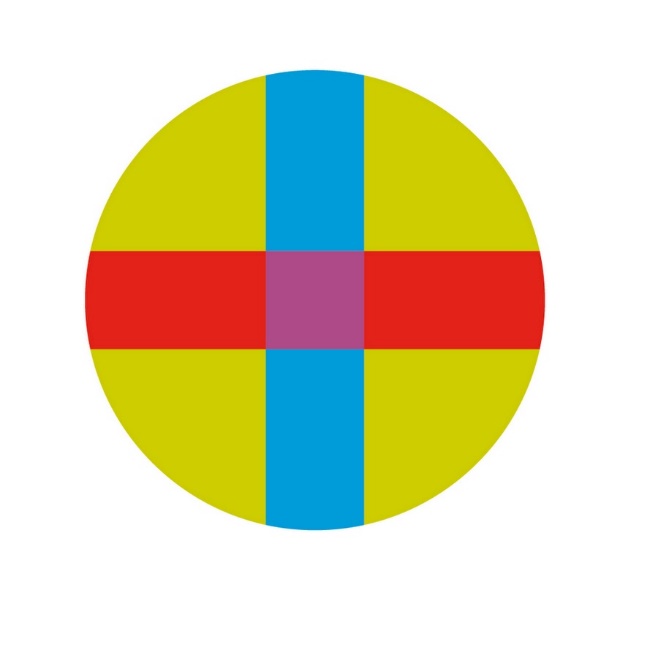
UNIVERSIDAD SAN PABLO - CEU

ESCUELA POLITÉCNICA SUPERIOR

GRADO EN INGENIERÍA DE SISTEMAS DE TELECOMUNICACIÓN



TRABAJO FIN DE GRADO

**Desarrollo de una aplicación web y base de datos para el seguimiento de pacientes con problemas de movilidad cervical**

Autor: Borja González Diaz

Director: Nombre y apellidos del director

Octubre de 2017

UNIVERSIDAD SAN PABLO-CEU

ESCUELA POLITÉCNICA SUPERIOR

División de Ingeniería

**Calificación del Trabajo Fin de Grado**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Datos del alumno   |  | | --- | | NOMBRE: |   Datos del Trabajo   |  | | --- | | TÍTULO DEL PROYECTO: |   Tribunal calificador   |  |  | | --- | --- | | PRESIDENTE: | FDO.: |      |  |  | | --- | --- | | SECRETARIO: | FDO.: |      |  |  | | --- | --- | | VOCAL: | FDO.: |  |  | | --- | | Reunido este tribunal el \_\_\_\_\_\_/\_\_\_\_\_\_/\_\_\_\_\_\_, acuerda otorgar al Trabajo Fin de Grado presentado por Don\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_la calificación de \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_. | |

A toda mi familia y en especial a mi padre Peps, con el que me hubiese encantado compartir este momento.

**Resumen**

Con la llegada de las nuevas tecnologías al mercado la mayoría de ámbitos profesionales se están viendo forzados a adaptarse mediante la digitalización de todos sus sistemas, actualmente manuales y prácticamente obsoletos. Quedarse atascado en estos sistemas manuales significaría una gran desventaja a la hora de competir contra otros sistemas digitalizados, que son mucho más rápidos y eficientes, con una necesidad nula o mínima de mantenimiento. Esta disyuntiva también se aplica a los sistemas médicos, que se pueden beneficiar enormemente de la digitalización de la información relativa a los pacientes, permitiendo análisis más avanzados y una mayor agilidad para tratar a los pacientes.

Con un enfoque moderno a los métodos de visualización y recopilación de datos he construido una aplicación web con su correspondiente base de datos, que permiten al usuario acceder rápidamente a los datos de los pacientes permitiendo mostrar toda la información de manera sencilla y clara.

La aplicación web que he desarrollado se centra en el seguimiento de pacientes con problemas de movimiento cervical, específicamente en el rango de movimiento. El rango de movimiento cervical puede verse afectado o disminuido por varias causas como podrían ser los accidentes y más concretamente, el síndrome del latigazo vertical (SLC). Para poder medir el rango de movimiento del paciente, hay disponibles una serie de instrumentos que nos facilitan ésta tarea. Para este proyecto he utilizado un sensor inercial que proporciona una forma flexible y económica de medir la movilidad. Concretamente, el instrumento que vamos a utilizar se llama IMU (Inertial Measurement Unit).

Una parte fundamental de este trabajo ha sido la captura de los requisitos con el cliente, donde el cliente expresó las necesidades que requería para crear una herramienta para la medición del rango de movimiento cervical.

Para empezar el trabajo se han realizado pruebas en las que se han realizado varias mediciones del rango de movimiento de un grupo de 5 personas donde participaban tanto hombres como mujeres, ya que cada sexo tiene un rango de movimiento distinto.

Utilizando varios lenguajes de programación, plataformas y sistemas he conseguido satisfacer las necesidades dicha aplicación, cumpliendo así sus requisitos.

**Agradecimientos**

Tras estos años de carrera puedo afirmar que ha sido una experiencia que me ha permitido crecer de formar tanto profesional como personal. Al mirar atrás, obtengo una visión en perspectiva y me doy cuenta que todo el esfuerzo y dedicación han merecido la pena, y a día de hoy me encuentro un paso más cerca de convertirme en un ingeniero. Por su puesto todo esto no hubiese sido posible sin la ayuda de muchas personas que me han acompañado a lo largo del camino.

En primer lugar, quiero agradecer a toda mi familia el apoyo tanto sentimental como económico que me han ofrecido a lo largo de estos años. Hacer una mención especial a mi Madre Maite, que siempre me ha apoyado en todas mis decisiones y a estado conmigo en los momentos buenos y sobretodo en los malos. A mi Padre Peps que siempre fue un apoyo incondicional y al que agradeceré durante toda mi vida por la honestidad y franqueza con las que me ha brindado. Quiero agradecer a mis abuelas Christine y Encarnación que son personas a las que admiro y que me han ofrecido consejos y sabiduría que me han servido para afrontar los retos de esta carrera. Por último, agradecer a mi Tío Ignacio, al que también admiro enormemente por su espíritu de superación y por sus consejos que me han ayudado en mi desarrollo personal.

Agradecer también el apoyo que me han brindado mis amigos, tanto a las amistades con los que entré a esta universidad y como a las nuevas que he hecho durante el desarrollo de la carrera. Una mención especial a mi amigo Jorge con el que puedo decir que he compartido los mejores momentos de esta carrera, pero más importante aún, ha estado en los momentos más difíciles.

Para finalizar, quería agradecer todo el apoyo del personal del CEU, en especial mis profesores, que me han sabido guiar en la dirección correcta en todo momento y de los que he podido aprender una cantidad de conocimientos que valoro como incalculables. Menciones especiales a David que ha sabido guiarme y aconsejarme de la forma adecuada en todo momento y a Rodrigo que ha sido un elemento fundamental en el desarrollo de este proyecto y en mi desarrollo a lo largo de la carrera, ya que ha sabido guiar mi atención en la dirección adecuada a lo largo de toda la carrera.

**Índice general**

[1. Introducción 8](#_Toc494496032)

[2. Estado del arte 9](#_Toc494496033)

[2.1. Diseño de web estático 9](#_Toc494496034)

[2.1.1. HTML 9](#_Toc494496035)

[2.1.2. CSS 9](#_Toc494496036)

[2.1.3. JavaScript 10](#_Toc494496037)

[2.1.4. Chart.js 10](#_Toc494496038)

[2.1.5. Papa Parse 10](#_Toc494496039)

[2.2. Diseño del lado del Servidor 11](#_Toc494496040)

[2.2.1. NodeJS 11](#_Toc494496041)

[2.2.1 Express.js 12](#_Toc494496042)

[2.2.2. Socket.io 12](#_Toc494496043)

[2.3. Bases de Datos 13](#_Toc494496044)

[2.3.1. SQLite 14](#_Toc494496045)

[2.4. Sensor Inercial - IMU 15](#_Toc494496046)

[2.4.1. Werium Basic Pro 15](#_Toc494496047)

[3. Diseño 16](#_Toc494496048)

[3.1. Descripción del problema 17](#_Toc494496049)

[3.2. Requisitos 17](#_Toc494496050)

[3.2.1. Requisitos Funcionales 17](#_Toc494496051)

[3.2.2. Requisitos no Funcionales 18](#_Toc494496052)

[3.3. Casos de uso 18](#_Toc494496053)

[3.4. Matriz de trazabilidad 23](#_Toc494496054)

[3.5. Arquitectura del sistema 24](#_Toc494496055)

[3.5.1. Diseño visual (Storyboard) de la aplicación web 24](#_Toc494496056)

[3.5.2 Esquema del modelo de datos 26](#_Toc494496057)

[3.5.3 Estructura del archivo CSV 30](#_Toc494496058)

[4. Implementación 31](#_Toc494496059)

[4.1. Comunicación Cliente-Servidor 31](#_Toc494496060)

[4.1.1. Servidor 31](#_Toc494496061)

[4.1.2. Cliente 32](#_Toc494496062)

[4.1.3 Despliegue del servidor 33](#_Toc494496063)

[4.2. SQLite 33](#_Toc494496064)

[4.2.1. Compatibilidad con el Servidor 33](#_Toc494496065)

[4.3. Funciones 34](#_Toc494496066)

[4.3.1. Obtener pacientes 34](#_Toc494496067)

[4.3.2. Borrar Paciente 36](#_Toc494496068)

[4.3.3. Añadir un Paciente 39](#_Toc494496069)

[4.3.4. Obtener datos de movimiento de un paciente 42](#_Toc494496070)

[4.3.5. Añadir datos de movimiento 46](#_Toc494496071)

[4.3.6. Borrar una sesión de movimientos 50](#_Toc494496072)

[4.3.7 Mostrar un gráfico de un movimiento 52](#_Toc494496073)

[4.3.8 Mostrar un gráfico de evolución de un movimiento 55](#_Toc494496074)

[4.4. Diagrama de flujo 63](#_Toc494496075)

[5. Pruebas 64](#_Toc494496076)

[5.1. Pruebas de sistema 64](#_Toc494496077)

[5.1.1. Obtener pacientes 65](#_Toc494496078)

[5.1.2. Añadir un paciente 65](#_Toc494496079)

[5.1.3. Borrar un paciente 66](#_Toc494496080)

[5.1.4. Obtener datos de movimiento de un paciente 68](#_Toc494496081)

[5.1.5. Añadir datos de movimiento 69](#_Toc494496082)

[5.1.6. Borrar una sesión de movimientos 69](#_Toc494496083)

[5.1.7. Mostrar un gráfico de un movimiento 71](#_Toc494496084)

[5.1.8. Mostrar un gráfico de evolución de un movimiento 72](#_Toc494496085)

[6. Resultados y conclusiones 73](#_Toc494496086)

[6.1. Resultados 73](#_Toc494496087)

[6.2. Conclusiones 73](#_Toc494496088)

[6.3. Líneas de trabajo futuras 74](#_Toc494496089)

[7. GitHub 75](#_Toc494496090)

# 

# 1. Introducción

La motivación de este trabajo es facilitar el seguimiento de pacientes con problemas de movilidad cervical, debido a que, actualmente, este seguimiento se hace de una forma manual y no es posible llevar a cabo un seguimiento de forma. Este trabajo busca facilitar este seguimiento para que tanto paciente como médico y fisioterapeuta puedan hacer un mejor uso de su tiempo y recursos. Para hacer esto posible vamos a recurrir a las tecnologías de la información y las comunicaciones para crear una aplicación web que permita analizar información de movilidad cervical almacenada en una base de datos. Por tanto, el objetivo de esta aplicación web será facilitar el acceso y visualización de esta información.

Por tanto, la aplicación permitirá el acceso a una base de datos donde se almacena un listado de pacientes y de registros de sus movimientos cervicales, tomados utilizando un sensor comercial ya existente que los exporta en un formato particular del mismo. El sensor comercial utilizado para la obtención de medidas corresponde al dispositivo Werium Basic Pro, que ofrece unas altas prestaciones. A partir de dicha base de datos el sistema permitirá consultar movimientos cervicales en todos los planos y observar la evolución con el tiempo de estos movimientos, para así comprobar si se encuentran dentro de la normalidad. Estos intervalos de normalidad dependen del sexo del paciente, pues cada uno posee rangos de movilidad cervical ligeramente distintos. Los datos referentes a los intervalos de normalidad se han obtenido del Journal of Orthopaedic Research, concretamente del documento “Active range of motion of the head and cervical spine: a three-dimensional investigation in healthy young adults”.

La aplicación incluirá una sección específica para cada paciente, que mostrará un conjunto de movimientos; en los planos transversal, coronal y sagital, con una fecha asociada que se corresponde a la fecha de la medición. Se podrá consultar cada movimiento por separado en forma de gráfico. Además, también se podrá visualizar, de nuevo en forma de gráfico, la evolución de cada movimiento a medida que pasa el tiempo, usando para ello medidas tomadas en diferentes sesiones. En este gráfico de evolución se mostrarán también los valores de normalidad antes nombrados, para que sea posible observar si el paciente entra dentro de dichos valores. Se podrán añadir y borrar sesiones de movimientos, teniendo en cuenta que para añadir una habrá que seleccionar un archivo local en el formato producido por el software del sistema de captura de movimientos (en formato CSV) y asociar una fecha y hora de medición a la sesión, para así soportar más de una medición diaria.

# 2. Estado del arte

## 2.1. Diseño de web estático

### 2.1.1. HTML

HTML, acrónimo de HyperText Markup Language (lenguaje de marcas de hipertexto), se refiere al lenguaje de programación que se utiliza hoy en día para la parte principal del diseño de las páginas web. Este lenguaje de programación dictamina el contenido de una página web pero no su funcionalidad. De la apariencia de la página se encarga CSS y de la funcionalidad sería JavaScript. Para poner ejemplos, HTML define elementos de la página web como podrían ser el texto, las imágenes, los títulos o los juegos entre otros objetos. En el caso de mi aplicación web utilizo HTML para posicionar los distintos elementos, como podrían ser el título, una tabla, el pie de página u otros elementos, en un orden específico. También lo utilizo para añadir enlaces a las distintas pestañas de mi página web, por lo que HTML da la posibilidad de enlazar contenidos con otras páginas web.

La forma de escribir en HTML es mediante el uso de etiquetas. Todo lo que se encuentre entre el etiquetado será lo que se visualice en la página. Algunos ejemplos típicos de estas etiquetas son los siguientes:

<head> 🡪 cabecera de la página.

<title> 🡪 Título de la página.

<body> 🡪 El cuerpo o contenido principal de la página web.

<img> 🡪 Imagen.

Y muchos otros más.

### 2.1.2. CSS

CSS, acrónimo de Cascading Style Sheets (Hojas de estilo en cascada) es un lenguaje de programación simple que es utilizado para definir el aspecto o presentación de documentos HTML o XML, por lo que podríamos decir que es un lenguaje de diseño gráfico. Como he mencionado antes CSS se utiliza en conjunto a HTML y JavaScript para definir una página web. Con una misma hoja de estilos es posible definir el diseño de varios documentos HTML, incluyendo características como el color, las fuentes y las capas.

Para mi página web he compartido una hoja de estilos para definir el aspecto de los tres documentos HTML que utilizo. De forma más específica, he definido el diseño de las pestañas que dan acceso a cada documento HTML, el diseño de las tablas y los botones que se muestran. Para el caso de los botones, CSS me ha permitido definir la interactividad con los botones y las imágenes que corresponden al botón.

### 2.1.3. JavaScript

JavaScript es un lenguaje de programación interpretado, lo que quiere decir que el navegador interpreta el código línea a línea mientras la página web se va cargando. También se define como un lenguaje orientado a objetos, lo que quiere decir que las instrucciones escritas en este lenguaje son en realidad llamadas a métodos o propiedades de objetos del navegador.

JavaScript es el estándar de ECMAScript y es conocido principalmente como el lenguaje de script de las páginas web. También es utilizado en otros ámbitos distintos al navegador, como pueden ser node.js, el cual utilizo en esta aplicación web, y Apache CouchDB.

JavaScript posee una sintaxis que es muy parecida a la utilizada en el lenguaje de programación C, ya que se utilizan sentencias comunes como pueden ser if, for o switch entre otras. Esta similitud se debe a que la sintaxis de JavaScript proviene de C y de Java. La principal diferencia con estos lenguajes es que JavaScript no tiene clases y que las funciones son objetos.

Para el desarrollo de mi aplicación web he utilizado este lenguaje en la mayor parte de mi código. Algunos ejemplos de uso son la utilización de bucles for para rellenar las tablas, alertas que avisan al usuario de ciertos eventos y la obtención de valores introducidos por el usuario.

### 2.1.4. Chart.js

Chart.js es una plataforma de JavaScript que nos permite crear gráficos simples, pero a la vez flexibles.

* A través del elemento <canvas> se consiguen crear gráficos simples.
* Con Chart.js podemos crear hasta 8 tipos de gráficos personalizables y con los que se puede interactuar.
* Posee un gran rendimiento en todos los navegadores actuales (IE9+).
* Redibuja los gráficos a la hora de ampliar o reducir sobre ellos para conseguir una escala perfecta de granularidad.
* Chart.js funciona muy bien a la hora de visualizar los gráficos en navegadores de otros dispositivos como tablets y móviles

### 2.1.5. Papa Parse

Papa parse es un analizador sintáctico que convierte archivos de texto delimitados, principalmente archivos CSV, en estructuras de datos y viceversa. Papa Parse es el analizador más rápido en navegadores para JavaScript. Según la RFC 4180, Papa parse es el analizador sintáctico más fiable y por lo tanto el más recomendable. Posee las siguientes características:

* Muy fácil de usar.
* Parsea ficheros CSV en un entorno local o a través de la red directamente.
* Transmite archivos de gran tamaño (incluso a través de HTTP).
* El análisis sintáctico con la conversión se realizan, además, de forma inversa(JSON a CSV).
* Detección automática de delimitadores.
* Pausa, reanudación y anulación del parseo.
* Papa parse no tiene dependencias.
* Uno de los únicos analizadores que maneja correctamente los saltos de línea y las comillas.
* Puede convertir números y booleanos a sus tipos.

Soporte del encabezado de fila.

## 2.2. Diseño del lado del Servidor

### 2.2.1. NodeJS

NodeJS es un entorno de JavaScript situado en el lado del servidor. Es una arquitectura basada en eventos, la cual permite a las operaciones no bloqueantes comunicar señales de éxito o de error en su terminación. Las operaciones no bloqueantes de entrada y salida se apoyan en un único hilo que utiliza multi-threading para las operaciones E/S.

NodeJS ejecuta el código JavaScript mediante el motor V8, que ha sido desarrollado por Google. Este motor permite a NodeJS ejecutar y compilar el código de una forma increíblemente rápida.

Junto con la alta velocidad de ejecución como característica fundamental, encontramos el Bucle de eventos. Este bucle nos permite manejar una gran cantidad de clientes, ya que en NodeJS todas las operaciones pesadas I/O se realizan de forma asíncrona. El problema de algunos servidores actuales es que asignan un hilo a cada cliente y junto a este hilo va una cantidad me memoria RAM asociada, por lo que el numero de clientes está limitado al tamaño de la RAM del servidor. Node resuelve este problema, ya que en cada conexión se utiliza el motor V8 para ejecutar un evento.



### 2.2.1 Express.js

Express es el entorno de trabajo web más utilizado en NodeJS y está diseñado para la construcción de aplicaciones web, aplicaciones móviles y APIs. Es posible crear una API de forma sencilla y rápida que proporciona una gran cantidad de métodos de utilidad para HTTP y a la lógica de intercambio de información entre programas o middleware. Las características principales de Express se pueden resumir en las siguientes:

* Infraestructura mínima, flexible y rápida.
* Enrutamiento robusto.
* Alto rendimiento.
* Ejecución sencilla que permite generar aplicaciones rápidamente.

### 2.2.2. Socket.io

Socket.io es una librería de JavaScript utilizada para aplicaciones web en tiempo real. Permite una comunicación bidireccional y en tiempo real entre cliente y servidor. Sockets.io está dividido en dos partes, una librería del cliente que se ejecuta en el navegador y una librería del servidor para Node.js. Ambas partes tienen prácticamente la misma API. Al igual que Node.js, Socket.io presenta una arquitectura orientada a eventos.

Generalmente Socket.io utiliza el protocolo WebSocket, y como opción alternativa, en el caso de que el cliente no soporte el protocolo WebSocket, utiliza polling, que consiste en una consulta constante al servidor por parte del cliente, pero siempre utiliza la misma interfaz. El protocolo WebSocket se basa en una comunicación bidireccional y full-duplex a través de un socket TCP. Esto significa que cliente y servidor pueden intercambiar información al mismo tiempo. Además de ofrecer la funcionalidad de WebSocket, Socket.io es capaz de emitir mensajes a varios sockets, almacenar datos asociados a cada cliente y E/S asíncrona.

## 2.3. Bases de Datos

Los sistemas de gestión de datos (DBMS) proporcionan la capacidad de almacenarinformación y permiten un acceso eficiente, fiable, conveniente y seguro a cantidades muy grandes de información que pueden persistir varios usuarios. Las características de estos sistemas son las siguientes:

* Eficientes, ya que permiten miles de peticiones o modificaciones por segundo.
* Fiables debido a una disponibilidad casi total.
* Convenientes ya que son independientes de los datos físicos y con un lenguaje de peticiones de alto nivel.
* Seguros tanto a nivel de software, hardware, energía y usuarios.
* Con grandes volúmenes de información. Del orden de Terabytes.
* Persistentes. La información se guarda de forma permanente.
* Accesibles por múltiples usuarios gracias a un control de acceso concurrente.

Las aplicaciones más comunes son para la gestión de empresas e instituciones públicas. También son ampliamente utilizadas en entornos científicos con el objeto de almacenar la información experimental.



### 2.3.1. SQLite

SQLite es un sistema que gestiona bases de datos relacionales (DBMS). Éste sistema es completamente ACID y está contenido en una biblioteca en C que se podría considerar pequeña para ciertos casos. El problema del tamaño se ha resuelto en la una de las últimas versiones que permite bases de datos de hasta 2 Terabytes.

Que el sistema sea completamente ACID, significa que las propiedades de las transacciones del sistema garantizan la fiabilidad del sistema incluso en casos de errores. Las características de un sistema ACID son:

* Atomicidad: O se ejecutan todos los pasos de una operación o no se ejecuta ninguno.
* Consistencia: Solo se ejecutarán las transacciones que cumplan con las reglas de integridad de la base de datos.
* Aislamiento: Cada transacción se realiza de forma autónoma a las demás para evitar errores entre transacciones sobre la misma información.
* Durabilidad: Las transacciones realizadas aseguran que los datos se guarden de forma permanente, incluso cuando el sistema falle, por lo que los datos han de almacenarse en una memoria no volátil.

SQLite es un proyecto de dominio público, por lo que cualquier persona puede colaborar en el proyecto para añadir funcionalidad o sugerir cambios.

A diferencia de otros gestores de bases de datos de tipo cliente-servidor, SQLite forma parte del programa con el que se comunica, ya que la mayoría de gestores están implementados de forma autónoma al programa con el que se comunican. Esto clasifica a SQLite como un gestor de base de datos embebido o empotrado. Estos gestores permiten que todas las operaciones se realicen dentro de la aplicación mediante las funciones o llamadas de las librerías de SQLite (INSERT, SELECT, DELETE).

Para habilitar la interacción entre NodeJS y la base de datos SQLite he utilizado la librería sqlite3. Esta librería nos permite ejecutar consultas de una forma simple y la posibilidad de pasar parámetros a dichas consultas.

## 2.4. Sensor Inercial - IMU

Mediante el uso combinado de acelerómetros, giroscopios y magnetómetros, hay ciertos aparatos electrónicos que miden y dan información acerca de la velocidad, orientación y fuerzas gravitacionales de un objeto en movimiento.

El acelerómetro mide las aceleraciones y tiene la capacidad de medir la orientación de una plataforma fija respecto a la superficie terrestre. El giróscopo es un dispositivo que mide, mantiene o cambia la orientación en el espacio de un objeto. Un magnetómetro sirve para medir el campo magnético producido por la Tierra, y sirve a modo de compás para determinar la orientación en el plano N-E-S-W.

Combinando estos tres sensores, un aparato puede determinar la posición y la orientación de un objeto, lo cual es muy útil a la hora de medir y obtener datos sobre movimientos cervicales.

orie﷽﷽﷽﷽﷽﷽iñon locidaddratos electrometros rticular tera

### 2.4.1. Werium Basic Pro

Basic Pro es una herramienta que se ha diseñado para facilitar el trabajo de los terapeutas. Es un sistema muy preciso a la hora de medir el rango de movimiento articular cervical.  Captura los movimientos en los planos de flexo extensión, inclinaciones laterales y rotación.

Está compuesto por tres giroscopios y tres acelerómetros que a su vez integran un reloj que permite asociar valores temporales a las medidas. Este tipo de herramienta implementa tres ejes ortogonales que referencian a los sensores y así se puede asignar a cada eje un acelerómetro y un giroscopio. La información que proporciona un IMU es la velocidad angular y la aceleración lineal en cada eje, asociada a los valores temporales.

Para el uso correcto de Basic Pro es necesario utilizar dos dispositivos. Para el caso de mediciones de movimientos cervicales, uno de los dispositivos se colocaría en el tronco y otro en la cabeza. Esto permite una medida muy estable debido a la posición relativa entre cabeza y tronco.

Basic Pro incluye la tecnología Bluetooth, lo cual permite compartir las mediciones en tiempo real, por lo que el trabajo del terapeuta se facilita en gran medida, ya que puede medir, visualizar y comprar movimientos de una forma muy sencilla.



Sensor inercial Werium

# 

# 3. Diseño

En este capítulo se intenta analizar en profundidad el problema que se pretende resolver, intentando abarcar la mayor cantidad de información posible y estudiando los principales aspectos que habrían de cubrirse en el desarrollo del proyecto. En este punto se enumerarán las características principales del sistema a desarrollar, por lo que se establecen también los casos de uso que deberá poder efectuar el sistema resultado del proyecto para cumplir con los requisitos que se establezcan.

## 3.1. Descripción del problema

El objetivo principal es construir una aplicación web con una base de datos asociada para el seguimiento de pacientes con problemas de movilidad cervical. Esta aplicación web permitirá el acceso de uno o más usuarios o clientes, y que estos puedan trabajar con la base de datos obteniendo información de los pacientes. Para describir el problema en cuestión voy a seguir un proceso que se divide en varios apartados. Empezaré con el diseño de la arquitectura del sistema para satisfacer las necesidades descritas previamente, seguiré con la implementación de dicho sistema, y terminaré describiendo el despliegue de las aplicaciones y pruebas.

## 3.2. Requisitos

La fase de análisis incluye el estudio y la definición de los requisitos del sistema. Los requisitos definen en detalle los servicios que el sistema debería poder ofrecer, identificando las restricciones y características de este.

### 3.2.1. Requisitos Funcionales

RF1 – Una vez dentro la aplicación web, el sistema debe mostrar una sección de inicio con una breve descripción de las distintas funcionalidades de la página.

RF2 – El sistema ofrecerá un listado de los clientes con su nombre, apellidos y un link para acceder a sus datos de movimiento.

RF3 – El usuario tendrá la opción de añadir pacientes, produciéndose un error en el caso de que el usuario no rellene algún dato solicitado.

RF4 – El usuario tendrá la opción de borrar pacientes, borrando así todos sus datos asociados.

RF5 – Para cada paciente, el sistema mostrará una tabla con los tres movimientos disponibles.

RF6 – El sistema deberá ser capaz de mostrar las distintas sesiones de movimientos.

RF7 – El sistema debe proporcionar a los clientes la opción de añadir datos de movimientos con una fecha asociada.

RF8 - El sistema debe proporcionar a los clientes la opción de borrar datos de movimientos.

RF9- El sistema ofrece la opción de mostrar la evolución de cada movimiento.

### 3.2.2. Requisitos no Funcionales

RNF1 - El sistema debe permitir el acceso a la aplicación web desde cualquier dispositivo. Una vez activo el servidor, cualquier usuario puede acceder a la página web desde cualquier dispositivo.

RNF2 - Solo se admitirán ficheros CSV. Esto quiere decir que a la hora de que el usuario seleccione un archivo local para añadir un set de datos, solo se le permitirá seleccionar un archivo del tipo CSV y no cualquier otro.

RNF3 - Dentro de los gráficos de evolución el sistema marcará unos límites de normalidad dependiendo del sexo del paciente.

RNF4 - El sistema deberá ser capaz de almacenar los datos de los pacientes entre sesiones.

RNF5 - Cuando se realice un cambio en un dispositivo (añadir paciente, borrar sesión de datos, etc…) se reflejará automáticamente en el resto de dispositivos, mostrando la tabla correspondiente con el dato añadido o sin el dato borrado.

RNF6 – Cada vez que el usuario muestre un gráfico, el sistema debe ser capaz de eliminar el gráfico anterior y poner en su lugar el nuevo gráfico elegido por el usuario.

## 3.3. Casos de uso

Los casos de uso representan las diferentes posibles interacciones que pueden existir entre los diferentes actores dentro de un sistema. Los actores de un sistema son entidades externas al sistema que pueden interactuar con el. Los actores pueden ser humanos u objetos (servidores, aplicaciones, servicios...). Los actores que interactúan con nuestra aplicación son los siguientes:

* Usuarios del sistema a través del navegador de cualquier dispositivo.

|  |
| --- |
| **CU1: Acceso a la página web.** |
| El usuario introduce la dirección del servidor en su navegador accediendo instantáneamente al servidor. |
| Actores implicados: Usuario/cliente. |
| Precondiciones: El sistema debe estar correctamente funcionando. |
| Secuencia de acciones:   1. El cliente accede a la aplicación web introduciendo la dirección IP del servidor en su navegador. |

|  |
| --- |
| **CU2: Acceso a la pestaña de pacientes/Listado de pacientes.** |
| El usuario accede a un listado de pacientes. |
| Actores implicados: Usuario/cliente. |
| Precondiciones: Acceso a la página web. |
| Secuencia de acciones:   1. El cliente accede a la aplicación web introduciendo la dirección IP del servidor en su navegador. 2. El cliente accede a la sección de pacientes mediante la pestaña de pacientes. 3. El navegador solicita la lista de pacientes a la base de datos. 4. El cliente recibe la lista de pacientes y se la presenta al usuario en una tabla. |

|  |
| --- |
| **CU3: Añadir un paciente.** |
| El usuario añade un paciente a la base de datos. |
| Actores implicados: Usuario/cliente. |
| Precondiciones: Acceso a la pestaña de pacientes/Listado de pacientes. |
| Secuencia de acciones:   1. El cliente accede a la aplicación web introduciendo la dirección IP del servidor en su navegador. 2. El cliente accede a la sección de pacientes mediante la pestaña de pacientes. 3. El usuario introduce el nombre, apellidos y sexo del paciente y presiona el botón “Añadir”. 4. El navegador manda la petición, para añadir el paciente, al servidor. 5. El servidor añade el nuevo paciente a la base de datos, persistiendo los cambios. |
| Post-condiciones: El nuevo paciente tiene que ser mostrado inmediatamente en el listado de pacientes, sin que el usuario tenga que refrescar la página. |

|  |
| --- |
| **CU4: Borrar un paciente.** |
| El usuario elimina un paciente de la base de datos. |
| Actores implicados: Usuario/cliente. |
| Precondiciones: Acceso a la pestaña de pacientes/Listado de pacientes. |
| Secuencia de acciones:   1. El cliente accede a la aplicación web introduciendo la dirección IP del servidor en su navegador. 2. El cliente accede a la sección de pacientes mediante la pestaña de pacientes. 3. El navegador solicita la lista de pacientes a la base de datos. 4. El cliente recibe la lista de pacientes y se la presenta al usuario en una tabla. 5. El usuario elige el paciente a borrar y presiona el botón de borrado correspondiente, ya que cada paciente tiene su botón de borrado. 6. El navegador pregunta al usuario si desea realizar esta operación, teniendo en cuenta que borraría todos los datos de movimientos asociados a ese paciente. 7. El usuario acepta la operación, teniendo la opción de cancelarla. 8. El navegador manda la petición para borrar el paciente al servidor. 9. El servidor borra al paciente y todos sus datos de movimientos asociados de la base de datos, persistiendo los cambios. |
| Post-condiciones: La lista de pacientes ha de actualizarse, mostrando todos los pacientes menos el borrado. Se actualiza de forma automática. Todos los movimientos asociados a este paciente tienen que desaparecer. |

|  |
| --- |
| **CU5: Listado de movimientos de un paciente.** |
| El usuario accede a un listado de movimientos del paciente. |
| Actores implicados: Usuario/cliente. |
| Precondiciones: Acceso a la pestaña de pacientes/Listado de pacientes. |
| Secuencia de acciones:   1. El cliente accede a la aplicación web introduciendo la dirección IP del servidor en su navegador. 2. El cliente accede a la sección de pacientes mediante la pestaña de pacientes. 3. El navegador solicita la lista de pacientes a la base de datos. 4. El cliente recibe la lista de pacientes y se la presenta al usuario en una tabla. 5. El cliente elige el paciente del que quiere mostrar sus datos de movimiento presionando el botón adecuado, ya que cada paciente tiene su propio botón para mostrar sus datos. 6. El navegador se posiciona en la pestaña de Datos. 7. El navegador solicita la lista de movimientos del paciente a la base de datos. 8. El navegador recibe la lista de movimientos y se la presenta al usuario en una tabla. |

|  |
| --- |
| **CU6: Añadir datos de movimiento.** |
| El usuario añade un set de movimientos al paciente. |
| Actores implicados: Usuario/cliente. |
| Precondiciones: Acceso al listado de movimientos de un paciente. |
| Secuencia de acciones:   1. El cliente accede a la aplicación web introduciendo la dirección IP del servidor en su navegador. 2. El cliente accede a la sección de pacientes mediante la pestaña de pacientes. 3. El navegador solicita la lista de pacientes a la base de datos. 4. El cliente recibe la lista de pacientes y se la presenta al usuario en una tabla. 5. El cliente elige el paciente del que quiere mostrar sus datos de movimiento presionando el botón adecuado, ya que cada paciente tiene su propio botón para mostrar sus datos. 6. El navegador se posiciona en la pestaña de Datos. 7. El usuario selecciona un archivo, que solo podrá ser del tipo CSV, y una fecha de medición del set movimientos y presiona el botón de “Añadir datos”. 8. El navegador manda la petición para añadir los datos de movimiento al servidor. 9. El servidor añade el set de datos de movimientos al la base de datos persistiendo los cambios. |
| Post-condiciones: El nuevo set de datos tiene que ser mostrado inmediatamente en el listado de datos de movimientos, sin que el usuario tenga que refrescar la página. |

|  |
| --- |
| **CU7: Borrar datos de movimiento.** |
| El usuario borra un set de movimientos del paciente. |
| Actores implicados: Usuario/cliente. |
| Precondiciones: Acceso al listado de movimientos de un paciente. |
| Secuencia de acciones:   1. El cliente accede a la aplicación web introduciendo la dirección IP del servidor en su navegador. 2. El cliente accede a la sección de pacientes mediante la pestaña de pacientes. 3. El navegador solicita la lista de pacientes a la base de datos. 4. El cliente recibe la lista de pacientes y se la presenta al usuario en una tabla. 5. El cliente elige el paciente del que quiere mostrar sus datos de movimiento presionando el botón adecuado, ya que cada paciente tiene su propio botón para mostrar sus datos. 6. El navegador se posiciona en la pestaña de Datos. 7. El navegador solicita la lista de movimientos del paciente a la base de datos. 8. El navegador recibe la lista de movimientos y se la presenta al usuario en una tabla. 9. El usuario elige los movimientos a borrar y presiona el botón de borrado correspondiente, ya que cada set de movimientos tiene su botón de borrado. 10. El navegador pregunta si desea realizar esta operación. 11. El usuario acepta la operación, teniendo la opción de cancelarla. 12. El navegador manda la petición para borrar los datos de movimiento al servidor. 13. El servidor borra el set de datos de movimiento del paciente de la base de datos y persiste los cambios. |
| Post-condiciones: La nueva lista de movimientos, sin el set de movimientos borrado, tiene que ser mostrada inmediatamente en el listado de movimientos, sin que el usuario tenga que refrescar la página. |

|  |
| --- |
| **CU8: Mostar datos del paciente.** |
| El usuario muestra un movimientos del paciente. |
| Actores implicados: Usuario/cliente. |
| Precondiciones: Acceso al listado de movimientos de un paciente. |
| Secuencia de acciones:   1. El cliente accede a la aplicación web introduciendo la dirección IP del servidor en su navegador. 2. El cliente accede a la sección de pacientes mediante la pestaña de pacientes. 3. El navegador solicita la lista de pacientes a la base de datos. 4. El cliente recibe la lista de pacientes y se la presenta al usuario en una tabla. 5. El cliente elige el paciente del que quiere mostrar sus datos de movimiento presionando el botón adecuado, ya que cada paciente tiene su propio botón para mostrar sus datos. 6. El navegador se posiciona en la pestaña de Datos. 7. El navegador solicita la lista de movimientos del paciente a la base de datos. 8. El navegador recibe la lista de movimientos y se la presenta al usuario en una tabla. 9. El usuario elige el movimiento (Sagital, Coronal, Transversal) con su fecha asociada presionando en el botón adecuado. 10. El navegador muestra por pantalla un gráfico con los datos requeridos. |
| Post-condiciones: El navegador tiene que borrar el contenido del gráfico anterior (en el caso de que exista) y graficar los datos requeridos. |

|  |
| --- |
| **CU9: Mostar datos de evolución del paciente.** |
| El usuario muestra la evolución de un movimiento del paciente. |
| Actores implicados: Usuario/cliente. |
| Precondiciones: Acceso al listado de movimientos de un paciente. |
| Secuencia de acciones:   1. El cliente accede a la aplicación web introduciendo la dirección IP del servidor en su navegador. 2. El cliente accede a la sección de pacientes mediante la pestaña de pacientes. 3. El navegador solicita la lista de pacientes a la base de datos. 4. El cliente recibe la lista de pacientes y se la presenta al usuario en una tabla. 5. El cliente elige el paciente del que quiere mostrar sus datos de movimiento presionando el botón adecuado, ya que cada paciente tiene su propio botón para mostrar sus datos. 6. El navegador se posiciona en la pestaña de Datos. 7. El navegador solicita la lista de movimientos del paciente a la base de datos. 8. El navegador recibe la lista de movimientos y se la presenta al usuario en una tabla. 9. El usuario elige la evolución del movimiento (Sagital, Coronal, Transversal) que quiere mostrar presionando el botón “Evolución del movimiento”. 10. El navegador muestra por pantalla un gráfico con los datos requeridos |
| Post-condiciones: El navegador tiene que borrar el contenido del gráfico anterior (en el caso de que exista) y graficar los datos requeridos. |



## 3.4. Matriz de trazabilidad

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Casos de Uso | RF1 | RF2 | RF3 | RF4 | RF5 | RF6 | RF7 | RF8 | RF9 | RNF1 | RNF2 | RNF3 | RNF4 | RNF5 |
| CU1 | ✓ |  |  |  |  |  |  |  |  | ✓ |  |  |  |  |
| CU2 |  | ✓ |  |  |  |  |  |  |  |  |  |  |  |  |
| CU3 |  |  | ✓ |  |  |  |  |  |  |  |  |  | ✓ | ✓ |
| CU4 |  |  |  | ✓ |  |  |  |  |  |  |  |  | ✓ | ✓ |
| CU5 |  |  |  |  | ✓ |  |  |  |  |  |  |  |  |  |
| CU6 |  |  |  |  |  | ✓ |  |  |  |  |  |  | ✓ | ✓ |
| CU7 |  |  |  |  |  |  | ✓ |  |  |  | ✓ |  | ✓ | ✓ |
| CU8 |  |  |  |  |  |  |  | ✓ |  |  |  |  |  |  |
| CU9 |  |  |  |  |  |  |  |  | ✓ |  |  | ✓ |  |  |

## 3.5. Arquitectura del sistema

### 3.5.1. Diseño visual (Storyboard) de la aplicación web

#### 3.5.1.1 Sección de Inicio



En la sección de inicio encontramos una breve descripción de cómo funciona la página web. Desde esta sección podemos navegar a la sección de pacientes o de datos, simplemente haciendo clic en la pestaña correspondiente.

#### 3.5.1.2 Sección de Pacientes



En la sección de pacientes encontramos un listado de pacientes con el nombre, apellidos, un botón para acceder a los datos del paciente y otro botón para borrar al paciente. También existe la posibilidad de añadir pacientes, introduciendo su nombre, apellidos y sexo.

#### 3.5.1.3 Sección de Datos



En la sección de datos tenemos un listado de conjuntos de movimientos (movimientos en los planos Coronal, Transversal y Sagital) con una fecha de medición asociada a cada conjunto. A demás es posible añadir un conjunto de datos con su fecha asociada y borrar un conjunto de datos.

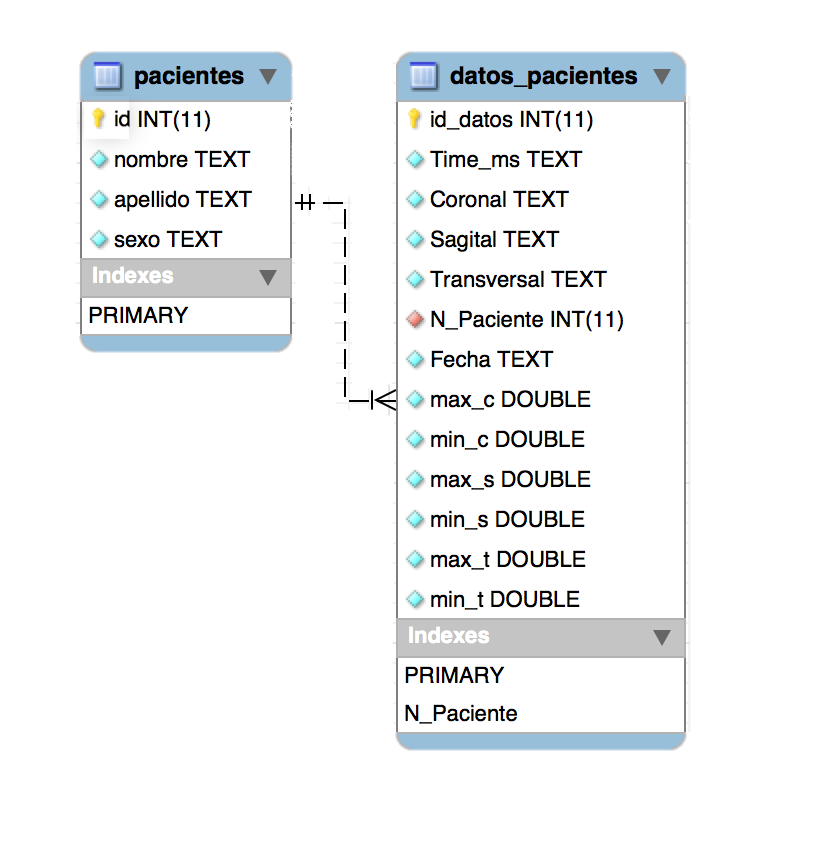
### 3.5.2 Esquema del modelo de datos

#### 3.5.2.1 Modelo de datos: EER

El modelo de Entidad-Relación es una herramienta de análisis usada para crear bases de datos que permite representar las entidades de una base de datos así como la relación entre estas entidades y sus propiedades.

Una entidad es un tipo de objeto que existe en el mundo real y cada entidad esta compuesta por una o más propiedades. La relación denota como dos entidades están relacionadas y existen tres tipos de relaciones:

* Uno a uno.
* Uno a muchos.
* Muchos a muchos.



La base de datos está formada por dos tablas o entidades relacionadas (Uno a muchos), lo que quiere decir que por cada paciente podrán existir varios sets de datos. En este tipo de relaciones siempre existe un Foreign Key, que es la clave que relaciona el set de datos con el paciente.

#### 3.5.2.2. Tabla de pacientes

La tabla de pacientes consiste de los siguientes elementos o propiedades:

**Id**: Identificador único (“PRIMARY KEY”) para distinguir a cada paciente que consiste en un tipo de dato entero. Este dato también es del tipo “AUTOINCREMENT” que permite que al introducir un paciente en la base de datos se le asigne id automáticamente que no exista y se le asignará el valor más alto de id existente mas uno. Este dato tiene asignada la característica “NOT NULL” para que no se permita que el campo esté vacío.

**Nombre:** Campo de tipo texto que corresponde al nombre del paciente. Este dato tiene asignada la característica “NOT NULL” para que no se permita que el campo esté vacío.

**Apellido:** Campo de tipo texto que corresponde a los apellidos del paciente. Este dato tiene asignada la característica “NOT NULL” para que no se permita que el campo esté vacío.

**Sexo:** Campo de tipo texto que corresponde al sexo del paciente. Contiene uno de dos valores posibles (h o m). Este dato tiene asignada la característica “NOT NULL” para que no se permita que el campo esté vacío.

#### 3.5.2.3. Tabla de datos de pacientes

La tabla de datos de los pacientes consiste de los siguientes elementos o propiedades:

**Id\_datos:** Identificador único (“PRIMARY KEY”) para distinguir a cada sesión de datos que consiste en un tipo de dato entero. Este dato también es del tipo “AUTOINCREMENT” que permite que al introducir un paciente en la base de datos se le asigne id automáticamente que no exista y se le asignará el valor más alto de id existente mas uno. Este dato tiene asignada la característica “NOT NULL” para que no se permita que el campo esté vacío.

**Time\_ms:** Campo de tipo texto que corresponde a un array de valores de instantes de tiempo en milisegundos, que corresponden a una sesión de movimiento. Para la base de datos, este campo es una cadena de texto muy larga, pero cuando la aplicación extrae estos datos de la base de datos, los separa y los almacena en un array. Este dato tiene asignada la característica “NOT NULL” para que no se permita que el campo esté vacío.

**Coronal:** Campo de tipo texto que corresponde a un array de valores que corresponden a una medida en grados del movimiento cervical en el plano coronal, que corresponden a una sesión de movimiento. Para la base de datos, este campo es una cadena de texto muy larga, pero cuando la aplicación extrae estos datos de la base de datos, los separa y los almacena en un array. Este dato tiene asignada la característica “NOT NULL” para que no se permita que el campo esté vacío.

**Sagital:** Campo de tipo texto que corresponde a un array de valores que corresponden a una medida en grados del movimiento cervical en el plano sagital, que corresponden a una sesión de movimiento. Para la base de datos, este campo es una cadena de texto muy larga, pero cuando la aplicación extrae estos datos de la base de datos, los separa y los almacena en un array. Este dato tiene asignada la característica “NOT NULL” para que no se permita que el campo esté vacío.

**Transversal:** Campo de tipo texto que corresponde a un array de valores que corresponden a una medida en grados del movimiento cervical en el plano transversal, que corresponden a una sesión de movimiento. Para la base de datos, este campo es una cadena de texto muy larga, pero cuando la aplicación extrae estos datos de la base de datos, los separa y los almacena en un array. Este dato tiene asignada la característica “NOT NULL” para que no se permita que el campo esté vacío.

**N\_Paciente:** Este dato corresponde al mismo que id en la tabla de pacientes, por lo tanto es del mismo tipo y tiene los mismos atributos, excepto “AUTOINCREMENT” y “PRIMARY KEY” ya que para la tabla en cuestión este datos se puede repetir debido a que para cada paciente único pueden existir varias sesiones de movimientos.

**Fecha:** Campo de tipo texto que corresponde a la fecha de una sesión de movimiento. Este dato tiene asignada la característica “NOT NULL” para que no se permita que el campo esté vacío.

**Max\_c:** Campo de tipo punto flotante que corresponde al ángulo máximo alcanzado para el rango de movimiento en el plano coronal. Este dato tiene asignada la característica “NOT NULL” para que no se permita que el campo esté vacío.

**Min\_c:** Campo de tipo punto flotante que corresponde al ángulo mínimo alcanzado para el rango de movimiento en el plano coronal. Este dato tiene asignada la característica “NOT NULL” para que no se permita que el campo esté vacío.

**Max\_s:** Campo de tipo punto flotante que corresponde al ángulo máximo alcanzado para el rango de movimiento en el plano sagital. Este dato tiene asignada la característica “NOT NULL” para que no se permita que el campo esté vacío.

**Min\_s:** Campo de tipo punto flotante que corresponde al ángulo mínimo alcanzado para el rango de movimiento en el plano sagital. Este dato tiene asignada la característica “NOT NULL” para que no se permita que el campo esté vacío.

**Max\_t:** Campo de tipo punto flotante que corresponde al ángulo máximo alcanzado para el rango de movimiento en el plano transversal. Este dato tiene asignada la característica “NOT NULL” para que no se permita que el campo esté vacío.

**Min\_t:** Campo de tipo punto flotante que corresponde al ángulo mínimo alcanzado para el rango de movimiento en el plano transversal. Este dato tiene asignada la característica “NOT NULL” para que no se permita que el campo esté vacío.

### 3.5.3 Estructura del archivo CSV

La utilización de los archivos CSV es fundamental para el desarrollo de esta aplicación web, ya que el dispositivo Werium Basic Pro, al realizar un medición del rango de movimiento, genera un archivo CSV del cual hay que extraer los datos para que posteriormente se almacenen en la base de datos.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Time\_ms | Coronal | Sagital | Transversal | 1 | 2 |
| 266 | -0.09740291299729731 | 0.16478853427653659 | 0.17688254772586492 | #DCM=0.38,0.43,-0.82,-0.13,0.90,0.41,0.92,-0.05,0.40,303081 | #DCM=-0.31,0.01,0.95,-0.02,-1.00,0.01,0.95,-0.02,0.31,240617 |
| 287 | -0.2750210837754741 | 0.16478853427653659 | -0.36713315963460835 | #DCM=0.38,0.43,-0.82,-0.13,0.90,0.41,0.92,-0.05,0.40,303081 | #DCM=-0.31,0.02,0.95,-0.02,-1.00,0.01,0.95,-0.02,0.31,240658 |
| 307 | -0.2750210837754741 | 0.16478853427653659 | -0.36713315963460835 | #DCM=0.38,0.43,-0.82,-0.13,0.90,0.41,0.92,-0.05,0.40,303081 | #DCM=-0.31,0.02,0.95,-0.02,-1.00,0.01,0.95,-0.02,0.31,240699 |

La primera tabla muestra el contenido original del archivo CSV. La tabla siguiente muestra cómo se almacenarían los datos en la base de datos, donde se guardan los números redondeados a dos números decimales para que la sesión de movimiento no ocupe mucho espacio. Aunque pueda parecer que se pierde exactitud, no ocurre así, ya que a la hora de mostrar los gráficos, nos encontramos valores de entre 50 y -50 grados en los distintos rangos de movimientos, por lo que un cambio de 0.001 grados no se aprecia en absoluto en los gráficos.

|  |  |  |  |
| --- | --- | --- | --- |
| Time\_ms | Coronal | Sagital | Transversal |
| 266 | -0,10 | 0,16 | 0,18 |
| 287 | -0,10 | 0,16 | -0,37 |
| 307 | -0,10 | 0,16 | -0,37 |
| 328 | -0,10 | 0,16 | -0,37 |
| 348 | -0,10 | 0,16 | -0,37 |
| 369 | -0,10 | 0,16 | -0,37 |
| 389 | -0,10 | 0,16 | -0,37 |
| 410 | -0,10 | 0,16 | -0,37 |
| 430 | -0,10 | 0,16 | -0,37 |
| 451 | -0,10 | 0,16 | -0,37 |
| 471 | -0,10 | 0,16 | -0,37 |
| 492 | -0,10 | 0,16 | -0,37 |
| 512 | -0,10 | 0,17 | -0,60 |
| 533 | -0,10 | 0,17 | -0,60 |
| 553 | -0,10 | 0,17 | -0,05 |

El archivo CSV es un documento de texto delimitado por un punto y coma (semicolon) que divide los datos en tiempo, movimiento coronal, movimiento sagital, movimiento transversal y datos adicionales que ayudan a calcular los previamente dichos. Éstos últimos datos no serán utilizados ya que no son necesarios. De una forma simplificada podemos ver este archivo como un listado de columnas donde se encuentran los datos de tiempo (valores enteros) y valores del rango de movimiento en los planos coronal, sagital y transversal (valores de punto flotante). La primera fila contiene los nombres de cada columna.

# 4. Implementación

En este capítulo se explica paso a paso el proceso de implementación que se ha llevado a cabo para el sistema de acuerdo a los requisitos funcionales y no funcionales establecidos. A continuación se mostrarán algunos extractos de código y ficheros para que el lector pueda comprender en detalle los procesos y funcionalidades implementadas.

## 4.1. Comunicación Cliente-Servidor

Para la comunicación entre el cliente/navegador y el servidor hemos utilizado sockets.io para establecer una comunicación bidireccional. A continuación se puede ver cómo he implementado estos sockets en mi aplicación web.

### 4.1.1. Servidor

Para establecer una conexión entre el cliente y el servidor, la parte del servidor realiza los siguientes pasos:

1. Solicitar las librerías correspondientes para que el servidor haga uso de sus funcionalidades (express, socket.io, http):

|  |
| --- |
| **var** express **=** require**(**'express'**);**  **var** http **=** require**(**'http'**);**  **var** io **=** require**(**'socket.io'**);** |

2. Utilizando el middleware estático, especificamos la carpeta pública del servidor para disponer de las páginas HTML. Estas páginas corresponden al código que muestra el diseño visual de la aplicación web al usuario. Este diseño se ha mostrado en el apartado 3.5. En mi caso el código HTML (index.html, paciente.html y evolución.html) se encuentra en la carpeta pagina\_web.

|  |
| --- |
| **var** app **=** express**();**  app**.**use**(**express**.static(**'./../pagina\_web'**));** |

3. Creamos una instancia “server” con un puerto asociado (8124) y una dirección IP que corresponde a la dirección que se asocia a mi terminal. La combinación de dirección IP y puerto será la que el usuario tendrá que introducir en su navegador (192.168.1.41:8124) para acceder a la aplicación web. Con la instancia creada se iniciara una comunicación websocket pasando la instancia como parámetro.

|  |
| --- |
| **var** server **=** http**.**createServer**(**app**).**listen**(8124,** '172.20.10.5'**);**  io **=** io**.**listen**(**server**);** |

4. Asociamos una función callback que se ejecutará cuando el cliente visite la página y establezca un websocket. A partir de aquí el servidor estará escuchando permanentemente a peticiones del cliente y mandará respuestas según lo que se solicite.

|  |
| --- |
| io**.**sockets**.**on**(**"connection"**,function(**socket**){** |

### 4.1.2. Cliente

La parte del cliente es muy sencilla ya que solo tiene que inicializar una conexión que activará la función callback del servidor.

|  |
| --- |
| **var** socket **=** io**.**connect**(**"http://172.20.10.5:8124"**);** |

A partir de aquí el cliente utilizará la instancia creada “socket” para enviar datos al servidor o recibir datos del servidor.

### 4.1.3 Despliegue del servidor

Para que la comunicación entre el cliente y el servidor funcione es necesario ejecutar el código del servidor para que empiece a escuchar peticiones de clientes.

Es un paso muy sencillo que requiere posicionarse en el terminal de la máquina donde se encuentre el código del servidor y cambiar al directorio donde se encuentre el fichero JavaScript ejecutable.



## 4.2. SQLite

### 4.2.1. Compatibilidad con el Servidor

Para que el servidor pueda hacer un uso adecuado de SQLite necesitará un script llamado sql.js, que es el que le permitirá manejar la base de datos de los pacientes.

#### 4.2.1.1. Sql.js

Sql.js es un puerto de SQLite a JavaScript. Utiliza una base de datos virtual que se almacena en memoria y por los tanto los cambios realizados a la base de datos no se persisten. Lo interesante y que es muy funcional para este desarrollo es que permite importar cualquier fichero SQLite y exportar la base de datos creada como un array escrito en JavaScript. Usaremos el array exportado para poder persistir los cambios realizados con la ayuda de node.js. Cada vez que se realiza un cambio en la base de datos (añadir/borrar paciente, añadir/borrar sesión de movimiento) se persistirán los cambios, y no se esperará a que el usuario cierre la sesión, si no que al realizar cualquiera de las operaciones mencionadas se persistirán los cambios.

#### 4.2.1.2. Uso de sql.js con node.js

Todas las operaciones que se realizan para manejar la base de datos (consultar, borrar, ordenar o insertar) requieren de una lectura de la base de datos y/o de una operación para guardar los cambios realizados en la base de datos.

1. Lectura de una base de datos de memoria:



1. Escritura de una base de datos en memoria:

Hace falta convertir el resultado de db.export a un buffer



## 4.3. Funciones

A continuación expondré el funcionamiento de las distintas funciones que piden, añaden, borran u organizan la base de datos. Todas estas funciones se realizan gracias a la comunicación entre el navegador y el servidor, que utilizan sockets para establecer una comunicación bidireccional, así como node.js para pedir datos a la base de datos o para persistir cambios en la base de datos.

### 4.3.1. Obtener pacientes

Para que el navegador obtenga la lista de pacientes de la base de datos es necesario que se realizar una conexión bidireccional con el servidor y que desde el servidor se haga una consulta a la base de datos, enviando el resultado de la consulta al navegador mediante el websocket establecido. Todo esto ha sido explicado en los apartados anteriores. En este apartado veremos cómo se realizan todas las operaciones en conjunto, además de las consultas de SQL.

#### 4.3.1.1. Funcionalidad en el lado del cliente

Cuando el usuario entra en la sección de pacientes, se genera una tabla y se llama a una función que devuelve un array de pacientes. Utilizando un bucle for que recorre el array rellenamos la tabla.

|  |
| --- |
| **<script** type="text/javascript"**>**  **var** tabla **=** document**.**getElementById**(**"miTabla"**);**  get\_paciente\_node**(function(**paciente**){**  **var** tabla **=** document**.**getElementById**(**"miTabla"**);**  **for(**i**=0;** i**<(**paciente**.**length**);** i**++){**  **var** filas **=** tabla**.**rows**.**length**;**  **var** fila **=** tabla**.**insertRow**(**filas**);**  fila**.**insertCell**(0).**innerHTML **=** '<button class="btn\_borrar" type="button" onClick="borrar\_paciente('**+**paciente**[**i**][0]+**',\'' **+** paciente**[**i**][1]** **+** '\')"></button>'**;**  fila**.**insertCell**(0).**innerHTML **=** '<button class="btn" type="button" onClick="datos('**+**paciente**[**i**][0]+**',\'' **+** paciente**[**i**][1]** **+** '\',\'' **+** paciente**[**i**][2]** **+** '\',\'' **+** paciente**[**i**][3]** **+** '\')"></button>'**;**  fila**.**insertCell**(0).**innerHTML **=** paciente**[**i**][2];**  fila**.**insertCell**(0).**innerHTML **=** paciente**[**i**][1];**  **}**  console**.**log**(**"Lista de pacientes disponible en el navegador"**);**  **});**  **</script>** |

La función “get\_paciente\_node()” es la que se encarga de establecer la conexión con el servidor a través de un websocket.

|  |
| --- |
| **function** get\_paciente\_node**(**callback**){**  **var** socket **=** io**.**connect**(**"http://172.20.10.5:8124"**);**  console**.**log**(**"Conexíon establecida con el servidor"**);**  socket**.**on**(**"message"**,function(**message**){**  message **=** JSON**.**parse**(**message**);**  *//console.log(message.data);*  **});**  **var** datos5 **=** **{**  operacion**:** "Pacientes"  **}**  socket**.**send**(**JSON**.**stringify**(**datos5**));**  console**.**log**(**"Solicitud de listado de pacientes enviada"**);**  socket**.**on**(**"pacientes"**,** **function** **(**data**)** **{**  console**.**log**(**"Lista de pacientes recibida"**);**  callback**(**data**);**  **});**  **}** |

Como se mencionó en la sección 4.1.2, el cliente utiliza la instancia socket para escuchar al servidor a través de socket.on y para enviar mensajes a través de socket.send(). Todas las solicitudes al servidor se realizan enviando un JSON con una “cabecera” llamada operación que permitirá al servidor distinguir entre las distintas operaciones, y así devolver unos datos específicos. En este caso la cabecera contiene la operación Pacientes.

#### 4.3.1.2. Funcionalidad en el lado del servidor

Ahora podremos ver como el servidor maneja la solicitud para enviar el listado de pacientes al cliente.

|  |
| --- |
| socket**.**on**(**"message"**,function(**info**){**  datos **=** JSON**.**parse**(**info**);**  console**.**log**(**timestamp**(**'hh:mm:ss:iii'**)+**" Petición del cliente: "**+**datos**.**operacion**);**  **if** **(**datos**.**operacion **==** "Pacientes"**){**  **var** filebuffer **=** fs**.**readFileSync**(**'./Pacientes\_DB.db'**);**  **var** db **=** **new** SQL**.**Database**(**filebuffer**);**  console**.**log**(**timestamp**(**'hh:mm:ss:iii'**)+**" Base de datos abierta"**);**  **var** pacientes **=** db**.**exec**(**"SELECT \* FROM pacientes"**);**  socket**.**emit**(**"pacientes"**,**pacientes**[0].**values**);**  console**.**log**(**timestamp**(**'hh:mm:ss:iii'**)+**" Listado de pacientes enviado al cliente"**);**  db**.**close**();**  console**.**log**(**timestamp**(**'hh:mm:ss:iii'**)+**" Base de datos cerrada"**);**  **}** |

La variable datos es la que contiene el JSON con la cabecera que ha enviado el navegador. El servidor comprueba si esta cabecera es equivalente a pacientes y así reconoce la petición del cliente. Podemos observar la operación de lectura de la base de datos descrita en el apartado 4.2.2.2. Al tener acceso a la base de datos se realiza una consulta para obtener los pacientes mediantes db.exec(). El resultado de esta consulta (array de pacientes) se envía al cliente mediante el websocket con la operación socket.emit() y el cliente muestra los pacientes en la tabla.

### 4.3.2. Borrar Paciente

Para borrar un paciente seguiremos una dinámica similar a la de añadir un paciente. Cambiará la cabecera de la operación y el comando SQL que en vez de extraer datos de la base de datos borrará datos de la tabla de pacientes y de la tabla de datos de paciente, ya que, si un paciente tiene datos de movimientos y lo borramos, todos sus datos de movimientos asociados serán eliminados.

#### 4.3.2.1. Funcionalidad en el lado del cliente

Cuando el usuario entra en la sección de pacientes, se genera una tabla y se llama a una función que devuelve un array de pacientes. Utilizando un bucle for que recorre el array rellenamos la tabla. Dentro de cada fila se rellenan varias elementos, incluyendo nombre, apellidos, botón de acceso a los datos y botón de borrado. El usuario presionara el botón de borrado del paciente que quiera borrar.

|  |
| --- |
| var filas = tabla.rows.length;  var fila = tabla.insertRow(filas);  fila.insertCell(0).innerHTML = '**<button** class="btn\_borrar" type="button" onClick="borrar\_paciente('+paciente[i][0]+',\'' + paciente[i][1] + '\')"**></button>**'; |

En este trozo de código vemos que una de las filas insertadas es un botón y cuando se presiona sobre este botón se llama a una función llamada borrar\_paciente() a la que le pasamos el id del paciente y el nombre.

|  |
| --- |
| **function** borrar\_paciente**(**N\_p**,**nombre**){**  **var** y **=** confirm**(**"¿Esta seguro de que quiere borrar a este paciente?. Al eliminar un paciente borrara todos sus datos asociados."**);**  **if** **(**y **==** **true){**  **var** socket **=** io**.**connect**(**"http://172.20.10.5:8124"**);**  console**.**log**(**"Conexíon establecida con el servidor"**);**  socket**.**on**(**"message"**,function(**message**){**  console**.**log**(**"El servidor ha enviado un mensaje:"**);**  message **=** JSON**.**parse**(**message**);**  *//console.log(message.data);*  **});**  **var** data **=** **{**  operacion**:** "Borrar paciente"**,**  id**:** N\_p**,**  n**:** nombre  **}**  socket**.**send**(**JSON**.**stringify**(**data**));**  console**.**log**(**"Solicitud para borrar paciente: ("**+**data**.**n**+**") enviada"**);**  **}**  **else{**  console**.**log**(**"Datos no borrados"**);**  **}**  **}** |

En este caso le salta un mensaje de confirmación al usuario para asegurar que quiera borrar al paciente y sus datos de movimientos asociados. Si confirma la operación se establecerá un websocket con el servidor donde el navegador escuchará al servidor para recibir la confirmación de conexión. Después el navegador enviará un objeto JSON con la cabecera “Borrar paciente”, el identificador único del paciente y su nombre.

Para este caso es necesario que cuando borremos el paciente se actualice la tabla con los pacientes y así no se muestre el paciente borrado. Para realizar esta función el servidor manda un mensaje que el cliente recibe y le sirve para saber cuando tiene que actualizar el contenido de la tabla.

|  |
| --- |
| **var** socket **=** io**.**connect**(**"http://172.20.10.5:8124"**);**  socket**.**on**(**"reload"**,** **function** **(**data**)** **{**  location**.**reload**();**  **});** |

Este código se encuentra en el navegador y espera a que el servidor confirme que se han borrado los datos. Una vez recibida la confirmación, el navegador realiza un location.reload() que básicamente actualiza la página web, lo que obliga a que se vuelva a llamar a la función que lista los pacientes. En esta ocasión se mostrará la lista de pacientes actualizada.

#### 4.3.2.2. Funcionalidad en el lado del servidor

|  |
| --- |
| socket**.**on**(**"message"**,function(**info**){**  datos **=** JSON**.**parse**(**info**);**  **function** borrar\_paciente**(**N\_p**,**nombre**){**  **var** y **=** confirm**(**"¿Esta seguro de que quiere borrar a este paciente?. Al eliminar un paciente borrara todos sus datos asociados."**);**  **if** **(**y **==** **true){**  **var** socket **=** io**.**connect**(**"http://172.20.10.5:8124"**);**  console**.**log**(**"Conexíon establecida con el servidor"**);**  socket**.**on**(**"message"**,function(**message**){**  console**.**log**(**"El servidor ha enviado un mensaje:"**);**  message **=** JSON**.**parse**(**message**);**  *//console.log(message.data);*  **});**  **var** data **=** **{**  operacion**:** "Borrar paciente"**,**  id**:** N\_p**,**  n**:** nombre  **}**  socket**.**send**(**JSON**.**stringify**(**data**));**  console**.**log**(**"Solicitud para borrar paciente: ("**+**data**.**n**+**") enviada"**);**  **}**  **else{**  console**.**log**(**"Datos no borrados"**);**  **}**  **}** |

Como vimos para el caso para añadir un paciente el servidor escucha mediante socket.on(). Cuando recibe un mensaje comprueba su cabecera y al reconocer la operación “Borrar Paciente” realiza la conexión con la base de datos y elimina al paciente y cualquier set de movimientos que tenga. A continuación guarda los cambios y cierra la base de datos. Como hemos dicho en la parte del cliente, cuando se realizan los cambios el servidor ejecuta la función io.sockets.emit(). Básicamente fuerza a cualquier cliente conectado a actualizar su lista de clientes, por lo que si hay varios dispositivos conectados, en todos estos se actualizarán los cambios.

### 4.3.3. Añadir un Paciente

Para este caso, existirá un procesa similar al de borrar un paciente. Cambiará la cabecera u operación enviada al servidor y el comando SQL que añadirá datos a la base de datos.

#### 4.3.3.1. Funcionalidad en el lado del cliente

Para poder añadir un paciente, el usuario tendrá que posicionarse en la pestaña de pacientes y rellenar el formulario que se encuentra justo después del listado de pacientes.

|  |
| --- |
| **<p>**Para añadir a un paciente rellene el formulario y haga click en 'Añadir':**</p><br>**  **<p1>**Nombre: **</p1>**  **<input** type="text" id="nombre" placeholder="Nombre del Paciente"**><br>**  **<p1>**Apellido: **</p1>**  **<input** type="text" id="apellido" placeholder="Apellidos del Paciente"**><br>**  **<p1>**Sexo:&nbsp;&nbsp;&nbsp;&nbsp; **</p1>**  **<input** type="radio" name="sexo" value="h" id="sexo1"**>** Hombre  **<input** type="radio" name="sexo" value="m" id="sexo2"**>** Mujer**<br>**  **<script** type="text/javascript"**>**  **function** Validar**(){**  **if(**document**.**getElementById**(**"sexo1"**).**checked**){**  **var** sexo **=** document**.**getElementById**(**"sexo1"**).**value**;**  **}**  **else** **if(**document**.**getElementById**(**"sexo2"**).**checked**){**  **var** sexo **=** document**.**getElementById**(**"sexo2"**).**value**;**  **}**  **else{**  alert**(**"Añada un sexo"**);**  **return;**  **}**  **if(**document**.**getElementById**(**"nombre"**).**value **==** "" **||** document**.**getElementById**(**"apellido"**).**value **==** ""**){**  alert**(**"Rellene nombre y apellidos"**);**  **return;**  **}**  save\_paciente**(**document**.**getElementById**(**"nombre"**).**value**,** document**.**getElementById**(**"apellido"**).**value**,**sexo**);**  **}**  **</script>**  **<input** type="button" onclick="Validar()" value="Añadir"**><br><br>** |

El formulario solicitará un nombre, apellidos y el sexo del paciente. Cuando se presione el botón añadir se llamará a una función “Validar()” que comprobará si hay algún campo vacío. En ese caso saltará un mensaje solicitando al usuario que rellene el campo que no ha rellenado. Cuando todos los campos estén adecuadamente rellenos y el usuario pulse “Añadir” se llamará a una función llamada “save\_paciente()” a la que se le pasarán los datos requeridos.

|  |
| --- |
| **function** save\_paciente**(**nombre**,** apellido**,** sexo**){**  **var** socket **=** io**.**connect**(**"http://172.20.10.5:8124"**);**  console**.**log**(**"Conexíon establecida con el servidor"**);**  socket**.**on**(**"message"**,function(**message**){**  console**.**log**(**"El servidor ha enviado un mensaje:"**);**  message **=** JSON**.**parse**(**message**);**  *//console.log(message.data);*  **});**  **var** data **=** **{**  operacion**:** "Añadir paciente"**,**  n**:** nombre**,**  a**:** apellido**,**  s**:** sexo  **}**  socket**.**send**(**JSON**.**stringify**(**data**));**  console**.**log**(**"Solicitud para añadir paciente: ("**+**data**.**n**+**" "**+**data**.**a**+**") enviada"**);**  **}** |

Aquí vemos un código muy similar al de los apartados anteriores, donde se establece un websocket con el servidor y se escucha un mensaje entrante de este. Después se envía al servidor un objeto JSON que contiene la cabecera “Añadir paciente” y los datos del paciente (nombre, apellidos y sexo). Como sabemos la cabecera es útil para que el servidor pueda distinguir las distintas operaciones que el navegador le pide.

Para este caso es necesario que cuando añadamos al paciente se actualice la tabla de pacientes y así se mostrará la tabla de paciente con el nuevo paciente añadido. Para realizar esta función el servidor manda un mensaje que el cliente recibe y le sirve para saber cuando tiene que actualizar el contenido de la tabla.

|  |
| --- |
| **var** socket **=** io**.**connect**(**"http://172.20.10.5:8124"**);**  socket**.**on**(**"reload"**,** **function** **(**data**)** **{**  location**.**reload**();**  **});** |

Este código se encuentra en el navegador y espera a que el servidor confirme que se han borrado los datos. Una vez recibida la confirmación, el navegador realiza un location.reload() que básicamente actualiza la página web, lo que obliga a que se vuelva a llamar a la función que lista los pacientes. En esta ocasión se mostrará la lista de pacientes actualizada.

#### 4.3.3.2. Funcionalidad en el lado del servidor

|  |
| --- |
| socket**.**on**(**"message"**,function(**info**){**  datos **=** JSON**.**parse**(**info**);**  **if(**datos**.**operacion **==** "Añadir paciente"**){**  console**.**log**(**timestamp**(**'hh:mm:ss:iii'**)+**" Paciente a añadir: "**+**datos**.**n**+**" "**+**datos**.**a**);**  **var** filebuffer **=** fs**.**readFileSync**(**'./Pacientes\_DB.db'**);**  **var** db **=** **new** SQL**.**Database**(**filebuffer**);**  console**.**log**(**timestamp**(**'hh:mm:ss:iii'**)+**" Base de datos abierta"**);**  db**.**run**(**"INSERT INTO pacientes VALUES (:id, :nombre, :apellido, :sexo)"**,** **{**':nombre'**:**datos**.**n**,** ':apellido'**:**datos**.**a**,**':sexo'**:**datos**.**s**});**  console**.**log**(**timestamp**(**'hh:mm:ss:iii'**)+**" Se ha añadido al paciente "**+**datos**.**n**+**" "**+**datos**.**a**+**" a la base de datos"**);**  **var** data **=** db**.export();**  **var** buffer **=** **new** Buffer**(**data**);**  fs**.**writeFileSync**(**"./Pacientes\_DB.db"**,** buffer**);**  db**.**close**();**  console**.**log**(**timestamp**(**'hh:mm:ss:iii'**)+**" Base de datos cerrada"**);**  **var** ack\_to\_client **=** **{**  data**:**"El servidor ha añadido un paciente a la db"  **}**  socket**.**send**(**JSON**.**stringify**(**ack\_to\_client**));**  io**.**sockets**.**emit**(**"reload"**,{});**  **}** |

Como vimos para el caso para borrar un paciente el servidor escucha mediante socket.on(). Cuando recibe un mensaje comprueba su cabecera y al reconocer la operación “Añadir paciente” realiza la conexión con la base de datos y realiza un INSERT pasando los datos del paciente a añadir como parámetros. A continuación guarda los cambios y cierra la base de datos. Como hemos dicho en la parte del cliente, cuando se realizan los cambios el servidor ejecuta la función io.sockets.emit(). Básicamente fuerza a cualquier cliente conectado a actualizar su lista de clientes, por lo que si hay varios dispositivos conectados, en todos estos se actualizarán los cambios.

### 4.3.4. Obtener datos de movimiento de un paciente

Para obtener estos datos seguiremos con la dinámica que se ha visto en las funciones anteriores (Conexión cliente-servidor/intercambio de datos entre cliente-servidor/apertura y cierre de la base de datos).

#### 4.3.4.1. Funcionalidad en el lado del cliente

En primer lugar, el usuario deberá posicionarse en la pestaña de pacientes y elegir el paciente del que quiere mostrar los datos de movimientos.

|  |
| --- |
| fila.insertCell(0).innerHTML = '**<button** class="btn" type="button" onClick="datos('+paciente[i][0]+',\'' + paciente[i][1] + '\',\'' + paciente[i][2] + '\',\'' + paciente[i][3] + '\')"**></button>**'; |

Cuando se genera la tabla de pacientes con el bucle for, una de las filas insertadas es un botón que permite el acceso a los datos de movimientos. Cuando el usuario presiona este botón se llama a una función “datos()” a la que se le pasan el id, nombre y apellidos del paciente.

|  |
| --- |
| **function** datos**(**id**,**nombre**,**apellido**,**sexo**){**  window**.**location**.**href **=** "http://172.20.10.5:8124/../evolucion.html?var1="**+**id**+**"&var2="**+**nombre**+**"&var3="**+**apellido**+**"&var4="**+**sexo**;**  **}** |

La función datos posiciona al usuario en la pestaña de datos y pasa los datos del paciente a través de la URL. Así desde la pestaña de datos el navegador tendrá acceso a los datos del paciente del que tiene que mostrar los datos de movimientos.

|  |
| --- |
| document.getElementById("paciente\_info").innerHTML = "Paciente: "+nombre+" "+apellido;  get\_datos\_node(id, function(datos){  if (datos.length==0){  document.getElementById("paciente\_info\_1").innerHTML = "Todavía no hay datos de movimieto disponibles para este paciente";  }  else{  var tabla = document.getElementById("miTabla");  for(i=0; i<(datos[0].values.length); i++){  var filas = tabla.rows.length;  var fila = tabla.insertRow(filas);  fila.insertCell(0).innerHTML = '**<button** class="btn\_borrar" type="button" onClick="borrar\_datos('+datos[0].values[i][0]+',\''+nombre+'\')"**></button>**';  fila.insertCell(0).innerHTML = datos[0].values[i][6].substring(0,10)+" "+datos[0].values[i][6].substring(11,16);  fila.insertCell(0).innerHTML = '**<button** class="btn" type="button" onClick="crearGrafico(\''+datos[0].values[i][1]+'\', \'' + datos[0].values[i][4] + '\',1)"**></button>**';  fila.insertCell(0).innerHTML = '**<button** class="btn" type="button" onClick="crearGrafico(\''+datos[0].values[i][1]+'\', \'' + datos[0].values[i][3] + '\',2)"**></button>**';  fila.insertCell(0).innerHTML = '**<button** class="btn" type="button" onClick="crearGrafico(\''+datos[0].values[i][1]+'\', \'' + datos[0].values[i][2] + '\',3)"**></button>**';  }  var fila1 = tabla.insertRow(tabla.length);  fila1.insertCell(0).innerHTML = '**<button** class="bt" type="button" onClick="Evolucion(1)"**>**Evolución del movimieto**</button>**';  fila1.insertCell(0).innerHTML = '**<button** class="bt" type="button" onClick="Evolucion(2)"**>**Evolución del movimieto**</button>**';  fila1.insertCell(0).innerHTML = '**<button** class="bt" type="button" onClick="Evolucion(3)"**>**Evolución del movimieto**</button>**';  }  }); |

Una vez posicionados en la pestaña de datos, se mostrará el nombre y los apellidos del paciente y se generará una tabla con distintos botones que al ser presionados permitirán al usuario mostrar los datos de movimientos para distintas fechas además de poder mostrar la evolución de estos datos con el tiempo. En el caso de que el paciente no tenga datos de movimientos se mostrará un mensaje comunicándolo.

Los únicos datos que necesitaremos para la tabla de datos de movimientos serán las fechas para cada sesión de movimientos. Para obtener estos datos llamamos a la función “get\_datos\_node()” pasándole el id del paciente, que hemos obtenido del URL.

|  |
| --- |
| **function** get\_datos\_node**(**id\_p**,** callback**){**  **var** socket **=** io**.**connect**(**"http://172.20.10.5:8124"**);**  socket**.**on**(**"message"**,function(**message**){**  console**.**log**(**"El servidor ha enviado un mensaje:"**);**  message **=** JSON**.**parse**(**message**);**  console**.**log**(**message**);**  **});**  **var** datos1 **=** **{**  operacion**:** "Datos paciente"**,**  id**:** id\_p**,**  n**:** nombre  **}**  socket**.**send**(**JSON**.**stringify**(**datos1**));**  socket**.**on**(**"datos\_paciente"**,** **function** **(**data**)** **{**  callback**(**data**);**  **});**  **}** |

Observamos que se repite el proceso en el que se establece un websocket con el servidor, escuchando un mensaje de este y enviándole un objeto JSON con la cabecera “Datos paciente”. Se vuelve a escuchar al servidor mediante socket.on(), y se reciben los datos requeridos.

#### 4.3.4.2. Funcionalidad en el lado del servidor

|  |
| --- |
| socket**.**on**(**"message"**,function(**info**){**  datos **=** JSON**.**parse**(**info**);**  **if(**datos**.**operacion**==**"Datos paciente"**){**  console**.**log**(**"Mostrar datos de: "**+**datos**.**n**);**  **var** filebuffer **=** fs**.**readFileSync**(**'./Pacientes\_DB.db'**);**  **var** db **=** **new** SQL**.**Database**(**filebuffer**);**  console**.**log**(**timestamp**(**'hh:mm:ss:iii'**)+**" Base de datos abierta"**);**  **var** datos\_paciente **=** db**.**exec**(**"SELECT \* FROM datos\_pacientes WHERE N\_Paciente = "**+**datos**.**id**+**" ORDER BY datetime(FECHA) asc LIMIT (select count() from datos\_pacientes)"**);**  socket**.**emit**(**"datos\_paciente"**,**datos\_paciente**);**  db**.**close**();**  console**.**log**(**timestamp**(**'hh:mm:ss:iii'**)+**" Base de datos cerrada"**);**  **}** |

Volvemos a observar el mismo comportamiento que en los aparados anteriores donde el servidor distingue la cabecera enviada por el cliente y opera en consecuencia. En este caso el comando SQL pide los datos de la base de datos ordenados por fecha ya que a la hora de mostrar los datos de movimiento es conveniente mostrarlos ordenados por fecha, pero sobretodo a la hora de mostrar la evolución de los movimientos es esencial que los datos se muestren en progresión con el tiempo, ya que no tendría sentido un gráfico de evolución temporal sin orden cronológico.

### 4.3.5. Añadir datos de movimiento

Para esta función tendremos una funcionalidad similar a la de añadir un paciente. Cambiará la forma de introducir los datos, que en vez de introducirlos en el navegador se seleccionará un archivo local.

#### 4.3.5.1. Funcionalidad en el lado del cliente

|  |
| --- |
| **<button** id="boton\_x" type="button"**>**Añadir datos**</button>**  **<script** type="text/javascript"**>**  **var** Boton\_pres **=** document**.**getElementById**(**"boton\_x"**);**  **var** fileInput **=** document**.**getElementById**(**"csv"**);**  readFile **=** **function** **()** **{**  **var** reader **=** **new** FileReader**();**  reader**.**onload **=** **function** **()** **{**  Papa**.**parse**(**reader**.**result**,** **{**  complete**:** **function(**results**)** **{**  **if(!**document**.**getElementById**(**"miFecha"**).**value**){**  alert**(**"Introduzca la fecha"**);**  **}**  **else{**  **var** Fecha **=** document**.**getElementById**(**"miFecha"**).**value**;**  console**.**log**(**"Datos introducidos de forma correcta"**);**  add\_datos**(**results**.**data**,** Fecha**);**  **}**  **}**  **});**  **};**  **if** **(**fileInput**.**files**[0]** **instanceof** Blob**){**  reader**.**readAsBinaryString**(**fileInput**.**files**[0]);**  **}**  **else{**  alert**(**"Seleccione un archivo"**);**  **}**  **};**  Boton\_pres**.**addEventListener**(**"click"**,**readFile**);**  **</script>** |

Como ocurría en el apartado de añadir pacientes tenemos una sección debajo de la tabla de datos de movimientos donde se requiere un archivo y una fecha de medición de la sesión de movimientos. A la hora de añadir un archivo se le pide al usuario que seleccione un archivo local. Este archivo solo podrá ser del formato CSV ya que este archivo es el que genera el aparato utilizado para medir los movimientos (Werium Basic Pro). Cuando el usuario ha seleccionado el archivo adecuado utilizamos la tecnología paparse para transformar el archivo CSV en un JSON ya que JavaScript no maneja archivos de texto delimitados mientras que si puede manejar objetos JSON. Cuando el usuario presione el botón para añadir los datos se comprobará que hay un archivo seleccionado y una fecha introducida. Si es correcto se llamará a la función “add\_datos()”, pasándole como parámetros el archivo CSV en formato JSON y la fecha.

|  |
| --- |
| **function** add\_datos**(**datos**,**fecha**){**  **var** Time **=** **[];**  **var** Coronal **=** **[];**  **var** Sagital **=** **[];**  **var** Transversal **=** **[];**  **for(var** i **=** **1;** i **<** datos**.**length**-1;** i**++){**  Time**.**push**(**datos**[**i**][0]);**  Coronal**.**push**(**parseFloat**(**datos**[**i**][1]).**toFixed**(2));**  Sagital**.**push**(**parseFloat**(**datos**[**i**][2]).**toFixed**(2));**  Transversal**.**push**(**parseFloat**(**datos**[**i**][3]).**toFixed**(2));**  **}**  **var** max\_c **=** Math**.**max**.**apply**(null,** Coronal**);**  **var** min\_c **=** Math**.**min**.**apply**(null,** Coronal**);**  **var** max\_t **=** Math**.**max**.**apply**(null,** Transversal**);**  **var** min\_tr **=** Math**.**min**.**apply**(null,** Transversal**);**  **var** max\_s **=** Math**.**max**.**apply**(null,** Sagital**);**  **var** min\_s **=** Math**.**min**.**apply**(null,** Sagital**);**  **var** t **=** Time**.**join**();**  **var** c **=** Coronal**.**join**();**  **var** s **=** Sagital**.**join**();**  **var** tr **=** Transversal**.**join**();**  **var** socket **=** io**.**connect**(**"http://172.20.10.5:8124"**);**  socket**.**on**(**"message"**,function(**message**){**  console**.**log**(**"El servidor ha enviado un mensaje:"**);**  message **=** JSON**.**parse**(**message**);**  *//console.log(message);*  **});**  **var** datos3 **=** **{**  operacion**:** "Añadir datos de paciente"**,**  id**:** url1**.**searchParams**.**get**(**"var1"**),** */\*creating a Js ojbect to be sent to the server\*/*  n**:** url1**.**searchParams**.**get**(**"var2"**),**  t1**:** t**,**  c1**:** c**,**  s1**:** s**,**  t1**:** tr**,**  mxc**:** max\_c**,**  mnc**:** min\_c**,**  mxt**:** max\_t**,**  mntr**:** min\_tr**,**  mxs**:** max\_s**,**  mns**:** min\_s**,**  f**:** fecha  **}**  socket**.**send**(**JSON**.**stringify**(**datos3**));**  **}** |

En este caso hay que realizar varias operaciones para obtener la información que nos interesa del JSON. El JSON está compuesto por varios arrays. El bucle for se dedica a recorrer estos arrays para sacar los datos de tiempo y de los movimientos en los planos coronal, sagital y transversal. Cuando se extraen los datos para los movimientos se redondean los datos a dos decimales ya que el archivo CSV, generado por el aparato, genera números con 16 decimales, y guardar tantos números con tantos números decimales sería una carga muy pesada para la base de datos. Una vez extraídos estos datos sacamos los máximos y los mínimos de los tres movimientos, que nos servirán para mostrar la evolución de cada movimiento.

Una vez obtenidos todos los datos de interés crearemos un websocket para conectarnos al servidor, escucharemos el mensaje de confirmación de este y enviaremos los datos extraídos del JSON con la cabecera adecuada para que el servidor reconozca la acción que debe realizar “Añadir datos de paciente”. Podemos observar que el objeto JSON enviado al servidor tiene bastantes elementos que son los que hemos descrito anteriormente.

#### 4.3.5.2. Funcionalidad en el lado del servidor

|  |
| --- |
| socket**.**on**(**"message"**,function(**info**){**  datos **=** JSON**.**parse**(**info**);**  **if(**datos**.**operacion **==** "Añadir datos de paciente"**){**  console**.**log**(**"Paciente a añadir: "**+**datos**.**n**);**  **var** filebuffer **=** fs**.**readFileSync**(**'./Pacientes\_DB.db'**);**  **var** db **=** **new** SQL**.**Database**(**filebuffer**);**  console**.**log**(**timestamp**(**'hh:mm:ss:iii'**)+**" Base de datos abierta"**);**  db**.**run**(**"INSERT INTO datos\_pacientes VALUES (:id\_datos, :Time\_ms, :Coronal, :Sagital, :Transversal, :N\_Paciente, :Fecha, :max\_c, :min\_c, :max\_s, :min\_s, :max\_t, :min\_t)"**,** **{**':Time\_ms'**:**datos**.**t1**,** ':Coronal'**:**datos**.**c1**,**':Sagital'**:**datos**.**s1**,** ':Transversal'**:**datos**.**t1**,** ':N\_Paciente'**:**datos**.**id**,** ':Fecha'**:**datos**.**f**,** ':max\_c'**:**datos**.**mxc**,** ':min\_c'**:**datos**.**mnc**,** ':max\_s'**:**datos**.**mxs**,** ':min\_s'**:**datos**.**mns**,** ':max\_t'**:**datos**.**mxt**,** ':min\_t'**:**datos**.**mntr**});**  **var** data **=** db**.export();**  **var** buffer **=** **new** Buffer**(**data**);**  fs**.**writeFileSync**(**"./Pacientes\_DB.db"**,** buffer**);**  db**.**close**();**  console**.**log**(**timestamp**(**'hh:mm:ss:iii'**)+**" Base de datos cerrada"**);**  **var** ack\_to\_client **=** **{**  data**:**"El servidor ha añadido datos de un paciente a la db"  **}**  socket**.**send**(**JSON**.**stringify**(**ack\_to\_client**));**  io**.**sockets**.**emit**(**"reload"**,{});**  **}** |

Aquí vemos el comportamiento que ha tenido el servidor para el resto de funciones. Se reconoce la cabecera contenida en el objeto JSON y se actúa en consecuencia. Después se vuelve a acceder a la base de datos y se insertan todos los datos que se han extraído del objeto JSON. Se guardan los datos mediante un db.export() y el uso de un buffer como ya se ha explicado anteriormente y se emite un mensaje para que todos los clientes conectados actualicen la tabla de movimientos.

### 4.3.6. Borrar una sesión de movimientos

Para borra un set de movimientos de un paciente seguiremos una dinámica similar a la de borrar un paciente.

#### 4.3.6.1. Funcionalidad en el lado del cliente

|  |
| --- |
| var filas = tabla.rows.length;  var fila = tabla.insertRow(filas);  fila.insertCell(0).innerHTML = '**<button** class="btn\_borrar" type="button" onClick="borrar\_datos('+datos[0].values[i][0]+',\''+nombre+'\')"**></button>**'; |

Como vimos en el aparatado de obtención de datos de un paciente, cuando se genera la tabla con movimientos se van insertando filas con los distintos movimientos, la fecha y por último un fila que incluye un botón para borrar esa sesión de movimientos. Como vemos en el código cuando el usuario pulsa el botón se llama a una función borrar datos() a la que se le pasan el identificador único de la sesión de movimientos y el nombre del paciente al que pertenece dicha sesión.

|  |
| --- |
| **function** borrar\_datos**(**N\_p**,**nombre**){**  **var** r **=** confirm**(**"¿Esta seguro de que quiere borrar estos datos?"**);**  **if** **(**r **==** **true){**    **var** socket **=** io**.**connect**(**"http://172.20.10.5:8124"**);**  socket**.**on**(**"message"**,function(**message**){**  console**.**log**(**"El servidor ha enviado un mensaje:"**);**  message **=** JSON**.**parse**(**message**);**  *//console.log(message);*  **});**  **var** data **=** **{**  operacion**:** "Borrar datos de paciente"**,**  id**:** N\_p**,**  n**:** nombre  **}**  socket**.**send**(**JSON**.**stringify**(**data**));**  **}**  **else{**  console**.**log**(**"Datos no borrados"**);**  **}**  **}** |

Como en todas las operaciones en las que se eliminan datos, se pregunta al usuario si quiere borrar los datos. Cuando el usuario confirma la operación se crea una conexión con el servidor y se sigue el procedimiento visto en funciones anteriores (escuchar mensaje del servidor y envío de datos). Como ya hemos visto se manda un objeto JSON con la cabecera “Borrar datos de paciente” para que el servidor proceda con la operación correspondiente.

Para este caso es necesario que cuando borremos la sesión de datos se actualice la tabla con los movimientos y así no se muestre la sesión borrada. Para realizar esta función el servidor manda un mensaje que el cliente recibe y le sirve para saber cuando tiene que actualizar el contenido de la tabla.

|  |
| --- |
| **var** socket **=** io**.**connect**(**"http://172.20.10.5:8124"**);**  socket**.**on**(**"reload"**,** **function** **(**data**)** **{**  location**.**reload**();**  **});** |

Este código se encuentra en el navegador y espera a que el servidor confirme que se han borrado los datos. Una vez recibida la confirmación, el navegador realiza un location.reload() que básicamente actualiza la página web, lo que obliga a que se vuelva a llamar a la función que obtiene las sesiones de movimientos. En esta ocasión se mostrará la lista de sesiones de movimientos actualizada.

#### 4.3.6.2. Funcionalidad en el lado del servidor

|  |
| --- |
| socket**.**on**(**"message"**,function(**info**){**  datos **=** JSON**.**parse**(**info**);**  **if(**datos**.**operacion**==**"Borrar datos de paciente"**){**  console**.**log**(**"Datos de paciente a borrar: "**+**datos**.**n**);**  **var** filebuffer **=** fs**.**readFileSync**(**'./Pacientes\_DB.db'**);**  **var** db **=** **new** SQL**.**Database**(**filebuffer**);**  console**.**log**(**timestamp**(**'hh:mm:ss:iii'**)+**" Base de datos abierta"**);**  db**.**run**(**"DELETE FROM datos\_pacientes WHERE id\_datos="**+**datos**.**id**);**  **var** data **=** db**.export();**  **var** buffer **=** **new** Buffer**(**data**);**  fs**.**writeFileSync**(**"./Pacientes\_DB.db"**,** buffer**);**  db**.**close**();**  console**.**log**(**timestamp**(**'hh:mm:ss:iii'**)+**" Base de datos cerrada"**);**  **var** ack\_to\_client **=** **{**  data**:**"El servidor ha eliminado los datos del paciente de la db"  **}**  socket**.**send**(**JSON**.**stringify**(**ack\_to\_client**));**  io**.**sockets**.**emit**(**"reload"**,{});**  **}** |

Como vimos para el caso para añadir una sesión de movimientos, el servidor escucha mediante socket.on(). Cuando recibe un mensaje comprueba su cabecera y al reconocer la operación “Borrar datos de paciente” realiza la conexión con la base de datos y elimina la sesión de movimientos con el identificador que ha pasado el cliente. A continuación guarda los cambios y cierra la base de datos. Como hemos dicho en la parte del cliente, cuando se realizan los cambios el servidor ejecuta la función io.sockets.emit(). Básicamente fuerza a cualquier cliente conectado a actualizar su lista de sesiones de movimientos, por lo que si hay varios dispositivos conectados, en todos estos se actualizarán los cambios.

### 4.3.7 Mostrar un gráfico de un movimiento

Para este caso no seguiremos el mismo procedimiento que para el resto de funciones, ya que no será necesaria una conexión con el servidor y además se hará uso de la herramienta chart.js para convertir los arrays de datos en gráficos.

#### 4.3.7.1. Funcionalidad en el lado del cliente

|  |
| --- |
| fila.insertCell(0).innerHTML = '**<button** class="btn" type="button" onClick="crearGrafico(\''+datos[0].values[i][1]+'\', \'' + datos[0].values[i][4] + '\',1)"**></button>**';  fila.insertCell(0).innerHTML = '**<button** class="btn" type="button" onClick="crearGrafico(\''+datos[0].values[i][1]+'\', \'' + datos[0].values[i][3] + '\',2)"**></button>**';  fila.insertCell(0).innerHTML = '**<button** class="btn" type="button" onClick="crearGrafico(\''+datos[0].values[i][1]+'\', \'' + datos[0].values[i][2] + '\',3)"**></button>**'; |

Este fragmento de código corresponde a la tabla de sesiones de movimientos. Tres de las filas insertadas para cada columna corresponden a un botón que permite crear un gráfico. Cada botón corresponde a un movimiento en un plano (Coronal, Sagital y Transversal). Dependiendo del botón que el usuario presione se mostrará uno de los tres movimientos. Los tres botones llaman a la misma función cuando son presionados “crearGrafico()”, y se le pasan los arrays de tiempo y del movimiento y un número que identifica el movimiento seleccionado.

|  |
| --- |
| **var** var\_i **=** **0;**  **function** crearGrafico**(**time**,** mov**,** n**){**  var\_i**+=1;**  **var** url **=** window**.**location**.**href**;**  **var** url1 **=** **new** URL**(**url**);**  **var** nombre **=** url1**.**searchParams**.**get**(**"var2"**);**  **var** apellido **=** url1**.**searchParams**.**get**(**"var3"**);**  **if(**n**==3){**  x**=**"Coronal"**;**  **}**  **else** **if(**n**==2){**  x**=**"Sagital"**;**  **}**  **else** **if(**n**==1){**  x**=**"Transversal"**;**  **}**  **var** Movimiento **=** mov**.**split**(**','**);**  **var** Tiempo **=** time**.**split**(**','**);**  **const** CHART **=** document**.**getElementById**(**"lineChart"**);**  **if** **(**var\_i**>=2){**  lineChart**.**destroy**();**  **}**  lineChart **=** **new** Chart**(**CHART**,** **{**  type**:** 'line'**,**  data**:** **{**  labels**:** Tiempo**,**  datasets**:** **[**  **{**  label**:** "Movimiento "**+**x**+**" de "**+**nombre**+**" "**+**apellido**,**  fill**:** **false,**  lineTension**:** **0.5,**  backgroundColor**:** "rgba(75,192,192,0.4)"**,**  borderColor**:** "rgba(75,192,192,1)"**,**  borderCapStyle**:** 'butt'**,**  borderDash**:** **[],**  borderDashOffset**:** **0.0,**  borderJoinStyle**:** 'miter'**,**  pointBorderColor**:** "rgba(75,192,192,1)"**,**  pointBackgroundColor**:** "#fff"**,**  pointBorderWidth**:** **1,**  pointHoverRadius**:** **5,**  pointHoverBackgroundColor**:** "rgba(75,192,192,1)"**,**  pointHoverBorderColor**:** "rgba(220,220,220,1)"**,**  pointHoverBorderWidth**:** **2,**  pointRadius**:** **1,**  pointHitRadius**:** **10,**  data**:** Movimiento**,**  spanGaps**:** **false,**  **}**  **]**  **},**options**:** **{**  scales**:** **{**  yAxes**:** **[{**  ticks**:** **{**  beginAtZero**:** **false**  **}**  **}]**  **}**  **}**  **});**  **}** |

Dentro de esta función reorganizamos los dos arrays para que la herramienta chart.js pueda utilizarlos. Con Chart.js creamos un nuevo tipo de gráfico que en este caso será lineal y le pondremos un título. También le pasaremos los dos arrays y ajustaremos las características del gráfico (color de la línea, grosor de la línea, interactividad con el gráfico, etc).

A continuación mostraremos un ejemplo de un gráfico generado por la aplicación web gracias a chart.js:



Este gráfico corresponde al movimiento de un paciente en el plano sagital. El eje vertical denota los grados del rango de movimiento y el eje horizontal denota el tiempo en milisegundos. Este gráfico es interactivo, por lo que se pueden realizar las siguientes acciones:

* Ampliar y reducir el gráfico.
* Interactuar con los distintos puntos, mostrando para cada punto el ángulo de y su tiempo correspondiente.

### 4.3.8 Mostrar un gráfico de evolución de un movimiento

Para este caso se realizarán interacciones muy similares a las interacciones que se mostraron en el caso anterior. Simplemente se pasarán arrays de datos diferentes, por lo que el gráfico será distinto. Al contrario que en el caso anterior, sí se realizará una conexión con el servidor, ya que los datos que se obtienen cuando se muestra el listado de sesiones de movimientos, se pueden obtener los valores de una fila de la base de datos pero no los de varias como es necesario en este caso.

#### 4.3.8.1. Funcionalidad en el lado del cliente

|  |
| --- |
| var fila1 = tabla.insertRow(tabla.length);  fila1.insertCell(0).innerHTML = '**<button** class="bt" type="button" onClick="Evolucion(1)"**>**Evolución del movimieto**</button>**';  fila1.insertCell(0).innerHTML = '**<button** class="bt" type="button" onClick="Evolucion(2)"**>**Evolución del movimieto**</button>**';  fila1.insertCell(0).innerHTML = '**<button** class="bt" type="button" onClick="Evolucion(3)"**>**Evolución del movimieto**</button>**'; |

Este fragmento de código corresponde a la tabla de sesiones de movimientos. Tres de las filas insertadas para cada columna corresponden a un botón que permite crear un gráfico. Cada botón corresponde a un gráfico de evolución en un plano (Coronal, Sagital y Transversal). Dependiendo del botón que el usuario presione se mostrará una de las evoluciones. Los tres botones llaman a la misma función cuando son presionados “Evolución()”, y se le pasa un numero para distinguir entre los tres planos distintos.

|  |
| --- |
| **function** Evolucion**(**move**){**  **var** socket **=** io**.**connect**(**"http://172.20.10.5:8124"**);**  socket**.**on**(**"message"**,function(**message**){**  console**.**log**(**"El servidor ha enviado un mensaje:"**);**  message **=** JSON**.**parse**(**message**);**  *//console.log(message);*  **});**  **var** datos2 **=** **{**  operacion**:** "Datos de Evolucion paciente"**,**  id**:** url1**.**searchParams**.**get**(**"var1"**),**  n**:** url1**.**searchParams**.**get**(**"var2"**)**  **}**  socket**.**send**(**JSON**.**stringify**(**datos2**));**  socket**.**on**(**"datos\_evolucion\_paciente"**,** **function** **(**max\_minimo**)** **{**  **var** genero **=** url1**.**searchParams**.**get**(**"var4"**);**  **var** max **=** **[];**  **var** max\_max **=** **[];**  **var** max\_min **=** **[];**  **var** min **=** **[];**  **var** min\_max **=** **[];**  **var** min\_min **=** **[];**  **var** fecha **=** **[];**  **if(**move**==1){**  **for(**i**=0;**i**<**max\_minimo**[0].**values**.**length**;**i**++){**  max**.**push**(**max\_minimo**[0].**values**[**i**][0]);**  min**.**push**(**max\_minimo**[0].**values**[**i**][1]);**  fecha**.**push**(**max\_minimo**[0].**values**[**i**][6])**  **if** **(**genero **==** "h"**){**  max\_max**.**push**(49.2);**  max\_min**.**push**(32.6);**  min\_max**.**push**(-44.3);**  min\_min**.**push**(-28.3);**  **}**  **else{**  max\_max**.**push**(54.8);**  max\_min**.**push**(35.8);**  min\_max**.**push**(-53.1);**  min\_min**.**push**(-37.9);**  **}**  **}**  grafico\_evolucion**(**max**,**min**,**fecha**,**"Coronal"**,**max\_max**,**max\_min**,**min\_max**,**min\_min**,**"Flexión Lateral"**,**"Extensión Lateral"**);**  **}**  **else** **if(**move**==2){**  **for(**i**=0;**i**<**max\_minimo**[0].**values**.**length**;**i**++){**  max**.**push**(**max\_minimo**[0].**values**[**i**][2]);**  min**.**push**(**max\_minimo**[0].**values**[**i**][3]);**  fecha**.**push**(**max\_minimo**[0].**values**[**i**][6])**  **if** **(**genero **==** "h"**){**  max\_max**.**push**(72.5);**  max\_min**.**push**(48.3);**  min\_max**.**push**(-82.6);**  min\_min**.**push**(-57.2);**  **}**  **else{**  max\_max**.**push**(68.2);**  max\_min**.**push**(48.8);**  min\_max**.**push**(-90.7);**  min\_min**.**push**(-64.3);**  **}**  **}**  grafico\_evolucion**(**max**,**min**,**fecha**,**"Sagital"**,**max\_max**,**max\_min**,**min\_max**,**min\_min**,**"Flexión"**,**"Extensión"**);**  **}**  **else{**  **for(**i**=0;**i**<**max\_minimo**[0].**values**.**length**;**i**++){**  max**.**push**(**max\_minimo**[0].**values**[**i**][4]);**  min**.**push**(**max\_minimo**[0].**values**[**i**][5]);**  fecha**.**push**(**max\_minimo**[0].**values**[**i**][6])**  **if** **(**genero **==** "h"**){**  max\_max**.**push**(87.4);**  max\_min**.**push**(72.2);**  min\_max**.**push**(-83.5);**  min\_min**.**push**(-67.1);**  **}**  **else{**  max\_max**.**push**(89);**  max\_min**.**push**(74.6);**  min\_max**.**push**(-87.8);**  min\_min**.**push**(-72.4);**  **}**  **}**  grafico\_evolucion**(**max**,**min**,**fecha**,**"Transversal"**,**max\_max**,**max\_min**,**min\_max**,**min\_min**,**"Rotación derecha"**,**"Rotación izquierda"**);**  **}**  **});**  **}** |

Como observamos se realiza la conexión con el servidor y de este recibimos los datos necesarios. Una vez obtenidos los datos, dependiendo del valor de “move” se realizará un gráfico u otro. En estas capturas solo vemos el caso de move=1 (evolución en el plano coronal) ya que para los otros dos casos se realiza el mismo procedimiento. Este procedimiento consiste en los siguientes pasos:

1. Extraer los valores máximos y mínimos del movimiento y almacenarlos en dos arrays, uno con los valores máximos y otro con los valores mínimos.
2. Extraer las fechas de cada máximo y mínimo y almacenarlas en otro array.
3. Distinguir entre sexos para crear 4 arrays que formarán los límites de normalidad.
4. Llamar a la función “grafico\_evolucion()” y pasar todos los valores extraídos como argumentos de la función.

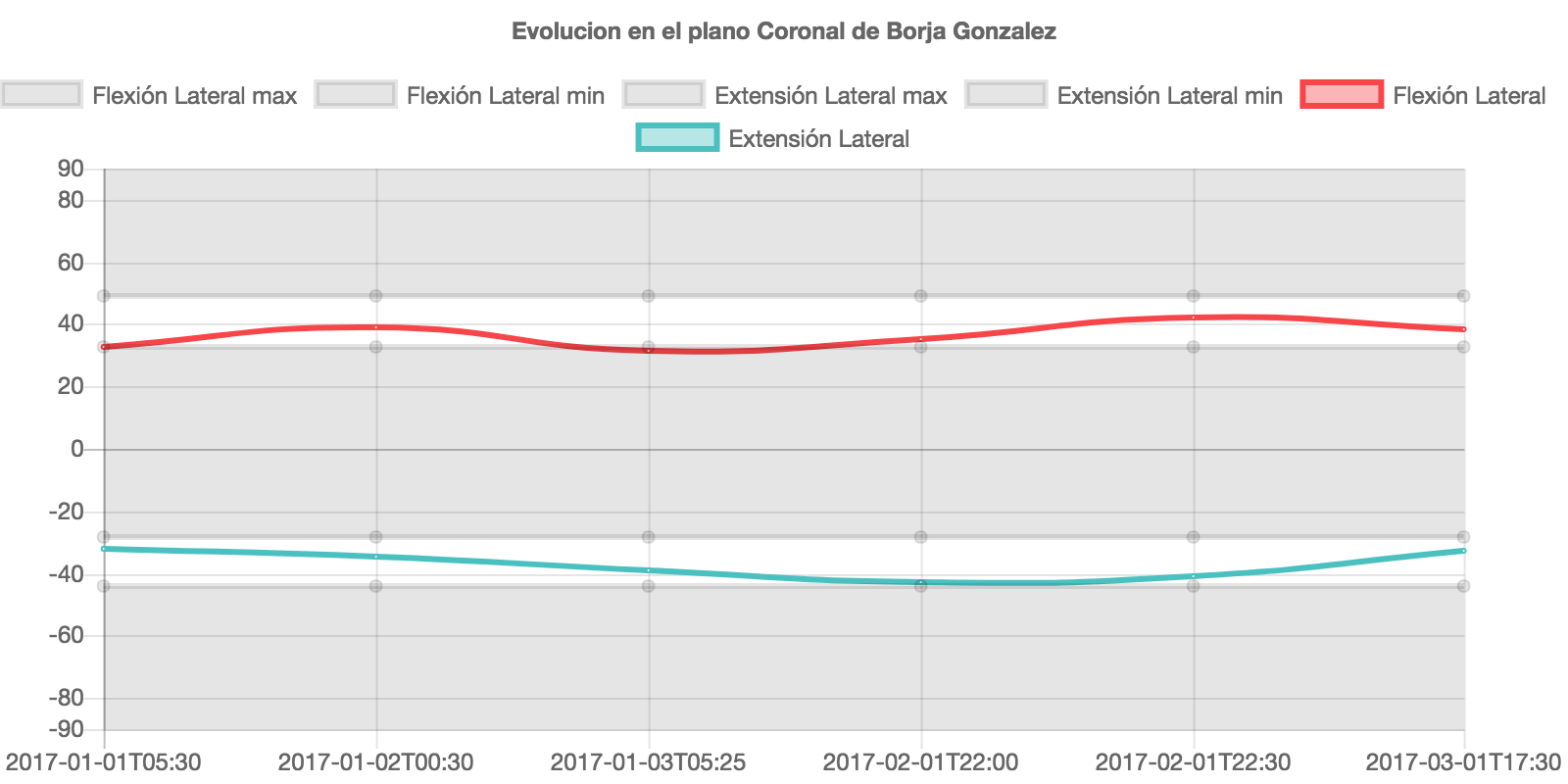
|  |
| --- |
| **function** grafico\_evolucion**(**maximo**,** minimo**,** fechas**,** x**,** maximo\_max**,** maximo\_min**,** minimo\_max**,** minimo\_min**,**titulo1**,**titulo2**){**  var\_i**+=1;**  **var** url **=** window**.**location**.**href**;**  **var** url1 **=** **new** URL**(**url**);**  **var** nombre **=** url1**.**searchParams**.**get**(**"var2"**);**  **var** apellido **=** url1**.**searchParams**.**get**(**"var3"**);**  **var** sexo **=** url1**.**searchParams**.**get**(**"var4"**);**  **const** CHART **=** document**.**getElementById**(**"lineChart"**);**  **if** **(**var\_i**>=2){**  lineChart**.**destroy**();**  **}**  lineChart **=** **new** Chart**(**CHART**,** **{**  type**:** 'line'**,**  data**:** **{**  labels**:** fechas**,**  datasets**:** **[**  **{**  label**:** titulo1**+**" max"**,**  fill**:**'end'**,**  data**:**maximo\_max**,**  **},**  **{**  label**:** titulo1**+**" min"**,**  fill**:true,**  data**:**maximo\_min**,**  **},**  **{**  label**:** titulo2**+**" max"**,**  fill**:**'bottom'**,**  data**:**minimo\_max**,**  **},**  **{**  label**:** titulo2**+**" min"**,**  fill**:true,**  data**:**minimo\_min**,**  **},**  **{**  label**:** titulo1**,**  fill**:** **false,**  lineTension**:** **0.5,**  backgroundColor**:** "rgba(247,70,74,0.4)"**,**  borderColor**:** "rgba(247,70,74,1)"**,**  borderCapStyle**:** 'butt'**,**  borderDash**:** **[],**  borderDashOffset**:** **0.0,**  borderJoinStyle**:** 'miter'**,**  pointBorderColor**:** "rgba(247,70,74,1)"**,**  pointBackgroundColor**:** "#fff"**,**  pointBorderWidth**:** **1,**  pointHoverRadius**:** **5,**  pointHoverBackgroundColor**:** "rgba(247,70,74,1)"**,**  pointHoverBorderColor**:** "rgba(220,220,220,1)"**,**  pointHoverBorderWidth**:** **2,**  pointRadius**:** **1,**  pointHitRadius**:** **10,**  data**:** maximo**,**  spanGaps**:** **false,**  **},**  **{**  label**:** titulo2**,**  fill**:** **false,**  lineTension**:** **0.5,**  backgroundColor**:** "rgba(75,192,192,0.4)"**,**  borderColor**:** "rgba(75,192,192,1)"**,**  borderCapStyle**:** 'butt'**,**  borderDash**:** **[],**  borderDashOffset**:** **0.0,**  borderJoinStyle**:** 'miter'**,**  pointBorderColor**:** "rgba(75,192,192,1)"**,**  pointBackgroundColor**:** "#fff"**,**  pointBorderWidth**:** **1,**  pointHoverRadius**:** **5,**  pointHoverBackgroundColor**:** "rgba(75,192,192,1)"**,**  pointHoverBorderColor**:** "rgba(220,220,220,1)"**,**  pointHoverBorderWidth**:** **2,**  pointRadius**:** **1,**  pointHitRadius**:** **10,**  data**:** minimo**,**  spanGaps**:** **false,**  **},**  **]**  **},**options**:** **{**  title**:** **{**  display**:** **true,**  text**:** "Evolucion en el plano "**+**x**+**" de "**+**nombre**+**" "**+**apellido  **},**  scales**:** **{**  yAxes**:** **[{**  ticks**:** **{**  beginAtZero**:** **false,**  min**:** **-90,**  max**:** **90**  **},**  gridLines**:** **{**  drawBorder**:** **true,**  **}**  **}]**  **}**  **}**  **});**  **}** |

En esta función vemos como se comprueba si existe un gráfico, y en caso afirmativo se eliminará para sustituirlo por el gráfico de evolución en cuestión.

Vemos como se han incluido varios sets de datos en el gráfico. Estos valores corresponden a:

* Cuatro sets de datos para los valores de normalidad. Dos para los valores máximos y dos para los mínimos.
* Un set de datos correspondiente a los valores máximos alcanzados en el movimiento por el paciente.
* Un set de datos correspondiente a los valores mínimos alcanzados en el movimiento por el paciente.

A continuación se mostrará un ejemplo de un gráfico de evolución y se explicará más en detalle:



Este gráfico corresponde a la evolución de un movimiento en el plano coronal de un paciente. El eje vertical denota los grados del rango de movimiento y el eje horizontal denota las fechas de las distintas mediciones. Este gráfico es interactivo, por lo que se pueden realizar las siguientes acciones:

* Ampliar y reducir el gráfico.
* Interactuar con los distintos puntos, mostrando para cada punto el ángulo de y su tiempo correspondiente.

La zona que no está oscurecida corresponde al rango de normalidad, por lo tanto si los valores caen en la zona más clara, podemos asumir que el paciente tiene un rango de movimiento normal.

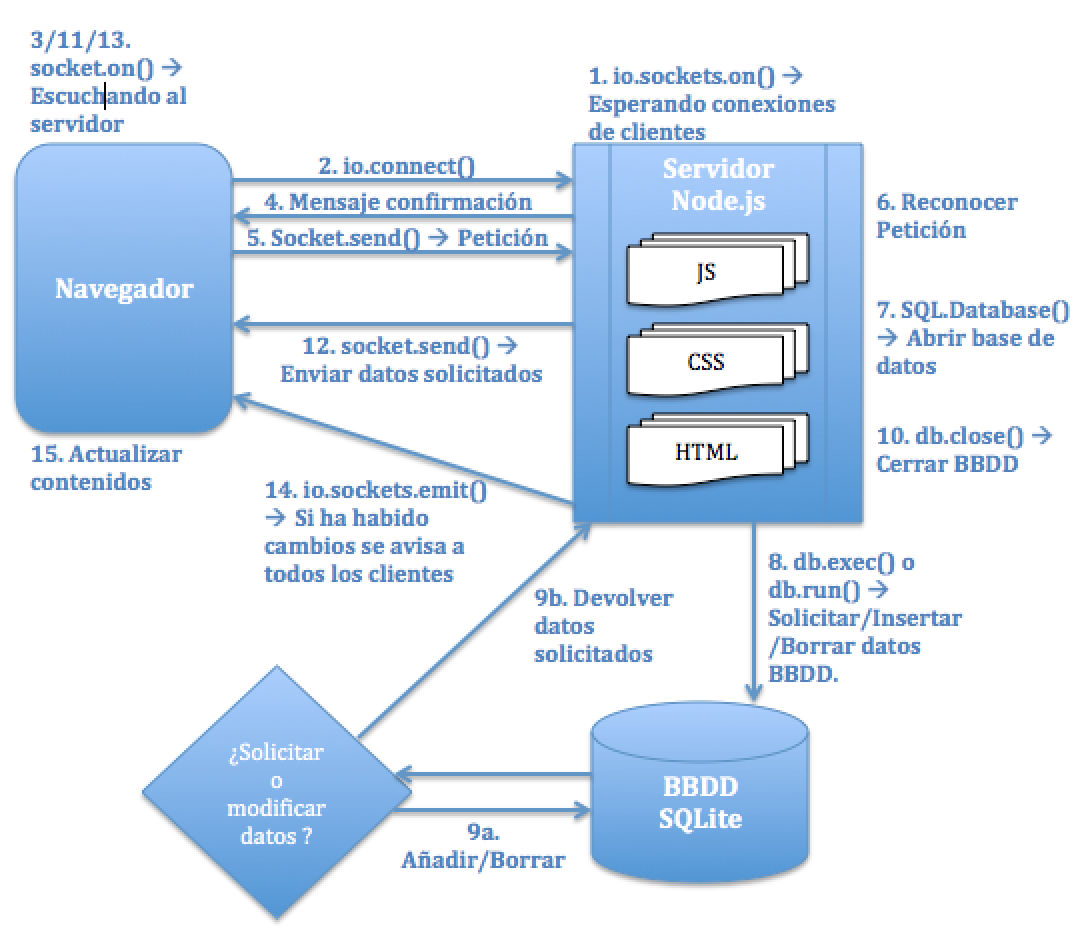
#### 4.3.8.2. Funcionalidad en el lado del servidor

|  |
| --- |
| socket**.**on**(**"message"**,function(**info**){**  datos **=** JSON**.**parse**(**info**);**  **if(**datos**.**operacion**==**"Datos de Evolucion paciente"**){**  console**.**log**(**"Mostrar datos de evolucion de: "**+**datos**.**n**);**  **var** filebuffer **=** fs**.**readFileSync**(**'./Pacientes\_DB.db'**);**  **var** db **=** **new** SQL**.**Database**(**filebuffer**);**  console**.**log**(**timestamp**(**'hh:mm:ss:iii'**)+**" Base de datos abierta"**);**  **var** datos\_evolucion\_paciente **=** db**.**exec**(**"SELECT max\_c, min\_c, max\_s, min\_s, max\_t, min\_t, Fecha FROM datos\_pacientes WHERE N\_PACIENTE ="**+**datos**.**id**+**" ORDER BY datetime(FECHA) asc LIMIT (select count() from datos\_pacientes)"**);**  socket**.**emit**(**"datos\_evolucion\_paciente"**,**datos\_evolucion\_paciente**);**  db**.**close**();**  console**.**log**(**timestamp**(**'hh:mm:ss:iii'**)+**" Base de datos cerrada"**);**  **}** |

Como en el resto de casos observamos un comportamiento idéntico donde se espera a que un cliente realice una conexión con el servidor y una vez establecida la conexión se distingue la operación a realizar, se abre la base de datos y se requieren los datos de máximos y mínimos ordenados por fecha para que la evolución se muestre en orden cronológico. Finalmente enviamos los datos al cliente y cerramos la base de datos.

## 4.4. Diagrama de flujo

El diagrama de flujo es la representación gráfica de todas las interacciones posibles entre el navegador, el servidor y la base de datos.



En este diagrama podemos ver todas las operaciones posibles de las distintas funciones existentes. Cada función realizará ciertos pasos pero no necesariamente todos los mostrados en el diagrama de flujo. Por ejemplo:

* **Obtener pacientes:** El recorrido a seguir es del 1 al 12 y el paso 9 se correspondería al b.
* **Añadir/Borrar paciente:** El recorrido a seguir es del 1 al 12 y el paso 9 correspondería al b. Cuando el usuario presiona el botón de borrado, el recorrido a seguir es del 1 al 15 y el paso 9 se corresponde al a.
* **Listado de movimientos:** El recorrido a seguir es del 1 al 12 y el paso 9 se correspondería al b. Una vez situados en la lista de pacientes, y cuando el usuario presiona el botón para acceder a los datos, el recorrido a seguir es del 1 al 12 y el paso 9 se correspondería al b.
* **Añadir/Borrar datos de movimiento:** El recorrido a seguir es del 1 al 12 y el paso 9 se correspondería al b. Una vez situados en la lista de pacientes, y cuando el usuario presiona el botón para acceder a los datos, el recorrido a seguir es del 1 al 12 y el paso 9 se correspondería al b. Finalmente el usuario se posiciona en el listado de movimientos, y cuando añade/borra datos de movimiento, el recorrido a seguir es del 1 al 15 y el paso 9 se corresponde al a.
* **Mostrar un gráfico de un movimiento:** El recorrido a seguir es del 1 al 12 y el paso 9 se correspondería al b. Una vez situados en la lista de pacientes, y cuando el usuario presiona el botón para acceder a los datos, el recorrido a seguir es del 1 al 12 y el paso 9 se correspondería al b. Como los datos de los movimientos del paciente ya están descargados, el usuario solo tiene que presionar el botón del gráfico a mostrar y se graficará el movimiento correspondiente.
* **Mostrar un gráfico de evolución de un movimiento:** El recorrido a seguir es del 1 al 12 y el paso 9 se correspondería al b. Una vez situados en la lista de pacientes, y cuando el usuario presiona el botón para acceder a los datos, el recorrido a seguir es del 1 al 12 y el paso 9 se correspondería al b. Finalmente el usuario se posiciona en el listado de movimientos, y cuando presiona el botón para mostrar la evolución, el recorrido a seguir es del 1 al 12 y el paso 9 se correspondería al b.

# 5. Pruebas

El objetivo del plan de pruebas es comprobar que el sistema funciona según lo especificado y responde a los requisitos del cliente. Para ello hemos llevado a cabo una serie de pruebas de sistema que aseguran que se cumplen todos los requisitos funcionales enumerados en el capítulo 3.

## 5.1. Pruebas de sistema

En este proyecto se utilizará la escritura de valores por pantalla, ya que es una estrategia simple y efectiva. Por lo tanto, se observarán valores en la interfaz web y en el servidor que irán mostrando como cada función se va ejecutando. Para el caso del cliente/navegador se mostrarán valores en la consola del inspector de Google Chrome y para el caso del servidor se mostrarán valores en un terminal donde se ejecuta el código del servidor. Para poder observar que se realiza todo en el orden apropiado, se imprimirá un tiempo asociado a cada valor por pantalla.

### 5.1.1. Obtener pacientes

Consola del navegador:

|  |
| --- |
| Navigated to http://192.168.1.33:8124/pacientes.html  22:05:34.084 VM88 pacientes\_node.js:28 Conexíon establecida con el servidor  22:05:34.087 VM88 pacientes\_node.js:39 Solicitud de listado de pacientes enviada  22:05:34.294 VM88 pacientes\_node.js:41 Lista de pacientes recibida  22:05:34.296 pacientes.html:45 Lista de pacientes disponible en el navegador |

Terminal (Servidor):

|  |
| --- |
| 22:05:34:094 Conexión establecida con el cliente  22:05:34:099 Conexión establecida con el cliente  22:05:34:136 Petición del cliente: Pacientes  22:05:34:177 Base de datos abierta  22:05:34:286 Listado de pacientes enviado al cliente  22:05:34:292 Base de datos cerrada |

Como se puede observar todos los eventos ocurren en orden cronológico. En primer lugar se establece la conexión mediante un websocket y después el cliente manda la solicitud para obtener el listado de pacientes. A continuación el servidor reconoce la petición, abre la base de datos, envía el listado al cliente y cierra la base de datos. Finalmente observamos como el cliente recibe la lista y la muestra en el navegador.

### 5.1.2. Añadir un paciente

Consola del navegador:

|  |
| --- |
| 22:08:24.928 Navigated to http://192.168.1.33:8124/pacientes.html  22:08:25.452 VM88 pacientes\_node.js:28 Conexíon establecida con el servidor  22:08:25.457 VM88 pacientes\_node.js:39 Solicitud de listado de pacientes enviada  22:08:25.673 VM88 pacientes\_node.js:41 Lista de pacientes recibida  22:08:25.675 pacientes.html:45 Lista de pacientes disponible en el navegador  22:08:52.862 VM88 pacientes\_node.js:8 Conexíon establecida con el servidor  22:08:52.864 VM88 pacientes\_node.js:22 Solicitud para añadir paciente: (María Constancia) enviada  22:08:52.993 Navigated to http://192.168.1.33:8124/pacientes.html  22:08:53.446 pacientes\_node.js:28 Conexíon establecida con el servidor  22:08:53.451 pacientes\_node.js:39 Solicitud de listado de pacientes enviada  22:08:53.551 pacientes\_node.js:41 Lista de pacientes recibida  22:08:53.553 pacientes.html:45 Lista de pacientes disponible en el navegador |

Terminal (Servidor):

|  |
| --- |
| 22:08:25:463 Conexión establecida con el cliente  22:08:25:466 Conexión establecida con el cliente  22:08:25:511 Petición del cliente: Pacientes  22:08:25:552 Base de datos abierta  22:08:25:665 Listado de pacientes enviado al cliente  22:08:25:671 Base de datos cerrada  22:08:52:868 Conexión establecida con el cliente  22:08:52:884 Petición del cliente: Añadir paciente  22:08:52:884 Paciente a añadir: María Constancia  22:08:52:887 Base de datos abierta  22:08:52:935 Se ha añadido al paciente María Constancia a la base de datos  22:08:52:941 Base de datos cerrada  22:08:53:446 Conexión establecida con el cliente  22:08:53:450 Conexión establecida con el cliente  22:08:53:517 Petición del cliente: Pacientes  22:08:53:522 Base de datos abierta  22:08:53:536 Listado de pacientes enviado al cliente  22:08:53:537 Base de datos cerrada |

Cuando añadimos un paciente a la base de datos, primero se listan los pacientes, por lo que observaremos los mismos valores por pantalla que en el apartado anterior. Después se establecerá otra conexión con el servidor para añadir al paciente. Finalmente volveremos a observar, por segunda vez, los valores por pantalla para obtener el listado de pacientes, ya que como se explicó en la sección de implementación, cuando se añade un paciente, el servidor obliga al navegador a refrescarse, listando de nuevo los pacientes y por lo tanto mostrando el paciente añadido.

### 5.1.3. Borrar un paciente

Consola del navegador:

|  |
| --- |
| Navigated to http://192.168.1.33:8124/pacientes.html  22:10:15.257 VM88 pacientes\_node.js:28 Conexión establecida con el servidor  22:10:15.261 VM88 pacientes\_node.js:39 Solicitud de listado de pacientes enviada  22:10:15.467 VM88 pacientes\_node.js:41 Lista de pacientes recibida  22:10:15.469 pacientes.html:45 Lista de pacientes disponible en el navegador  22:10:18.368 VM88 pacientes\_node.js:52 Conexión establecida con el servidor  22:10:18.368 VM88 pacientes\_node.js:65 Solicitud para borrar paciente: (María) enviada  22:10:18.628 Navigated to http://192.168.1.33:8124/pacientes.html  22:10:19.088 pacientes\_node.js:28 Conexión establecida con el servidor  22:10:19.093 pacientes\_node.js:39 Solicitud de listado de pacientes enviada  22:10:19.206 pacientes\_node.js:41 Lista de pacientes recibida  22:10:19.208 pacientes.html:45 Lista de pacientes disponible en el navegador |

Terminal (Servidor):

|  |
| --- |
| 22:10:15:269 Conexión establecida con el cliente  22:10:15:272 Conexión establecida con el cliente  22:10:15:310 Petición del cliente: Pacientes  22:10:15:353 Base de datos abierta  22:10:15:459 Listado de pacientes enviado al cliente  22:10:15:465 Base de datos cerrada  22:10:18:371 Conexión establecida con el cliente  22:10:18:376 Petición del cliente: Borrar paciente  22:10:18:378 Base de datos abierta  22:10:18:428 Paciente María y sus datos asociados eliminados de la base de datos  22:10:18:429 Base de datos cerrada  22:10:19:088 Conexión establecida con el cliente  22:10:19:090 Conexión establecida con el cliente  22:10:19:163 Petición del cliente: Pacientes  22:10:19:167 Base de datos abierta  22:10:19:179 Listado de pacientes enviado al cliente  22:10:19:180 Base de datos cerrada |

Para este caso volvemos a observar el mismo comportamiento que se dio para añadir un paciente, ya que se obtiene la lista de pacientes en dos ocasiones (Listado inicial y listado de pacientes actualizado).







### 5.1.4. Obtener datos de movimiento de un paciente

Consola del navegador:

|  |
| --- |
| 20:49:22.115 Navigated to http://192.168.1.33:8124/pacientes.html  20:49:22.634 VM196 pacientes\_node.js:28 Conexión establecida con el servidor  20:49:22.638 VM196 pacientes\_node.js:39 Solicitud de listado de pacientes enviada  20:49:22.847 VM196 pacientes\_node.js:41 Lista de pacientes recibida  20:49:22.848 pacientes.html:45 Lista de pacientes disponible en el navegador  20:49:27.368 Navigated to http://192.168.1.33:8124/evolucion.html?var1=3&var2=Javier&var3=Perez&var4=h  20:49:27.810 evolucion\_node.js:8 Conexión establecida con el servidor  20:49:27.813 evolucion\_node.js:21 Solicitud de listado de movimientos de Javier enviada  20:49:27.981 evolucion\_node.js:23 Lista de movimientos de Javier recibida  20:49:27.983 evolucion.html?var1=3&var2=Javier&var3=Perez&var4=h:114 Lista de movimietos de Javier Perez disponible en el navegador |

Terminal (Servidor):

|  |
| --- |
| 20:49:22:647 Conexión establecida con el cliente  20:49:22:650 Conexión establecida con el cliente  20:49:22:688 Petición del cliente: Pacientes  20:49:22:727 Base de datos abierta  20:49:22:838 Listado de pacientes enviado al cliente  20:49:22:843 Base de datos cerrada  20:49:27:418 Conexión establecida con el cliente  20:49:27:813 Conexión establecida con el cliente  20:49:27:894 Petición del cliente: Datos paciente  20:49:27:897 Base de datos abierta  20:49:27:965 Listado de movimientos de Javier enviado al cliente  20:49:27:966 Base de datos cerrada |

Aquí se observa el mismo comportamiento que se da en la función para obtener pacientes. Cuando el listado de pacientes está disponible en el navegador, el usuario presiona el botón para mostrar los datos asociados al paciente, y se vuelva a hacer una petición al servidor, donde se abre la base de datos, se obtienen los datos requeridos, se envían al cliente y se cierra la base de datos. En última instancia el cliente recibe los datos y se listan en el navegador.

### 5.1.5. Añadir datos de movimiento

Consola del navegador:

|  |
| --- |
| 10:58:02.367 Navigated to http://192.168.1.33:8124/pacientes.html  10:58:02.906 :8124/JS/pacientes\_node.js:28 Conexión establecida con el servidor  10:58:02.911 :8124/JS/pacientes\_node.js:39 Solicitud de listado de pacientes enviada  10:58:03.138 :8124/JS/pacientes\_node.js:41 Lista de pacientes recibida  10:58:03.140 :8124/pacientes.html:45 Lista de pacientes disponible en el navegador  10:58:05.236 Navigated to http://192.168.1.33:8124/evolucion.html?var1=3&var2=Javier&var3=Perez&var4=h  10:58:05.664 VM216 evolucion\_node.js:8 Conexión establecida con el servidor  10:58:05.669 VM216 evolucion\_node.js:21 Solicitud de listado de movimientos de Javier enviada  10:58:05.814 VM216 evolucion\_node.js:23 Lista de movimientos de Javier recibida  10:58:32.277 evolucion.html?var1=3&var2=Javier&var3=Perez&var4=h:57 Datos introducidos de forma correcta  10:58:32.290 VM216 evolucion\_node.js:119 Conexión establecida con el servidor  10:58:32.291 VM216 evolucion\_node.js:144 Datos de movimiento de Javier enviados  10:58:32.436 Navigated to http://192.168.1.33:8124/evolucion.html?var1=3&var2=Javier&var3=Perez&var4=h  10:58:32.904 evolucion\_node.js:8 Conexión establecida con el servidor  10:58:32.912 evolucion\_node.js:21 Solicitud de listado de movimientos de Javier enviada  10:58:33.050 evolucion\_node.js:23 Lista de movimientos de Javier recibida  10:58:33.053 evolucion.html?var1=3&var2=Javier&var3=Perez&var4=h:114 Lista de movimientos de Javier Perez disponible en el navegador |

Terminal (Servidor):

|  |
| --- |
| 10:58:02:919 Conexión establecida con el cliente  10:58:02:921 Conexión establecida con el cliente  10:58:02:972 Petición del cliente: Pacientes  10:58:03:019 Base de datos abierta  10:58:03:128 Listado de pacientes enviado al cliente  10:58:03:136 Base de datos cerrada  10:58:05:297 Conexión establecida con el cliente  10:58:05:666 Conexión establecida con el cliente  10:58:05:751 Petición del cliente: Datos paciente  10:58:05:754 Base de datos abierta  10:58:05:804 Listado de movimientos de Javier enviado al cliente  10:58:05:805 Base de datos cerrada  10:58:32:297 Conexión establecida con el cliente  10:58:32:310 Petición del cliente: Añadir datos de paciente  10:58:32:314 Base de datos abierta  10:58:32:385 Datos de movimiento de Javier guardados en la base de datos  10:58:32:385 Base de datos cerrada  10:58:32:495 Conexión establecida con el cliente  10:58:32:906 Conexión establecida con el cliente  10:58:32:980 Petición del cliente: Datos paciente  10:58:32:987 Base de datos abierta  10:58:33:038 Listado de movimientos de Javier enviado al cliente  10:58:33:039 Base de datos cerrada |

### Como podemos observar se realizan las mismas interacciones que en el caso en el que se obtienes los pacientes. Cuando el usuario selecciona el archivo CSV local, le asocia una fecha y presiona el botón para añadir los datos, se realiza una conexión con el servidor, donde se abre la base de datos, se guardan los datos seleccionados por el usuario y se cierra la base de datos. Como se ha realizado un cambio en la base de datos, el servidor establece una conexión con el cliente para forzarle a actualizar su lista de movimientos y así se visualizará el nuevo movimiento añadido.

### 5.1.6. Borrar una sesión de movimientos

Consola del navegador:

|  |
| --- |
| 21:54:16.374 Navigated to http://192.168.1.33:8124/pacientes.html  21:54:16.867 VM90 pacientes\_node.js:28 Conexión establecida con el servidor  21:54:16.874 VM90 pacientes\_node.js:39 Solicitud de listado de pacientes enviada  21:54:17.082 VM90 pacientes\_node.js:41 Lista de pacientes recibida  21:54:17.084 pacientes.html:45 Lista de pacientes disponible en el navegador  21:54:19.187 Navigated to http://192.168.1.33:8124/evolucion.html?var1=3&var2=Javier&var3=Perez&var4=h  21:54:19.609 VM118 evolucion\_node.js:8 Conexión establecida con el servidor  21:54:19.615 VM118 evolucion\_node.js:21 Solicitud de listado de movimientos de Javier enviada  21:54:19.786 VM118 evolucion\_node.js:23 Lista de movimientos de Javier recibida  21:54:19.789 evolucion.html?var1=3&var2=Javier&var3=Perez&var4=h:114 Lista de movimientos de Javier Perez disponible en el navegador  21:54:23.016 VM118 evolucion\_node.js:353 Conexión establecida con el servidor  21:54:23.017 VM118 evolucion\_node.js:367 Solicitud para borrar datos de Javier enviada  21:54:23.300 Navigated to http://192.168.1.33:8124/evolucion.html?var1=3&var2=Javier&var3=Perez&var4=h  21:54:23.779 evolucion\_node.js:8 Conexión establecida con el servidor  21:54:23.785 evolucion\_node.js:21 Solicitud de listado de movimientos de Javier enviada  21:54:23.902 evolucion\_node.js:23 Lista de movimientos de Javier recibida |

Terminal (Servidor):

|  |
| --- |
| 21:54:16:878 Conexión establecida con el cliente  21:54:16:882 Conexión establecida con el cliente  21:54:16:927 Petición del cliente: Pacientes  21:54:16:968 Base de datos abierta  21:54:17:073 Listado de pacientes enviado al cliente  21:54:17:079 Base de datos cerrada  21:54:19:240 Conexión establecida con el cliente  21:54:19:611 Conexión establecida con el cliente  21:54:19:696 Petición del cliente: Datos paciente  21:54:19:698 Base de datos abierta  21:54:19:770 Listado de movimientos de Javier enviado al cliente  21:54:19:771 Base de datos cerrada  21:54:23:021 Conexión establecida con el cliente  21:54:23:032 Petición del cliente: Borrar datos de paciente  21:54:23:035 Base de datos abierta  21:54:23:076Datos de movimiento del paciente Javier borrados  21:54:23:077 Base de datos cerrada  21:54:23:390 Conexión establecida con el cliente  21:54:23:782 Conexión establecida con el cliente  21:54:23:869 Petición del cliente: Datos paciente  21:54:23:877 Base de datos abierta  21:54:23:898 Listado de movimientos de Javier enviado al cliente  21:54:23:899 Base de datos cerrada |

En esta situación se seguirán los mismos pasos que en el apartado anterior. Veremos que lo único que cambio es la solicitud al servidor, que en vez de guardar datos, se borrarán, por lo que se forzará al cliente a actualizar su lista de movimientos para visualizar los cambios realizados.

### 5.1.7. Mostrar un gráfico de un movimiento

Consola del navegador:

|  |
| --- |
| 21:59:03.602 Navigated to http://192.168.1.33:8124/pacientes.html  21:59:04.096 VM86 pacientes\_node.js:28 Conexión establecida con el servidor  21:59:04.100 VM86 pacientes\_node.js:39 Solicitud de listado de pacientes enviada  21:59:04.313 VM86 pacientes\_node.js:41 Lista de pacientes recibida  21:59:04.315 pacientes.html:45 Lista de pacientes disponible en el navegador  21:59:06.035 Navigated to http://192.168.1.33:8124/evolucion.html?var1=1&var2=Borja&var3=Gonzalez&var4=h  21:59:06.456 evolucion\_node.js:8 Conexión establecida con el servidor  21:59:06.460 evolucion\_node.js:21 Solicitud de listado de movimientos de Borja enviada  21:59:06.708 evolucion\_node.js:23 Lista de movimientos de Borja recibida  21:59:06.715 evolucion.html?var1=1&var2=Borja&var3=Gonzalez&var4=h:114 Lista de movimietos de Borja Gonzalez disponible en el navegador  21:59:09.157 evolucion\_node.js:88 Gráfico de movimiento en el plano Transversal de Borja Gonzalez disponible en el navegador |

Terminal (Servidor):

|  |
| --- |
| 21:59:04:108 Conexión establecida con el cliente  21:59:04:112 Conexión establecida con el cliente  21:59:04:148 Petición del cliente: Pacientes  21:59:04:194 Base de datos abierta  21:59:04:305 Listado de pacientes enviado al cliente  21:59:04:311 Base de datos cerrada  21:59:06:084 Conexión establecida con el cliente  21:59:06:458 Conexión establecida con el cliente  21:59:06:541 Petición del cliente: Datos paciente  21:59:06:543 Base de datos abierta  21:59:06:658 Listado de movimientos de Borja enviado al cliente  21:59:06:661 Base de datos cerrada |

Para mostrar un gráfico de movimiento, el usuario tendrá que situarse en el listado de movimientos de un paciente, por lo que observamos el mismo comportamiento que en el apartado de obtención de datos de un movimiento. Cuando el usuario esté situado en el listado, presionará el botón del movimiento que desee visualizar, y como los datos de movimiento ya están disponibles, solo hará falta pasar los datos a la herramienta chart.js que se encargará de graficar el movimiento en la pantalla del navegador.

### 5.1.8. Mostrar un gráfico de evolución de un movimiento

Consola del navegador:

|  |
| --- |
| 22:05:33.586 Navigated to http://192.168.1.33:8124/pacientes.html  22:05:34.084 VM88 pacientes\_node.js:28 Conexión establecida con el servidor  22:05:34.087 VM88 pacientes\_node.js:39 Solicitud de listado de pacientes enviada  22:05:34.294 VM88 pacientes\_node.js:41 Lista de pacientes recibida  22:05:34.296 pacientes.html:45 Lista de pacientes disponible en el navegador  22:05:35.582 Navigated to http://192.168.1.33:8124/evolucion.html?var1=1&var2=Borja&var3=Gonzalez&var4=h  22:05:36.006 evolucion\_node.js:8 Conexión establecida con el servidor  22:05:36.009 evolucion\_node.js:21 Solicitud de listado de movimientos de Borja enviada  22:05:36.257 evolucion\_node.js:23 Lista de movimientos de Borja recibida  22:05:36.264 evolucion.html?var1=1&var2=Borja&var3=Gonzalez&var4=h:114 Lista de movimietos de Borja Gonzalez disponible en el navegador  22:05:39.533 evolucion\_node.js:150 Conexión establecida con el servidor  22:05:39.534 evolucion\_node.js:164 Solicitud de datos de evolución de Borja enviada  22:05:39.667 evolucion\_node.js:166 Datos de evolución de Borja recibidos  22:05:39.750 evolucion\_node.js:345 Gráfico de Evolución en el plano Transversal de Borja Gonzalez disponible en el navegador |

Terminal (Servidor):

|  |
| --- |
| 22:05:34:094 Conexión establecida con el cliente  22:05:34:099 Conexión establecida con el cliente  22:05:34:136 Petición del cliente: Pacientes  22:05:34:177 Base de datos abierta  22:05:34:286 Listado de pacientes enviado al cliente  22:05:34:292 Base de datos cerrada  22:05:35:629 Conexión establecida con el cliente  22:05:36:008 Conexión establecida con el cliente  22:05:36:090 Petición del cliente: Datos paciente  22:05:36:095 Base de datos abierta  22:05:36:206 Listado de movimientos de Borja enviado al cliente  22:05:36:209 Base de datos cerrada  22:05:39:536 Conexión establecida con el cliente  22:05:39:544 Petición del cliente: Datos de Evolucion paciente  22:05:39:554 Base de datos abierta  22:05:39:662 Listado de datos de evolución de movimiento de Borja enviado al cliente  22:05:39:665 Base de datos cerrada |

Para mostrar un gráfico de evolución de un movimiento, el usuario tendrá que situarse en el listado de movimientos de un paciente, por lo que observamos el mismo comportamiento que en el apartado de obtención de datos de un movimiento. Cuando el usuario esté situado en el listado, presionará el botón de evolución del movimiento que desee visualizar, y se realizará una nueva conexión con el servidor, donde se abre la base de datos, se obtienen los datos requeridos, se cierra la base de datos y se envían los datos al cliente. Finalmente, el cliente pasa los datos a la herramienta chart.js, que se encarga de graficar los datos de evolución en la pantalla del navegador.

# 6. Resultados y conclusiones

Como se ha explicado al principio de esta memoria, el objetivo principal del trabajo era crear una aplicación web con su base de datos correspondiente cuyo fin fuera facilitar el seguimiento y análisis de la información de movilidad cervical de pacientes. A continuación indico si se han cumplido los requisitos que en su momento se determinaron como necesarios y enumero posibles líneas de trabajo futuras asociadas a este proyecto.

## 6.1. Resultados

Como se pudo ver en el capítulo de pruebas, el proyecto ha cumplido con todos los requisitos solicitados por el cliente. En resumen:

La aplicación web es capaz de comunicarse con un servidor que tiene acceso a una base de datos para solicitar o modificar datos en consecuencia a lo que solicite el cliente. Además la aplicación tiene la capacidad de mostrar los datos que solicite el usuario, pudiendo ser en forma de tablas de pacientes o tablas de datos de movimientos de pacientes y en forma de gráficos de sesiones de movimiento o gráficos de evolución de movimientos.

Además, durante el desarrollo del trabajo se ha expuesto la aplicación web a fisioterapeutas del Hospital de La Salle (un ejemplo de usuario final), que han transmitido su satisfacción con el resultado y han resaltado que esta tiene una enorme utilidad a la hora tanto de tratar pacientes con problemas cervicales como de realizar un análisis de datos para estudios clínicos. A consecuencia de esto, la aplicación acabará siendo utilizada en el hospital antes nombrado, integrándose con el flujo de trabajo del equipo de fisioterapia. En línea con esto, los fisioterapeutas hicieron algunas sugerencias para mejorar la aplicación web, algunas de las cuales se implementaron en un tiempo muy breve, debido al escaso coste en tiempo de desarrollo de las mismas. Otras, sin embargo, se dejaron para más adelante, convirtiéndose posibles desarrollos futuros.

Por último, cabe mencionar que la aplicación web desarrollada es de fácil uso y no es necesario que el usuario (probablemente un fisioterapeuta) tenga conocimientos específicos sobre las tecnologías utilizadas en este proyecto.

## 6.2. Conclusiones

Al implementar este trabajo he conseguido reforzar conocimientos aprendidos a lo largo de la carrera. En especial en el desarrollo de plataformas web, el manejo de bases de datos, la implementación de un servidor con su lógica de negocio, el uso de sockets para establecer una comunicación bidireccional y la integración de librerías para añadir la funcionalidad necesaria para cumplir en los requisitos.

Otra cuestión que merece mención es el proceso de investigación conjunto entre mi tutor y yo para encontrar las tecnologías adecuadas que se ajustasen a las necesidades y requisitos del proyecto. Para mí, la capacidad para llevar a cabo esta tarea con éxito es una cualidad fundamental que debe adquirir cualquier ingeniero, pues en el mundo laboral se valora su capacidad de afrontar los problemas, independientemente de si se poseían de antemano los conocimientos necesarios o si estos debieron adquiriste durante el desarrollo.

Como he mencionado antes, durante el desarrollo de la aplicación me he reunido con el cliente y juntos hemos trabajado para crear un proyecto real. Este hecho es de una importancia fundamental, ya que me ha enseñado a trabajar con distintas personas para alcanzar unos objetivos concretos, algo que ocurre constantemente en el mundo laboral y que de alguna forma complementa a mi formación para introducirme a los procesos de una empresa a la hora de desarrollar un producto.

## 6.3. Líneas de trabajo futuras

A lo largo del desarrollo de este proyecto, se han planteado diferentes rutas que se podrían seguir para mejorar el proyecto realizado, pero que por limitaciones de tiempo y recursos no han podido implementarse en el trabajo presentado.

Como ya se ha explicado, la aplicación web obtiene sus datos de movimiento del dispositivo Werium Basic Pro. Éste dispositivo no solo puede medir el rango de movimiento cervical, sino que es capaz de medir rangos de movimiento de todas las extremidades del cuerpo, por lo que es posible utilizarlo para recoger datos de otros movimientos y así poder evaluar diferentes situaciones y dolencias. Sería relativamente sencillo incorporar otros rangos de movimiento a la aplicación web, ya que la parte más compleja de la misma es la relativa a la extracción de los datos, el almacenamiento de y su representación en forma de gráfico, aspectos comunes a todos los tipos de movimiento que el sensor nombrado puede capturar. Por ello, podrían incluirse con poco esfuerzo otras secciones en la interfaz gráfica cuyo fin fuera mostrar otros rangos de movimiento.

Otra mejora fácil de implementar sería el despliegue en remoto, posiblemente en un servidor en la nube, de la aplicación web.

Como último apunte, también se podría implementar un sistema más seguro, ya que la aplicación web no dispone de ningún control de acceso para acceder a los datos. Este hecho no es problemático ahora mismo ya que solo puede accederse a la aplicación en su despliegue actual desde una red privada, pero podría resultar muy importante en el futuro, sobre todo para el seguimiento de pacientes individuales. No es un aspecto tan relevante para el estudio clínico, ya que los nombres podrían anonimizarse para evitar vulnerar leyes de protección de datos.

# 7. GitHub

Quiero acabar esta memoria mencionando que he estado trabajando con un repositorio remoto GitHub. He utilizado este repositorio para tener mi proyecto en más de una ubicación y además para compartir los progresos con mi tutor. Además de tener disponibilidad para ver el código, mi tutor y yo hemos podido realizar cambios sobre el repositorio para hacer correcciones, por lo que hemos ido actualizando las versiones del repositorio, trabajando así de forma conjunta en un proyecto común, algo que se realiza de forma habitual en el mundo empresarial.

El link para acceder al repositorio que contiene todo lo que hace falta para poner en funcionamiento la aplicación web y además esta memoria, es el siguiente:

https://github.com/BorjaGD94/TFG/

**Bibliografía**

Virgilio F. Ferrario \*, Chiarella Sforza, Graziano Serrao, GianPiero Grassi, Erio Mossi - Active range of motion of the head and cervical spine: a three-dimensional investigation in healthy young adults

<https://en.wikipedia.org/wiki/Socket.IO>

https://socket.io/docs/

https://developer.mozilla.org/es/docs/Web/JavaScript

<https://es.wikipedia.org/wiki/JavaScript>

<https://es.wikipedia.org/wiki/Node.js>

https://www.netconsulting.es/blog/nodejs/

https://developer.mozilla.org/es/docs/Web/HTML

<https://es.wikipedia.org/wiki/HTML>

https://developer.mozilla.org/es/docs/Web/CSS

<https://es.wikipedia.org/wiki/Hoja_de_estilos_en_cascada>

<https://es.wikipedia.org/wiki/SQLite>

https://www.sqlite.org/

https://github.com/mapbox/node-sqlite3

<https://es.wikipedia.org/wiki/Base_de_datos>

<https://es.wikipedia.org/wiki/ACID>

<https://github.com/kikinteractive/app>