no se mostrará nada en la parte del navegador cuyo identificador sea `noRegistrado`, y en caso contrario se mostrará en el mismo lugar la respuesta enviada por el servidor.

Por otro lado, cabe mencionar que es en los archivos CSS dónde están descritos estos identificadores mencionados anteriormente. Además de tener esta funcionalidad también describen los estilos que va a tener la página en el navegador. Para eso hemos utilizado el estilo propio de Bootstrap descargado íntegro de internet y otra hoja de estilos creada posteriormente para modificar o añadir ciertos estilos que se ha precisado cambiar.

Tal y como hemos dicho con anterioridad, la forma en la que hemos desarrollado la parte del Login es muy similar a la del Signin (registro), el funcionamiento principal de la parte del cliente es semejante y para su desarrollo ha sido necesario seguir los mismo pasos, pero en este caso se ha utilizado otra plantilla de formulario de Bootstrap diferente.

Servidor

Debido a que el juego no presenta gran complejidad en cuanto al número de clases java, se ha optado por que todas estén incluidas en un solo paquete.

Para esta parte, el servidor tiene como único objetivo comunicarse con la base de datos MongoDB y realizar comprobaciones e inserciones en ella.

Cabe mencionar que se ha utilizado el driver para MongoDB 3.0.4, este no ha sido necesario descargarlo ya que al utilizar maven lo incluye automáticamente en nuestro proyecto.

Para poder hacer dichas comprobaciones e inserciones inicialmente debemos crear una base de datos y una colección o tabla en la cual almacenar los datos del usuario. En este caso se ha creado una base de datos llamada *webservice* y una colección *login* (véase sección 3.2).

Es necesario que exista una comunicación entre el servidor java y la base de datos, para ello será necesario crearla en el propio servidor de la siguiente manera:

```
@Path("login")
public class LoginResource {

    MongoClient mongoClient;
    MongoDatabase db;
    MongoCollection<Document> coleccion;

 @Context
public LoginResource() {

        mongoClient = new MongoClient("localhost", 27017); //Objeto cliente
        db = mongoClient.getDatabase("webservice"); //Obtiene base de datos
        coleccion = db.getCollection("login"); //Obtiene colección
    }

/* Aquí se encuentran el resto de métodos. Ej: @POST */
}
```

Una vez creada ya podemos operar con la base de datos a través del servidor. Cuando la función AJAX (cliente) recoge los datos introducidos por el usuario, envía mediante un POST los datos al servidor. Los datos llegan al método que etiquetado como `@POST`, y es en este método en el cual se opera con el contenido de la colección login. Las operaciones realizadas son distintas en los procesos del login y el registro:

- Login: En este caso, se recorre toda la colección login en busca de coincidencias de nombre de usuario y contraseña. Si existen se abre un nuevo navegador con el juego, en caso

contrario se le sugiere al usuario que escriba de forma correcta los datos y que se asegure de haberse registrado previamente.

- Signin: En el registro, inserta en la colección los datos introducidos por el usuario en caso de no existir coincidencias de nombre de usuario y dni. En tal caso aparecerá un mensaje informando al usuario que o bien ya está registrado o que el nombre de usuario ya está en uso. Una vez se haya registrado correctamente, el usuario tendrá que volver a la pestaña de login e introducir tanto su nombre de usuario como su contraseña para poder iniciar el juego.

Una vez realizados estos pasos el usuario finalizará la fase de registro y podrá empezar a jugar.

5.3 Juego

Una vez se abre la pestaña del juego observamos que el centro del navegador contiene un rectángulo en el cual se ubican todos los elementos que intervienen en el juego, desde diversos botones hasta el propio tablero donde se desarrolla el juego.

Al igual que el punto anterior, está compuesto por una parte que actúa como cliente y otra como servidor, cada una realizando tareas distintas resumidas en lo siguiente:

- La parte del cliente tiene como finalidad:
    i. Mostrar el contenido en el navegador, las vistas del juego y apariencia del juego mediante HTML y CSS.
    ii. Contiene la mayor parte de la lógica del juego, controla los eventos del teclado y los eventos de mensajes que provienen del servidor, utilizando JavaScript.
- El servidor por su parte realiza, en Java, las siguientes funciones:
    i. Controlar los eventos de recepción de mensajes procedentes del cliente.
    ii. Contiene la lógica del movimiento del enemigo mediante un hilo. Los distintos tipos de movimientos del enemigo están realizados mediante una factoría Java.
    iii. Propagación de los cambios a todos los clientes dentro de una misma sesión (envío broadcast[16]).

En esta parte de la aplicación web la comunicación entre cliente y servidor se efectúa mediante la tecnología WebSocket.

Es necesario que se tenga presente cómo se produce el intercambio de mensajes entre cliente y servidor, el cual se muestra de forma muy simplificada en la Figura 19:

[16] Broadcast: es una forma de transmisión de información donde un nodo emisor envía información a una multitud de nodos receptores de manera simultánea, sin necesidad de reproducir la misma transmisión nodo por nodo.

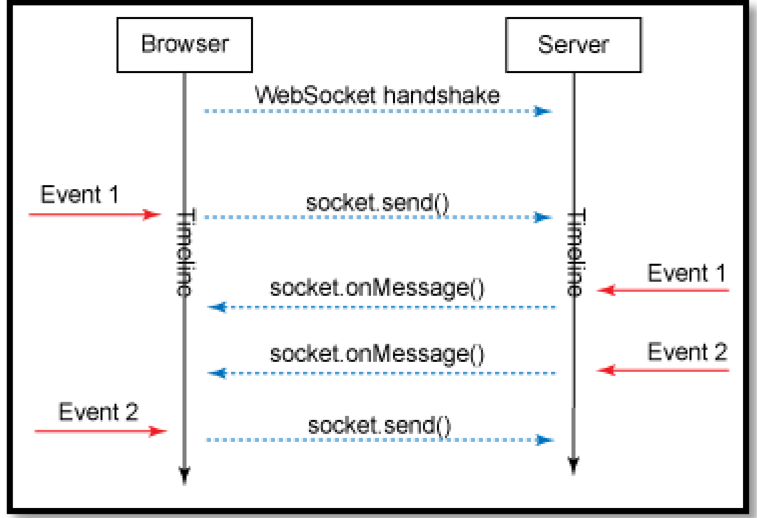


Figura 19: Comunicación cliente-servidor WebSocket

Inicialmente el cliente se une a una sesión HTTP a través de un establecimiento de sesión con el servidor a través de TCP. Este declara el WebSocket que se va a utilizar en la comunicación con el servidor y además contiene el nombre del endpoint al cual irán dirigidos los datos. El endpoint es el punto del servidor que contiene algunos métodos principales que se ejecutan cuando se abre o se cierra una sesión, y cuando un mensaje es recibido por parte del cliente.

El cliente iniciará la sesión de comunicación con el servidor cuando esté cargada por el navegador la parte que contiene la declaración del WebSocket y el enpoint.

Cliente

Como tarea principal, tiene la de procesar los eventos que le llegan del usuario mediante el teclado y los eventos que recibe por parte del servidor. Además, también es en el cliente donde se desarrolla toda la parte visual del juego: botones, imágenes... Y donde se colocan todos los elementos.

1. **Apariencia:** Tal y como se puede ver en el código adjunto a la memoria (archivo `inicioJuego.html`), para el desarrollo en HTML de esta parte se ha dividido el diseño en tres bloques:
   - Bloque izquierdo: En este aparecen cinco botones. Dos de ellos destinados al movimiento del cliente, y los otros tres al movimiento del enemigo. Cada botón se define de la siguiente forma:

```
<button id="button1" onclick="movimiento_enemigo(1)">Lineal</button>
```

Al pulsar sobre el botón le pasamos a la función definida en el cliente `movimiento_enemigo` el parámetro 1. Y el cliente avisará al servidor de que el usuario ha

solicitado un cambio en el movimiento del enemigo y será el servidor quien se encargue de cambiarlo.

- Bloque central: En él se encuentra el propio tablero del juego, este irá cambiándose de manera dinámica a medida que avance el juego. Es en esta parte donde se inicializa nuestro objeto canvas.

```
<canvas id="canvas" width="601" height="601" class="contenedor"></canvas>
```

- Bloque derecho: En este bloque se muestran los puntos y el nivel del jugador cuyos resultados también se obtienen a través de una función del cliente en JavaScript. Justo debajo de esto, aparecen las distintas alternativas que se le ofrece al usuario para cambiar de vista de jugador. Se ha realizado mediante `radio buttons`, acompañado cada uno de ellos por la imagen de la vista que aparecería al seleccionarlo.

```
<input id="naranja" type="radio" name="forma" value="pato_lucas"
onclick="inputPulsado(this)">

<img height="85" width="85" src="img/pato_lucas.png">
```

2. **Eventos:** Es una de las partes más importantes de la aplicación junto con la del servidor. Ya que a través de los eventos que llegan por el usuario o en forma de mensaje por el servidor, el juego avanza de forma dinámica y reacciona de forma diferente ante estos eventos. Esta sección está desarrollada en la parte del código que se encuentra en los archivos: `tablero.js` y `webSocket.js`

Eventos recibidos por el servidor:

Los eventos recibidos por parte del servidor son recogidos en por el método `onMessage(msg)` alojado en el archivo `webSocket.js` Lo que hace principalmente este script es crear el WebSocket que apunta al endpoint creado en la clase de java `MiTableroDeJuego` y definir varias funciones:

- `Function onEerror (evt)`: Esta función es llamada si se produce algún tipo de error, en cuyo caso se muestra por consola el mensaje de error que le llega como atributo.
- `Function sendText (msg)`: Esta función es invocada por parte del cliente y su finalidad es enviar los JSON para el endpoint (servidor).
- `Function onMessage (msg)`: A esta función llegarán los datos que el servidor le envíe al cliente. Y es esta la encargada de decidir qué hacer con ellos. En el caso de este proyecto, este método le pasa los datos recibidos a otra función (drawImage) del cliente, encargada de dibujar las figuras en el tablero de juego de acuerdo a los datos recibidos.

Eventos recibidos por el usuario:

Los eventos recibidos por parte del usuario son recogidos por un método (alojado en `tablero.js`) que permanece en constante escucha para captar los posibles eventos que pueden llegar por parte del usuario a través del teclado. Esta función es la siguiente:

```
window.addEventListener("keydown", mover, true);
```

Cuando se pulsa una tecla esta función automáticamente se activa y pasa el valor de la tecla pulsada a la función definida como `mover`. Esta función pasa el valor de la tecla a otra función dedicada al movimiento del jugador, en esta se compara el valor recibido con los valores de cuatro teclas preestablecidas como las utilizadas para el movimiento (las flechas de subida, bajada, derecha e izquierda). Si la tecla pulsada es alguna de esas cuatro, el jugador se moverá en la posición indicada.

Una vez se ha desplazado, el cliente debe mandar un mensaje al servidor informando de la nueva posición. Para que el servidor sepa qué dato está recibiendo, el cliente envía la información de todo el personaje en formato JSON, el cual tendría los siguientes campos:

```
{
    "tipo": "Figure",
    "x":1,
    "y":501,
    "forma": "conejo",
    "puntos":0,
    "otro":0,  //indica el nivel del jugador
    "id":"1"
};
```

El campo “`tipo`” contiene el tipo de elemento que se envía al servidor, en este trabajo siempre será de tipo Figure. Los campo “`x`” e “`y`” indican la posición de la figura, el campo “`forma`” está asociado a la vista del elemento, el campo “`puntos`” es necesario para ir contabilizando los puntos del jugador y únicamente se utiliza para esta figura. Por otro lado existe un campo “`otro`” que en el jugador indica el nivel del juego en el que se encuentra, y en el enemigo muestra el tipo de movimiento de persecución que está seleccionado. Por último el campo “`id`” es el que diferencia los objetos JSON que se envían y reciben, puede tratarse del jugador, enemigo, vida u obstáculo.

Cliente y servidor tienen que contener toda la información de las figuras del juego, necesarias para mantener ambos extremos sincronizados y que ambos posean toda la información actualizada.

Además de esta función correspondiente al movimiento manual del jugador, existen otras que cambian el tipo de movimiento, su vista, otras generan nuevas vidas y una muy importante que pinta el tablero de juego cuando los elementos que se ubican en él cambian. El concepto de funcionamiento general de estas funciones es el siguiente:

- Cambio de movimiento:

En la parte html se asocian los botones del movimiento del jugador a una función llamada modo_auto(), a la cual se le pasa como argumento true/false y en función de estos valores el cliente javascript seleccionará un tipo de movimiento diferente (manual o automático):

```
<button id="button1" onclick="modo_auto(true)">Automático</button>
<button id="button2" onclick=" modo_auto(false)">Manual</button>
```

Se hace uso de la variable auto la cual almacena el valor recibido a través de la función de cambio de movimiento. Esta variable inicialmente toma el valor false, por lo que si ningún botón es pulsado el jugador se mueve de forma manual. Al seleccionar el botón de movimiento automático la variable auto cambia a true.

Cada vez que el usuario pulsa una tecla, para mover al jugador, se comprueba el valor de esta variable, y en función de si su valor es true o false se selecciona una función u otra.

```
var auto=false;

function modo_auto(decision){
    auto = decision;
}
function mover(evt){
    if(auto===false)
        moverFig(evt.keyCode);
    else
        moverFig_auto(evt.keyCode);
}
```

Estas funciones (moverFig() y moverFig_auto()) comprueban el código de la tecla pulsada y varían la posición del jugador en esa dirección. Cuando el movimiento es manual la posición cambia una vez por cada tecla pulsada, y si el movimiento es automático se modifica la posición del jugador en esa dirección cada cierto tiempo, previamente preestablecido, mientras que no se pulse otra tecla.

Por otra parte, además de estas acciones se comprueba si el jugador ha cogido una vida o si el usuario ha seleccionado un cambio de vista del jugador. Una vez hecho esto, envía de nuevo al servidor todos los objetos.

- Cambio de Vistas:

Tal y como se ha explicado dentro de este punto (apartado de apariencia), en el bloque derecho del código html se definen las diferentes vistas que el usuario puede elegir. El cliente javaScript comprueba cual vista está seleccionada y obtiene el valor asociado a esa vista.

```
function forma_jugador(){
    for (i = 0; i < document.inputForm.forma.length; i++) {
        if (document.inputForm.forma[i].checked) {
            var forma = document.inputForm.forma[i];
            return forma.value;
            break;
        }
    }
}
```

En las funciones de movimiento se obtiene la vista que está seleccionada y se modifica el campo "forma" del JSON del jugador. Cuando se pinta el tablero con sus elementos, se dibujará la vista asociada a ese valor "forma".

- Generación de nuevas vidas:

Es necesario comprobar si el jugador o el enemigo atrapan vidas. Si es así la vida cogida ha de desaparecer creándose otra en una nueva posición. Además, esta posición será diferente a la posición en la que se encuentren los demás elementos del tablero, es decir, las coordenadas no podrán ser las mismas a las del obstáculo o las que tengan en ese momento el enemigo y el jugador

- Pintar el tablero:

Inicialmente, cuando el juego empieza, se dibuja una cuadrícula, el tablero de juego, y en él se dibujan el enemigo, el jugador, la vida y el obstáculo. Los valores iniciales de cada objeto están definidos tanto en el cliente como en el servidor e irán variando a lo largo del juego. El cliente por su parte cada vez que realiza algún cambio manda la información al servidor, este la procesa y se la vuelve a enviar al cliente, recibiéndola en la función `onMessage(msg)`, la cual se la pasa a `drawImage(msg)` encargada de pintar de nuevo todo el tablero. El argumento `msg` que le llega a ambas funciones es la información de los objetos JSON.

El servidor por su parte cuando realiza cambios sobre algún objeto los envía al cliente para que este los dibuje a través de las mismas funciones explicadas anteriormente y para que actualice los campos de estos objetos.

Es necesario saber, que antes de dibujar el nuevo tablero borra el actual y vuelve a pintarlo todo, los cambios son tan rápidos que no es perceptible.

Ambas funciones descritas con anterioridad se han desarrollado de la siguiente forma:

```
/* onMessage(msg) */
function onMessage(msg) {
    drawImage(msg.data);
}

/* drawImage(msg) */
function drawImage(data){
    tablero.clearRect(0,0,canvas.width,canvas.height); // borra el tablero
```

```
    dibujarTablero(); //función que dibuja el nuevo tablero

    var json = JSON.parse(data); //convierte los datos recibidos a JSON

    /* asocia cada JSON con la figura a la que se refiere */
    figura = json[0];
    obstaculo = json [1];
    vida = json [2];
    enemigo = json[3];


    var img = new Image;

    /* pinta en el tablero cada elemento */
    var paso;
    for (paso = 0; paso < json.length; paso++) {
        var id = json[paso].id;
        var forma_fig = json[paso].forma;

    switch(id)
        {
        case "1":
            if (forma_fig==="personaje1"){
            img.src = "img/Bugs_Bunny.png";
            tablero.drawImage(img, json[paso].x, json[paso].y, 99, 99);
        }
        else if (forma_fig==="personaje2"){
            img.src = "img/silvestre.png";
            tablero.drawImage(img, json[paso].x, json[paso].y, 99, 99);
        }
        else if (forma_fig==="personaje3"){
            img.src = "img/tweety.png";
            tablero.drawImage(img, json[paso].x, json[paso].y, 99, 99);
        }
        else if (forma_fig==="personaje4"){
            img.src = "img/pato_lucas.png";
            tablero.drawImage(img, json[paso].x, json[paso].y, 99, 99);
        }
        break;

        case "2":
            img.src = "img/piedra.png";
            tablero.drawImage(img, json[paso].x, json[paso].y, 99, 99);
            break;

        case "3":
        img.src = "img/demonio.png";
        tablero.drawImage(img, json[paso].x, json[paso].y, 99, 99);
        break;

    case "4":
            img.src = "img/vida.png";
            tablero.drawImage(img, json[paso].x, json[paso].y, 99, 99);
            break;

    default:
            break;

    }
    }
}
```

Servidor

El servidor es el encargado de realizar una de las partes más importante de la aplicación, pues es el responsable de controlar los eventos procedentes del cliente, procesarlos, modificarlos si es necesario y propagar los cambios a todos componentes de una sesión, haciendo un envío broadcast a todos ellos. Además de esto, es en el servidor donde se lleva a cabo la lógica del enemigo, la cual es diferente dependiendo del tipo de movimiento seleccionado para este.

A continuación se va a describir tanto la funcionalidad que presta el servidor como el concepto de sesión, necesario para entender de mejor manera este apartado:

1. Sesión.

Se conoce como la duración de una conexión en un determinado sistema o red. En ella se intercambia información entre un cliente y un servidor. Es habitual que el usuario deba ingresar un nombre de usuario y contraseña para iniciar una sesión, en un procedimiento conocido como login.

Sabemos que una vez hecho esto la aplicación crea una sesión, a la cual pueden unirse diferentes compañeros o usuarios que hayan hecho login con los mismos datos pero desde ubicaciones distintas. A estos usuarios se les denomina compañeros, ya que todos forman parte de una misma sesión.

Para esta aplicación, a una misma sesión pueden estar vinculados varios compañeros, de tal manera que los cambios que realice uno de ellos serán visibles por los demás, pudiendo todos intervenir en cualquier momento a lo largo del transcurso del juego.

2. Recepción de eventos y envío broadcast.

El intercambio de mensajes entre cliente y servidor mediante la tecnología WebSocket es la principal tarea que mantienen ambos. Cuando el cliente envía un mensaje al servidor, lo hace a su punto final (endpoint). El servidor contiene un método etiquetado como @onMessage, el cual se invoca cada vez que recibe un mensaje WebSocket del cliente. Antes de que el servidor opere con los datos recibidos es necesario que se decodifiquen los mensajes y se transformen a objetos de la clase Figure, ya que este tipo son los utilizados por el servidor.

Una vez que el servidor ha decodificado los datos y después de haber operado con ellos, se los vuelve a enviar al cliente pero antes es necesario codificarlos como cadenas JSON. Los datos se envían al cliente en el mismo formato en el que son recibidos, el servidor únicamente los transforma para poder utilizarlos.

Cuando el servidor envía los datos al cliente lo hace en broadcast, es decir, manda los datos a todos los miembros de una misma sesión, independientemente de qué miembro de la sesión se los mandó. De esta forma todos los miembros poseen la información actualizada del juego.

El siguiente código muestra el método en el cual el cliente recibe el mensaje del servidor y lo envía de vuelta a todos los miembros de una misma sesión.

```java
private static Figure fig[] = new Figure[4];
@OnMessage
    public void broadcastFigure(Figure f[], Session session) throws IOException,
EncodeException {
        fig = f;
        JSONArray jArray = new JSONArray();

        for (int i = 0; i < f.length; i++) {
            jArray.put(fig[i].toJSONObject());
        }
        s = session;
        s.getBasicRemote().sendObject(jArray);
        for (Session peer : peers) {
            if (!peer.equals(s)) {
                peer.getBasicRemote().sendObject(jArray);
            }
        }
    }
```

3. Lógica del enemigo. Factoría e hilos.

La lógica del enemigo en el tablero de juego se desarrolla en la parte del servidor. Para ello se ha creado un hilo que se ejecuta periódicamente, tiene por función modificar la posición del enemigo.

Al inicio del juego se ha definido un timer de 1.5 segundos, el cual irá disminuyendo conforme el usuario sube de nivel, por lo tanto cada vez que el nivel del juego aumenta, el tiempo de ejecución del hilo disminuirá y el enemigo se moverá con mayor rapidez.

Tal y como se ha comentado en apartados anteriores, el usuario puede elegir el tipo de movimiento del enemigo, aportando unos mayor dificultad que otros a la partida. Es el cliente el que le proporciona la información a comprobar, el tipo de movimiento seleccionado, y a través de una factoría se escoge la clase que modificará la posición del enemigo, siguiendo cada uno unos criterios de funcionamiento diferentes.

En el servidor se ha creado una nueva clase llama `Hilos.java`, la cual es llamada cuando se abre una nueva sesión en la clase `MiTableroDeJuego.java`. Se muestra a continuación:

```java
@OnOpen
    public void onOpen (Session peer) {
        peers.add(peer);
        if(peers.size()==1){
            Hilos mihilo = new Hilos(fig, peer, peers);
        }
    }
```

Como se puede apreciar se crea un único hilo, que será compartido por todos los miembros o compañeros de una misma sesión. Este hilo tiene como argumentos toda la información de las figuras, el identificador de la sesión y el conjunto de compañeros que forman parte de una misma sesión.

Cada vez que haya vencido el timer se ejecuta un método llamado `run()`, dentro se comprueba el tipo de movimiento seleccionado y en función de este se selecciona una clase u otra de la factoría, modificándose en ella la posición del enemigo. Una vez hecho esto se propagan los cambios a todos los componentes de la sesión. Este método está implementado de la siguiente manera:

```
public void run(){
       while(session.isOpen()){

          /* Escogemos el array de figuras actualizado */
          fig_mod = tj.getArrayFigure();

          if(fig_mod != fig_old){
             fig = fig_mod;
             fig_old = fig_mod;
         } /*  */

          /* Factory Method: Movimiento del enemigo */
          Movimiento mov = null;
          if (fig[3].otro==1){
               mov = TiposMovimientos.CrearMovimiento("lineal");
          }
          else if(fig[3].otro==2){
              mov = TiposMovimientos.CrearMovimiento("aleatorio");
          }
          else if(fig[3].otro == 3){
              mov = TiposMovimientos.CrearMovimiento("inteligente");
          }

          fig = mov.obtenerMovimientoFigura(fig);
          /*  */


          /* Empaquetamos todas las figuras en un Array de JSON */
          JSONArray jArray = new JSONArray();

          for (Figure fig1 : fig) {
               jArray.put(fig1.toJSONObject());
           }
           /*  */

          try {
              sendCoord(jArray); //Propaga los cambios a toda la sesión
              t.sleep(timer - 50*fig[0].otro); //Decrementa el timer

               if((fig[0].otro==17) || (fig[3].x == fig[0].x && fig[3].y ==
fig[0].y) || (fig[0].puntos<0)){ //Comprueba que no haya acabado la partida
              System.exit(0);

                  }
```

```
            }

            catch(IOException | EncodeException | InterruptedException e) {
                try {
                    session.close();
                } catch (IOException e1) {
                    System.out.println("Excepción al cerrar sesión");

                }
        }
    }
}
```

El usuario tiene la opción de elegir tres formas distintas de desplazamiento del enemigo en el tablero:

- Lineal: Es previsible por el jugador, ya que sigue un patrón fijo. Se ha generado un método que va sumando posiciones mientras no choque con un obstáculo o con el límite del tablero, en ese caso cambia la dirección y continúa. Este movimiento facilita la partida al usuario.
- Aleatoria: Su movimiento es aleatorio, es decir, es imprevisible para el jugador. Para realizarlo, generamos dos números aleatorios, uno indica la dirección a seguir, y el otro el número de veces que se mueve en dicha dirección. Proporciona una complejidad media a la partida.
- Inteligente: Es la parte del juego que contiene inteligencia artificial, ya que el enemigo persigue al jugador y a las vidas. Este aporta una mayor complejidad al juego. Para su desarrollo se han seguido los siguientes pasos:

    1. Calcular la distancia enemigo-jugador y enemigo-vida.
    2. Seleccionar como objetivo el de menor distancia.
    3. De entre todas las direcciones de movimiento descartar las imposibles, por ejemplo si limita con un obstáculo o con el propio tablero.
    4. Coger las direcciones posibles y volver a calcular la distancia que abría enemigo-objetivo desde ese punto. Seleccionar la dirección que se acerque más al objetivo.

5. Construcción de un Juego mínimo.

A continuación se explicará en detalle los diferentes pasos a seguir para la realización de un juego básico utilizando WebSocket y Canvas principalmente, y haciendo uso de los distintos tipos de lenguajes de programación descritos en apartados anteriores.

5.1 Pasos a seguir

Para obtener como resultado el juego web que tenemos actualmente se comenzó realizando un juego sencillo y sobre él se han ido añadiendo más funcionalidades y características. Explicar paso a paso cómo se ha creado la aplicación en su totalidad sería largo y se podría perder el objetivo principal de este trabajo, conocer la realización de un juego con tecnología WebSocket. Por lo tanto, para explicar de forma clara el funcionamiento y realización se va a utilizar el juego básico inicial pues aporta los conceptos necesarios. Por este motivo se considera, además, que podría ser de gran interés en uso docentes.

La aplicación creada en este apartado va a servir de guía para iniciar a los alumnos en las tecnologías WebSocket, en la creación de aplicaciones con canvas y en su utilización.

La presente aplicación contiene algunos archivos JavaScript que se ejecutan en el navegador del cliente cuando se carga la página o cuando se invoca desde un formulario en la página web, una clase `WebSocket endpoint`[17] que maneja conversaciones WebSocket, una case `decoder` y `encoder` que codifica y decodifica interfaces, y una página web sobre la que se muestran los cambios.

Vamos a seguir una serie de pasos en la creación de la aplicación web:

- <u>Creación del proyecto de la aplicación Web</u>

El objetivo de este ejercicio es crear el proyecto de la aplicación web en NetBeans. Al crear el proyecto elegimos Java EE 7 Web como versión de Java EE, y GlassFish 4.1 como servidor de aplicaciones.

1. Seleccionar File > New Project del menú principal.
2. Seleccionar la categoría Maven y Web Application de la lista de proyectos. Hacer clic en Next.
3. Escribir `JuegoMinimo` en el nombre del proyecto y establecer la ubicación donde queremos guardar el proyecto. Hacer click en Next.
4. Seleccionar GlassFish Server 4.1 para el servidor
5. Establecer la versión de Java EE a Java EE 7 Web. Hacer click en Finalizar.

Al hacer click en Finalizar, el IDE crea el proyecto y lo abre en la ventana de proyectos.

---

[17] Endpoint: el punto de entrada a un servicio, un proceso o un destino en la arquitectura orientada a servicios.

Es necesario crear una nueva carpeta en la cual se guardan las imágenes de los distintos jugadores y elementos del juego.

1. Seleccionar la carpeta Web Pages. Pulsar botón derecho y seleccionar New Folder.
2. Escribir `img` como nombre de la carpeta. Hacer clic en finish
3. Será en esta carpeta donde se almacenarán todas las imágenes, ya bien sea arrastrándolas a esta carpeta o a través del propio directorio donde está guardado el proyecto en el pc.

- Creación del endpoint WebSocket

En esta sección se creará una clase endpoint y un archivo JavaScript. La clase endpoint contiene algunos métodos básicos que se ejecutan cuando se abre una sesión. En archivo JavaScript crearemos el WebSockt que se iniciará cuando se cargue la página.

Clase endpoint.

1. Hacer clic derecho en el Source Packages de nuestro proyecto y seleccionar New > Other
2. Seleccionar el tipo de archivo WebSocket Endpoint de la categoría Web. Hacer clic en Next.
3. Escribir `MiTableroDeJuego` como nombre de la clase
4. Seleccionar `org.sample.juegominimo` en la lista desplegable del paquete
5. En WebSocket URI añadir: `/endpoint`. Hacer clic en Finish

Al hacer clic en Finalizar el IDE genera la clase WebSocket endpoint y abre el archivo en el editor del código fuente. En el editor se puede ver que el IDE genera algunas anotaciones que forman parte de la API WebSocket. La clase se anota con `@ServerEndpoint` para identificar la clase como punto final y la URI WebSocket se especifica como un parámetro de la anotación. El IDE también genera un método `onMessage` predeterminado que se anota con `@OnMessage`. Un método anotado con `@OnMessage` se invoca cada vez que recibe un mensaje WebSocket del cliente.

6. Agregamos el código que aparece en negrita:

```
@ServerEndpoint ("/endpoint")
 MiTableroDeJuego public class {
                private static Set<Session> peers = Collections.synchronizedSet(new
                                                        HashSet<Session>());

    @OnMessage
    public String onMessage(String mensaje) {
        return null;
    }

    @OnOpen
    public void onOpen (Session peer) {
        peers.add(peer);
    }
```

```
    @OnClose
    public void onClose (Session peer) {
        peers.remove(peer);
    }
}
```

Se puede ver que los métodos `onOpen` y `onClose` están anotados con `@OnOpen` y `@OnClose`. Un método anotado con `@OnOpen` se llama cuando se abre la sesión websocket. En este ejemplo, el método de `onOpen` añade componentes al grupo de la sesión actual y el método `onClose` los elimina del grupo.

7. Añada las librerías que sugiere el editor y guarde los cambios. Las declaraciones de importación para las clases en javax.websocket y java.util se añaden al archivo.

Archivo WebSocket e inicio de sesión

Se crea un archivo JavaScript donde se iniciará la sesión WebSocket. El cliente se une a una sesión HTTP a través de un establecimiento de sesión con el servidor a través de TCP. En el archivo JavaScript se especifica el nombre del endpoint y se declara el WebSocket.

1. Hacer clic sobre el proyecto en la ventana de Proyectos y seleccione New> Other
2. Seleccionar un archivo JavaScript en la categoría Web. Hacer clic en Next
3. Llamamos al archivo `webSocket`. Hacer clic en Finish
4. Añadir lo siguiente al archivo JavaScript

```
    var wsUri = "ws://" + document.location.host +
                          document.location.pathname +
                          "endpoint";

   var websocket = new WebSocket(wsUri);
   console.log("URI: " + websocket);

   websocket.onerror = function(evt) { onError(evt); };

   function onError(evt) {
console.log("onError text: " + evt.data);
}

   websocket.onmessage = function(evt) { onMessage(evt); };

   function sendText(msg) {
      console.log("sending text: " + msg);
websocket.send(msg);
}

   function onMessage(msg) {
      console.log("received:" + msg.data);
      drawImage(msg.data);

}
```

Lo que hace principalmente este script es crear el WebSocket que apunta al endpoint creado en la clase de java `MiTableroDeJuego` y definir varias funciones:

- Function `onEerror (evt)`: Esta función es llamada si se produce algún tipo de error, en cuyo caso se muestra por consola el mensaje de error que le llega como atributo.
- Function `sendText (msg)`: Esta función es invocada por parte del cliente y su finalidad es enviar los JSON para el endpoint (servidor).
- Function `onMessage (msg)`: A esta función llegarán los datos que el servidor le envíe al cliente. Y es esta la encargada de decidir qué hacer con ellos. para la elaboración de la imagen cuando se recibe un mensaje desde el punto final.

Este script iniciará la sesión de comunicación con el servidor cuando esté cargado por el navegador.

5. Abrir el `index.html` que se ha creado con el proyecto, y añadir el siguiente código. Este incluye el script `websocket.js` que se cargará al terminar de cargar la página

```
<body>
   <h1>Juego</h1>
   <script src="webSocket.js" type="text/javascript"> </script>
</body>
```

Antes de añadirle más funcionalidad al juego debemos comprobar que el endpoint está trabajando correctamente, que se ha iniciado la sesión y el cliente se añadirá a la sesión. Una vez verificado este punto podremos añadir funcionalidad al juego.

Prueba del endpoint

Para comprobar el buen funcionamiento del endpoint es necesario añadir algunos métodos simples en el archivo websocket.js. Estos métodos imprimirán el wsURI en la ventana del navegador cuando este se conecta con el punto final.

1. Añadir al `index.html` la siguiente línea en negrita

```
<body>
   <h1>Juego</h1>
   <div id="canvas"></div>
   <script src="webSocket.js" type="text/javascript"></script>
</body>
```

2. Añadir temporalmente el siguiente código a `webSocket.js`

```
    // test functions
   var canvas = document.getElementById("canvas");
   websocket.onopen = function(evt) { onOpen(evt); };
```

```
   function writeToScreen(message) {
  canvas.innerHTML += message + "<br>";
}

function onOpen() {
    writeToScreen("Connected to " + wsUri);
}
// End test functions
```

Cuando la página carga las funciones del JavaScript se imprimirá el mensaje de que el navegador se conecta al punto final. Debe eliminar el código anterior después de confirmar que está funcionando correctamente.

3. Haga clic derecho sobre el proyecto y seleccione Run. Si todo ha ido correctamente deberás visualizar el contenido de la Figura 20 en el navegador, el cual indica el punto final donde se aceptan los mensajes:

Juego

Connected to ws://localhost:8080/JuegoMinimo/endpoint

Figura 20: Juego mínimo. Prueba de conexión

- Creación del juego

En este apartado creamos las clases y los archivos JavaScript para enviar y recibir mensajes JSON. Se añadirá un elemento Canvas de HTML5 para pintar el tablero del juego y representar todos los elementos que intervienen.

Página Web

1. Abre el archivo index.html
2. Se agrega el elemento canvas `<canvas>`, y añadimos el siguiente `<table>` para añadir botones de selección de diferentes jugadores. El código de este archivo quedaría de la siguiente forma:

```
<html>
    <head>
        <title>Juego</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

    </head>
    <body>

            <h1 >Juego Minimo</h1>
```

```
                    <canvas id="canvas" width="401" height="401"></canvas>

            <script src="webSocket.js" type="text/javascript"></script>

    </body>
</html>
```

En la tabla creada, cada botón lleva una imagen asociada para que el usuario visualice la opción de cambio de vista del jugador. Además también lleva una función asociada, necesaria, para quitarle el foco[18] al botón y que los cambios sean persistentes.

Tanto la forma, coordenadas de cualquier figura, identificador, etc, se convierte en una cadena con una estructura JSON y se envían como un mensaje al punto final WebSocket.

Clase Figure

Se define esta clase para establecer el tipo de objeto que vamos a recibir por parte del cliente.

1. Hacer clic derecho sobre el proyecto y elegir New > Java Class
2. Establecer `Figure` como nombre de la clase y elegir el paquete `org.sample.juegominimo`. Hacer clic en Finalizar.
3. Cuando se abra la clase, escribir dentro de ella:

```
public class Figure {
private JSONObject json;
}
```

4. Al escribir esta línea la clase se obtiene un error debido a que es necesario importar las dependencias y librerías de JSON.
5. Añadir un método `toString()` que convierte un objeto JSON a String, un método `toJSONObject()` que convierte un objeto Figure a JSON, un constructor `Figure(JSONObject json)`, y algunas variables necesarias:

```
    int x,y,puntos,nivel;
    String color, tipo, id;

    public Figure(JSONObject json) {
```

[18] Foco (focus): Hace referencia a posicionamiento sobre un elemento de una página web. Esto, por ejemplo, nos puede ser muy útil para posicionarnos en un campo concreto de un formulario, no solo nos sirve para hacer foco en campos de formulario. También podemos hacer foco en otros elementos seleccionables de una página como podría ser el caso de un enlace, un botón, etc.El método que nos sirve para hacer foco es focus().

```
        tipo = json.getString("tipo");

        if (tipo.compareTo("Figure") != 0){
            throw new JSONException("Figure.Constructor.No es una figure");
        }
        x = json.getInt("x");
        y = json.getInt("y");
        color = json.getString("color");
        puntos = json.getInt("puntos");
        nivel = json.getInt("nivel");
        id = json.getString("id");
    }


    public String toString() {
        return toJSONObject().toString();
    }

    public JSONObject toJSONObject() {
        JSONObject jObj = new JSONObject();

        jObj.put("tipo", "Figure");
         jObj.put("x", x);
         jObj.put("y", y);
         jObj.put("color", color);
         jObj.put("puntos", puntos);
         jObj.put("nivel", nivel);
         jObj.put("id", id);
        return jObj;
    }
```

Clase Hilos

Se crea esta clase para que el servidor envíe mensajes periódicos al cliente modificando uno de los JSON que obtiene. Se realiza a través de un hilo java, que ejecuta un método de forma regular, de acuerdo a un timer.

1. Hacer clic en el nodo del proyecto. New> Java class
2. Escribir `Hilos` como el nombre de la clase y elegir `org.sample.juego` en la lista desplegable del paquete. Hacer clic en Finish
3. En el editor de código fuente, hacer que la clase `Hilos` implemente a `Runnable` y agrega los métodos abstractos sugeridos:

```
    public class Hilos implements Runnable{
    @Override
    public void run() {
        throw new UnsupportedOperationException("Not supported yet.");
    }
  }
```

4. Completar esta función para que modifique un elemento JSON y lo envíe al servidor cada segundo. Se crea además un método que envía el mensaje a todos los miembros de una sesión.

```
public class Hilos implements Runnable {

    private static Set<Session> peers_p = Collections.synchronizedSet(new
HashSet<Session>());
    private static Figure [] fig_mod,fig_old, fig = new Figure [4];
    private static Session session;
    Thread t;
    MiTableroDeJuego tj = new MiTableroDeJuego();
    int tiempo = 800;

    public Hilos(Figure [] figure, Session s, Set<Session> peers ){
        fig = figure;
        fig_old = tj.getArrayFigure();
        session = s;
        peers_p = peers;
        t = new Thread(this,"miHilo");
        t.start();
    }
    public void run(){
       while(session.isOpen()){
          fig_mod = tj.getArrayFigure();

          if(fig_mod != fig_old){
             fig = fig_mod;
             fig_old = fig_mod;
         }
           //Movimiento que bordea tablero
           if(fig[3].x==1 && (fig[3].y>=1 && fig[3].y<361) ){
                 fig[3].y +=40;
                 fig[3].x=1;
            }
```

```
            else if((fig[3].x>=1 && fig[3].x<361) && fig[3].y==361){
                 fig[3].x += 40;
                 fig[3].y=361;
            }
            else if(fig[3].x==361 && (fig[3].y>1 && fig[3].y<=361)){
                 fig[3].y -= 40;
                 fig[3].x = 361;
            }
            else if(fig[3].y==1 && (fig[3].x>1 && fig[3].x<=361)){
                 fig[3].y = 1;
                 fig[3].x -=40;
            }
           JSONArray jArray = new JSONArray();

            for (int i = 0; i < fig.length; i++){
                 jArray.put(fig[i].toJSONObject());
            }

            try {
                    sendCoord(jArray);
                    t.sleep(tiempo);
            }
            catch(Exception e) {
                try {
                    session.close();
                } catch (IOException e1) {
                    System.out.println("Exception");
                }
            }
        }
    }

    private void sendCoord(JSONArray jArray) throws IOException, EncodeException{
         session.getBasicRemote().sendObject(jArray);
            for (Session peer : peers_p) {
                if (!peer.equals(session)) {
                    peer.getBasicRemote().sendObject(jArray);
                }
            }
     }
}
```

5. Añadimos todas las librerías sugeridas para importar.
6. Obtendremos un error en el método: `getArrayFigure`, ya que aún no está creado en la clase `MiTableroDeJuego`.java. Una vez creado, este error desaparecerá.

Generar JSON

En este ejercicio se creará un archivo JavaScript que utiliza los distintos campos del objeto Figure, utilizando una estructura JSON, para dibujar en Canvas y enviar la información al punto final WebSocket (servidor).

1. Hacer clic sobre el proyecto en la ventana de Proyectos y seleccione New> JavaScript File
2. Como nombre del archivo poner `Tablero`. Hacer clic en Finish
3. Añadir el siguiente código para inicializar el elemento canvas, añadir un evento listener, declarar las figuras y dibujarlas.

```
var canvas = document.getElementById("canvas");
var tablero = canvas.getContext("2d");
window.addEventListener("keydown", mover, true);

var figura = {
    "tipo":"Figure",
    "x":41,
    "y":41,
    "color":"amarillo",
    "puntos":0,
    "nivel":0,
    "id":"1"};

var obstaculo = {
    "tipo":"Figure",
    "x":241,
    "y":241,
    "color":"rojo",
    "puntos":0,
    "nivel":0,
    "id":"2"};

var enemigo = {
    "tipo":"Figure",
    "x":1,
    "y":1,
    "color":"azul",
```

```
        "puntos":0,
        "nivel":0,
        "id":"3"};

    var vida ={
       "tipo":"Figure",
        "x":81,
        "y":41,
        "color":"vida",
        "puntos":0,
        "nivel":0,
        "id":"4"};

        var img = new Image;
        img.src = "img/jugador.png";
        tablero.drawImage(img, figura.x, figura.y, 39, 39);

        var img2 = new Image;
        img2.src = "img/obstaculo.png";
        tablero.drawImage(img2, obstaculo.x, obstaculo.y, 39, 39);

        var img3 = new Image;
        img3.src = "img/enemigo.jpg";
        tablero.drawImage(img3, enemigo.x, enemigo.y, 39, 39);

        var img4 = new Image;
        img4.src = "img/vida.png";
        tablero.drawImage(img4, vida.x, vida.y, 39, 39);

    var coorXY = [1, 41, 81, 121, 161, 201, 241, 281, 321, 361];
    dibujarTablero();
```

Se puede ver que el método `mover` se invoca cuando el usuario hace clic en el elemento `canvas`.

4. Añadir los siguientes métodos para la construcción del JSON y enviarlo al endpoint.

```
     /* MÉTODOS */
    function dibujarTablero(){
        for (var x = 0.5; x < 401; x += 40) {
          tablero.moveTo(x, 0);
```

```
            tablero.lineTo(x, 401);
            }
          for (var y = 0.5; y < 401; y += 40) {
            tablero.moveTo(0, y);
            tablero.lineTo(401, y);
            }
          tablero.strokeStyle = "#000";
          tablero.stroke();
      }

      function coge_vida(){
          if (figura.x===vida.x && figura.y === vida.y){
              genera_vida();
          }
      }

      function genera_vida(){
         var generaCoor = elegirXY();
         var x = generaCoor[0];
         var y = generaCoor[1];
         vida = {
         "tipo":"Figure",
          "x":coorXY[x],
          "y":coorXY[y],
          "color":"vida",
          "puntos":0,
          "nivel":0,
          "id":"4"};
      }

      function fig_to_string(f){
          return JSON.stringify({
          "tipo":f.tipo,
          "x":f.x,
          "y":f.y,
          "color":f.color,
          "puntos":f.puntos,
          "nivel":f.nivel,
          "id":f.id});
      }
```

```
function enviarMensaje(){
    var evt_f = fig_to_string(figura);
    var evt_o = fig_to_string(obstaculo);
    var evt_v = fig_to_string(vida);
    var evt_e = fig_to_string(enemigo);
    var msg = "[" + evt_f + "; " + evt_o +  "; " + evt_v + "; " + evt_e + "]";
    sendText(msg);
}

function mover(tecla){
   tecla = tecla.keyCode:
        if(tecla===39){
           figura.x += 40;
           if(figura.x===obstaculo.x && figura.y===obstaculo.y)
               figura.x -= 40;
           if(figura.x > 361)
               figura.x = 361;
       }
        else if(tecla===37){
           figura.x -= 40;
           if(figura.x===obstaculo.x && figura.y===obstaculo.y)
               figura.x += 40;
           if(figura.x < 1)
               figura.x = 1;
       }
        else if(tecla===38){
           figura.y -= 40;
           if(figura.x===obstaculo.x && figura.y===obstaculo.y)
               figura.y += 40;
           if(figura.y < 1)
               figura.y = 1;
       }
        else if(tecla===40){
           figura.y +=40;
           if(figura.x===obstaculo.x && figura.y===obstaculo.y)
               figura.y -= 40;
           if(figura.y > 361)
               figura.y = 361;
       }
    coge_vida();
    enviarMensaje();
}
```

```
function drawImage(data){
    tablero.clearRect(0,0,canvas.width,canvas.height);
    dibujarTablero();

    var json = JSON.parse(data);

    figura = json[0];
    obstaculo = json [1];
    vida = json [2];
    enemigo = json[3];

    if ((json[3].x === json[0].x && json[3].y === json[0].y) || figura.puntos <
0)
           {
               alert("Has perdido :( ");
           }

    if (json[3].x === json[2].x && json[3].y === json[2].y)
           {
               figura.puntos -=2;
               genera_vida();
               json[2] = vida;
               enviarMensaje();
           }

    var img = new Image;
    var paso;
    for (paso = 0; paso < json.length; paso++) {
        var id = json[paso].id;

     switch(id) {
    case "1": {
         img.src = "img/indice.png";
         tablero.drawImage(img, json[paso].x, json[paso].y, 39, 39);
         break;
     }
     case "2": {
         img.src = "img/Stop.png";
         tablero.drawImage(img, json[paso].x, json[paso].y, 39, 39);
         break;
     }
```

```
          case "3": {
              img.src = "img/azul.jpg";
              tablero.drawImage(img, json[paso].x, json[paso].y, 39, 39);
              break;
          }
          case "4": {
                  img.src = "img/vida.png";
                  tablero.drawImage(img, json[paso].x, json[paso].y, 39, 39);
                  break;
              }
          default:
              break;
        }
     }
}
     function crear_escenario()
     {
         var aleatorio = Math.round(Math.random()*9);
         return aleatorio;


     }
     function elegirXY (){
         var x = crear_escenario();
         var y = crear_escenario();
         var xy = [x,y];
         while ((coorXY[x]===figura.x && coorXY[y]===figura.y) ||
     (coorXY[x]===obstaculo.x && coorXY[y]===obstaculo.y) || (coorXY[x]===enemigo.x
     && coorXY[y]===enemigo.y)){
             x = crear_escenario();
             y = crear_escenario();
             xy = [x,y];
         }
         return xy;


     }
```

5. Añadir la siguiente línea en negrita al archivo `index.html`

```
<script src="webSocket.js" type="text/javascript"></script>
<script src="Tablero.js" type="text/javascript"></script>
```

Interfaces de codificación y decodificación

En este ejercicio se crean dos clases para implementar interfaces codificadoras y decodificadoras. Tal y como su nombre indica, se utilizan para decodificar mensajes provenientes del cliente web en forma de JSON y transformarlos a objetos de la clase Figure. Del mismo modo se codificarán los tipo Figure como cadenas JSON para mandarlos al punto final.

7. Hacer clic en el nodo del proyecto. New> Java class
8. Escribir `FigureEncoder` como el nombre de la clase y elegir `org.sample.juegominimo` en la lista desplegable del paquete. Hacer clic en Finish
9. En el editor de código fuente, implementar la interfaz WebSocket del codificador añadiendo la siguiente línea de código:

```
    public class FigureEncoder implements Encoder.Text<Figure []> {
}
```

10. Importar `javax.websocket.Encoder` y añadir los métodos abstractos.
11. Modificar estos métodos abstractos para que queden de la siguiente forma:

```
 @Override
  public String encode(Figure [] figure) throws EncodeException {
      StringBuilder sb = new StringBuilder();
      sb.append("[");
      for (int i = 0; i < figure.length - 1; i++){
          sb.append(figure[i].toString()).append(";");
      }
      sb.append(figure[figure.length-1]).append("]");
      return sb.toString();
  }

  @Override
  public void init(EndpointConfig config) {
    System.out.println("init");
  }

  @Override
  public void destroy() {
      System.out.println("destroy");
  }
```

12. Hacer clic en el nodo del proyecto. New> Java class
13. Escribir `FigureDecoder` como el nombre de la clase y elegir `org.sample.juegominimo` en la lista desplegable del paquete. Hacer clic en Finish

14. En el editor de código fuente, implementar la interfaz WebSocket del decodificador añadiendo la siguiente línea de código:

```
    public class FigureDecoder implements Decoder.Text<Figure []> {
}
```

15. Importar javax.websocket.Dencoder y añadir los métodos abstractos.
16. Modificar estos métodos abstractos para que queden de la siguiente forma:

```
    @Override
    public Figure [] decode(String string) throws DecodeException {
        String SubString = string.substring(1,string.length());
        String[] fig_s = SubString.split(";");
        JSONObject[] jObject = new JSONObject[4];
        Figure [] Array_fig = new Figure[4];

        for (int i=0; i<fig_s.length; i++) {
            JSONObject jsonO = new JSONObject(fig_s[i]);
            jObject [i] = jsonO;
            Array_fig[i] = new Figure(jObject[i]);
        }
        return  Array_fig;
    }

    @Override
    public boolean willDecode(String string) {
        try {
           String SubString = string.substring(1,string.length());
           String[] fig_s = SubString.split(";");
           for (String fig_ : fig_s) {
                Json.createReader(new StringReader(fig_)).readObject();
            }
            return true;
        } catch (JSONException ex) {
            ex.printStackTrace();
            return false;
        }
    }

    @Override
    public void init(EndpointConfig ec) {
        System.out.println("init");
    }
```

```
    @Override
    public void destroy() {
        System.out.println("destroy");
    }
```

17. Añadir las importaciones necesarias que se indican.

Ejecutar la aplicación

Ahora la aplicación está casi lita para ejecutarse. En este ejercicio se modifica el punto final de la clase `MiTableroDeJuego.java` para especificar el codificador y decodificador de la cadena JSON y se añadirá un método para enviar la cadena JSON a los clientes conectados, cuando se recibe un mensaje. Además de esto, se inicializarán los elementos JSON y se creará un nuevo método que envía la información del enemigo a la clase `Hilos.java`

1. Abrir `MiTableroDeJuego.java`
2. Modificar la anotación `@ServerEndpoint` para especificar el codificador y decodificador del endopoint.

```
@ServerEndpoint(value="/endpoint", encoders = {FigureEncoder.class}, decoders =
{FigureDecoder.class})
```

3. Eliminar el método `@onMessage` que se genera por omisión
4. Agregue el siguiente método `broadcastFigure` y anotar el método con `@OnMessage`
5. Completar todos los métodos de la clase y añadir los demás métodos y variables para que quede de la siguiente manera:

```
    public class MiTableroDeJuego  {
    private static Set<Session> peers = Collections.synchronizedSet(new
HashSet<Session>());


    private static Figure fig [] = new Figure[4];
    private static JSONObject figura = new JSONObject();
    private static JSONObject obstaculo = new JSONObject();
    private static JSONObject vida = new JSONObject();
    private static  JSONObject enemigo = new JSONObject();
    private static Session s;


    @OnMessage
    public void broadcastFigure(Figure f [], Session session) throws IOException,
EncodeException{


        fig = f;
        System.out.println("broadcastFigure");
```

```
        JSONArray jArray = new JSONArray();

        for (int i = 0; i < f.length; i++){
             jArray.put(fig[i].toJSONObject());
        }

        s = session;
        s.getBasicRemote().sendObject(jArray);

        for (Session peer : peers) {
            if (!peer.equals(s)) {
                peer.getBasicRemote().sendObject(jArray);
            }
        }
    }

    @OnOpen
    public void onOpen (Session peer) {
        peers.add(peer);
        if(peers.size()==1){
            figura.put("tipo", "Figure");
            figura.put("x", 41);
            figura.put("y", 41);
            figura.put("color", "amarillo");
            figura.put("puntos", 0);
            figura.put("nivel", 0);
            figura.put("id", "1");
            fig [0] = new Figure(figura);

            obstaculo.put("tipo", "Figure");
            obstaculo.put("x", 241);
            obstaculo.put("y", 241);
            obstaculo.put("color", "rojo");
            obstaculo.put("puntos", 0);
            obstaculo.put("nivel", 0);
            obstaculo.put("id", "2");
            fig [1] = new Figure(obstaculo);

            vida.put("tipo", "Figure");
            vida.put("x", 81);
            vida.put("y", 41);
            vida.put("color", "vida");
```

```
            vida.put("puntos", 0);
            vida.put("nivel", 0);
            vida.put("id", "4");
            fig [2] = new Figure(vida);

            enemigo.put("tipo", "Figure");
            enemigo.put("x", 1);
            enemigo.put("y", 1);
            enemigo.put("color", "demonio");
            enemigo.put("puntos", 0);
            enemigo.put("nivel", 0);
            enemigo.put("id", "3");
            fig [3] = new Figure(enemigo);

            Hilos mihilo = new Hilos(fig, peer, peers);
        }
    }

    @OnClose
    public void onClose (Session peer) {
        peers.remove(peer);
    }

    public Figure[] getArrayFigure(){
        System.out.println("fig[2] --> "+fig[2]);
        return fig;
    }
}
```

6. Haga clic en el editor y corrija las importaciones.
7. Observar que el error ha desaparecido en la clase `Hilos.java` al haber creado el método `getArrayFigure()`.
8. Haga clic en el proyecto en la ventana de Proyectos y seleccione Run.

6. Cambio de vistas

Inicialmente se ha creado un juego basado en los dibujos animados de los Looney Tunes, es una temática que se ha considerado podría despertar interés al usuario. Aun así de la misma forma en la que se ha escogido esta vista para el juego se podría haber escogido cualquier otra.

Cambiar el aspecto y diseño visual del juego es algo bastante sencillo de hacer, y cambiando escasas líneas de código podríamos obtener otro juego totalmente diferente en cuanto a interfaz se refiere, manteniéndose la lógica.

A continuación se describen los pasos a seguir para cambiar un juego con temática Looney Tunes a otro sobre Minions, por ejemplo.

Se escogen (de internet o del propio pc) todas las vistas que se deseen utilizar. En este caso necesitaremos: Un personaje que actúe como enemigo, cuatro personajes que ejerzan como jugadores, un obstáculo, una vida, una imagen de fondo para el tablero, una imagen con el nombre del juego y otras dos para la parte del login y signin. Es bastante importante tener en cuenta que las imágenes tiene que ser .png para que la apariencia sea la mejor posible. Además de todo esto sería recomendable obtener varios sonidos que se ajusten a la nueva temática.

Una vez obtenidas todas las imágenes y sonidos las guardamos dentro de la carpeta *img* que se encuentra dentro del proyecto.

Ahora se irá cambiando una a una cada vista:

<u>Jugadores:</u> Las imágenes o vistas de los jugadores a utilizar se han de definir tanto en el cliente como en el servidor. En la parte del cliente modificamos la parte sombreada del archivo `inicioJuego.html`.

```
<tr>
   <td><input type="radio" name="forma" value= "personaje1" checked="checked"
onclick="inputPulsado(this)"><img height="85" width="85" src="img/minion.png">
   </td>
</tr>
<tr>
  <td><input type="radio" name="forma" value= "personaje2"
onclick="inputPulsado(this)"><img height="85" width="85" src="img/m_bailon.png">
  </td>
</tr>
<tr>
  <td><input type="radio" name="forma" value= "personaje3"
onclick="inputPulsado(this)"><img height="85" width="85" src="img/m_eroe.png">
  </td>
</tr>
<tr>
  <td><input type="radio" name="forma" value= "personaje4"
onclick="inputPulsado(this)"><img height="85" width="85" src="img/m_golf.png">
  </td>
</tr>
```

Y en la parte del servidor, archivo `tablero.js`, se modifica de la misma manera la parte sombreada que se muestra a continuación:

```
    //código ubicado en la parte inicial del documento
    var img = new Image;
    img.src = "img/minion.png";
    tablero.drawImage(img, figura.x, figura.y, 99, 99);
```

```
   //código ubicado dentro de la función drawImage
switch(id)
        {
        case "1":
            if (forma_fig==="personaje1"){
            img.src = "img/minion.png";
            tablero.drawImage(img, json[paso].x, json[paso].y, 99, 99);

        else if (forma_fig==="personaje2"){
            img.src = "img/m_bailon.png";
            tablero.drawImage(img, json[paso].x, json[paso].y, 99, 99);
        }
        else if (forma_fig==="personaje3"){
            img.src = "img/m_eroe.png";
            tablero.drawImage(img, json[paso].x, json[paso].y, 99, 99);
        }
        else if (forma_fig==="personaje4"){
            img.src = "img/m_golf.png";
            tablero.drawImage(img, json[paso].x, json[paso].y, 99, 99);
        }
        break;
    }
```

Enemigo: La imagen del enemigo únicamente es necesario declararla en el servidor (archivo `tablero.js`):

```
    //código ubicado en la parte inicial del documento
    var img3 = new Image;
    img3.src = "img/m_malo.png";
    tablero.drawImage(img3, enemigo.x, enemigo.y, 99, 99);
```

```
   //código ubicado dentro de la función drawImage
   case "3":{
     img.src = "img/m_malo.png";
     tablero.drawImage(img, json[paso].x, json[paso].y, 99, 99);
     break;
    }
```

Obstáculo: La vista del obstáculo a modificar se ubica en el servidor (archivo tablero.js):

```
//código ubicado en la parte inicial del documento
var img2 = new Image;
img2.src = "img/piedras.png";
tablero.drawImage(img2, obstaculo.x, obstaculo.y, 99, 99);
```

```
//código ubicado dentro de la función drawImage
 case "2":{
    img.src = "img/piedras.png";
    tablero.drawImage(img, json[paso].x, json[paso].y, 99, 99);
         break;
  }
```

Vida: De igual manera que el enemigo y el obstáculo, la vida se varía dentro del servidor en:

```
//código ubicado en la parte inicial del documento
var img4 = new Image;
img4.src = "img/banana.png";
tablero.drawImage(img4, vida.x, vida.y, 99, 99);
```

```
//código ubicado dentro de la función drawImage
case "4":{
   img.src = "img/banana.png ";
   tablero.drawImage(img, json[paso].x, json[paso].y, 99, 99);
   break;
 }
```

Imágenes de fondo (Juego): Ahora se procede a cambiar las vistas del contenedor del tablero de juego, del propio tablero, y del nombre del juego. Estos cambios se realizan en el archivo child.css. En el caso del tablero contenedor se ha considerado mejor poner el fondo de color amarillo en lugar de poner una imagen que produjese el mismo efecto. Por otro lado, es necesario modificar el estilo del nombre del juego, además de su vista, para centrarlo en el tablero. Han de modificarse las siguientes partes sombreadas:

```
//código del tablero interior
 .contenedor{
     background-image: url("img/fondo_tablero.png");
 }
 //código tablero contenedor
 .cuadro_exterior{
    background-color: yelow;
  }
```

```
    //código nombre del juego
     .seccion_h1{
        width: 366px;
        height: 64px;
        background-image: url(img/titulo_juego.png);
        margin: 10px 260px;
   }
```

Imágenes de fondo (Login-Signin): Además de los cambios anteriores, será necesario cambiar también las imágenes de fondo del login y del signin. Como se muestra a continuación, los cambios se hacen sobre los archivos `login_css.css` y `signin_css.css` respectivamente:

```
  //código en el login_css.css
body {
        background: url(img/fondo_login.jpg);

    }

   //código en el signin_css.css
body {
        //En este caso, únicamente se dejaría esta línea de coódigo
        background: url(img/fondo_signin.png);

    }
```

Sonidos: Para modificar los diferentes sonidos que se reproducen en el juego, es necesario cambiar el nombre del sonido a utilizar. Estos cambios se realizan en el archivo `inicioJuego.html`:

```
  <audio src="img/Little-christmas-bell-sound.wav" id ="player"></audio> //Al coger
vidas
  <audio src="img/minions_inicio.mp3" id ="inicio"></audio> //Al comienzar de juego
  <audio src="img/minion_risa.mp3" id ="ganar"></audio> // Al ganar la partida
  <audio src="img/persecucion.mp3" id ="perder"></audio> //Al perder la partida
```

Una vez realizados todos estos cambios obtenemos un juego aparentemente diferente pero con la misma lógica. Como se ha podido apreciar, realizar las modificaciones para los cambios de vistas es sencillo y cualquier usuario con unos conceptos básicos de programación podría realizarlos sin problema alguno siguiendo estos pasos.

A continuación se muestran tres figuras. La primera (Figura 21) expone cómo quedaría la parte del Login, en la siguiente (Figura 22) se ve la nueva vista de la página del registro, y por último (Figura 23) el juego en sí, con la nueva temática de los Minions.

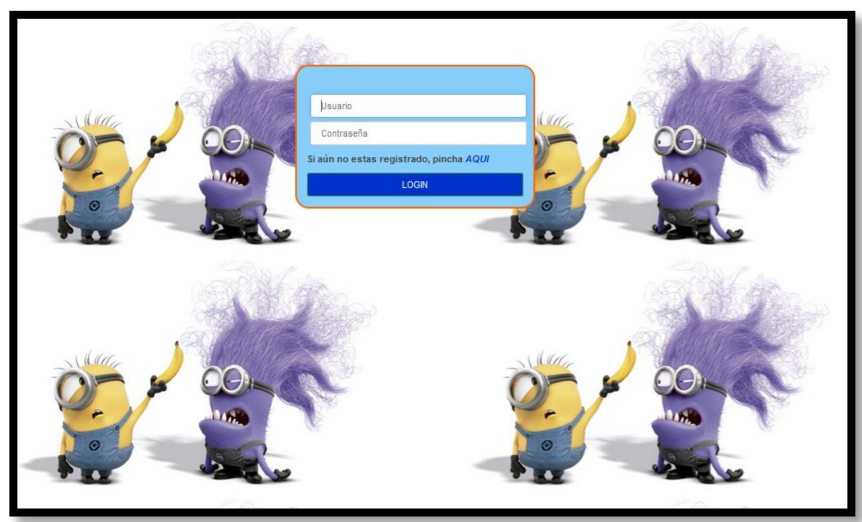


Figura 21: Login (cambio de vista)

Figura 22: Registro (cambio de vista)

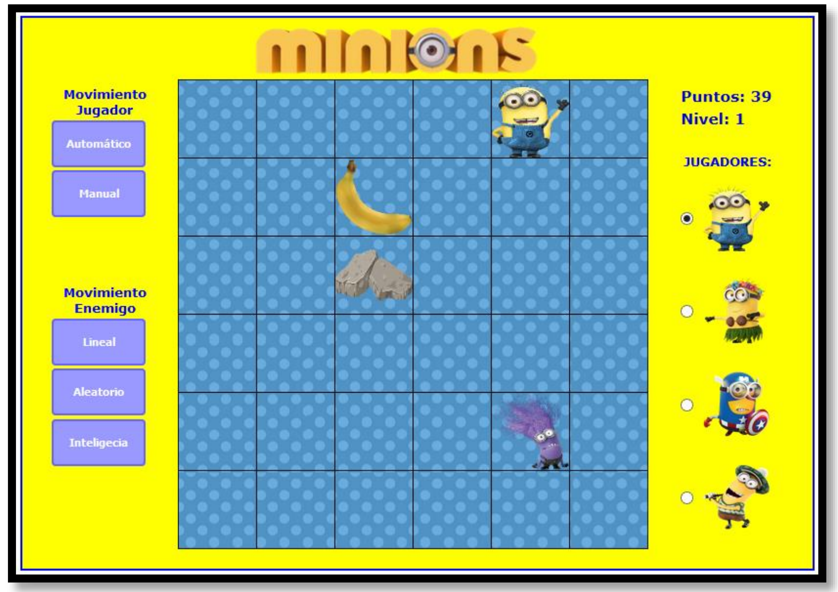


Figura 23: Juego Minions (cambio de vista)

7. Conclusiones

Tras finalizar el desarrollo de la aplicación web y la redacción de la memoria, es el momento de reflexionar sobre lo aprendido, lo conseguido y lo que se podría mejorar.

En este apartado se van a comentar las conclusiones sobre los objetivos del proyecto y los objetivos finalmente conseguidos.

7.1 Puntos fuertes y objetivos conseguidos

Sin lugar a dudas puede decirse que se ha cumplido con mayor o menor éxito con los objetivos planteados inicialmente en el proyecto. Se ha desarrollado una aplicación web plenamente funcional que ha conseguido satisfacer los requisitos propuestos inicialmente.

Respecto a los objetivos planteados, es interesante resaltar la utilidad que ha tenido el estudio de los WebSocket para conseguir una aplicación web interactiva. Tras el estudio de estos se ha aprendido la multitud de ventajas que ofrecen tanto para la comunicación entre cliente y servidor como para la propagación de la información a diferentes puntos de una sesión. Uno de los principales problemas que se ha solventado ha sido el de la concurrencia entre varios jugadores, la simultaneidad entre partidas y la distinción entre ellas ha sido posible permitiendo que todas puedan coexistir.

Se considera también una particular fortaleza del proyecto el haber ido más allá del uso de los WebSocket y haber diseñado una aplicación web en la que interaccionan distintas técnicas para el desarrollo web. Para cada objetivo marcado en el proyecto se han valorado las necesidades requeridas y se ha utilizado las tecnologías más adecuadas para cada fin.

El empleo de tecnologías REST y AJAX ha servido para aprender tanto el funcionamiento de cada una de ellas como su utilidad. Además de aclarar con qué fines es más beneficioso usar estas técnicas de desarrollo frente a otras.

Por último, al margen de los objetivos establecidos, el proyecto ha servido para profundizar y asentar los conocimientos adquiridos a lo largo de la carrera, así como el aprendizaje de nuevas tecnologías y técnicas de programación.

7.2 Mejoras y trabajos futuros

A pesar de lo comentado anteriormente, aún queda trabajo por hacer. Toda aplicación siempre es mejorable, ya sea añadiendo nuevas funcionalidades, eliminando pequeños fallos, etc. Son algunas las mejoras posibles en la aplicación, y a continuación se detallan algunas de ellas:

- **Mejora del código de la aplicación**: Aunque el código realizado funciona perfectamente puede que todavía pueda perfeccionarse más, haciéndolo más flexible y eficiente.
- **Mejoras en la interfaz de usuario:** Aunque la interfaz gráfica que se ha creado ha sido pensada para aportar entretenimiento y diversión al usuario, quedan algunas mejoras

que aportarían mejor sensación, como el desarrollo de animaciones mejoradas que comuniquen al jugador que ha perdido o ganado la partida.

- **Recuperación de partidas pasadas:** Aunque existe comunicación entre el servidor y una base de datos, se considera que una buena mejora del juego sería añadir la funcionalidad de guardar las partidas inacabadas que el usuario desee y poder recuperarlas más tarde para poder continuar a partir de ellas. Para eso sería necesario almacenar la información de los cuatro objetos JSON en la base de datos, y posteriormente extraerlos de ella.

8. Bibliografía

WebSocket: Lightweight Client-Server Communications, de Andrew Lombardi, ASIN: B015D78JVQ

Professional Java for Web Applications: Featuring Websockets, Spring Framework, JPA Hibernate, and Spring Security, Nicholas S. Williams, John Wiley & Sons Inc, ISBN-13: 978-1118656464

RESTful Web Services, Leonard Richardson, O'Reilly Media, ISBN-13: 978-0596529260
Desarrollo de Juegos en HTML5, Egor Kuryanovich, Shy Shalom, Russell Goldenberg, Mathias Paumgarten, Anaya Multimedia, ISBN 9788441532021

9. Referencias

[1] NetBeans: https://netbeans.org/

[2] Maven: https://maven.apache.org/

[3] Bower: http://bower.io/

[4] Npm: https://www.npmjs.com/

[5] WebSocket: http://www.websocket.org/book.html

[6] REST: http://www.tutorialspoint.com/restful/restful_introduction.htm

[7] Java: https://www.java.com/es/about/whatis_java.jsp

[8] JavaScript: http://www.w3schools.com/js/

[9] CSS: http://www.w3schools.com/css/default.asp

[10] HTML: http://www.w3schools.com/html/default.asp

[11] Canvas: http://www.w3schools.com/canvas/

[12] Bootstrap: http://getbootstrap.com/

[13] AJAX: http://www.w3schools.com/ajax/

[14] jQuery: https://jquery.com/

[15] JSON: http://www.w3schools.com/json/

[16] GlassFish: https://glassfish.java.net/

[17] MongoDB: https://www.mongodb.org/