Índice general

1. Diseño **1**

1.1. Diseño de la interfaz 1
1.1.1. Principios de diseño de la experiencia de usuario 1
1.1.2. Principios de diseño de la interfaz de usuario 1
1.1.3. Fuentes y tipografía 3
1.1.4. Interacción persona-ordenador 4
1.1.5. *Mockup* 6
1.2. Diseño de la navegación 6
1.2.1. Semántica de la navegación 6
1.2.2. Sintaxis de navegación 7
1.3. Tecnologías utilizadas 7
1.3.1. Frameworks 7
1.3.2. Lenguajes de programación 16
1.3.3. *Modules* 17
1.3.4. Librerías 17
1.3.5. *Middleware* 18
1.4. Herramientas utilizadas 18
1.5. Seguridad 19
1.5.1. ¿Qué es Bcrypt? 19
1.5.2. Ventajas de Bcrypt 20
1.5.3. Funcionamiento 21
1.5.4. Cifrado Blowfish 21
1.6. Diseño de la arquitectura 21
1.6.1. SPA 21
1.6.2. Particularidades de JavaScript 21
1.6.3. *Callbacks* en JavaScript 22
1.6.4. Patrones de diseño 23
1.6.5. *Observer* 26
1.6.6. *Factory Method* 28
1.6.7. *Proxy* 28
1.6.8. *Data Mapper* 28
1.6.9. API REST 29

1.6.10. Estructura de directorios 29
1.7. Diagramas 32

2. Implementación **37**
2.1. Implementación del primer incremento 37
2.2. Implementación del segundo incremento 40
2.3. Implementación del tercer incremento 49

Índice de figuras

1.1. USN-001 8
1.2. USN-002 8
1.3. USN-003 9
1.4. USN-004 9
1.5. USN-005 10
1.6. USN-006 10
1.7. USN-007 11
1.8. USN-008 11
1.9. USN-009 12
1.10. USN-010 12
1.11. USN-011 (Paciente) 13
1.12. USN-011 (Psicólogo) 13
1.13. USN-012 13
1.14. Mapa de navegación 14
1.15. Mean STACK - MVC y MVVM 25
1.16. Esquema *data-binding* 27
1.17. *Proxy pattern*[46] 29
1.18. Estructura de directorios 31
1.19. Patrón MVC-MVVM 32
1.20. Data mapper 33
1.21. Diagrama de secuencia del CU-003 34
1.22. Diagrama de secuencia del CU-005 35

Índice de tablas

1.1. Perfil paciente 2
1.2. Perfil psicólogo 2
1.3. Correspondencia entre CU y USN 7
1.4. Comparativa Java y JavaScript 22

Capítulo 1

Diseño

1.1. Diseño de la interfaz

El diseño de la interfaz de usuario crea un medio eficaz de comunicación entre los seres humanos y la computadora.

1.1.1. Principios de diseño de la experiencia de usuario

Para poder realizar un diseño de interfaz centrado en el usuario, primero hemos de conocer quiénes y cómo son nuestros usuarios. Durante el desarrollo del modelo de negocio pudimos entrevistar a nuestros posibles clientes (futuros usuarios) y pudimos hacernos una idea de cuál es nuestro nicho de mercado.

En nuestra plataforma, tenemos dos tipos de perfiles: Los pacientes1.1 y los psicólogos1.2.

Hacer un diseño centrado en el usuario implica tenerlo presente a lo largo de todo el proceso de diseño y tratar de entender cuáles son sus necesidades, intereses y limitaciones.

1.1.2. Principios de diseño de la interfaz de usuario

Layout

Nuestra interfaz seguirá la regla de los tercios para ayudar a concretar el enfoque del usuario. La regla de los tercios es una forma de composición para ordenar objetos dentro de la imagen al dividirla en nueve partes iguales utilizando dos líneas imaginarias paralelas y equiespaciadas de forma horizontal y otras dos con la mismas características de forma vertical. Los puntos donde se cortan las líneas son los puntos de intersección y sirven para distribuir los elementos de la página. Entre los puntos de intersección se ha de ubicar el centro de atención para crear una imagen estéticamente agradable y equilibrada[38].

Pacientes	
Demografía	España
Edad	Mayores de 16 años
Experiencia laboral	Estudios mínimos
Contexto profesional	Cualquiera
Necesidades e intereses	Buscar al psicólogo más adecuado para tratar su problemática.
	Reducir las preocupaciones de aquellas poblaciones que no puedan acudir físicamente a una consulta por motivos de desplazamiento, urgencias, estigmas sociales...
¿Cuándo y dónde utilizaran este servicio?	Cuando surja la necesidad de contactar con un psicólogo y a través de un ordenador.

Tabla 1.1: Perfil paciente

Psicólogos	
Demografía	España
Edad	Mayoría de edad
Experiencia laboral	Estudios superiores
Contexto profesional	Psicólogos con especialidad clínica o sanitaria que estean colegiados para el ejercicio de la actividad profesional.
Necesidades e intereses	Conseguir un flujo constante de pacientes.
¿Cuándo y dónde utilizaran este servicio?	Periódicamente y a través de un ordenador.

Tabla 1.2: Perfil psicólogo

La librería Bootstrap, que es una de las librerías escogidas para el diseño de la página, posee un sistema de cuadrículas[29] que permite dividir la página en filas y columnas. La cuadrícula siempre divide la página en 12 columnas. Como 12 es múltiplo 3, se puede aplicar fácilmente la regla de los tercios a este sistema de cuadrículas.

Por otra parte, también se debe tener en cuenta el público que va a tener nuestra página a la hora de disponer los objetos en ella. La jerarquía visual es importante porque dirige la atención de los ojos del usuario. En nuestro caso, nuestra población es occidental, por lo que el usuario comienza a leer la página desde la esquina superior izquierda. Por este motivo, es interesante situar el logo de nuestra plataforma en esta esquina y la navegación a su lado de forma horizontal.

Colores

En la página predominan los colores que, para nuestros futuros usuarios pertenecientes a la población occidental, transmiten:

- Verde: Seguridad, salud
- Azul: Seguridad, confianza, estabilidad, veracidad, lealtad

El verde y azul son colores análogos, puesto que se encuentran uno pegado a otro en la rueda de color.

Es importante tener en cuenta la cultura de nuestros usuarios, puesto que entre culturas pueden existir connotaciones de los colores totalmente opuestas.

Para el color de las letras se han utilizado el blanco y el negro (colores neutros) y hacen contraste con el color predominante del background y de los elementos de la página.

1.1.3. Fuentes y tipografía

La tipografía predominante en toda la página es Lato[32], pero para aquellos mensajes que queramos resaltar en algún momento puntual se utilizará Merriweather[33].

La elección de las fuentes se debe a que la tipografía contextualiza el tipo de contenido de la página, no sólo lo hace a nivel verbal sino que también de forma visual. El lector, primero identifica los patrones gráficos de la página, y después, analiza el lenguaje y lee.

- Lato Es de tipo *sans serif*[1] transicional: Los trazos son fuertes y los caracteres son derechos y uniformes. Transmite sencillez y modernez. Se suele utilizar en tecnología y aplicaciones portables.

[1] ***Sans serif*:** Fuentes con ausencia de *serif*.

- Merriweather Es de tipo egipcio (*slab serif*): Existe muy poco contraste entre los trazos, y la *serif* es gruesa. Transmite autoridad pero en tono amistoso. Se suele utilizar en *marketing* y aplicaciones promocionales.

1.1.4. Interacción persona-ordenador

Sociedad de la información

La tendencia que existe hoy en día es digitalizar toda clase de servicios. Los servicios TIC permiten acceder a la salud, el ocio, el bienestar, la formación... derechos básicos que pertenecen a toda la ciudadanía.

A pesar de ello, existen personas que no tienen fácil acceso a este tipo de servicios ya sea por razones de limitaciones geográficas como es el caso del rural, aspectos de género culturales o religiosos, la edad, aspectos socieconómicos (personas bajo el umbral de difícil acceso a las TIC) y la discapacidad. Aunque el proyecto no pueda abarcar a toda clase de colectivos por limitaciones de tiempo, sí es importante tenerlos en cuenta para el futuro.

Diversidad funcional

La diversidad funcional es inherente al ser humano: Una persona experimenta variaciones en su capacidad de dependencia a lo largo de su vida, aunque sea de manera temporal. Por ejemplo, a veces nos sentimos limitados por no poder utilizar el ordenador portátil al no tener batería; o con la edad, la gente obtiene discapacidades que antes no tenía como la pérdida de vista.

Accesibilidad

La accesibilidad la interacción persona-ordenador es el conjunto de propiedades que debe incorporar un producto, servicio o sistema, de forma que el mayor número posible de personas, en el mayor número posible de circunstancias, que sea comercialmente práctico tener en cuenta, puede acceder a él y usarlo.[36]

Un producto es accesible si, por mucho tiempo o esfuerzo que requiera, la tarea puede realizarse.

Usabilidad

La usabilidad es la efectividad, eficiencia y satisfacción con la que usuarios específicos pueden abarcar unos objetivos determinados en un entorno particular[7].

Por tanto, los sistemas deben ser usables y accesibles para que la inmensa mayoría de las personas puedan utilizarlo con calidad.

Diseño universal

El diseño universal es la estrategia que tiene como objetivo diseñar productos y servicios que puedan se utilizados por el mayor número de personas, considerando que existe una amplia variedad de habilidades humanas y no una habilidad media, sin necesidad de llevar a cabo una adaptación o diseñi especializado, simplificando la vida de todas las personas con independencia de su edad, talla o capacidad[39].

En la práctica, esto supone un auténtico reto, por lo que el diseño para todos acaba equivaliendo a diseño para "la mayoría". Aquí es donde se entra a valorar la posibilidad de añadir productos de apoyo para que pueda ser utilizado por las minorías.

En España hay leyes que tratan de combatir con este tipo de discriminación como la Ley 51/2005, de 2 de diciembre, de igualdad de oportunidades, no discriminación y accesibilidad universal de las personas con discapacidad. Por otra parte, AENOR posee la norma UNE 139803:2012. Requisitos de Accesibilidad para contenidos en la web[27].

Para lograr que nuestra plataforma pueda ser utilizada por el mayor número de personas posible se ha decidido seguir los principios heurísticos de Nielsen y Molich.

Principios heurísticos

La Interacción Persona Ordenador (IPO) presenta a la Evaluación Heurística (EH) como un método de evaluación de la usabilidad por inspección que debe ser llevado a cabo a través de unos principios heurísticos previamente establecidos. Por ser un método de evaluación de la usabilidad, tiene como objetivo medir la calidad de la interfaz de cualquier sistema interactivo en relación a su facilidad para ser aprendido y usado por un determinado grupo de usuarios en un determinado contexto de uso.

Aplicar los principios heurísticos nos sirve de guía para el proceso de diseño y nos permite identificar problemas de usabilidad en las interfaces de usuario. En nuestra aplicación, tendremos en cuenta los principios heurísticos de Nielsen y Molich:

1. Visibilidad del estado del sistema El sistema debe siempre mantener a los usuarios informados del estado del sistema, con una realimentación apropiada y en un tiempo razonable.

2. Lenguaje de los usuarios El sistema debe hablar el lenguaje de los usuarios, utilizando convenciones del mundo real, disponiendo la información en un orden natural y lógico.

3. Control y libertad para el usuario Los usuarios eligen a veces funciones del sistema por error y necesitan una salida del estado indeseado sin tener que pasar por un diálogo extendido.

4. Consistencia y estándares Se deben seguir las normas y convencios de la plataforma para las que se implementa el sistema.

5. Ayuda a los usuarios para reconocimiento, diagnóstico y recuperación de errores Los mensajes de error deben expresarse en un lenguaje claro y explicativo.

6. Prevención de errores Se debe prevenir la aparición de errores que mejor que generar buenos mensajes de error.

7. Reconocimiento antes de cancelación El usuario no debería tener que recordar la información de una parte del diálogo a la otra.

8. Flexibilidad y eficiencia de uso Las instrucciones para el uso del sistema deben ser visibles o fácilmente accesibles siempre que se necesiten.

9. Estética de diálogos y diseño minimalista No deben contener la información que sea inaplicable o se necesite raramente. Cada unidad adicional de la información en un diálogo compite con las unidades relevantes de la información y disminuye su visibilidad.

10. Ayuda general y documentación Aunque es mejor si el sistema se pueda usar sin documentación, puede ser necesario disponer de ayuda y documentación. Ésta ha de ser fácil de buscar, centrada en las tareas del usuario, tener información de las etapas a realizar y que no sea muy extensa. [44]

1.1.5. *Mockup*

1.2. Diseño de la navegación

Para definir las rutas de navegación que permiten a los usuarios acceder al contenido y a las funciones de la plataforma web, debemos identificar la semántica de la navegación para los distintos usuarios del sitio y definir la sintaxis para efectuar la navegación.

1.2.1. Semántica de la navegación

Se define a partir del rol que toma un actor dentro de un caso de uso, ya que cada actor tiene distintos requisitos de navegación. A medida que un usuario interactúa con la web, encuentra una serie de unidades semánticas de navegación (USN) que son un conjunto de estructuras de información y navegación relacionadas que colaboran para el cumplimiento de un subconjunto de requisitos de dicho usuario".

Una USN está compuesta por un conjunto de elementos de navegación llamados formas de navegar (FdN) que representan la mejor ruta de navegación para

lograr una meta específica. Está formada por un conjunto de nodos de navegación conectados por vínculos, que en algún pueden tratarse de otra USN.

Para cada caso de uso, se procede a diseñar su USN. Las correspondencias se pueden apreciar en la tabla1.3.

Caso de uso	**Unidad semántica de navegación**
CU-001	USN-001 (Paciente)
CU-002	USN-002
CU-003	USN-003
CU-004	USN-004
CU-005	USN-005
CU-006	USN-006
CU-007	USN-007
CU-008	USN-008
CU-009	USN-009
CU-010	USN-010
CU-011	USN-011 (Paciente)
	USN-011 (Psicólogo)
CU-012	USN-012

Tabla 1.3: Correspondencia entre CU y USN

1.2.2. Sintaxis de navegación

La sintaxis de navegación refleja la mecánica de la navegación para cada USN. Para nuestra plataforma web, se ha diseñado un mapa del sitio que describe todas las posibles navegaciones existentes. Para cada tipo de usuario (perfil o psicólogo), existe un mapa1.14 diferente.

1.3. Tecnologías utilizadas

1.3.1. Frameworks

Un *framework* es una arquitectura de *software* que modela las relaciones generales de las entidades del dominio, y provee una estructura y una especial metodología de trabajo, la cual extiende o utiliza las aplicaciones del dominio.

Los *frameworks* involucrados en nuestro proyecto son MEAN Stack, Bootstrap y Semantic UI.

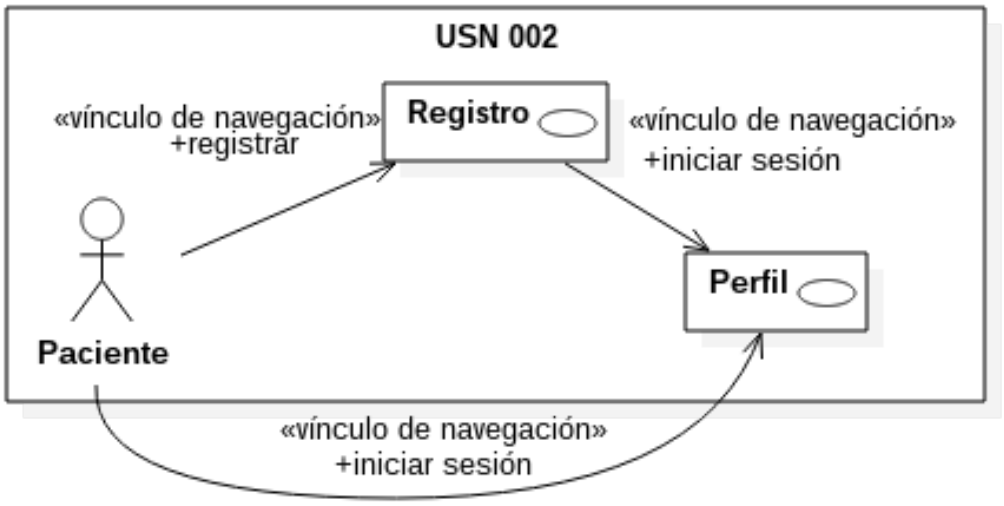

Figura 1.1: USN-001

Figura 1.2: USN-002

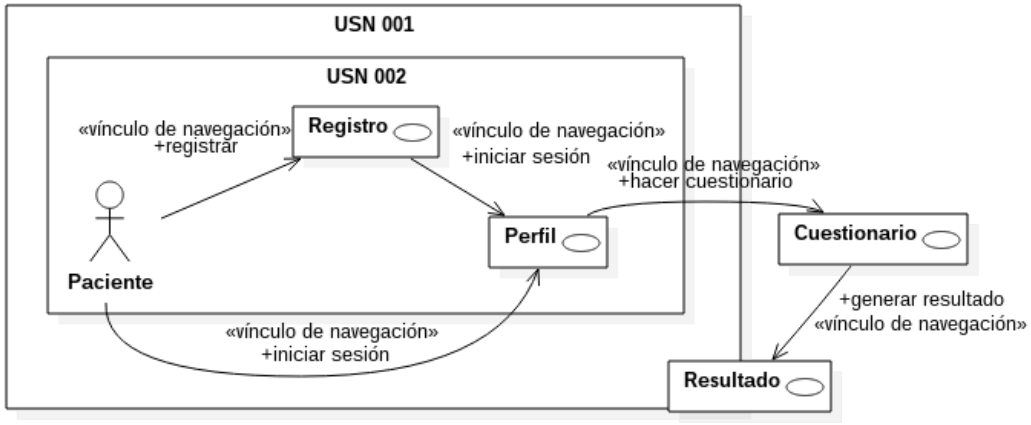

Figura 1.3: USN-003

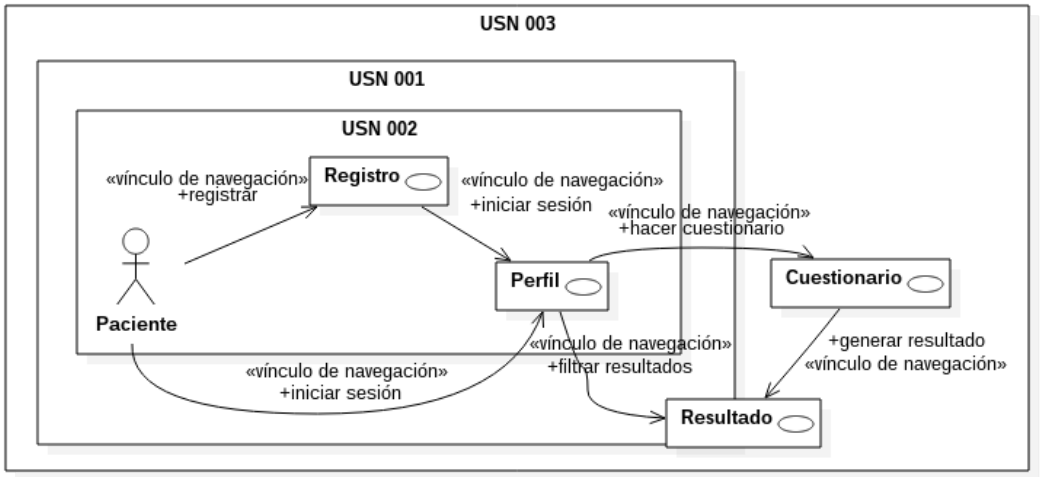

Figura 1.4: USN-004

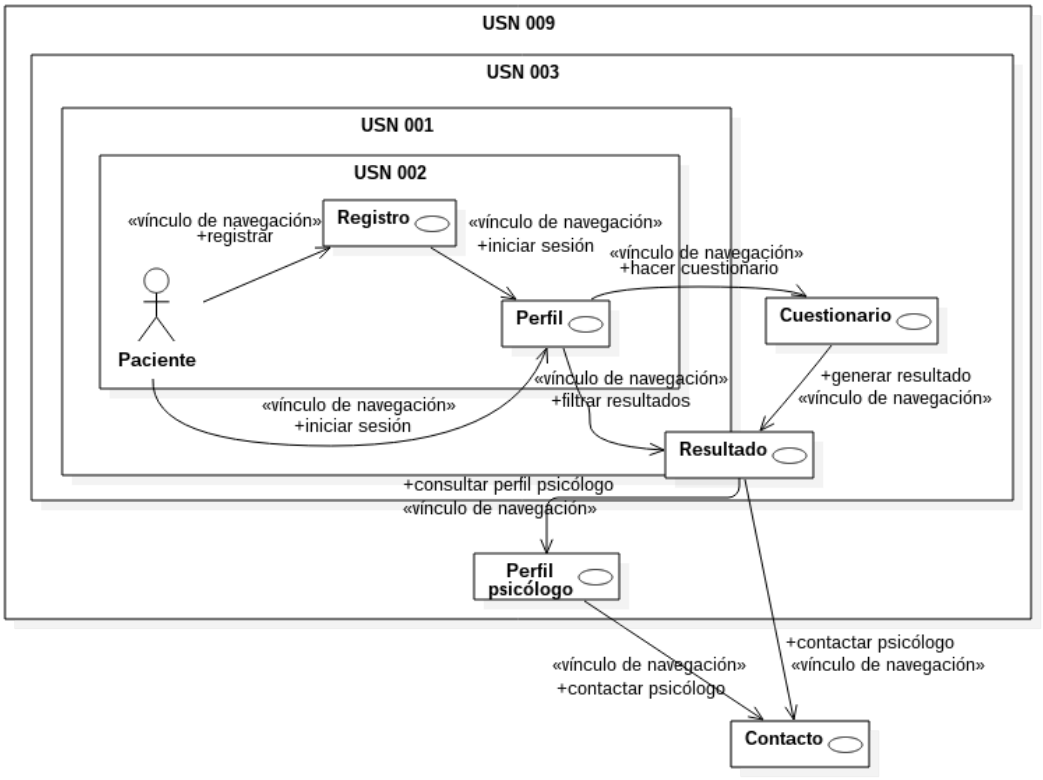

Figura 1.5: USN-005

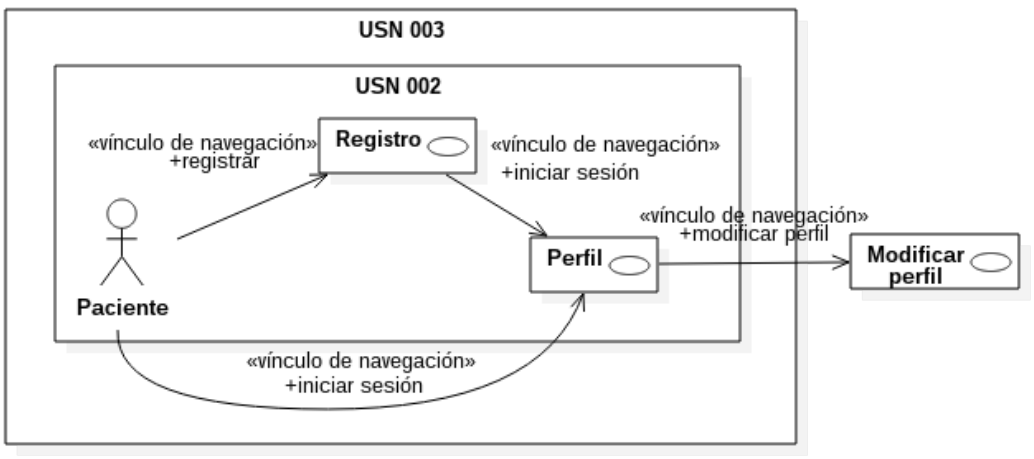

Figura 1.6: USN-006

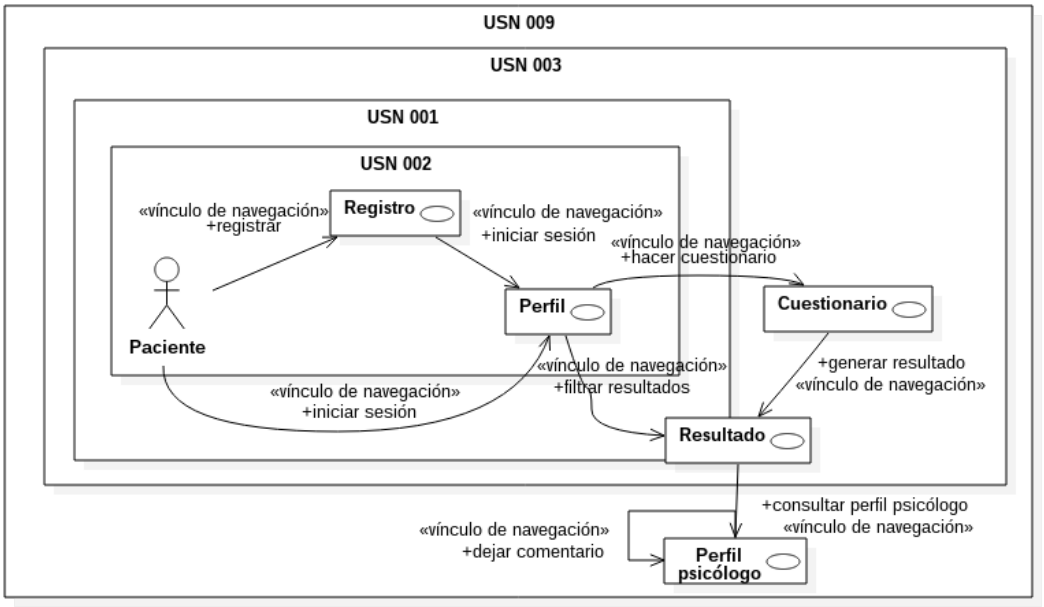

Figura 1.7: USN-007

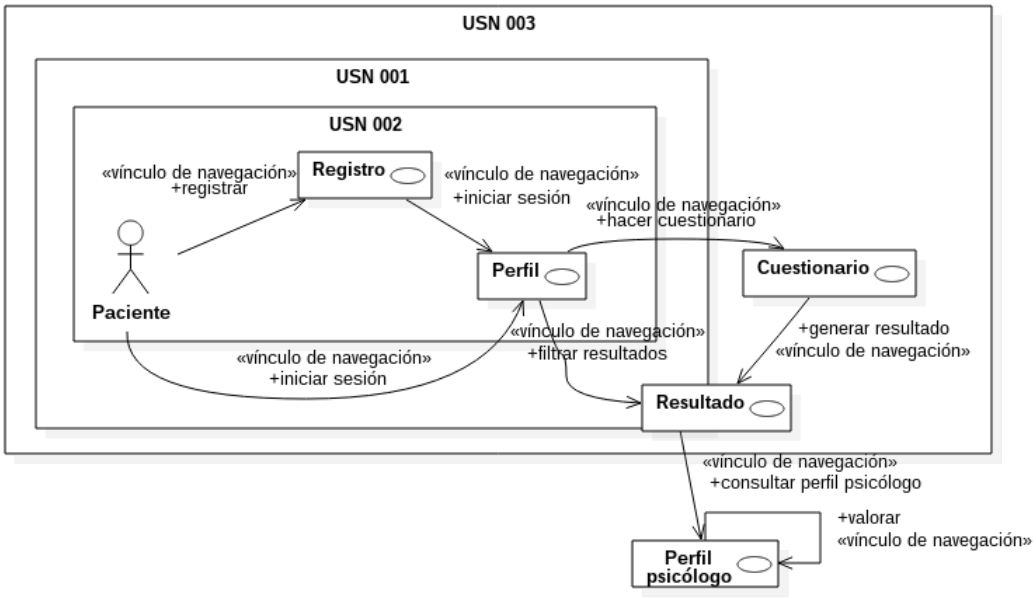

Figura 1.8: USN-008

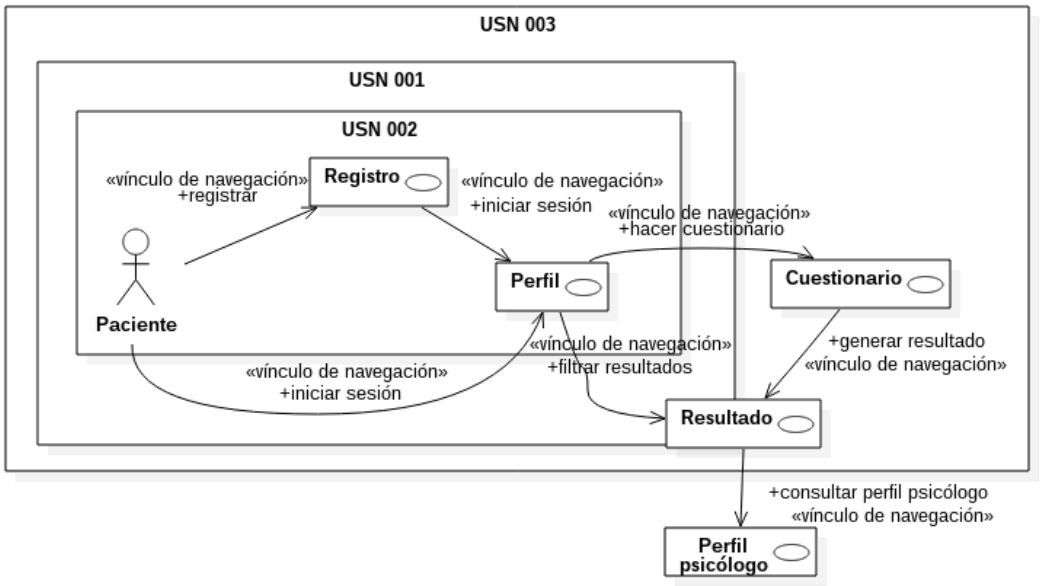

Figura 1.9: USN-009

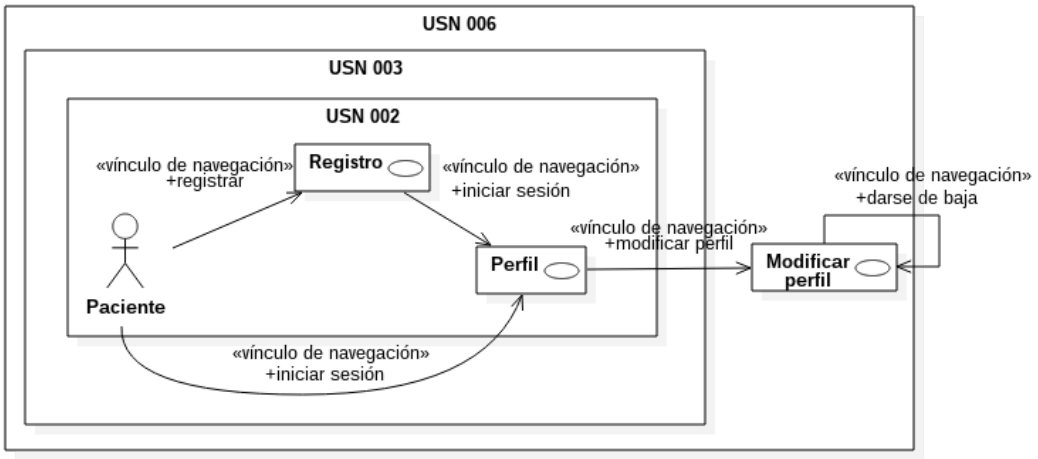

Figura 1.10: USN-010

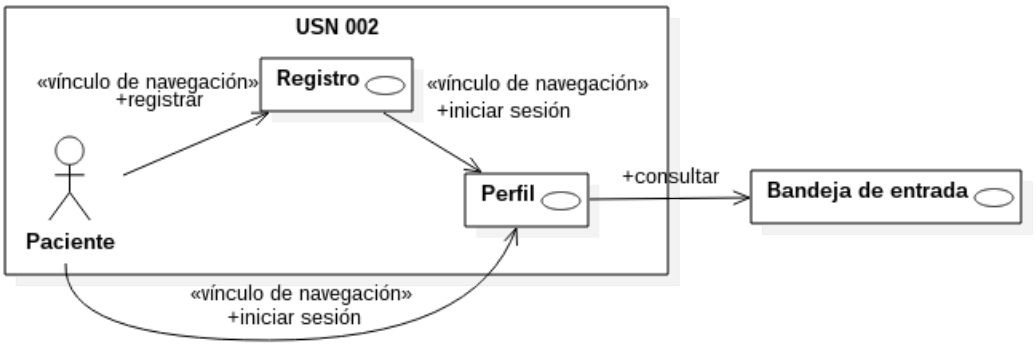

Figura 1.11: USN-011 (Paciente)

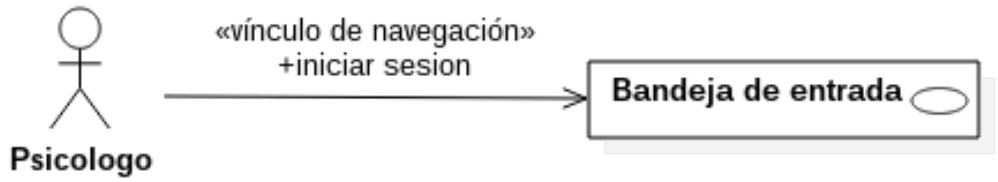

Figura 1.12: USN-011 (Psicólogo)

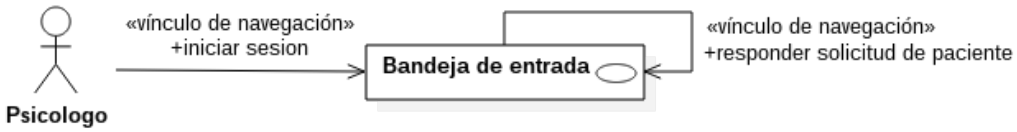

Figura 1.13: USN-012

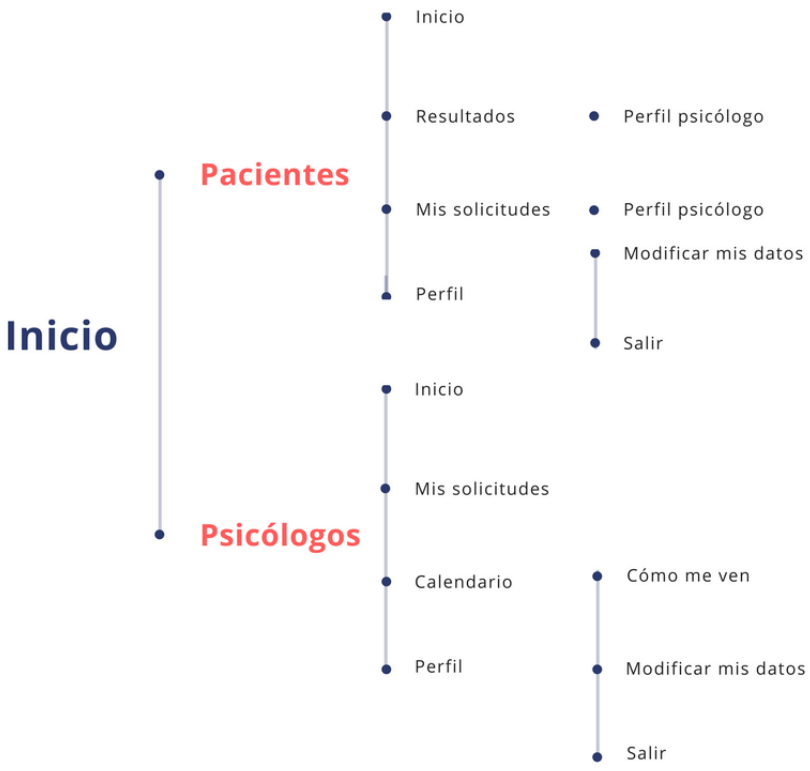

Figura 1.14: Mapa de navegación

MEAN Stack

MEAN stack es un *framework* para el desarrollo de aplicaciones, y páginas web dinámicas, basadas en JavaScript: MongoDB, ExpressJS, AngularJS y NodeJS, lo cual permite que se integren entre ellas eficazmente.

Con MongoDB podemos almacenar nuestros documentos en formato JSON, se pueden escribir consultas en nuestro servidor ExpressJS y NodeJS, y del mismo modo, pasar esos documentos JSON a nuestro *frontend* hecho con AngularJS.

El *debugging* y la administración de la base de datos se vuelven más sencillas cuando el objeto almacenado en la base de datos es idéntico al objeto que tu cliente JavaScript puede ver[41].

Algunos de los motivos por los que escogí este *framework* es por su escalabilidad, rapidez y flexibilidad, ya que permite que en un futuro sea sencillo poder adaptar la aplicación a plataformas móviles, o añadir cambios con facilidad.

Los componentes que forman el *framework* son:

- **MongoDB** MongoDB es una base de datos ágil NoSQL orientada a documentos que permite que los esquemas cambien rápidamente a medida que las aplicaciones evolucionan, proporcionando siempre la funcionalidad que los desarrolladores esperan de las bases de datos tradicionales, tales como índices secundarios, un lenguaje completo de búsquedas y consistencia. En resumen, MongoDB brinda escalabilidad, rendimiento y gran disponibilidad[42].

- **ExpressJS** ExpressJS es una middleware de aplicaciones web Node.js minimalista y flexible que proporciona un conjunto sólido de características para aplicaciones web y móviles[14]. Se trata de una API REST que utiliza métodos HTTP para obtener datos o generar operaciones sobre esos datos.

- **AngularJS** AngularJS es un framework para construir client applications en HTML y Javascript, aunque también puede ser otro lenguaje como TypeScript que sea compilado a JavaScript. El framework posee un conjunto de librerías funcionales, y otras opcionales[4].

  Angular permite fácilmente construir aplicaciones web, ya que combina declarative templates, dependency injection y end to end tooling, e integra buenas prácticas de programación. Las aplicaciones desarrolladas con Angular funcionan tanto para web, móvil o escritorio[2].

- **NodeJS** NodeJS es un entorno de ejecución para JavaScript construido con el motor de JavaScript V8 de Chrome. Node.js usa un modelo de operaciones entrada/salida sin bloqueo (asíncrono) y orientado a eventos, que lo hace liviano y eficiente, ya que nunca se bloquea.

  Node está diseñado sin hilos, este presenta un bucle de eventos como un entorno en vez de una librería. Node simplemente ingresa el bucle de eventos

después de ejecutar el script de entrada. Node sale del bucle de eventos cuando no hay más callbacks que ejecutar[1].

Bootstrap

Bootstrap es un *framework* de código abierto para desarrollar con HTML, CSS y JS. Contiene características como un grid responsive, docenas de componentes, JavaScript plugins, tipografía, control de formularios y capacidad de personalización[12].

Semantic UI

Semantic UI es un *framework* que permite a los desarrolladores contruir rápidamente sitios web, con HTML conciso, JavaScript intuitivo, y simplicando el *debugging*. Semantic está diseñado de manera *responsive* permitiendo que la aplicación sea escalable en múltiples dispositivos. Semantic está preparado para poder asociarse con otros frameworks como React, Angular, Meteor y Ember[5].

1.3.2. Lenguajes de programación

Los principales lenguajes de programación utilizados son:

CSS3

Hojas de Estilo en Cascada (Cascading Style Sheets) es el lenguaje utilizado para describir la presentación de documentos HTML o XML CSS describe como debe ser renderizado el elemento estructurado en pantalla, en papel, hablado o en otros medios[3].

HTML

HTML, que significa Lenguaje de Marcado para Hipertextos (HyperText Markup Language) es el elemento de construcción más básico de una página web y se usa para crear y representar visualmente una página web. Determina el contenido de la página web, pero no su funcionalidad[6].

JavaScript

Es un lenguaje ligero e interpretado, orientado a objetos con funciones de primera clase, más conocido como el lenguaje de script para páginas web, pero también usado en muchos entornos sin navegador, tales como node.js.

Es un lenguaje script multi-paradigma, basado en prototipos[2], dinámico, soporta estilos de programación funcional, orientada a objetos e imperativa[8].

1.3.3. *Modules*

Un *module* es cualquier fichero o directorio que puede ser cargado por NodeJS.

Mongoose

Mongoose proporciona una sencilla, solución basa en esquemas para el modelo de los datos de la aplicación[21].

Bcrypt

Bcrypt permite *hashear* y comparar contraseñas en Node.

NodeMailer

NodeMailer permite a las aplicaciones el envío de mensajes.

FullCalendar

FullCalendar es un calendario de eventos JavaScript personalizable y de código abierto[15].

1.3.4. Librerías

Una librería es un conjunto de implementaciones funcionales, codificadas en un lenguaje de programación, que ofrece una interfaz bien definida para la funcionalidad que se invoca.

Places de Google Maps JavaScript API

Las funciones de la biblioteca JavaScript de Google Places permite que una aplicación busque sitios (definidos en esta API como establecimientos, ubicaciones geográficas o puntos de interés destacados) dentro de un área definida, como los límites de un mapa o alrededor de un punto fijo. También, ofrece una función de autocompletado que puedes usar para dar a tus aplicaciones el comportamiento de escritura anticipada del campo de búsqueda de Google Maps. Cuando un usuario comienza a escribir una dirección, la función de autocompletado termina la tarea[10].

[2]La programación basada en prototipos es un estilo de programación orientada a objetos que reutiliza comportamientos de objetos existentes.

1.3.5. *Middleware*

Un *middleware* proporciona la lógica de intercambio entre aplicaciones. El *middleware* abstrae de la complejidad y heterogeneidad de las redes de comunicaciones subyacentes, así como de los sistemas operativos y lenguajes de programación, proporcionando una API para la fácil programación y manejo de aplicaciones distribuidas.

Passport

Passport es un *middleware* de autenticación para NodeJS. Extremadamente flexible y modular. También puede ser utilizado en cualquier aplicación web basada en Express. Tiene múltiples estrategias que soportan autenticación utilizando nombre y contraseña, Facebook, Twitter, y más[23].

1.4. Herramientas utilizadas

Las herramientas utilizadas en este trabajo fueron:

- **Brackets**
  - *Descripción:* Brackets es un editor de texto de código abierto[13].
  - *Uso:* Desarrollo software de la aplicación.
- **Git**
  - *Descripción:* Git es un sistema de código abierto de control de versiones distribuído diseñado para manipular cualquier tipo de proyectos con rapidez y eficacia[16].
  - *Uso:* Gestión de la configuración del proyecto.
- **GitHub**
  - *Descripción:* Es una plataforma de desarrollo colaborativo que permite alojar proyectos utilizando el sistema de control de versiones de Git[17].
  - *Uso:* Gestión de la configuración del proyecto.
- **Grunt**
  - *Descripción:* Es una herramienta que permite simplificar el proceso de construcción (build) de proyectos en JavaScript. Sirven para automatizar tareas repetitivas como minificación, compilación, testeo unitario...[18]
  - *Uso:* Durante el desarrollo de la aplicación.

- **LaTeX**
  - *Descripción:* Es un sistema tipográfico que incluye características diseñadas para la producción de documentación técnica y científica[19].
  - *Uso:* Elaboración de la memoria final.
- **LibreOffice**
  - *Descripción:* Es un conjunto de aplicaciones de oficina: Writer, el procesador de textos, Calc, la hoja de cálculos, Impress, el editor de presentaciones, Draw, nuestra aplicación de dibujo y diagramas de flujo; entre otros[20].
  - *Uso:* Borradores y algunos diagramas del proyecto.
- **Pencil**
  - *Descripción:* Pencil es una GUI de prototipado de código abierlo para creación de *mockups*[24].
  - *Uso:* Elaboración del *mockup*.
- **NPM**
  - *Descripción:* NPM es un gestor de paquetes JavaScript. Es el repositorio más grande de librerías de código abierto en el mundo[22].
  - *Uso:* Instalación de paquetes y librerías software.
- **Robomongo**
  - *Descripción:* Robomongo es una GUI que maneja la shell de MongoDB[25].
  - *Uso:* Gestión de la base de datos.
- **StarUML**
  - *Descripción:* StarUML es una herramienta de modelado UML que permite hacer multitud de tipos de diagramas como de clase, de objeto, de casos de uso, de componente, entre otros[26].
  - *Uso:* Diseño de los diagramas del proyecto.

1.5. Seguridad

1.5.1. ¿Qué es Bcrypt?

Niels Provos y David Maxieres diseñaron Bcrypt, una función de hashing de contraseñas basado en el tipo de cifrado Blowfish. Es empleado en algunas distribuciones de Linux por defecto.

Cuando se genera un hash asociado a la contraseña por lo general los algoritmos comunes (md5, sha-1,...) incorporan un valor de salt, este fragmento se emplea para generar dicho hash, de esta forma se consigue que dos contraseñas iguales que generarían el mismo hash no lo hagan, algo muy importante para luchar contra ataques de fuerza bruta, así como para dificultar los ataques de Rainbow table.

Los ataque de Rainbow table son efectivos cuando las contraseñas son hasheadas de la misma manera, de esta forma para dos contraseñas iguales el hash sería el mismo. Así es como surge el salt, añadir un hash a cada contraseña nos devuelve dos hashes distintos para la misma contraseña. Es importante evitar la reutilización de salt. Sin embargo no es suficiente el uso de salt para el almacenamiento de contraseñas debido a que salt pasa a ser inútil para ataques de fuerza bruta o de diccionario.

1.5.2. Ventajas de Bcrypt

En este punto es donde entra Bcrypt, para solucionar el problema en el que nos hallamos debemos hablar del número de iteraciones. Esto mejora a los hashes comunes gracias a su lentitud. ¿Qué quiere decir esto? Empleando una variante de cifrado Blowfish se introduce un factor de trabajo que permite controlar el coste de la función hash. Mediante este factor Bcrypt se actualiza de acuerdo a la ley de Moore con la evolución de la tecnología.

Para verlo más concretamente podemos ver la tabla de comparación siguiente, y recalcar que aunque las contraseñas no requieran de una protección tan elevada, gracias al factor de trabajo la optimización de bcrypt se consigue con un buen equilibrio entre velocidad y seguridad. El sacrificio de un poco de rendimiento se traduce en un aumento de la seguridad. Esta tabla muestra una prueba de la fuerza de bcrypt, que ha sido lanzado en un clúster de 25 GPU para la rotura de contraseñas en hash, han resultado los siguientes datos:

- md5($password)
  - 180 billones resultados/s
  - 9.4 Horas
- sha1($password)
  - 61 billones resultados/s
  - 27 Horas
- md5crypt
  - 77 millones resultados/s
  - 2.5 Años

- bcrypt con un factor de 5
  - 71 mil resultados/s
  - 2700 Años

1.5.3. Funcionamiento

La librería bcrypt nos permite crear el hash de una contraseña mediante saltRounds, este valor nos da control sobre el factor de coste de procesado de datos. A mayor valor de SaltRound mayor coste de cálculo del hash asociado a una contraseña. Por defecto este valor viene situado en 10.

En el caso del registro de usuario se envía la contraseña y el valor del parámetro SaltRounds a la librería para que la cifre, esta nos devuelve el hash asociado a la contraseña en el usuario de la base de datos. Cuando el usuario quiere entrar la contraseña introducida se cifra, para ello se busca el salt asociado al usuario de la base de datos. Una vez cifrada con el salt se comparan los hashes. La librería nos devuelve un booleano que indica si las contraseñas coinciden o no, con lo cual podemos dejar entrar el usuario o devolver un error.

1.5.4. Cifrado Blowfish

Bruce Schneier en 1993 diseña un codificador de bloques simétricos de 64 bits, con claves de hasta 448 bits, es empleado en un abundante número de productos de cifrado. No tiene técnicas de criptoanálisis que hayan resultado efectivas contra este algoritmo. Su licencia es totalmente libre.

1.6. Diseño de la arquitectura

1.6.1. SPA

Single page aplication, de ahora en adelante SPA, es una aplicación o sitio web que cabe en una sóla página web con el objeto de proveer una experiencia de usuario más fluída y una interfaz más enriquecida. Una de las ventajas de este tipo de aplicaciones es que son capaces de actualizar una parte de la interfaz, sin necesidad de enviar o recibir una petición de full-page.

1.6.2. Particularidades de JavaScript

En JavaSCript no existen las clases

JavaScript es un lenguaje orientado a objetos basado en prototipos en lugar de clases. A diferencia de los lenguajes orientados a objetos basados en clases, un

lenguaje basado en prototipos no hace distinción entre clases e instancias: Simplemente tiene objetos. Estos objetos son conocidos como objetos prototípicos, que son objetos que se utilizan como una plantilla a partir de la cual se obtiene el conjunto inicial de propiedades de un nuevo objeto[43].

Para poder entender cuáles son las diferencias entre un lenguaje orientado a objetos basados en clases (Java) y basados en prototipos (JavaScript) se han listado sus diferencias en la tabla 1.4.

Basados en clases (Java)	**Basados en prototipos (JavaScript)**
La clase y su instancia son entidades distintas	Todos los objetos pueden heredar de otro objeto
Define una clase en la definición de clase; se instancia una clase con los métodos constructores.	Define y crea un conjunto de objetos con funciones constructoras.
Se crea un objeto con el operador new.	Igual.
Se construye una jerarquía de objetos utilizando la definición de las clases para definir subclases de clases existentes.	Se construye una jerarquía de objetos mediante la asignación de un objeto como el prototipo asociado a una función constructor.
Se heredan propiedades siguiendo la cadena de clases.	Se heredan propiedades siguiendo la cadena de prototipos.
La definición de una clase especifica todas las propiedades de todas las instancias de esa clase. No se pueden añadir propiedades dinámicamente en tiempo de ejecución.	El conjunto inicial de propiedades lo determina la función constructor o el prototipo. Se pueden añadir y quitar propiedades dinámicamente a objetos específicos o a un conjunto de objetos.

Tabla 1.4: Comparativa Java y JavaScript

Por estos motivos, los patrones que se han diseñado con UML consisten en una aproximación de cómo se representaría el diseño de la plataforma.

1.6.3. *Callbacks* en JavaScript

Las *callbacks* JavaScript son utilizadas para gestionar eventos de manera responsable en el lado cliente ejecutando funciones asíncronas, y en Node, las *callbacks* también son utilizadas en el lado servidor para dar servicio a múltiples peticiones simultáneas de clientes.

Una *callback* es una función que es pasada como argumento a otra función, que espera ser invocada tanto inmediátamente como en algún momento futuro.

Las *callbacks* pueden ser vistas como una forma de "*continuation-passing style*" (CPS), cuyo control es pasado explícitamente de forma continuada, en el caso de las *callback* es pasado como argumento representando una continuación.

JavaScript utiliza un modelo *event-driven* con un único hilo de ejecución. Programar con *callbacks* es especialmente útil cuando el llamador no quiere esperar hasta que la llamada se complete. Para conseguirlo, la operación no bloqueante (*non-blocking operation*) es agendada (*scheduled*) como una *callback* y el hilo principal continua su síncrona ejecución. Cuando la operación se completa, un mensaje es encolado en una cola de tareas según la *callback* que lo provee. El bucle de eventos (event loop) en JavaScript prioriza que el hilo ejecute la pila de llamadas primero; cuando la pila está vacía, el bucle de eventos desencola un mensaje para la cola de tareas y ejecuta la correspondiente función *callback*. En JavaScript, las *callbacks* pueden llamarse "funciones" o "funciones anónimas". En el proyecto, para gestionar las *callbacks* se han utilizado promises. Las *promise* son una extensión del lenguaje JavaScript [40].

Una *promise* es un *proxy* para un valor no necesariamente conocido en el momento que es creada. Permite asociar manejadores que actuarán asincrónicamente sobre un eventual valor en caso de éxito, o la razón del fallo. Esto permite que métodos asíncronos devuelvan valores como si fueran síncronos: en vez de inmediatamente retornar el valor final, el método asíncrono devuelve una *promise* y suministra su valor en algún momento en el futuro[9]. Al objeto le enganchas las funciones *callback*, en vez de pasar funciones *callback* a una función[11]. Nos permiten mejorar la legibilidad de nuestro código y evitar tener que pasar el contenido de las funciones directamente como argumentos a nuestra llamada. Las *promise* en JavaScript sirven para evitar el "*callback-hell*"que surge de llamar a una función asíncrona en JavaScript.

Implementando patrones de diseño con JavaScript

En la mayoría de lenguajes tradicionales orientados a objetos existen las clases, las interfaces, la herencia, la encapsulación y el polimorfismo. Pero JavaScript, no, puesto que es muy sencillo: Trabaja por medio de callbacks (funciones).

Un objeto es la unidad responsable de proporcionar estados y comportamiento; y la declaración de las funciones en JavaScript proporcionan ambas[**?**].

1.6.4. Patrones de diseño

MVC y MVVM

El patrón Modelo Vista Controlador, de ahora en adelante MVC, es un patrón de arquitectura software que permite separar la representación visual de la información, de la interacción del usuario.

MVC es un patrón *composite*. Los patrones *composite* son aquellos patrones capaces de trabajar juntos para darle solución a un problema en común[37].

El patrón de diseño MVC está compuesto por:

- Modelo: Almacena toda la información, estados y lógica de la aplicación.

- Vista: Se trata de a interfaz de usuario que muestra una representación del modelo. La vista informa al controlador qué trata de hacer el usuario.

- Controlador: Toma la entrada del usuario, la gestiona y le transmite lo que ha interpretado al modelo. También, manipula cómo se debe ver la vista.

Actualmente, se han desarrollado nuevos tipos de modelo MV*, como el Modelo Vista VistaModelo, de ahora en adelante MVVM, que está basada en el patrón MVC y el Modelo Vista Presentador, que trata de separar con más claridad el desarrollo de la interfaz de usuario con el de la lógica de negocio y el comportamiento de la aplicación.

Los componentes de los patrones MVVM son:

- Modelo: Representa los datos específicos del dominio o información con la que nuestra aplicación debe trabajar. Almacena información, pero comunmente no posee ningún tipo de comportamiento. No formatean la información ni influencian en cómo los datos aparecen en el navegador, ya que no es su responsabilidad.

- Vista: Es la única parte de la aplicación con la que el usuario interactúa. Contiene la información obtenida de la sincronización entre la vista y el modelo (*data binding*[30]), los eventos y comportamientos. Es una interfaz de usuario interactiva que representa el estado de la VistaModelo. La vista no es la encargada de gestionar su estado, sólo se mantiene sincronizada con la VistaModelo.

- VistaModelo: Se puede considerar como un Controlador especializado que actúa como un conversor de datos. Transforma la información del Modelo en la información de la Vista[43].

En MEAN STACK, estos dos patrones se pueden aparecen combinados como se muestra en la figura1.15.

Page Controller

Page Controller tiene un único *controller* por cada página lógica de la página web[**?**].

Las responsabilidades del *Page Controller* son:

- Analizar la URL y extraer cualquier dato de formulario necesario para el comportamiento.

- Crear e invocar cualquier objeto del modelo para procesar datos. Cualquier dato relevante de una petición HTML ha de ser pasada al modelo, de esta forma el modelo no necesita ningún tipo de conexión a las peticiones HTML.

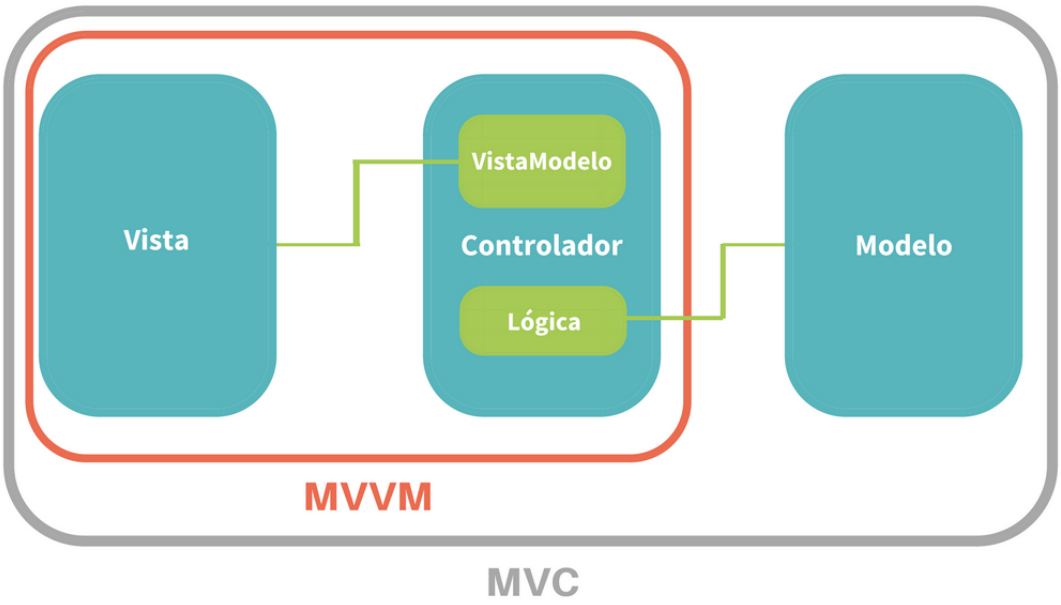

Figura 1.15: Mean STACK - MVC y MVVM

- Determina qué vista ha de ser mostrada a continuación como resultado de la página y proporcionar al modelo información sobre ello.

El patrón *Page Controller* acepta entradas desde la petición de una página, invoca las acciones pedidas en el modelo, y determina la vista que se ha de usar como página resultante[34].

En AngularJS, tenemos *controllers* los cuales tienen unas responsabilidades limitadas: Ellos no aceptan las peticiones del usuario porque es la responsabilidad de los *services* $route o $state y la renderización de la página es responsabilidad de la directiva ng-view. Para determinar qué vista ha de ser mostrada utiliza el *service* $location.

Al igual que *page controllers*, los *controllers* de AngularJS gestionan las interacciones de usuario, proveen y actualizan los modelos. El modelo está unido a la vista a través de $scope[28].

1.6.5. *Observer*

El patrón *Observer* es un patrón de diseño *software* en el cual un objeto, llamado *subject*, mantiene una lista de sus dependencias, llamadas *observers*, y las notifica automáticamente en cualquier cambio de estado, normalmente utilizando uno de sus métodos.

Data-binding

Data binding en Angular es la sincronización entre el modelo y la vista.

El *controller* proporciona comportamiento a través de $scope, mientras que se pueden utilizar en el código HTML para mostrar contenido del modelo en la vista. O la *directive* ng-model para asociar el modelo con la vista.

Cuando la información en el modelo cambia, la vista refleja dicho cambio, y cuando la vista es modificada, el modelo se actualiza.

Debida a la inmediata sincronización entre el modelo y la vista, el controlador puede estar completamente separado de la vista, y simplemente concentrarse en el modelo de datos[30].

El mostrado en la figura 1.16 representa el funcionamiento descrito a continuación. El *module* es el objeto de mayor nivel en Angular. Una aplicación Angular es especificada por la directiva ng-app siempre correspondiente a un módulo. Config es un objeto que configura la aplicación Angular, permite configurar las rutas (*routes*) necesarias en la SPA porque gestionan las *templates*. En Angular se empareja cada *template* con un *controller*. La ¡view¿representa la parte *front-end* y el *controller* y la *factory* representan el *back-end*. $scope vive entre ambos, y Angular permite asegurar que ambos tengan la última versión de la aplicación. El bucle de eventos y $scope permiten a Angular *two way data binding*, y como resultado es que ya no es necesario acceder al DOM desde JavaScript[**?**].

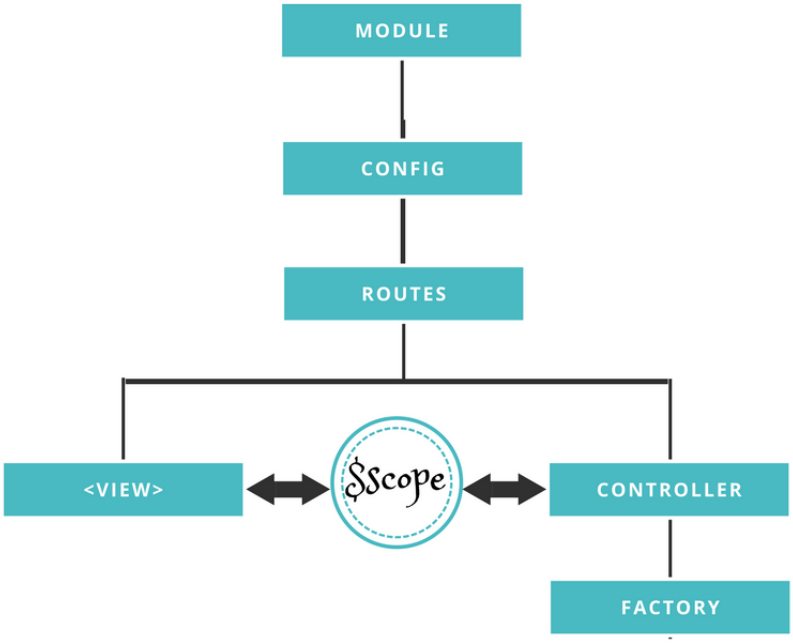

Figura 1.16: Esquema *data-binding*

Dependency Injection

Dependency injection permite obtener una instancia de las clases que se requieren y que una *factory* o *injector* te las provea tanto en tiempo de compilación como en el de ejecución[31].

Dependency injection suele ser utilizado para evitar al patrón *Singleton*[28].

1.6.6. *Factory Method*

Los propósitos del patrón *Factory* son:

- Crear objetos
- Realiza operaciones repetitivas cuando se preparan objetos similares.
- Ofrece que los consumidores de la *Factory* puedan crear objetos sin necesidad de conocer el tipo específico (de la clase) en tiempo de compilación[46].

El patrón *factory* provee una interfaz genérica para crear objetos, donde hay que especificar el tipo de objeto *factory* que deseamos obtener[45].

En AngularJS se hace por medio del *Factory Method*[28].

1.6.7. *Proxy*

En el patrón *proxy*, un objeto actúa como interfaz de otro objeto. El *proxy* se encuentra entre el cliente de un objeto y el objeto en sí, protegiendo el acceso a dicho objeto.

Al tratarse de un *proxy* virtual solventa la problemática que existe si inicializar el sujeto real es costoso, y puede ocurrir que el cliente lo inicialice pero nunca llegue a usarlo. En este caso, el *proxy* puede ayudar siendo la interfaz del sujeto real. El *proxy* recibe la petición de inicialización, pero nunca lo pasa a no ser que realmente el sujeto real sea utilizado.

La imagen 1.17, muestra el posible escenario de cuando un cliente realiza una petición de inicialización y el *proxy* responde que todo está bien, pero realmente no pasa el mensaje, a no ser que sea obvio que el cliente necesite hacer algo con el sujeto. Sólo en ese caso, el *proxy* le pasará ambos mensajes juntos[46].

1.6.8. *Data Mapper*

Un *Data Mapper* es una capa de acceso a datos (*Data Access Layer*) que permite la transferencia bidireccional de datos entre el lugar de almacenamiento de los datos persistentes y la representación en memoria de esos datos, manteniéndolos independientes[28].

En Angular a través del módulo ngResource, nos permite realizar conexiones REST a nuestra API para enviar y recibir información mediante el servicio $http.

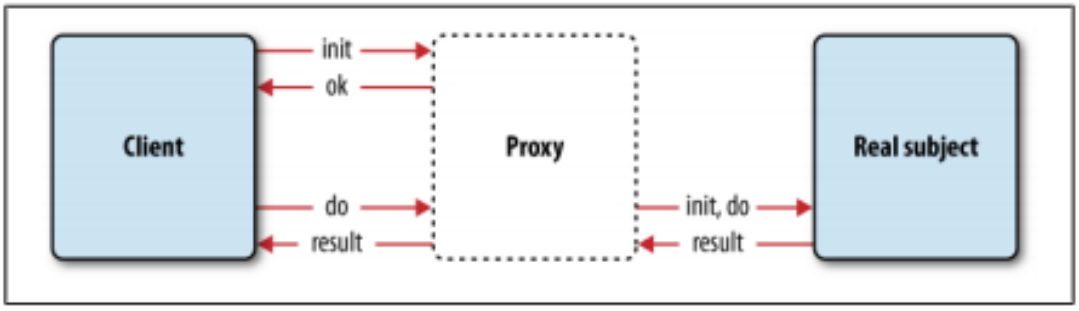

Figura 1.17: *Proxy pattern*[46]

Sin embargo, los datos pasados desde el servidor se encuentran en un formato apropiado gracias a la librería Mongoose.

Mongoose proporciona por un lado el esquema que da formato a los datos obtenidos de la base de datos, y por otro lado, el acceso a los mismos.

1.6.9. API REST

REST (*Representational State Transfer*) es un estilo de arquitectura diseñado para sistemas distribuídos. No está estandarizado pero posee una serie de directivas, como permanecer sin estado, tener relaciones cliente-servidor y una interfaz uniforme. Suele estar relacionado con HTTP.

Sus principios son:

- Expone recursos fácilmente con un directorio de URLs estructurado.
- Representa los data objects y atributos en formato JSON o XML.
- Los mensajes utilizan métodos HTTP explícitamente: GET, POST, PUT y DELETE.
- Protocolo cliente/servidor sin estado: Cada petición HTTP contiene toda la información necesaria para ejecutarla, lo que permite que ni cliente ni servidor necesiten recordar ningún estado previo para satisfacerla[35].

1.6.10. Estructura de directorios

Es importante mantener una estructura de directorios1.18 y de código organizada, pensando en el mantenimiento y en la escalabilidad de la aplicación.

A continuación, se describen (por orden de aparición) los distintos directorios y ficheros que pertenecen a la estructura básica del proyecto:

- app Contiene la parte frontend de la aplicación.

- assets Contiene algunos de los recursos de la aplicación.
- images Contiene las imágenes utilizadas en la aplicación.
- javascript Contiene todos los archivos JavaScript de la parte frontend de la aplicación.
- app.js Es nuestro fichero de inicio.
- controllers Contiene los controllers de los templates.
- routes.js Archivo que determina el enrutado de nuestra aplicación.
- factories Contiene los factories.
- vendor Contiene algunas librerías JavaScript utilizadas en la aplicación.
- semantic Contiene todos los archivos del paquete semantic-ui.
- styles Contiene algunas librerías CSS utilizadas en la aplicación.
- templates Contiene los templates de la aplicación.
- views Contiene la view de nuestra aplicación.
- app.js Archivo de configuración del arranque del servidor.
- node_modules Contiene todos los paquetes Node.js utilizados en la aplicación.
- npm-shrinkwrap.json Archivo que mantiene un registro de versiones de los paquetes.
- package.json Archivo que mantiene una lista de los paquetes de los que depende la aplicación.
- README.md Archivo que describe brevemente la aplicación.
- semantic.json Archivo que contiene configuraciones de compilación para Gulp.
- server Contiene la parte backend de la aplicación.
- config Contiene ficheros de configuración de la aplicación.
- expressConfig.js Archivo que contiene configuración de Express.js.
- models Contiene los models de la aplicación.
- routes Contiene las rutas API REST de la aplicación.
- routes.js Archivo que configura los módulos y enrutamiento de la aplicación.

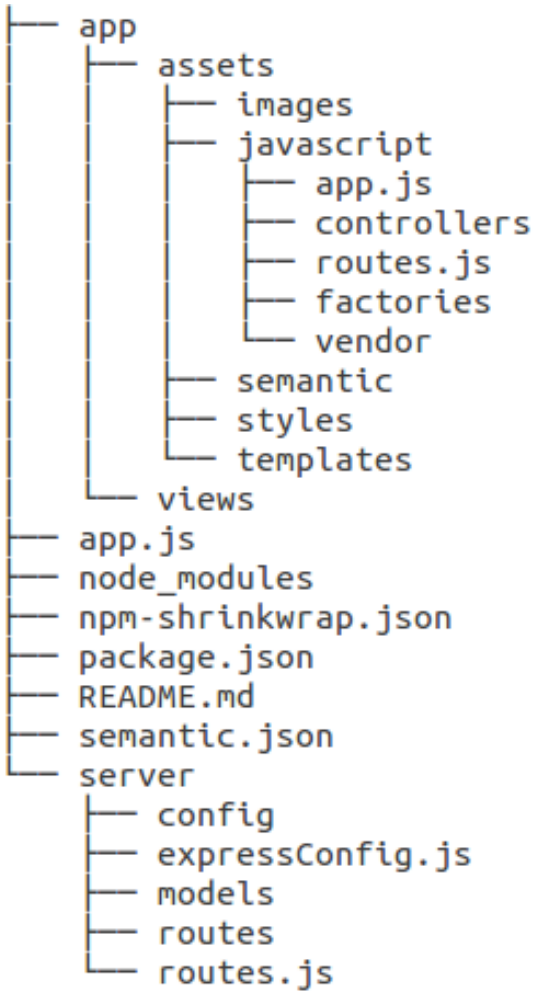

Figura 1.18: Estructura de directorios

1.7. Diagramas

Para una mayor comprensión de la composición del sistema, se han desarrollado diferentes tipos de diagramas.

En la figura 1.19 se muestra cómo interactúa el sistema a grandes rasgos a través del patrón MVC-MVVM. No se trata de un diagrama de clases, pero sirve para tener visión general de los distintos elementos que componen el sistema. En amarillo, aparecen elementos de Angular utilizados y se podrían interpretar como los patrones (en azul) que se indican a su lado.

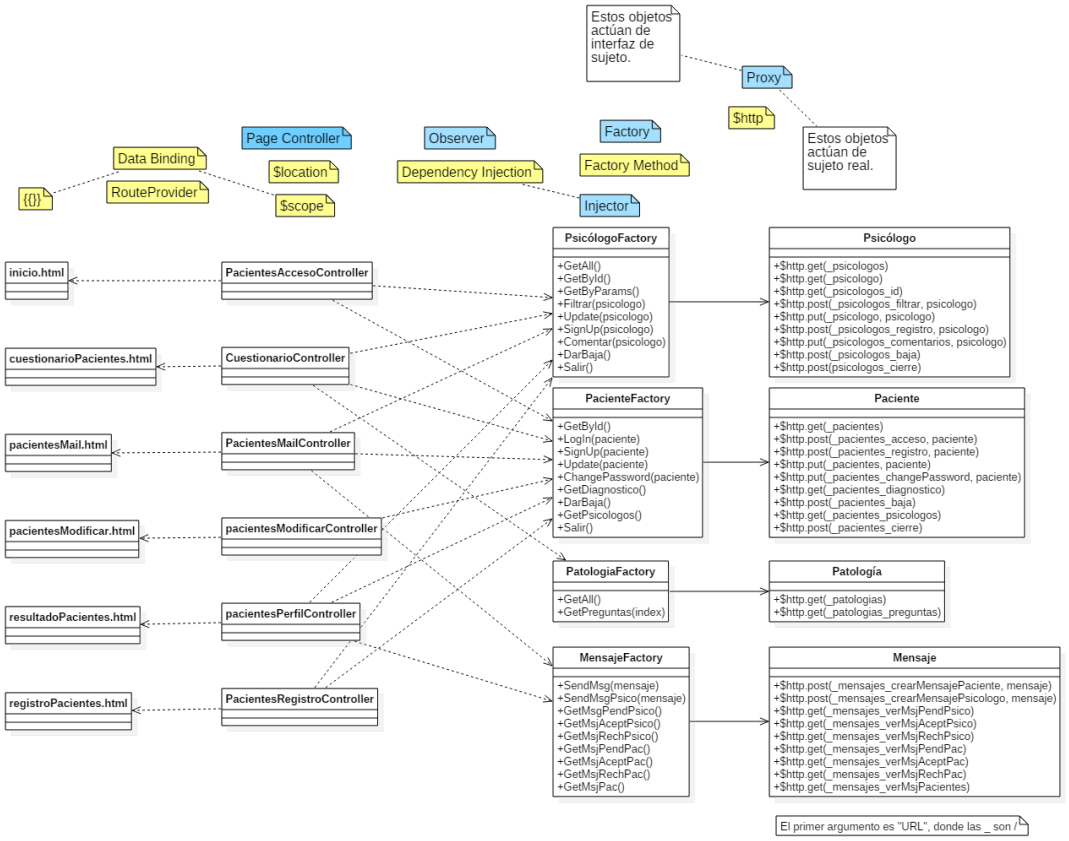

Figura 1.19: Patrón MVC-MVVM

En la figura 1.20, se muestra cómo se implementa la capa de datos *Data mapper* con ayuda de los *schemas* y funciones propias de la librería *Moongoose*.

Para ejemplificar las interacciones entre los distintos componentes del sistema, se han creado dos diagramas de secuencia: Uno referente al caso de uso CU-003 Realizar cuestionario1.21 y otro del CU-005 Contactar psicólogo1.22.

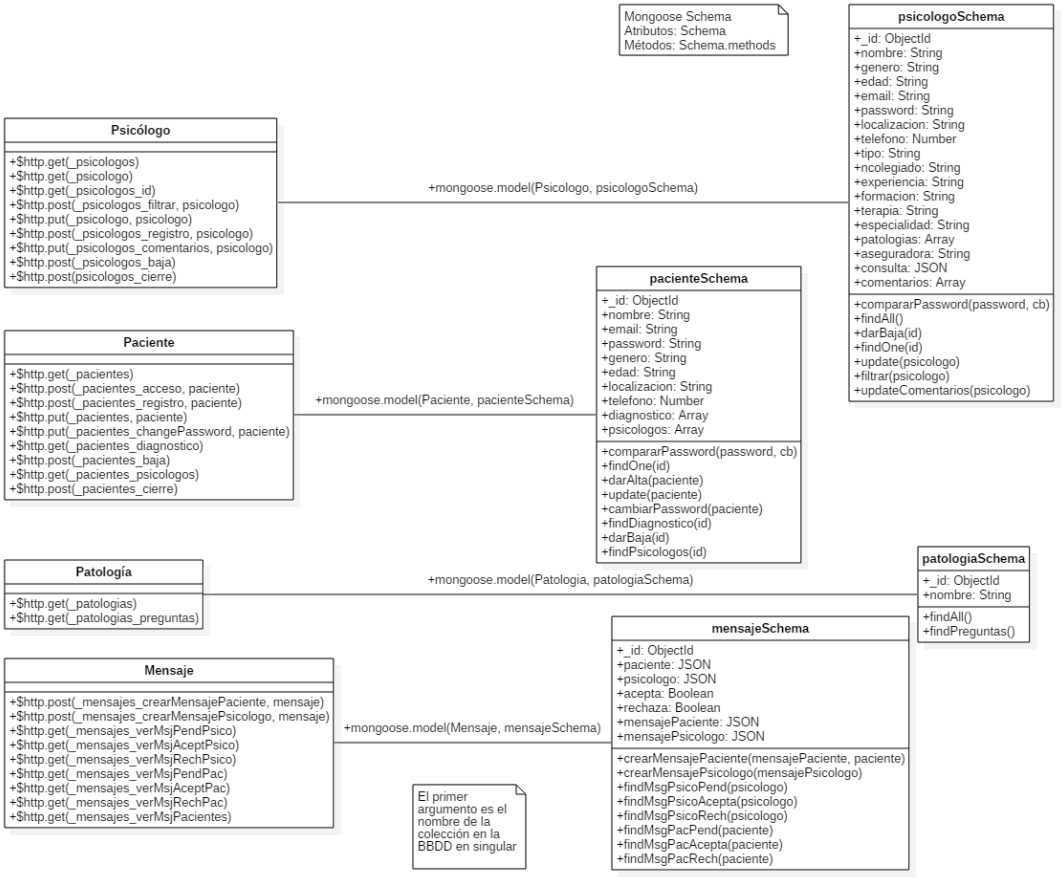

Figura 1.20: Data mapper

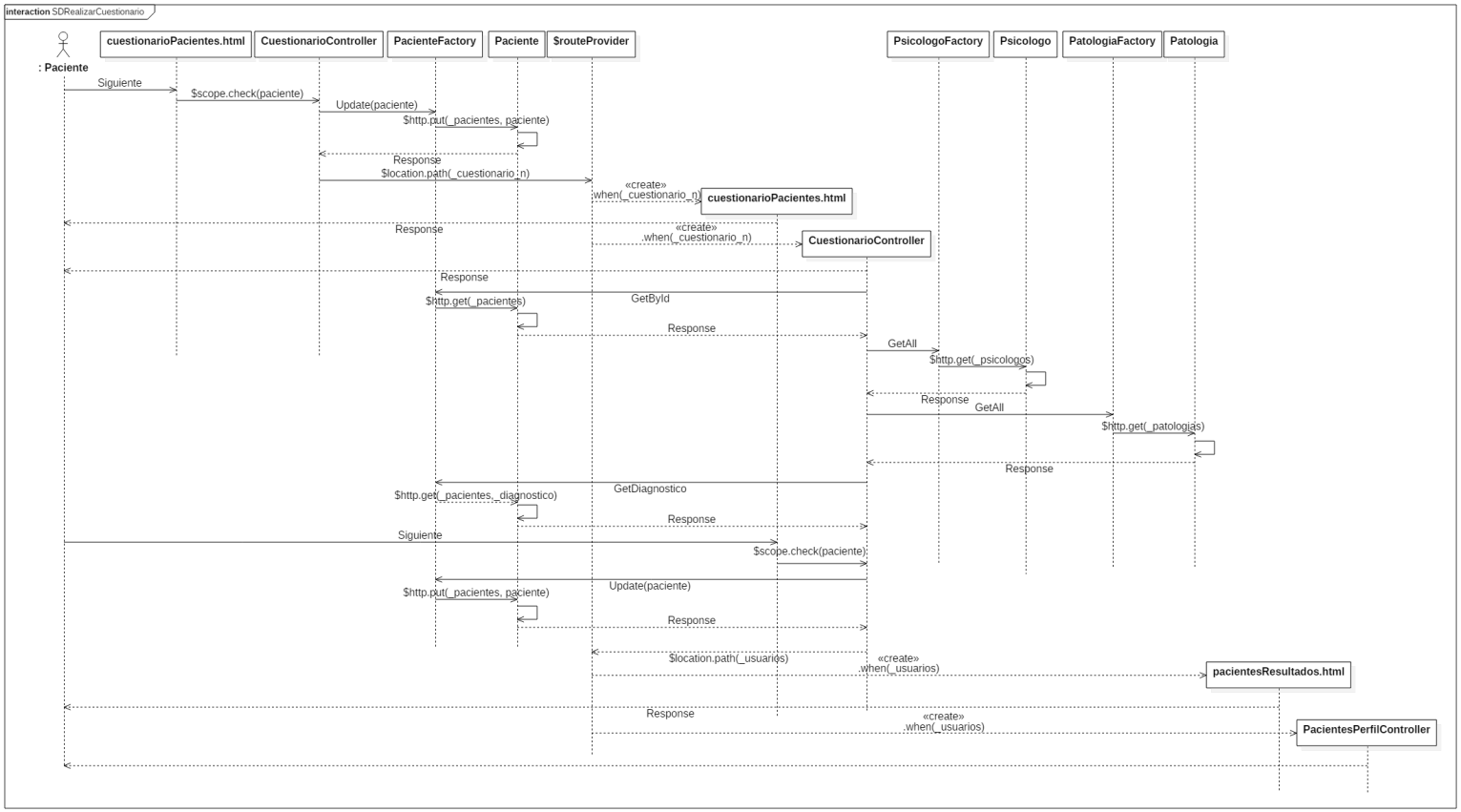

Figura 1.21: Diagrama de secuencia del CU-003

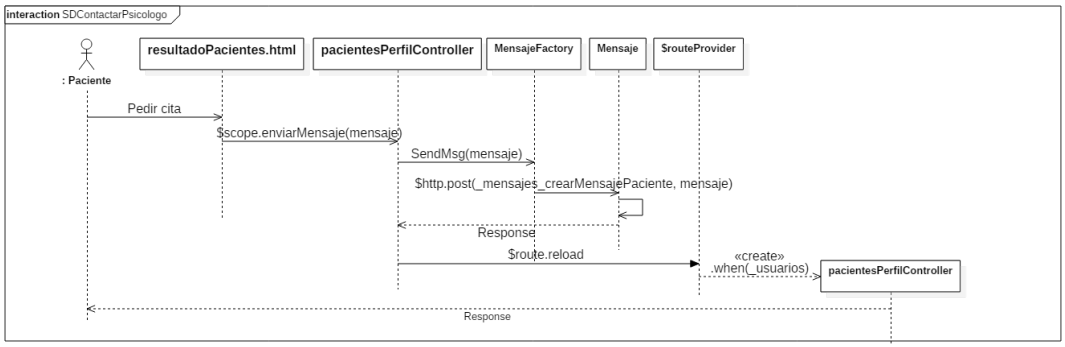

Figura 1.22: Diagrama de secuencia del CU-005

Capítulo 2

Implementación

En las secciones siguientes se mencionarán los aspectos más relevantes de la implementación.

2.1. Implementación del primer incremento

El primer incremento fue la primera toma de contacto con la tecnología utilizada tras la formación.

Los requisitos implementados en el primer incremento son los asociados a la parte del cuestionario de emparejamiento:

- RF-005: Emparejamiento
- RF-006: Filtrado de los resultados en base a distintos criterios
- RNF-002: Tiempo de respuesta de asignación

Las tareas más trascendentales realizadas en este incremento son las descritas a continuación:

Configuración Angular Todos los archivos Angular utilizados deben importarse en la *root view* (app/views/index.html en el proyecto): Librerías JavaScript, app.js, route.js, controllers y factories.

Para indicar a la aplicación que se trata de una aplicación Angular se debe especificar en la etiqueta ¡html¿de la *root view* (app/views/index.html) de la siguiente forma:

```
1 <html ng-app="Emozio"> </html>
```

Se define el módulo principal de la aplicación en el archivo app/assets/javascript/app.js:

```
1 angular.module('Emozio', ['ngRoute', 'ngResource']);
```

Entre corchetes se encuentran las dependencias que tiene nuestro módulo Emozio:

- `ngRoute`: Permite a la aplicación convertirse en una SPA, permitiendo la navegación ente distintas páginas sin necesidad de recargar. W3 angular-routing
- `ngResource`: Permite crear objetos para interactuar con los datos RESTful del lado servidor. El objeto devuelto tiene métodos de acción que proporcionan comportamiento sin necesidad de interactuar con el servicio $http a bajo nivel. Las acciones por defecto son:

```
1 { 'get':    {method:'GET'},
2   'save':   {method:'POST'},
3   'query':  {method:'GET', isArray:true},
4   'remove': {method:'DELETE'},
5   'delete': {method:'DELETE'} };
```

Llamar a estos métodos invoca a $http con el método http especificado, destino y parámetros.

Para indicar al $routeProvider dónde mostrar las templates se utiliza la siguiente directiva: `<div ui-view></div>`

Template	***Controller***
pacientes/cuestionarioPacientes.html	pacientes/cuestionarioController.js
pacientes/pacientesResultados.html	pacientes/perfilController.js

Implementación de las *templates* y *controllers* de los requisitos. Éstos se encuentran dentro del directorio app/assets.

Creación de las *factories*, *routes* y *models* de: Paciente, Psicologo y Patologia.

```
1 var MONGO_URL = 'mongodb://localhost:27017/emozio';
```

Se define la URL de la base de datos a la que se conecta la app.

```
1   var options = {
2     useMongoClient: true,
3     autoIndex: false, // No crear index
4     reconnectTries: Number.MAX_VALUE, // Nunca para de reintentar
          conectarse
5     reconnectInterval: 500, // Reconexion cada 500ms
6     poolSize: 10, // Mantener una conexion de 10 sockets
7     // Si no se conecta, devuelve un error inmediatamente antes
         de tratar de reconectarse
8     bufferMaxEntries: 0
9   };
```

Se establecen las opciones de conexión.

```
1 mongoose.Promise = global.Promise;
```

Se declara que las *promise* que va a utilizar Mongoose son las globales proporcionadas por Bluebird. En la base de datos utilizamos las *promise* definidas e implementadas en la librería Bluebird.

```
1 mongoose.connect(MONGO_URL, options, function(err, res) {
2   if(err) {
3     console.log('ERROR: Reconectando a la BBDD. ' + err);
4   }else{
5     console.log("Conectado a la BBDD");
6   }
7 });
```

Función de conexión de Mongoose a la base de datos con las opciones especificadas.

```
1   var db = mongoose.connection;
2   /* Si sucede un error, mostrarlo */
3   db.on('error', console.error.bind(console, 'Error de conexion:'
        ));
4   db.once('open', function() {
5     console.log("Con exito");
6   });
```

Se abren las conexiones a la base de datos.

```
1 app.use("/", express.static("app/"));
```

Con express.static se especifican los directorios donde se encuentran los archivos que Express va a leer. Facilita el acceso a los *assets* (bienes) de la carpeta app desde el servidor. Lo que nos permite tener la parte cliente separada del servidor.

```
1 app.set('views', __dirname + '/../app/views');
```

Define que las routes a las templates se rendericen con el render method dentro del directorio views.

```
app.use(bodyParser.urlencoded({ extended: true }));
```

Especifica que los datos recogidos de un formulario se pasen a través del método *post*.

```
app.use(bodyParser.json());
```

Mediante el paquete Body-parser, podemos tratar los objetos en formato JSON sin necesidad de manipularlos, o cambiar su tipo.

```
var app = express();
```

Se inicializa el servidor express.

```
  app.get('/', function(req, res){
    res.sendfile('index.html', {root: app.settings.views});
  });
```

Express establece cuál va a ser la *view* raíz (*root*) enviada tras el inicio del servidor.

Además, se importan los archivos de server/routes que sirven como *endpoint* (punto medio) entre la parte cliente y servidor, de esta forma durante la implementación, logramos mantener la parte del *frontend* independiente del *backend* haciéndola funcional.

2.2. Implementación del segundo incremento

Especificación del arranque del servidor y se establece cuál es la vista raíz en el archivo routes.js. También, se vinculan todos los archivos que va utilizar el servidor: Archivos de configuración, de conexiones al modelo... Los requisitos implementados en el segundo incremento son los asociados a la parte de la gestión de usuarios:

- RF-001: Acceso usuarios
- RF-002: Registro
- RF-003: Baja
- RF-004: Modificación de los datos
- RNF-001: Encriptado de datos

Template	***Controller***
inicio.html	pacientes/pacientesAccesoController.js
pacientes/registroPacientes.html	pacientes/pacientesRegistroController.js
pacientes/pacientesModificar.html	pacientes/pacientesModificarController.js
psicologos/psicologosModificar.html	psicologos/psicologosModificarController.js
psicologos/registroPsicologo.html	psicologos/psicologosRegistroController.js

Gestión de sesiones de usuario con PassportJS y Express-session. *La configuración de PassportJS se realizó en el directorio server/config/passport.js*

Las credenciales utilizadas para autenticar a un usuario sólo son transmitidas durante la petición de acceso (*login request*). Si la autenticación se realiza con éxito, la sesión será establecida y mantenida vía una cookie en el navegador del usuario. Cualquier petición posterior no contendrá las credenciales, únicamente la *cookie* que identifica la sesión. Para poder gestionar las sesiones de acceso (*login sessions*), Passport *serialize* y *deserialize* instancias de usuario.

```
1 passport.serializeUser(function(usuarios, done){
2   done(null, usuarios._id);
3 })
```

Sólo el ID de usuario es "creado" en la sesión, manteniendo mínima la cantidad de datos guardados. Cuando las siguientes peticiones sean recibidas, este ID será el utilizado para encontrar al usuario, el cual fue guardado en `req.user`.

```
1  passport.deserializeUser(function(id, done){
2    Paciente.findById(id, function(error, usuario){
3      if(usuario!=null){
4        done(null, usuario);
5      } else {
6        Psicologo.findById(id, function(error, usuario){
7          done(null, usuario);
8        });
9      }
10   });
11 })
```

`deserializeUser()` es invocado en cada petición por `passport.session`. Permite cargar información adicional a la información de usuario en cada petición; este objeto está asociado a la petición como `req.user` haciéndolo accesible en la gestión de peticiones.

```
1 passport.use(new LocalStrategy(
2   {
3     usernameField: 'email',
4     passwordField: 'password'
5   },
6   function (username, password, done) {
```

```
7     Paciente.findOne({email: username}, function(error, paciente)
          {
8       if(!paciente){
9         //                return done(null, false, {message: '
              Este email: '+email+'no esta registrado'});
10        Psicologo.findOne({email: username}, function(error,
              psicologo){
11          if(!psicologo) {
12            return done(null, false, {message: 'Este email: '+
                  username+'no esta registrado'});
13          } else {
14            psicologo.compararPassword(password, function(error,
                  sonIguales){
15              if(sonIguales){
16                return done(null, psicologo);
17              } else {
18                return done(null, false, {message: 'La contrasena
                      no es valida'});
19              }
20            });
21          }
22        });
23      } else {
24        paciente.compararPassword(password, function(error,
              sonIguales){
25          if(sonIguales){
26            return done(null, paciente);
27          } else {
28            return done(null, false, {message: 'La contrasena no
                  es valida'});
29          }
30        });
31      }
32    });
33  }
34 ));
```

Para poder utilizar la autenticación por `username` y `password`, Passport utiliza el mecanismo proporcionado por su módulo passport-local.

- `UsernameField` y `PassworfField` son los obtenidos del cuerpo de la petición (`req.body`) recibida cuando un usuario quiere acceder.
- Se busca al paciente que posea esos datos; y se pueden dar dos casos:
  - Si no existe, se busca al psicólogo que posea esos datos:
    - Si no existe: El usuario no está registrado.
    - Si existe: Se comprueba que la contraseña sea la misma a la introducida. Si son iguales, el acceso es correcto. Si no, la contraseña no es válida.

- Si existe: Se comprueba que la contraseña sea la misma a la introducida. Si son iguales, el acceso es correcto. Si no, la contraseña no es válida.

```
exports.estaAutenticado = function (req, res, next){
  if(req.isAuthenticated()){
    return next();
  }
  req.session.error = 'Please sign in!';
  res.redirect('/');
}
```

Función que comprueba si el usuario que está realizando una petición, está autenticado.

En la route de Pacientes del directorio server/routes/paciente.js:

```
app.route('/pacientes/acceso')
    .post(function(req, res, next){
    setTimeout(function(){
      passport.authenticate('local', function(error, paciente,
          info){
        if(error){
          return next(error);
        }
        if(!paciente) {
          return null;
        }else{
          req.login(paciente, {}, function(err) {
            if (err) {
              return null;
            };
            return res.json(paciente);
          });
        }
      })(req, res, next); //Funcion que devuelve passport y que
          debe ser invocada de esta forma
    }, 50);
  });
```

El formulario de acceso es enviado por el servidor a través del método POST. Utilizando `authenticate()` es como gestionamos la petición de acceso: En el caso de que el paciente exista, hacemos el `login`.

```
  app.use(session({
    /* Se utiliza para firmar el ID de sesion de la cookie*/
    secret: 'ESTO ES SECRETO',
    /* Por cada llamada realizada al servidor, la sesion se
       guardara en la BBDD */
    resave: true,
```

```
6     /* Cuando se realiza la llamada por primera vez, guarda un
          objeto vacio con informacion de esa session */
7     saveUninitialized: true,
8     store: new MongoStore({
9       url: MONGO_URL,
10      /* Si sucede un error, trata de volver a conectarse */
11      autoReconnect: true
12    })
13  }));
```

Se establecen las opciones de la sesión que será utilizada en la aplicación. Cada sesión es guardada en la base de datos a modo de registro, aunque por el momento no es un funcionamiento relevante para nuestra aplicación.

```
1   app.use(passport.initialize());
```

Se inicializa PassportJS.

```
1   app.use(passport.session());
```

Se determina que passport utilice las sesiones.

Encriptación bcrypt de las contraseñas En la aplicación, el paquete bcrypt es utilizado para encriptar las contraseñas de usuario.

Uno de los casos donde es necesario utilizar bcrypt es cuando es necesaria la comparación de contraseñas en el inicio de sesión. Se hace por medio de la siguiente función:

```
1   /* Comprobar una contrasena con su hash */
2   bcrypt.compare(password, this.password, function(error,
        sonIguales){
3     if(error){
4       return cb(error);
5     }
6     cb(null, sonIguales);
7   })
```

Compara la contraseña con la que está tratando de acceder el usuario después de cifrarla con el hash que existe actualmente en la base de datos. El hash que se encuentra en la base de datos tiene almacenados el coste, la sal y la contraseña, por lo que bcrypt sabe cómo cifrar la nueva contraseña introducida.

Un ejemplo práctico: El hash introducido en la base de datos podría ser el siguiente: `$2a$10$vI8aWBnW3fID.ZQ4/zo1G.q1lRps.9cGLcZEiGDMVr5yUP1KUOYTa`

Este hash contiene tres cadenas concatenadas con el símbolo "$":

- `2a` identifica la versión del algoritmo bcrypt utilizado.
- `10` es el factor de coste: Se han utilizado 2^{10} iteraciones de la función de derivación de la key.

- `vI8aWBnW3fID.ZQ4/zo1G.q1lRps.9cGLcZEiGDMVr5yUP1KUOYTOa` son la sal y la contraseña cifrados, concatenados y codificados en Base64. Los 22 primeros caracteres codificados en un valor para la sal de 16-byte. El resto de caracteres son la contraseña cifrada que va a ser comparada durante la autenticación.

Otro de los casos donde es necesario utilizar bcrypt es al dar de alta a un usuario.

```
1  bcrypt.genSalt(10, function(error, salt){
2    if(error) {
3      console.log("Error en la sal");
4    }
5    bcrypt.hash(paciente.password, salt, null, function(error,
        hash){
6      if(error) {
7      console.log("Error en el hash");
8      }
9    });
10 });
```

Primero se genera la sal con un factor de coste 10, y después, se crea el hash con la contraseña que ha introducido el paciente y la sal. Este hash será almacenado en la base de datos.

Google Maps JavaScript API para el autocompletado de la localización en los formularios de registro y modificación. Para autocompletar las localizaciones introducidas por el usuario tanto en el formulario de registro como en el de modificación, se ha utilizado la Autocompletado para direcciones y términos búsqueda de Google Maps JavaScript API.

Por ejemplo, en el formulario de registro de usuarios de la *template* pacientes/registroPacientes.html, fue incorporado al código HTML de la siguiente forma:

```
1  <div class="field" id="locationField">
2    <div class="ui left icon input">
3      <i class="marker icon"></i>
4      <input id="autocomplete" name="localizacion" placeholder="
          Localizacion" ng-focus="geolocate()" type="text"
          required>
5    </div>
6  </div>
```

Donde ng-focus ng-focus es una directiva que indica que cuando el input esté señalado, se ejecute la función `geolocate()`.

La función `geolocate` se encuentra en pacientes/pacientesRegistroController.js y viene definida por Google de la siguiente forma:

```
1  $scope.geolocate = function() {
2    if (navigator.geolocation) {
3      navigator.geolocation.getCurrentPosition(function(position)
           {
4        var geolocation = {
5          lat: position.coords.latitude,
6          lng: position.coords.longitude
7        };
8      });
9    }
10 }
```

Esta función, simplemente toma la ubicación donde se sitúa el usuario en ese momento. $scope se utiliza para pasar la función del *controller* a la *template*.

Por otra parte, la página se encuentra contenida bajo la siguiente etiqueta:

```
1 <div class='container' ng-init="initAutocomplete()">...</div>
```

La directiva ng-init ng-init evalúa `initAutocomplete()` al inicializar la aplicación.

La función `initAutocomplete` se encuentra en el *controller*.

```
1  $scope.initAutocomplete = function() {
2    autocomplete = new google.maps.places.Autocomplete(
3      (document.getElementById('autocomplete')),
4      { types: ['(cities)'], /* Se buscaran ciudades */
5       componentRestrictions: {country: "es"}}); /* Restringidas
           dentro de Espana */
6
7    /* Cuando el usuario selecciona una opcion del desplegable,
        se ejecuta la funcion indicada */
8    autocomplete.addListener('place_changed', fillInAddress);
9
10   /* Funcion que recupera el lugar escogido en el
        autocompletado */
11   function fillInAddress() {
12     /* Toma los detalles del lugar del objeto de autocompletado
           */
13     place = autocomplete.getPlace();
14   }
15 }
```

`google.maps.places.Autocomplete` crea un objeto tomando como argumentos el `input` donde se quiere autocompletar y una serie de opciones. Para la aplicación, se han restringido las opciones a ciudades españolas.

Nodemailer.js Cuando un psicólogo cubre el formulario de registro, sus datos son mandados al correo electrónico de Emozio para poder valorarlos antes de formar parte de nuestra base de datos. Para enviar los e-mails se ha utilizado Nodemailer. Algunas particularidades son descritas a continuación.

```
var transporter = nodemailer.createTransport({
  service: 'gmail',
  auth: {
    user: 'emozio.info@gmail.com',
    pass: '********************'
  }
});
```

Se crea un objeto transporter utilizado en el trasporte por defecto SMTP. SMTP es el transporte utilizado en Nodemailer para el envío de mensajes, pero también, es el protocolo utilizado por diferentes *host* de *email*.

```
var mailOptions = {
  from: 'emozio.info@gmail.com',
  to: 'emozio.info@gmail.com',
  subject: 'Emozio Web - Nuevo psicologo',
  generateTextFromHTML: true,
  html: /*Aqui iria el codigo HTML */
};
```

Se especifican las opciones de *email* que se quieren enviar.

```
transporter.sendMail(mailOptions, (error, info) => {
  if (error) {
    return console.log(error);
  }
  console.log('Message sent: %s', info.messageId);
});
```

Se envía el *email* con la función `sendMail` a través del `transporter`.

Validación de formularios con SemanticUI Para la validación de formularios se utilizó SemanticUI que funciona de la siguiente forma:

En la *template*, se han de tener todos los campos de formulario corréctamente nombrados y añadir un identificador a la etiqueta `<form>`.

En el *controller*, existe un objeto especial al que se le pueden pasar la lista de elementos del formulario a validar, las reglas que ha de cumplir cada campo y los mensajes de error en cada caso.

En el código de la aplicación, un ejemplo podría ser:

```
$('#access_form').form({
  on : 'blur', /* Cada elemento se evalua por separado */
  inline: 'false', /* Los mensajes de validacion no se disponen
      en linea */
  /* Campos a validar: Identificador y reglas a evaluar */
  fields : {
    email : {
      identifier : 'email',
      rules : [
        {
```

```
            type : 'empty',
            prompt : 'Por favor, introduzca un e-mail.'
          },
          {
            type: 'email',
            prompt: 'El formato del e-mail es incorrecto.'
          },
          {
            type: 'maxLength[50]',
            prompt: 'Demasiados caracteres.'
          }
        ]
      },
      password: {
        identifier: 'password',
        rules: [
          {
            type: 'empty',
            prompt: 'Por favor, introduzca una contrasena.'
          },
          {
            type: 'maxLength[50]',
            prompt: 'Demasiados caracteres.'
          }
        ]
      }
    }
  });
```

Los campos a validar en el formulario de acceso a la plataforma son email y password.

Para email, las reglas de validación son:

- El campo no debe estar vacío.
- El formato debe ser de tipo email.
- La logitud máxima de carácteres permitidos es 50.

Para password, las reglas de validación son:

- El campo no debe estar vacío.
- La logitud máxima de carácteres permitidos es 50.

Para evaluar que se cumplen las condiciones especificadas, se utiliza la siguiente función:

```
$('#access_form').form('is valid');
```

2.3. Implementación del tercer incremento

Los requisitos implementados en el segundo incremento son los asociados a la parte de la comunicación:

- RF-007: Contacto del paciente con el psicólogo
- RF-008: Valoración del psicólogo por parte del paciente

Además, aunque en primera instancia no estuviese contemplado, surgió la idea de gestionar la comunicación como un sistema de citas. Por ello, se incorporó a posteriori un calendario mensual donde el psicólogo pudiese ver las citas que tiene agendadas con una vista por mes, por semana y por día.

Las tareas más trascendentales realizadas en este incremento son las descritas a continuación:

Template	***Controller***
pacientes/pacientesMail.html	pacientes/pacientesMailController.js
psicologos/psicologosMail.html	psicologos/psicologosMailController.js
psicologos/psicologoCalendario.html	psicologos/psicologosCalendarioController.js

Implementación de las *templates* y *controllers* de los requisitos. Éstos se encuentran dentro del directorio app/assets.

Creación de la *factories*, *routes* y *models* de Mensaje.

Visualización de las citas agendadas por medio de FullCalendar En la *template* psicologoCalendario.html, el calendario es dispuesto de la siguiente forma:

```
1 <div id="calendar" ui-calendar ng-model="eventSources"></div>
```

Este elemento es gestionado por el psicologosCalendarioController.js.

```
1  $('#calendar').fullCalendar({
2    header: { /* Opciones de la barra de herramientas */
3        left: 'prev,next today', /* A la izquierda: Anterior,
             siguiente y hoy */
4        center: 'title', /* En el centro: Nombre del mes */
5        right: 'month,basicWeek,basicDay' /* A la derecha: Mes,
             semana y dia */
6      },
7      buttonText : { /* Nombre que aparece en las opciones de la
          barra de herramientas */
8        today:    'hoy',
```

```
9           month:     'mes',
10          week:      'semana',
11          day:       'dia',
12          prev:    '<',
13          next:    '>'
14        },
15        firstDay : 1, /* Empieza el lunes */
16        dayNames : ['Domingo', 'Lunes', 'Martes', 'Miercoles', '
              Jueves', 'Viernes', 'Sabado'], /* Nombres de los dias */
17        dayNamesShort : ['Dom.', 'Lun.', 'Mart.', 'Mierc.', 'Juev.'
              , 'Vier.', 'Sab.'], /* Nombres abreviados de los dias */
18        monthNames: ['Enero','Febrero','Marzo','Abril','Mayo','
              Junio','Julio','Agosto','Septiembre','Octubre','
              Noviembre','Diciembre'], /* Nombres de los meses */
19        monthNamesShort: ['Ene','Feb','Mar','Abr','May','Jun','Jul'
              ,'Ago','Sep','Oct','Nov','Dic'], /* Nombres abreviados
              de los meses */
20        navLinks: true, /* Permite navegar entre las distintas
              vistas: Mes, semana y dia */
21        editable: true, /* Los eventos pueden ser modificados */
22        eventLimit: true, /* Numero de eventos mostrados en un dia
              limitado */
23        theme: true, /* Activa el tema */
24        themeSystem:'bootstrap3', /* Establece el tipo de tema */
25        eventColor: '#ffffff', /* Establece el color del fondo de
              los eventos */
26        eventTextColor: '#008080', /* Establece el color del texto
              de los eventos */
27        eventClick: function (calEvent) { . . . },
28
29        height: 600 /* Establece la altura del calendario */
30      });
```

Para meter los eventos en el calendario se utiliza la función fullcalendar con el argumento `renderEvent`.

```
1 $('#calendar').fullCalendar('renderEvent', cita, true);
```

- Se inicializa el calendario.
- Opciones mostradas en los botones del calendario.
- Traducción de los mensajes de los botones (que traen por defecto) al castellano.
- Indica el día donde comienza la semana.
- Traducción de los días, abreviaturas de los días, meses y abreviaturas de los meses al castellano.
- Función que se ejecuta cuando se hace *click* en un evento y lo muestra.

Bibliografía

[1] Acerca de Node.js. https://nodejs.org/es/about/, Consultado de 29 de enero del 2018.

[2] Arquitectura AngularJS. https://angular.io/guide/architecture, Consultado de 29 de enero del 2018.

[3] CSS. https://developer.mozilla.org/es/docs/Web/CSS, Consultado de 29 de enero del 2018.

[4] Documentación de AngularJS. https://angular.io/docs, Consultado de 29 de enero del 2018.

[5] Github Semantic UI. https://github.com/Semantic-Org/Semantic-UI, Consultado de 29 de enero del 2018.

[6] HTML. https://developer.mozilla.org/es/docs/Web/HTML, Consultado de 29 de enero del 2018.

[7] ISO 9241-171:2008 Ergonomics of human-system interaction - Part 171: Guidance on software accessibility. https://www.iso.org/standard/39080.html, Consultado de 29 de enero del 2018.

[8] JavaScript. https://developer.mozilla.org/es/docs/Web/JavaScript, Consultado de 29 de enero del 2018.

[9] Objetos globales: Promise. https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Promise, Consultado de 29 de enero del 2018.

[10] Places de Google Maps JavaScript API. https://developers.google.com/maps/documentation/javascript/places-autocomplete?hl=es-419, Consultado de 29 de enero del 2018.

[11] Promise. https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Usar_promesas, Consultado de 29 de enero del 2018.

[12] Página oficial de Bootstrap. https://nodejs.org/es/about/, Consultado de 29 de enero del 2018.

[13] Página oficial de Brackets. http://brackets.io/, Consultado de 29 de enero del 2018.

[14] Página oficial de ExpressJS. http://expressjs.com/es/, Consultado de 29 de enero del 2018.

[15] Página oficial de FullCalendar. https://fullcalendar.io/, Consultado de 29 de enero del 2018.

[16] Página oficial de Git. https://git-scm.com/, Consultado de 29 de enero del 2018.

[17] Página oficial de GitHub. https://github.com/, Consultado de 29 de enero del 2018.

[18] Página oficial de GruntJS. https://gruntjs.com/, Consultado de 29 de enero del 2018.

[19] Página oficial de LaTeX. https://www.latex-project.org/, Consultado de 29 de enero del 2018.

[20] Página oficial de LibreOffice. https://es.libreoffice.org/, Consultado de 29 de enero del 2018.

[21] Página oficial de Mongoose. http://mongoosejs.com/, Consultado de 29 de enero del 2018.

[22] Página oficial de NPMJS. https://www.npmjs.com/, Consultado de 29 de enero del 2018.

[23] Página oficial de PassporJS. http://www.passportjs.org/, Consultado de 29 de enero del 2018.

[24] Página oficial de Pencil. https://pencil.evolus.vn/, Consultado de 29 de enero del 2018.

[25] Página oficial de Robomongo. https://robomongo.org/, Consultado de 29 de enero del 2018.

[26] Página oficial de StarUML. http://staruml.io/, Consultado de 29 de enero del 2018.

[27] UNE 139803:2012 Requisitos de accesibilidad para contenidos en la Web. http://administracionelectronica.gob.es/pae_Home/pae_Estrategias/pae_Accesibilidad/pae_normativa/pae_eInclusion_Normas_Accesibilidad.html#.WBbsEYVwZZ0, Consultado de 29 de enero del 2018.

[28] AngularJS in Patterns. https://github.com/mgechev/angularjs-in-patterns, Consultado el: 29 de Enero del 2018.

[29] Bootstrap Grids. w3schools. https://www.w3schools.com/bootstrap/bootstrap_grid_basic.asp, Consultado el: 29 de Enero del 2018.

[30] Data binding. w3schools. https://www.w3schools.com/angular/angular_databinding.asp, Consultado el: 29 de Enero del 2018.

[31] Dependency Injection in JavaScript. https://dzone.com/articles/dependency-injection-0, Consultado el: 29 de Enero del 2018.

[32] Lato. Google Fonts. https://fonts.google.com/specimen/Lato, Consultado el: 29 de Enero del 2018.

[33] Merriweather. Google Fonts. https://fonts.google.com/specimen/Merriweather, Consultado el: 29 de Enero del 2018.

[34] Page Controller. Microsoft. https://msdn.microsoft.com/en-us/library/ff649595.aspx, Consultado el: 29 de Enero del 2018.

[35] Understanding REST. https://spring.io/understanding/REST, Consultado el: 29 de Enero del 2018.

[36] Thorén C. *Nordic Guidelines for Computer Accessibility*. Nordic Cooperation on Disability, second edition, 1998.

[37] George Field. *Chromatics: Or, the Analogy, Harmony, and Philosophy of Colours*. David Bogue, Fleet Street, 1845.

[38] Ekberg J. *Un paso adelante: Diseño para todos. Proyecto INCLUDE.* CEAPAT-IMSERSO, Madrid., 2000.

[39] Ivan Beschastnikh Keheliya Gallaba, Ali Mesbah. *Don't Call Us, We'll Call You: Characterizing Callbacks in Javascript*. Empirical Software Engineering and Measurement (ESEM), 2015 ACM/IEEE International Symposium on, 2015.

[40] MongoDB. The MEAN Stack: MongoDB, ExpressJS, AngularJS and Node.js. https://www.mongodb.com/blog/post/the-mean-stack-mongodb-expressjs-angularjs-and, Consultado de 29 de enero del 2018.

[41] MongoDB. Reinventando la gestión de datos. `https://www.mongodb.com/es`, Consultado de 29 de enero del 2018.

[42] Fernando Monteiro. *Learning Single-page Web Application Development.* PACKT PUBLISHING, 2014.

[43] Afra Pascual y Jesús Lorés Mª Paula González. Evaluación heurística. Universitat de Lleida. `http://w.aipo.es/libro/pdf/15-Evaluacion-Heuristica.pdf`, Consultado de 29 de enero del 2018.

[44] Addy Osmani. *Learning JavaScript Design Patterns.* O'Reilly, 2012.

[45] Stoyan Stefanov. *JavaScript Patterns.* O'Reilly, 2010.