etsit
escuela técnica superior de
ingeniería de telecomunicación

Desarrollo de servicios web para consulta de datos en entornos médicos

Máster Universitario en Ingeniería en Telecomunicación

Trabajo Fin de Máster

Autor:
Jorge San Emeterio Villalaín
Tutor/es:
José Fernando Cerdán Cartagena
Andrés Cabrera Lozoya

Julio 2019

Universidad Politécnica de Cartagena | Campus de Excelencia Internacional

Desarrollo de servicios web para consulta de datos en entornos médicos

Universidad Politécnica de Cartagena

Autor
Jorge San Emeterio Villalaín

Tutor/es
José Fernando Cerdán Cartagena
Departamento de tecnologías de la información y las comunicaciones
Andrés Cabrera Lozoya
Departamento de física aplicada y tecnología naval

Máster Universitario en Ingeniería en Telecomunicación

etsit escuela técnica superior de ingeniería de telecomunicación

Universidad Politécnica de Cartagena | Campus de Excelencia Internacional

CARTAGENA, Julio 2019

Preámbulo

Desde los años 90, las tecnologías *Web* se han ido introduciendo poco a poco en cada uno de los hogares del planeta. Desde el famoso *Internet Explorer* a *Mozilla Firefox* no hay persona humana que no haya usado alguna vez un navegador para entrar en la red de redes y ser bienvenido por el famoso logotipo de *Google*. Este documento tiene como intención enfocar y analizar una parte de este mundo, el **Front-End**, aplicando su estudio a un caso concreto de uso: un servicio para una clínica médica.

A mis tutores por darme la oportunidad
para trabajar en algo que me fascina
y a mi familia por aguantarme en el proceso.

Un lenguaje que no altera
tu percepción sobre la programación
no merece ser aprendido.

Alan J. Perlis.

Índice general

Índice de figuras

Índice de tablas

Índice de Códigos

1 Introducción

La medicina es un campo en el que la interacción con el usuario, en este caso el paciente, forma una parte fundamental de su ser. Día tras día, miles de pacientes acuden a un sistema sanitario cuyo cometido se basa en la correcta recepción y atención de las personas para que estas puedan tener acceso al cuidado de un profesional cualificado. Es esta relación fundamental entre médico y paciente la que todo sistema sanitario debe cuidar, puesto que es la fluidez de esta correspondencia la que al final del día decide la satisfacción general que es percibida.

El problema surge cuando este proceso es entorpecido o sobre-saturado debido a aquellos pasos intermedios implícitos en la actividad, que acaban siendo tanto tediosos como poco óptimos. Tareas mecánicas como la petición de una cita con un doctor o la consulta del resultado de una analítica conllevan un consumo de recursos humanos, que a falta de ellos, pueden complicar el correcto funcionamiento de una clínica. Junto a ello, se añade la tendencia al error que conlleva el factor humano, siendo la corrección de dichos errores otro origen de consumo de más recursos.

Es por ello que minimizar el empleo de personas para este tipo de labores surge como un tema de importancia. Es en este punto donde las máquinas acuden como un gran solución. Los sistemas modernos están altamente especializados y preparados para solventar propósitos repetitivos y previsibles, justo aquellos que se pretende obviar, por lo que la idea de que sean estos sistemas los delegados de interactuar y gestionar datos y personas toma relevancia.

Es en este marco en donde se sitúa este proyecto. El interés que surge a lo largo de él es el del uso de las tecnologías modernas para gestionar la interacción más habitual que un paciente presenta con su clínica, liberando así carga de trabajo en el entorno y produciendo un servicio más relajado. A lo largo de este documento se tratará los problemas tecnológicos enfrentados con tal de corresponder este problema, así como se hará un pequeño inciso sobre la sensibilidad que el entorno médico presenta cuando se trabaja con su privacidad.

1.1 Objetivos del proyecto

La idea del proyecto consiste en proporcionar un entorno digital al usuario en el cual este sea capaz de solventar sus consultas médicas más comunes. Desde comprobar la fecha de su próxima cita hasta descargar la última radiografía que le ha sido realizada, la plataforma debe poder conectar a paciente y clínica de un modo en el que la intervención humana no sea necesaria.

Con esto en mente, el objetivo de este proyecto se centra pues en la construcción de una **plataforma** apta para el sustento y mantenimiento de una serie de servicios digitales que cumplan con los propósitos mencionados anteriormente. Al mismo tiempo, la plataforma debe contar con la heterogeneidad del público que ha de acceder a ella y, por lo tanto, debe mantenerse ajena a posibles restricciones en su uso bien sean debidas a dificultad de uso o a condiciones de acceso.

En síntesis, con lo que ha sido mencionado hasta el momento los objetivos del proyecto pueden ser listados tal como a continuación:

1. **Creación de un entorno interactivo expansible o *toolbox***: Un espacio en donde se puedan crear y presentar una selección de herramientas al usuario.

2. **Creación de un juego de herramientas básico**: El cual esta formado por la siguiente lista:

    a) Vista con los datos personales del paciente: nombre, correo electrónico, dni, etc.

    b) Vista con los resultados de las analíticas por las que el paciente ha pasado.

    c) Vista con un resumen general del historial clínico del paciente.

    d) Vista con un resumen general del historial de alergias detectadas en el paciente.

    e) Vista con notificaciones y eventos de interés a la persona.

    f) Acceso a un servicio para el consentimiento de actividades médicas puntuales.

    g) Acceso al servicio de consulta de citas mediante el cual el usuario puede o bien ver las citas que están en curso o generar nuevas.

    h) Acceso a un servicio a través del cual es posible establecer una comunicación con el equipo de gestión de la clínica.

3. **Accesibilidad global a la aplicación**: Significando que todo paciente debe tener la posibilidad de acceder al servicio por igual y desde cualquier situación.

4. **Compromiso de privacidad**: Los datos médicos son de una sensibilidad crítica y por ello deben ser tratados con respeto.

Una vez conocido cuales son los factores que se esperan de este proyecto, el siguiente paso consiste pues en el planteamiento y proposición de una solución capaz de cumplir con aquello que le ha sido propuesto. En consecuencia, los apartados próximos van a atender al razonamiento y propuesta de la solución que va a ser seguida a lo largo de este documento.

1.2 Presentando una solución

Entrando en contexto, el proyecto que aquí se cultiva esta plenamente inmerso en el entorno del desarrollo de *Software*. A fin de cuentas, el producto final que será obtenido no es más que una aplicación digital como las demás, que en su caso está pensada para un propósito concreto. Sin embargo, el mundo del *Software* es formidablemente extenso y variado. La metodología de trabajo y los conocimientos que han de ser adquiridos dependen completamente de la rama tecnológica que sea escogida.

Puesto que uno de los objetivos planteados con anterioridad consiste en maximizar el alcance y la accesibilidad de la plataforma, la gran constelación de tecnologías que cohabitan en el desarrollo del *Software* es enfocada y estrechada hasta alcanzar unas pocas opciones. En síntesis, las alternativas que han sido tenidas en cuenta son tales como se describen a continuación:

- Opción 1: Aplicación *Web* accesible desde un navegador *Web*.
- Opción 2: Aplicación móvil accesible de forma directa desde un dispositivo móvil.

(a) Aplicación *Web* **(b)** Aplicación móvil

Figura 1.1: Opciones de implementación

1.2.1 Comparando las alternativas

El proyecto está empezando a tomar forma, en este instante los objetivos de este así como los posibles caminos que se puedan tomar para cumplirlos se encuentran ya definidos. Es el momento pues de tomar una decisión, es la hora de comparar las alternativas presentes y sentenciar la forma que va a adoptar este trabajo. Obviamente, la elección que aquí se plantea depende en su mayoría del ajuste que las distintas posibilidades exhiben con respecto a los objetivos que estas deben cumplir. A su vez, la decisión también se puede ver afectada por circunstancias de carácter personal.

El análisis va a comenzar primero por estudiar el entorno de cada una de las opciones. Por lo tanto, el objetivo de este primer apartado es describir por encima el contexto en el que se sitúan las dos tecnologías aquí presentes. A partir de este momento ya se busca el presentar al proyecto como algo que es tangible, es decir, más cercano a lo que conlleva en la realidad. Con todo ello, se presenta la siguiente tabla a modo de entrada en calor:

Tecnología	Entorno	Lenguajes	Constructor	Despliegue
	Safari	JS6	Yarn	
	Firefox	CSS3	Babel	Apache2
Web	Chrome	HTML5	WebPack	Windows Server
Móvil	Android	Java	Gradle	Apk

Tabla 1.1: Comparativa de las dos tecnologías propuestas: Entorno de desarrollo

A simple vista, es posible comprobar como el desarrollo móvil es mucho más directo y simple. En este contamos con un sistema operativo: Android, el cual se trabaja utilizando un único lenguaje de programación y que cuenta con un sistema estandarizado para la construcción y despliegue de la aplicación.

Por el otro lado contamos con el desarrollo *Web*, este entorno se cimienta sobre los navegadores *Web*, como por ejemplo *Chrome* o *Safari*, y presenta un medio a varios niveles más heterogéneo que el descrito en el párrafo anterior. Por un lado, la variedad de los navegadores *Web* presenta un problema en sí mismo debido a que mientras que intentan evitarlo, el comportamiento entre ellos difiere en algunos casos. Por el otro lado se encuentran los lenguajes de programación y los constructores, la solución que se le da a los problemas que surgen del desarrollo *Web* consiste en la mezcla y convivencia de un montón de tecnologías que en su totalidad cumplen con la misma función que el constructor general del desarrollo móvil. Esto añade una mayor complejidad al proyecto debido a que el mantenimiento y enlazado de estas tecnologías requieren de un entendimiento adicional por sí solo.

En la segunda parte de la comparativa se va analizar las dos tecnologías anteriores de un modo más enfocado a los objetivos del proyecto. Por consiguiente, se presenta la siguiente tabla:

Tecnología	Funcionalidad	Conocimiento previos	Accesibilidad	Privacidad
Web	✓	✗	Completa	Segura
Móvil	✓	✓	Parcial	Segura

Tabla 1.2: Comparativa de las dos tecnologías propuestas: Cumplimiento de objetivos

En cuanto a funcionalidad y privacidad las dos opciones cumplen con los mismos requisitos, ambas son capaces de albergar las funciones interactivas de las que precisa la aplicación y la seguridad se presenta con el mismo formato para ambos casos: usuario y contraseña bajo comunicación cifrada. Otro detalle que como autor quisiera tener en cuenta resulta en que como desarrollador ya cuento con experiencia en el desarrollo de aplicación móviles, mientras que con los servicios *Web* sería una primera vez. Este factor afecta a la decisión debido a que dependiendo de la elección tomada el tiempo dedicado al aprendizaje varía considerablemente. Finalmente contamos con la accesibilidad, en este punto el servicio *Web* es el claro ganador debido a que se puede entrar a él desde cualquier dispositivo mientras que el móvil está limitado a aquellos terminales que cuenten con el sistema operativo *Android*.

1.2.2 Tomando una elección

Las cartas están puestas todas sobre la mesa, el análisis comparativo de las dos tecnologías ha concluido y es hora de tomar una decisión. A simple vista puede parecer que la respuesta es bastante trivial, el entorno de desarrollo móvil se presenta como una solución más simple y limpia que lo que pueda ofrecer el entorno *Web*. Sin embargo, esta rama presenta un error final decisivo, y esto recae sobre la accesibilidad.

Al comienzo de la proposición de este proyecto, uno de los objetivos principales de este resultaba ser el permitir que cualquier persona, no importa el dispositivo o procedencia, tuviera la posibilidad de acceder al servicio. También se mencionó como esto resulta ser debido a la heterogeneidad de la multitud que acude a un centro de salud cualquiera. Pues resulta ser este factor el que determina que la balanza se decante por el entorno *Web*. Puede ser más complejo sí, pero permite una mayor flexibilidad a ser accesible tanto desde terminales de escritorio como portátiles. A día de hoy se puede debatir como la gran mayoría de la población cuenta de antemano con un teléfono móvil, pero con tal de asegurar que el factor se satisfaga la decisión al final cae sobre **el servicio *Web***.

2 El mundo del Front-End

Se define como servicio *Web* a aquellas aplicaciones que se basan en un conjunto concreto de protocolos y estándares para el intercambio de datos entre sí. El servicio *Web* se presenta como uno de los fundamentos de *Internet*, que es el modo más popular o entre los más populares para la compartición de datos a través de una red. Un servicio *Web* no es un todo concreto, si no es más bien la acumulación de tecnologías de uso específico que en su conjunto dotan de funcionalidad a la comunicación. Ejemplo de estas tecnologías puede ser:

- **HTML**: Lenguaje etiquetado para la descripción de páginas *Web*.
- **HTTP**: Protocolo de telecomunicaciones fundamentado en la descripción de mensajes en base a palabras clave conocidas como *verbos*.
- **XML**: Lenguaje etiquetado para la descripción de contenidos.
- **REST**: Capa superior a *HTTP* que indica la interfaz de comunicación presente entre cliente y servidor.

La gran ventaja del servicio *Web* frente a otros tipos de comunicación es que se presenta como agnóstico del medio, esto es, la comunicación entre las máquinas se puede realizar sin tener en cuenta los dispositivos físicos que intervienen en ella o los lenguajes de programación usados para generar el *Software*. Es aquí donde el servicio *Web* presenta esa gran flexibilidad de la que se mencionaba anteriormente, puesto que las tecnologías envueltas en ello no tienen en cuenta el dispositivo y origen de la comunicación.

En el caso de este proyecto, el enfoque de la aplicación va a estar dirigido a la puesta en marcha de una página *Web*, esto es, una aplicación *dinámica* que se ejecuta dentro de un **navegador**. El encargado de producir y entregar las páginas correspondientes a un estado puntual de la aplicación es el **servidor**, un dispositivo que se dedica a la escucha de la red en busca de peticiones para generar respuestas que en su total formen la aplicación. El navegador es el encargado de contactar y recibir las páginas desde el servidor, y en apoyo a ellas generar una interfaz gráfica para la interacción con el usuario.

Este documento se centra en el trabajo realizado sobre la parte del navegador *Web*, esto es, la adquisición y presentación de datos procedentes desde el servidor. Inevitablemente, se hará mención de ciertas partes de funcionamiento del servidor, puesto que este es parte integra de la aplicación aquí propuesta. Sin embargo, a tener en cuenta que el foco de aquí no está sobre él.

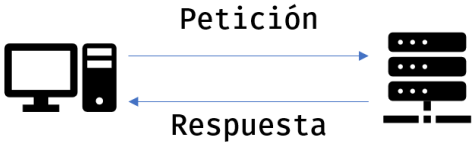


Figura 2.1: Diagrama funcionamiento cliente - servidor

En términos de desarrollo de aplicaciones *Web*, es común toparse con la distinción del mundo del navegador y del servidor mediante los términos: **Front-End** y **Back-End**, donde estos representan a uno y a otro respectivamente. En el caso que aquí encumbre, el foco de atención recae sobre la presentación de datos al usuario, y por ello, sobre el **Front-End**.

En su forma más básica, un navegador es un interprete capaz de entender tres lenguajes de programación:

1. **HyperText Markup Language (HTML)**: Lenguaje descrito mediante etiquetas predefinidas que indican los contenidos base de una página *Web*. Como tal, un navegador no precisa de nada más que de un documento *HTML* para presentar algo al usuario. Sin embargo, *HTML* es tan pobre de per sé que su uso aislado resulta inviable.

Código 2.1: Ejemplo de código HTML

```
<html>
  <body>
    Hello World!
  </body>
</html>
```

- **Cascading Style Sheets (CSS)**: Lenguaje de descripción de estilos. Este es uno de los primeros parches que se añaden al *HTML* básico. Mediante este lenguaje, es posible decirle al navegador como debe presentar la página *Web* al usuario. Ejemplos básicos de las opciones que otorga son: cambiar la fuente del texto, elegir el color de fondo, etc.

Código 2.2: Ejemplo de código CSS

```
body {
  color: blue
}
```

- **JavaScript (JS)**: Lenguaje de descripción de funciones. *JavaScript* presenta a día de hoy la parte más importante del desarrollo de *Front-End*. Sin este lenguaje, la *Web* no es más que un lugar estático donde solo se presentan textos. Gracias a él, es posible dotar de vida e interacción a las páginas.

Código 2.3: Ejemplo de código JS

```
function say_hello() {
  console.log("Hello")
}
```

A día de hoy, toda la generación de una página *Web* pasa de antemano por *JavaScript*. A través del lenguaje de programación se definen fragmentos de código, conocidos como **plantillas**, de los otros dos lenguajes mediante las cuales se genera dinámicamente el código final que se le entrega al navegador.

Por lo tanto, los mayores esfuerzos de este proyecto van a girar en torno a *JS*. Se estudiará como es un sistema de producción moderno y como es implementado este lenguaje de programación dentro él.

2.1 JavaScript

JavaScript es un lenguaje de programación de complicada definición. En su modo más básico se define como un lenguaje prototipado, más sobre esto posteriormente; funcional, dinámico y débilmente tipado. En términos generales se puede considerar que es un lenguaje terriblemente laxo, los códigos escritos sobre él no siguen una estructura prefijada como puede ocurrir en la orientación a objetos[1] de *Java* o *C++*. Esto último presenta sus ventajas y desventajas, pero ayuda en gran medida a la adaptación del lenguaje al medio, el servicio *Web*, donde no siempre es sencillo prever una estructura clara.

Figura 2.2: Logo de *JavaScript*

[1] OOP: Estilo de programación basado en el empaquetado de toda la información en contenedores conocidos como *objetos*.

2.1.1 El lenguaje

Con tal de dar una idea sobre con lo que se está trabajando, a continuación se describen las definiciones que particularizan a *JS*:

- **Prototipado**: La información está estructurada en contenedores que parten de la base de otros contenedores previos. Estos contenedores previos forman lo que se conoce como el prototipo del objeto. De esta forma es posible organizar la información como una cascada de tipos de datos en el que los niveles bajos adquieren las funciones de los que le preceden. El sistema de prototipado es similar a la herencia de objetos en una arquitectura OOP, pero en este caso la herencia es menos exigente que en el otro.

- **Débilmente tipado**: Las variables que contienen la información de un contenedor no tienen un tipo de dato fijo. En lenguajes fuertemente tipados como *Java*, el usuario está obligado a definir cualquier variable como la instancia de un arquetipo: *int, float, String, Object....* Por el otro lado, para un lenguaje débilmente tipado solo se cuenta con la palabra clave: **var**. Esto no significa que los datos de *JavaScript* no tengan un arquetipo, si no que está en la mano del programador mantener y prever cual será.

- **Dinámico**: Los contenedores de información ven su estructura modificada sobre la ejecución del programa. Esto es algo que los objetos de OOP prohiben por completo.

- **Funcional**: La estructura más importante del lenguaje es la **función**. Esta se comporta como un objeto en sí mismo, pudiendo albergar en su interior tanto información, como operaciones, como otras funciones, etc.

- **Scriptado**: Un fichero contenedor de código *JS* (.js) se considera auto-ejecutable. La aplicación no precisa del clásico método *Main* para ser capaz de ejecutar una porción de su código fuente.

- **Interpretado**: El lenguaje no es *compilado*[2] como un lenguaje tradicional. En su lugar, el navegador se encarga de ir leyendo el código linea a linea e indicar al ordenador los pasos que debe seguir con tal de ejecutar cada una de ellas. Al navegador se lo conoce como el interprete debido a que es capaz de entender y traducir el lenguaje a algo que el ordenador pueda ejecutar.

[2]Proceso por el cual se transforma el código fuente de la aplicación en un programa binario ejecutable por el ordenador.

El entorno de *JavaScript* está por lo general completamente limitado al navegador *Web*, es una tecnología que desde su nacimiento se enlazó con este entorno y realmente mantiene un sentido dentro de él. Fuera de esto simplemente existen lenguajes que hacen un trabajo mejor que él. Sin embargo, la gran flexibilidad y accesibilidad que proporciona el servicio *Web* ha propiciado que en los últimos tiempos haya un movimiento con el cual llevar a *JavaScript* a las aplicaciones de escritorio. El mayor testigo de esto último es la plataforma *Electron*, la cual alberga programas de importancia como *Spotify* o *Discord*. Pese a ello, el uso de este lenguaje en aplicaciones de escritorio sigue siendo un tema de debate debido a lo ineficiente que puede ser en algunos casos.

2.1.2 La evolución del lenguaje

Los años noventa, la revolución digital comenzaba a encontrar su curso y, con ella, una pequeña cosa llamada *Internet* se veía entrando poco a poco en cada vez más y más hogares. Como es de esperar, las empresas se dieron cuenta del potencial de esta nueva moda. Todo el mundo quería conseguir un cacho de este nuevo mundo, y qué cacho más grande puede haber que el de poseer el principal medio de acceso al mundillo. Es en este contexto donde se sitúa *JavaScript*, una época en la que los dos grandes grupos: *Microsoft* con *Internet Explorer* y la futura *Mozilla* con *Netscape Navigator*; peleaban entre sí con el fin de presentarse como la elección real ante estas nuevas tecnologías.

Por aquel entonces, *Internet* se encontraba en un estado de infancia. Las páginas no venían a ser mucho más que murales de texto con alguna foto embebida entre medias. Esto no era aceptable llegado ya el año 95, el mundo de la computación estaba creciendo y con ello aumentaban las exigencias de los usuarios. Era necesario dotar de una nueva vida a *Internet*, otorgando a sus páginas de mayor dinamismo y atractivo.

Con tal fin, *NetScape* decidió colaborar con la antigua *Sun Microsystems*[3] con la idea de integrar al lenguaje *Java* en su navegador. Gracias a las *Applets*[4], *Java* proporcionaba un buen entorno sobre el que desarrollar aplicaciones *Web*. Sin embargo, no era la solución definitiva. El código *Java* es espeso, que para las conexiones de la época era un problema, y su ejecución sigue contando con elementos externos al navegador. Sería perfecto si a esto lo acompañara un complemento más ligero que se encargue de tareas más básicas. Es en este último paso donde se introduce *JavaScript*.

[3] Actual *Oracle*.
[4] Pequeñas aplicaciones previstas para ser arrancadas por un visualizador.

Originariamente llamado *LiveScript* y posteriormente renombrado a *JavaScript* para seguir el tirón de *Java*, el nuevo lenguaje se presentaba como un medio ligero e integro al navegador listo para dotar a las páginas *Web* de cierta interactividad. La solución cayó en gracia, y fue cogiendo fuerza hasta el punto en el que competidores como *Microsoft* aceptaron integrarlo en sus propios navegadores. No obstante, con tal de diferenciarse y adaptarse mejor al navegador de turno, el lenguaje se veía modificado y desvirtuado dependiendo de las necesidades de cada caso. Esto es un problema para el programador, debido a que el código que pueda escribir en *NetScape* no tiene por qué funcionar completamente igual en *Internet Explorer*.

La descripción formal de *JavaScript* era muy endeble, no habían unas pautas claras con las cuales poder entender el comportamiento del lenguaje. Una definición real y estricta del funcionamiento de esto se necesitaba como agua de mayo. Afortunadamente, con este fin entra en juego la **ECMA**[5] con su estándar **ECMAScript**. *ECMAScript*, comúnmente *ES*, es un intento de normalización de *JavaScript* con la idea de que aquel navegador que implemente la norma tenga un comportamiento esperable al código. A lo largo del tiempo el estándar ha ido evolucionando siguiendo esta línea temporal:

1997	ECMAScript 1
1998	ECMAScript 2
1999	ECMAScript 3
2009-2011	ES5
2015-2018	ES6

Tabla 2.1: Línea temporal de *ECMAScript*

Siendo los dos más importantes **ES5** en el año 2009 y **ES6** en el 2015. Nunca llegó a haber una cuarta versión del estándar debido a la falta de consenso sobre lo que esta marca debía ser. A día de hoy, el *JavaScript* moderno corresponde con la versión 6 de la especificación mediante que el antiguo cae sobre la 5. Estas dos versiones difieren considerablemente la una de la otra, habiendo mecanismos en la 6 mucho más prácticos y comprensibles que en la 5. Por ejemplo:

Código 2.4: Comparando ES5 y ES6

```
/* Importando una librería */

// ES5: CommonJS
var module = require('./module');

// ES6: Modules
import module from './module';
```

[5] Organización europea de estandarización.

Código 2.5: Comparando ES5 y ES6

```
/* Llamada asíncrona */

// ES5
function example(action) {
    return spawn(function*() {
      yield action()
    });
}

// ES6
async function example(action) {
    await action()
}
```

El problema de estos estándares resulta en su lenta adopción. A día de hoy, los navegadores van integrando poco a poco las indicaciones de *ECMAScript 6* llegando hasta un punto de considerable adopción. Sin embargo, el programador aún no puede utilizar esta última versión con seguridad debido a los posibles detalles que aún estén por hacer. Pese a ello, posteriormente se mencionará un mecanismo que es utilizado a día de hoy como solución para el uso de *ECMAScript 6* sin ningún tipo de inseguridades.

2.2 El estado del Front-End en 2019

Hasta este punto, el foco ha estado fijado en explicar cuales son los fundamentos del servicio *Web*. Esto es un buen inicio, ya que las herramientas hasta ahora presentadas son elementalmente aquellas que se necesitan a la hora de crear una aplicación funcional. No obstante, todo esto fue ideado allá en los años 90 con la intención de cumplir con unas necesidades concretas que presentaba el *Internet* de la época. El gran **boom** que supuso la red de redes pilló completamente por sorpresa al sector, siendo su evolución y requerimientos mucho más grandes que los que el *Software* del momento podía afrontar.

Internet crecía demasiado rápido para *HTML*, *CSS* y *JavaScript*. Viendo como todo esto se quedaba anticuado, la solución ideal consiste en descartar todo aquello presente en aquel instante y volver a comenzar con proyectos mucho más adaptados a la situación. Sin embargo, las exigencias del mercado y la falta de tiempo manifestaron a esto como una remedio inviable. Es necesario extender de alguna forma rápida lo que ya hay con tal de adaptar los medios a los nuevos requerimientos que se imponen sobre ellos. Es en este contexto donde se ubican los **parches**, tecnologías y tecnologías que se adhieren a los soportes ya establecidos con tal de dotarlos de más fuerza.

A lo largo de este apartado se procede a explicar cuales son estos *parches* y como entre ellos se forma una plataforma de desarrollo moderna.

2.2.1 El lío de la modularidad

Toda persona que haya trabajado un poco con *C* estará acostumbrada a algo como esto:

Código 2.6: Importando código en C

```
#include<stdio.h>

int main() {
    printf("%s", "Hello World!");

    return 0;
}
```

Incluso para el programa más sencillo, este lenguaje necesita de la **inclusión de código proveniente de otros ficheros**: en este caso *stdio.h*. La organización del código en un proyecto es algo básico y clave, dividir las funciones en elementos de correcto tamaño y designación es una ayuda no solo para aquellas personas que deban trabajar con ello si no también para la propia persona que lo ha realizado. Es simplemente humano, una buena organización permite mantener una buena visión mental de la situación.

Para demostrar las limitaciones del *JavaScript* puro, en su definición no viene incluido un medio por el que soportar esta función. En su lugar, *JS* fomenta el uso de un estado global común a toda la aplicación por el cual todos los ficheros se comunican a través de. De esta forma, no hay una auténtica distinción al separar el código en distintos archivos puesto que realmente el lenguaje los trata como un todo común.

La forma en la cual se incluyen nuevos trozos a este estado global es a través de la etiqueta **script**:

Código 2.7: Importando código en JavaScript

```
<html>
  <body>
    <script src="example_1.js" />
    <script src="example_2.js" />
  </body>
</html>
```

Esto es simplemente intolerable en cuanto el proyecto crece en tamaño. El estado global es un concepto del cual los lenguajes modernos intentan huir y minimizar debido a la dificultad que conlleva su mantenimiento y depurado. Una variable cualquiera puede ser definida, modificada o eliminada en cualquier parte del código y no siempre está claro el flujo que la ha llevado a ese punto.

2.2.1.1 CommonJS

Muchos intentos de solución aparecieron a lo largo de los años, pero no hubo nunca alguno que fuera adoptado como el estándar. No fue hasta el año 2009 con la aparición de **CommonJS** cuando al fin se dotó a *JavaScript* de una estructura modular viable. *CommonJS* es una especificación, que no estándar, la cual indica qué debe hacer un interprete para permitir que fragmentos de código puedan ser reutilizados a los largo del proyecto. Estos fragmentos se los conoce como módulos y su definición es tal como en el siguiente ejemplo:

Código 2.8: Definiendo un módulo en CommonJS

```
const add = function(x, y) {
  return x + y
}

module.exports = {
  add
}
```

Este fragmento define un contenedor, el módulo, donde se introducen aquellos trozos del código *JavaScript* que se quiera exportar y compartir con el exterior. Con tal de utilizar esto en otro espacio, en necesario indicar la dependencia en el receptor:

Código 2.9: Importando código con CommonJS

```
const module = require("Nombre de fichero")

module.add(1, 1)
```

La importancia de *CommonJS* y una de las razones de su adopción es debido a que es el sistema utilizado por el famoso servidor *NodeJS*[6]. Aunque a día de hoy este ha avanzado por su lado modificando ciertas partes de la especificación.

2.2.1.2 ESM

CommonJS es una buena solución al problema de la modularidad en *JavaScript*. Sin embargo, pese a que es gratamente utilizado en el entorno de *NodeJS*, la especificación no forma parte de un estándar. Como se ha mencionado en el apartado 2.1.2, los navegadores están demasiado ocupados manteniendose al día con los estándares del momento, por lo que abarcar especificaciones externas a estos produce que por un lado el navegador difiera de los demás y por otro que el interprete crezca en complejidad.

Sería fenomenal entonces, que un estándar de la *ECMA* describa un mecanismo similar a *CommonJS* que dote de modularidad a *JavaScript* a nivel de lenguaje. Por fortuna esto ocurrió con *ES6*. Los conocidos como módulos ES, o ESM, son una reinterpretación del funcionamiento de los módulos de *CommonJS*, dotándolos de una sintaxis y versatilidad mucho más integral al lenguaje.

Un ejemplo similar al del apartado anterior en *ES6* es el siguiente:

Código 2.10: Definiendo un módulo en ESM

```
exports const add = function(x, y) {
  return x + y
}
```

Código 2.11: Importando código con ESM

```
import { add } from "URL de fichero"

add(1, 1)
```

[6]Un entorno de desarrollo que permite escribir código para servidor, fuera del navegador, con *JavaScript*.

Como se puede ver, el código importado forma en este caso parte del *script* actual, en lugar de ser envuelto en una variable, y la sintaxis resulta mucho más sencilla e intuitiva que la anterior. Es por su modernidad y mejoras por las que en este proyecto se prefiere esta solución.

Para finalizar queda por hacer mención especial a *RequireJS*, otra especificación como las dos anteriores que da solución a este problema. Pese a ser otra alternativa viable, su uso no fue considerado para este proyecto con tal de reducir complejidad.

2.2.2 El gestor de dependencias: Yarn

Crear cualquier tipo de *Software* desde cero existiendo la comunidad y los medios que se prestan a través de la red hoy en día resulta por un lado irresponsable y por otro inviable. Miles de proyectos han pasado previamente por los obstáculos por los que la creación de uno nuevo conlleva y, por lo general, en el camino se han producido y compartido públicamente soluciones con el objetivo de que otros no tengan que pasar por los mismos problemas. Resulta cuestionable no aprovecharse de todas estas herramientas que se encuentran al servicio de todos.

Es proporcionar un núcleo común en el que compartir la función principal de la cual un gestor de dependencias se hace cargo. Este *Software* se encarga por un lado de mantener una base de datos con miles y miles de librerías accesibles por medio de él, así como de gestionar la descarga e instalación de estas librerías dentro de un proyecto.

Gracias al gestor de dependencias, es sencillo acceder a grandes cantidades de código ya pre-hecho, es sencillo de descargar e instalar estos códigos para su uso y es relativamente sencillo mantener un control sobre todo los módulos que componen a un producto final.

En el mundo del *Front-End*, los dos gestores más comunes son **npm** y **yarn**. El primero es el más importante y más utilizado al ser el que viene por defecto dentro del entorno de desarrollo básico, mientras que el segundo engloba al anterior dotándolo de una mayor limpieza y sencillez. Al ofrecer las mismas funciones que *npm* pero de un modo más elegante, en este proyecto se prefirió el uso de *yarn*.

(a) NPM **(b)** Yarn

Figura 2.3: Gestores de dependencias para el desarrollo *Web*

Yarn nació de la mano de *Facebook* como una alternativa que solventa algunos fallos que llevan plagando a *npm* desde hace tiempo, siendo los dos más importantes la velocidad y la seguridad. Para el primer caso, *npm* es lento, no hay mayor debate, una mera prueba de cronómetro entre las dos opciones resultará en que *yarn* es en torno a cuatro veces más rápido que su rival para el mismo caso. En cuanto al segundo problema, *npm* permite que las dependencias ejecuten código durante su instalación. Mientras que útil, esto puede suponer un riesgo innecesario pues está abierto a brechas de seguridad por las que puede entrar *Software* mal intencionado.

Sea cual fuere, estas dos opciones basan su funcionamiento en torno a un fichero llamado **package.json**. Este no es más que un archivo de texto, definido en el formato *JSON*[7], que actúa como un descriptor del proyecto. Este fichero está compuesto por los siguientes puntos:

- **Información del proyecto**: Nombre de este, autores, versión, contacto, etc.
- **Tareas**: Lista de comandos que pueden ser ejecutados en la consola del sistema operativo como un extra de funcionalidad del gestor. Útil para la definición de labores tales como el arranque del servidor de desarrollo o la puesta en producción de la aplicación.
- **Dependencias**: Se indica el nombre de las librerías a descargar así como la versión correspondiente a cada una de ellas. Adicionalmente, también se permite diferenciar entre dependencias de producción y desarrollo, estando las segundas no presentes en la versión final del producto.
- **Información extra**: Navegadores objetivo, servidores objetivo, etc.

Y todo esto se refleja en un fichero tal como este:

Código 2.12: Ejemplo de package.json

```
{
  "name": "Ejemplo",
  "version": "0.1.0",
  "author": "Nombre / E-Mail",
  "scripts": {
      // Tareas
  },
  "dependencies": {
      // Dependencias de produccion
  },
  "devDependencies": {
      // Dependencias de desarrollo
  },
  "browserslist": [
      // Navegadores objetivo
  ],
  "engines": {
      // Servidores objetivo
  }
}
```

[7] Lenguaje de descripción de contenidos similar a *XML*.

El fichero *package.json* puede ser editado a mano como un fichero de texto o a través del propio gestor. Es recomendable el uso del segundo mecanismo debido a que el gestor no captura cambios en vivo sobre el fichero y necesita ser actualizado manualmente, mientras que con este método se actualiza por su propio pie.

Finalmente, a continuación se presenta una pequeña lista con las funciones más útiles de *yarn*:

- **yarn install**: Refresca el gestor con respecto al *package.json*. Esta es la actualización manual a la que se refiere en el párrafo anterior.
- **yarn add *nombre_librería***: Añade una nueva librería con su versión más moderna a las dependencias de producción.
- **yarn add –dev *nombre_librería***: Añade una nueva librería con su versión más moderna a las dependencias de desarrollo.
- **yarn run *nombre_tarea***: Ejecuta una de las tareas indicadas en el fichero.

2.2.3 El transpilador: Babel

En el apartado 2.1.2 se hizo mención a las diferencias que existen entre las dos últimas versiones de *JavaScript*: **ES5** y **ES6**. Posteriomente, en el apartado 2.2.1 se explicó como *ES6* dota de una modularidad deseable al lenguaje que *ES5* simplemente no posee. Es por esto y por otras ventajas por lo que el proyecto va a estar y debe estar escrito en *ES6*. Más todavía si se parte de la base del completo desconocimiento sobre la materia.

No obstante, hay que tener en cuenta que *ES6* es todavía un estándar en desarrollo. Existen ligeras diferencias entre *ES6 2016*, *ES6 2017*..., y que es por ello por lo que depende exclusivamente de la velocidad con la que el navegador se pone al día el presentar mayores o menores incompatibilidades. Lo ideal sería una forma de poder escribir código *ES6* sin tener que limitar el uso de funciones que se pueden usar de este o el número de navegadores a considerar objetivo. Por fortuna existe una solución que permite justamente eso, y esta solución es el **transpilador**.

Un transpilador es una especie de "compilador" que en lugar de traducir un código fuente a lenguaje máquina, lo que hace es traducir un lenguaje de programación en otro. Ejemplo comercial de un transpilador es por ejemplo con el que cuenta el popular *MatLab*. Mediante su módulo de traducción es posible transformar código *MatLab* a *C* con el objetivo de beneficiarse de la velocidad que posee este segundo lenguaje.

En el caso del desarrollo *Web*, el transpilador es una herramienta fantástica puesto que se presenta como otra solución con la que solventar las deficiencias de *JavaScript*. Si es posible coger un lenguaje más potente que él, y adaptarlo para que sea interpretado como código nativo al navegador entonces estamos ganando lo mejor de los dos mundos.

En el caso de *JavaScript* ya existen lenguajes construidos por encima de él, siendo los dos más populares: *TypeScript* y *CoffeeScript*. El primer caso dota al lenguaje de orientación a objetos y tipado fuerte, mientras que el segundo le otorga una sintaxis más potente.

Volviendo a centrar el tema en el que caso que aquí atañe, no es necesario irse a un caso tan lejano como un lenguaje distinto para usar un transpilador. Aplicando las mismas técnicas que se ven en los otros casos es posible transformar una versión de *JavaScript* en otra. Por ejemplo, escribir código *ES6* y que este se transforme en un código *ES5* totalmente compatible. Hay varias alternativas para realizar esta tarea, pero la más popular es sin duda **Babel**.

Figura 2.4: Logo de *Babel*

En este proyecto se va a utilizar la última versión de *Babel*, versión 7, como otra parte de la línea de producción del proyecto.

2.2.4 Generando estilos: Sass

Hasta este punto todo se ha enfocado siempre en torno a *JavaScript*. Sin embargo, anteriormente se enseñó como una página *Web* está formada por dos miembros adicionales: *HTML* y *CSS*. El primer caso sí está relacionado con *JS* directamente puesto que se genera dinámicamente desde él, y como tal su uso independiente es un tanto limitado. Aquellos límites con los que uno se tope en *HTML* son resueltos desde el otro lenguaje. Sin embargo, no es posible decir lo mismo para *CSS*.

CSS en sí mismo es lo suficientemente potente como para cumplir su función principal, que recae en la definición de las reglas de estilo de la *Web*. No obstante, su principal pecado resulta en ser una herramienta un tanto estática, es decir, las reglas en su código son escritas una única vez y durante la ejecución del proyecto no es posible su modificación. La única alternativa para proporcionar dinamismo a este apartado consiste en el intercambio de las reglas de estilo de forma condicional a través de *JavaScript*. Pero como tal, el código *CSS* siempre persiste estático.

Visto lo visto en el apartado 2.2.3, estaría fenomenal poder aplicar los conocimientos adquiridos ahí con tal de dotar al estilo de una *Web* de mayor dinamismo y responsividad. Pues bien, resulta que alguien pensó en esto mismo y dió con una solución: **Sass**.

Figura 2.5: Logo de *Sass*

Sass como concepto es muy similar a los transpiladores vistos antes, aunque este caso presenta alguna que otra diferencia y prefiere recibir el nombre de **pre-procesador**. Este añadido dota a *CSS* justamente de aquello en lo que peca, dinamismo. Así pues, las grandes razones para uso son las siguientes:

1. Permite definir variables reutilizables a lo largo del código.

2. Permite compartir esas variables con *JavaScript*, de tal forma que un lenguaje pueda alterar el comportamiento del otro.

3. Permite dividir el código en módulos, representados por un prefijo _ en su nombre, dotando de mayor contexto y división a las funciones.

4. Contiene funciones pre-instaladas para la reacción ante cambios en el navegador. Un ejemplo muy útil es la posibilidad de aplicar estilos en función del tamaño de la ventana del navegador. Esto último es vital para la transición entre una vista móvil y una de escritorio.

5. Añade mejoras en la sintaxis como la distribución de las etiquetas en forma de cascada.

6. Proporciona herencia entre los estilos, permitiendo que unos partan de la base de otros.

7. Permite realizar cálculos matemáticos a través de operadores.

Para una visualización más simple de esto, se presenta a continuación un ejemplo de como es uno frente al otro:

Código 2.13: Ejemplo código CSS

```
nav ul {
  margin: 0;
  padding: 0;
  list-style: none;
}

nav li {
  display: inline-block;
}

nav a {
  display: block;
  padding: 6px 12px;
  text-decoration: none;
}
```

Código 2.14: Ejemplo código SASS

```
// Variables
$color: #333

%example
  padding: 10px

// Cascada
nav
  ul
    margin: 0
    padding: 0
    list-style: none

  li
    display: inline-block
    color: $color

  a
    // Herencia
    @extend %example
    display: block
    padding: 6px 12px
    text-decoration: none
```

Yendo un poco más allá, en este proyecto no se utilizar un *Sass* puro, si no que se elige una rama de este lenguaje: **SCSS**. Este es prácticamente igual al anterior, solo que su intención es mantener una sintaxis más cercana a *CSS*.

Código 2.15: Ejemplo código SCSS

```
nav {
  ul {
    margin: 0;
    padding: 0;
    list-style: none;
  }

  li { display: inline-block; }

  a {
    display: block;
    padding: 6px 12px;
    text-decoration: none;
  }
}
```

Código 2.16: Ejemplo código SASS

```
nav
  ul
    margin: 0
    padding: 0
    list-style: none

  li
    display: inline-block

  a
    display: block
    padding: 6px 12px
    text-decoration: none
```

2.2.5 El framework: VueJS

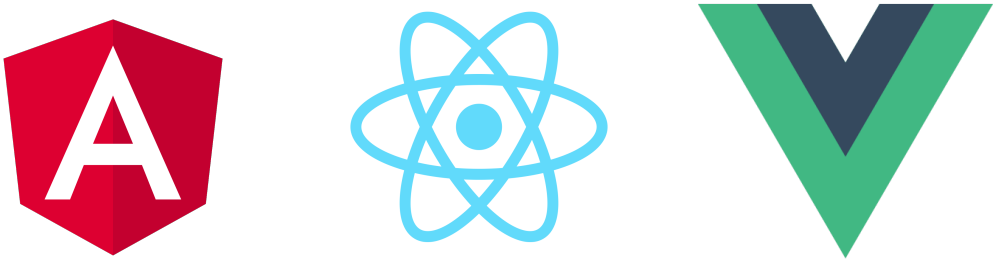
Un factor a tener en cuenta y quizás el de mayor importancia es el **framework**. Un *framework* es una librería masiva que se encarga de extender las capacidades de *JavaScript* llevándolo mucho más allá, construyendo y propiciando un entorno de desarrollo en donde el lenguaje cobra un nuevo significado. Gracias a esta librería, el desarrollador cuenta con una plataforma que se encarga de formar una estructura a partir de la cual puede olvidarse de la arquitectura del *Software* en sí y centrarse en el avance de la aplicación.

A modo de resumen, estas son las cosas de las cuales un *framework* se encarga:

- Definición de los límites y usos del lenguaje *JavaScript*. Por ejemplo, definición de los objetos base presentes en una aplicación y sus usos.
- Generación de código *HTML* dinámico.
- Gestión de la interfaz gráfica generada para el usuario: componentes ocultos, posición, interactividad, etc.
- Transmisión y reacción a eventos a lo largo de los componentes de la *Web*.
- Enrutamiento e intercambio dinámico entre *URLs*[8] y vistas.
- Almacenamiento de información de interés por medio de *Cookies*[9] y sus derivados.
- Gestión de un estado global para el intercambio de recursos a través de todos los miembros de la aplicación.
- Modularidad y aceptación de nuevas librerías.

[8]Dirección *Web* de la página.
[9]Pequeños paquetes de información almacenados en el navegador.

El *framework* facilita tanto el desarrollo de una aplicación que forma parte esencial de su ser. Ahora bien, en el mercado existen un gran número de opciones provenientes tanto desde grandes empresas como de proyectos *Open Source*[10]. Las opciones que aquí resultaron más resonadas son: **AngularJS** (Google), **ReactJS** (Facebook) y **VueJS** (Open Source); aunque hay muchas más que estas tres.



(a) AngularJS **(b)** ReactJS **(c)** VueJS

Figura 2.6: Frameworks de desarrollo *Web*

Realmente, desde el caso de alguien nuevo al menos, la elección de cual de ellos utilizar no es importante. Todos estos *frameworks* son capaces de soportar una página *Web* sea del tamaño que sea, lo significativo es utilizar uno. En el caso de este proyecto, la balanza se decantó por **VueJS** no más que debido a cuestiones de preferencia personal.

El proyecto hace pues uso de *VueJS* 2, la edición más moderna que contiene un gran número de cambios con respecto a su primera versión.

2.2.5.1 VueJS

VueJS se hace llamar a sí mismo un *framework* progresivo para el desarrollo de interfaces *Web*. Lo que esto quiere decir es que la plataforma cimienta su funcionamiento sobre la generación de código *HTML* a partir de información dinámica. De tal forma, la creación de una interfaz gráfica se basa en la creación de pautas con las que se guía a un generador para que produzca la página que se desea.

El nombre que se le da a estas pautas es el de **plantilla** o *template*. La forma que adoptan estas plantillas son generalmente muy parecidas al *HTML* estándar, pero a ello se le añade una gran cantidad de sintaxis nueva que le dotan al original de una mayor potencia. Un ejemplo de esto es el siguiente:

[10]Proyecto de código libre a los que cualquiera puede acceder. La propiedad del código se gestiona mediante el uso de licencias.

Código 2.17: Ejemplo de plantilla VueJS

```
<template>
  // Presenta el contenido de una variable
  <p> {{ var }} </p>

  // Relaciona el atributo de una etiqueta con un script
  <div v-bind:class="getTagClass()" />

  // Reacciona frente a eventos
  <input v-on:click="react()" />
</template>
```

Por lo general, estas plantillas se almacenan dentro de los "componentes de fichero único", o **single page component**. Ficheros de extensión **.vue** que aportan acceso a toda la especificación posible para la generación de un fragmento de código dinámico. Extendiendo el ejemplo anterior, este momento presenta ahora esta estructura:

Código 2.18: Ejemplo de componente VueJS

```
<template>
  // Presenta el contenido de una variable
  <p> {{ var }} </p>

  // Relaciona el atributo de una etiqueta con un script
  <div v-bind:class="getTagClass()" />

  // Reacciona frente a eventos
  <input v-on:click="react()" />
</template>

<script>
  export default {
    data: {
      var: 'Hello World'
    },
    methods: {
      getTagClass() {
        return 'example'
      },
      react: function (){
        // Do something
      }
    }
  }
</script>

<style>
  .example {
    background-color: red;
  }
</style>
```

Como se puede comprobar, un componente presenta los tres lenguajes vistos hasta ahora: *HTML*, *CSS* y *JavaScript*. Por lo que si se elimina todos los añadidos que introduce, *VueJS* puede ser utilizado como un entorno *Web* puro gracias a estos tres medios.

La parte más interesante de un componente recae como siempre en *JavaScript*. La forma en la que se usa este lenguaje es un tanto atípica. El único fragmento de código dinámico e interactuable con los otros dos lenguajes es aquel que está contenido dentro de la expresión: **export default**. A su vez, este fragmento está divido en una serie de zonas donde se permite definir un cierto aspecto del componente. Algunas de las zonas más interesantes son las siguientes:

1. **data**: Definición de variables.
2. **methods**: Definición de métodos.
3. **watch**: Definición de vigilantes reactivos a cambios en variables.
4. **computed**: Definición de variables auto-reactivas a cambios en sí mismas.
5. **components**: Declaración de componentes externos a utilizar en el actual.
6. **mixins**: Declaración de porciones de código externas adicionales para el componente actual.

Y de ellas, destacan las zonas: *components* y *mixins*. El primero es el fundamento de la modularidad en *VueJS*. A través de él, es posible importar otros ficheros *.vue* y utilizarlos como etiquetas adicionales del actual:

Código 2.19: Importando componentes en VueJS

```
<template>
  <example-component />
</template>

<script>
import Example from 'Example.vue'

export default {
  components: {
    'example-component': Example
  }
}
</script>
```

Mientras que el segundo permite definir y integrar porciones de código externas al actual. Muy útil para la reutilización de las funciones de llamada al servidor a lo largo del proyecto:

Código 2.20: Importando un mixin en VueJS

```
<template>
  {{ callServer() }}
</template>

<script>
const example = {
  methods: {
    callServer: function () {
      // Hi server
    }
  }
}

export default {
  mixins: [
    example
  ]
}
</script>
```

Otro gran interés de esta estructura peculiar, es la posibilidad del acceso al contexto de *Vue* desde él. Esto es, un modo en el que acceder a los mecanismos y herramientas que presenta la plataforma de antemano. Estas herramientas siempre se nombran con el prefijo $ y dan cabida a tareas como la emisión y recepción de eventos entre los componentes:

Código 2.21: Accediendo al contexto

```
<script>
  export default {
    methods: {
      sendEvent () {
        // El prefijo $ indica que esto pertenece al contexto
        this.$emit('event')
      }
    }
  }
</script>
```

Como último gran añadido, este contexto puede ser extendido a través de módulos que dotan al *framework* de una mayor funcionalidad a nivel global. Un nuevo módulo se añade mediante: *Vue.use(Modulo)*, está en la mano de cada uno definir cuales son las variables $ que añade. En el caso de este proyecto, los dos módulos más importantes que son añadidos son:

1. **Vue-Router**: Permite el intercambio de componentes en función de la *URL* actual, generando con ello una *Single Page Website*.

2. **Vuex**: Proporciona un estado global a la aplicación de estricta modificación. Útil para el almacenamiento de los credenciales del usuario donde debe ser accesible por todos, pero modificable por unos.

Y con esto se da un vista general de *VueJS*. A través de los módulos se dota de mayor potencia a la plataforma en sí. Mediante los *mixins* es posible reutilizar porciones de código a lo largo de los componentes. Y mediante la composición, utilizar unos componentes como etiquetas de otros, la página toma una estructura en forma de árbol en donde todo componente forma parte de otro estableciendo una relación padre-hijo.

Aún quedan muchos aspectos no explorados de este *framework*, como por ejemplo la comunicación de eventos entre componentes, pero serán tratados en el siguiente capítulo donde se pondrá en práctica todos estos conocimientos con tal de generar la aplicación.

2.2.6 Interconectando con el servidor: Axios

Como ya se ha mencionado anteriormente, un servicio *Web* está formado por dos partes: cliente y servidor. Donde por un lado el cliente dota la parte visual y el servidor la parte funcional a una aplicación. Como es de imaginar, un cliente no es de gran utilidad si no tiene un medio de comunicación con el servidor. El cliente no deja de ser más que una línea de comandos visual que exige contenidos al servidor. Por esto, es necesario establecer un túnel de comunicación entre estas dos partes. Es por ello por lo que se hace necesario el uso de una librería de comunicación como *Axios*.

La comunicación dentro de un servicio *Web* se produce con la ayuda del protocolo **HTTP**, mencionado en el apartado 2. Una ojeada muy rápida a este protocolo nos demuestra que sus paquetes de datos están formados por una intención, conocida como el verbo; una dirección *URL*, una serie de campos adicionales y por lo general un paquete de datos misceláneos.

```
PUT /create_page HTTP/1.1
Host: localhost:8000
Connection: keep-alive
Upgrade-Insecure-Requests:
Content-Type: text/html
Content-Length: 345

Body line 1
Body line 2
```

Figura 2.7: Ejemplo de mensaje HTTP

En este proyecto se hace uso del concepto de *REST API*, lo que significa que la respuesta del servidor va a estar condicionada por la *URL* que se invoque y el verbo usado en el mensaje. Un cuadro general de cuales son estos verbos es el siguiente:

GET	Obtén un recurso del servidor
POST	Crea un nuevo recurso en el servidor
PUT	Actualiza un recurso del servidor
DELETE	Elimina un recurso del servidor

Tabla 2.2: Listado de los verbos HTTP

Con esto, se puede resumir el funcionamiento del servidor como que la *URL* decide el recurso sobre el que se actúa mientras que el verbo decide la acción. Adicionalmente, tanto la *URL* como el verbo pueden ser alterados con parámetros adicionales que alteran la función que vaya a ser realizada finalmente.

Sabido esto, *Axios* es una librería de *JavaScript* externa a cualquier *framework* que proporciona una *API*[11] que permite gestionar las llamadas al servidor de una forma simple e intuitiva.

El uso de la interfaz es bien simple. Lo primero, se da acceso a varios generadores de paquetes *HTTP*, uno para cada verbo visto en la tabla anterior. Lo segundo, mediante los parámetros de la llamada se indica la *URL* destino y otra información que deba contener el paquete. Y lo tercero, se da una acción a realizar con la respuesta o posible error que pueda surgir. Así pues, recibir los datos de un usuario puede resultar similar a esto:

Código 2.22: Llamando al servidor con Axios

```
import axios from "axios"

axios.get('URL/Recursos', {
  // Parámetros adicionales de la llamada
  param_1 : 1,
  param_2 : 2
})
.then(
  function (response) {
    // Hacer algo con la respuesta
  }
)
.catch(
  function (error) {
    // Hacer algo con el error
  }
)
```

[11] Interfaz de programación.

El modo con el que se reacciona a una respuesta de *Axios* es lo que se conoce como asíncrono. Lo que esto significa es que el código de la aplicación no se queda bloqueado esperando a que el servidor responda, si no que prosigue con lo siguiente que toque y mantiene un ojo puesto para ver cuando hay que actuar. No obstante, hay un medio de mantener sincronismo con las llamadas de *Axios*. Desde la aparición de *ES6* se cuenta con la palabra **await**, cuyo significado es justamente el que se puede esperar: detente hasta que esta expresión termine. Un ejemplo de esto se vio anteriormente en el ejemplo 2.5.

Cabe decir que el proyecto que aquí se incuba tiene una naturaleza principalmente de consulta junto a unos breves esbozos de creación de contenidos. Por ello, el principal interés que se tiene de *Axios* se basa sobre sus funciones de **GET** y **POST**.

2.2.7 Otros componentes

Este capítulo tiene como foco e intención la introducción de los fragmentos básicos con los que se debe trabajar para poder poner en producción una aplicación *Web*. Como es de esperar, las secciones anteriores no abarcan la totalidad de los módulos funcionales con los que se cuenta en este entorno de desarrollo. Con tal de centrar los esfuerzos de este proyecto en los factores que realmente han intervenido en él, se descarta el análisis del elenco de herramientas adicionales posibles.

Pese a ello, en la siguiente lista se procede a hacer un esbozo de hasta donde se puede profundizar dentro de este círculo:

- Misceláneos:
  - **Lanzador de tareas (GruntJS)**: Permite la ejecución de comandos o pequeños *scripts* para el lanzamiento de operaciones en los demás módulos: procesar *CSS*, reducir *JS*, etc.
  - **Minimizador (UglifyJS)**: Reduce el tamaño de los ficheros de texto que forman el código fuente eliminando todos aquellos caracteres útiles para la compresión humana: espacios, comentarios, etc.
  - **Preprocesardor de *HTML* (PugJS)**: Concepto muy similar al visto anteriormente con *Sass*, pero aplicado sobre *HTML*.
  - **Generador de documentación (JSDoc)**: Aúna la documentación del código fuente y la presenta en forma de página *Web*.
  - **Corrector de estilo (ESLint)**: Vigila y corrige el estilo de programación presente en el código fuente para que siga unas pautas comunes a lo largo de él.

- Unit Testing:
  - **Entorno de pruebas (KarmaJS)**: Proporciona uno o más servidores configurables en donde ejecutar pruebas sobre el funcionamiento de la aplicación.
  - **Lanzador de Test (MochaJS)**: Reúne y controla la ejecución de los *tests*.
  - **Librería de aserciones (ChaiJS)**: Proporciona comprobaciones con las que decidir si un test es válido o incorrecto.
  - **Librería de generación de maquetas (SinonJS)**: Permite alterar el comportamiento de la aplicación tal que se encuentre en un estado deseable para el *test*.

2.2.8 El empaquetador: WebPack

A la hora de la verdad, pese a que contamos con toda esta cantidad de herramientas, un navegador *Web* solo es capaz de entender los pilares básicos: *HTML*, *CSS* y *JavaScript*; en su forma más pura. Todo el procesamiento que se dé a lo largo de la construcción del proyecto debe acabar tal que el resultado que se obtenga entre dentro de estos pilares. Es necesario pues controlar y secuenciar la ejecución de cada uno de los pasos de "compilación" con tal de dar un sentido a este proceso y evitar errores en su realización. Para esta tarea es para la que se utiliza el empaquetador, la última pieza del puzzle que es el desarrollo *Web*.

El empaquetador forma el corazón del proceso de construcción de una aplicación. Su objetivo principal es la de conseguir un código final tal que este pueda ser proporcionado por un servidor y, por lo tanto, puesto en producción. Para ello, el empaquetador proporciona una línea de ensamblado totalmente configurable donde se ejecutan los procesos vistos anteriormente siguiendo siempre una secuencia lógica y funcional. A necesidad de esto, el empaquetador debe ser lo suficientemente extendible con el propósito de permitir la comunicación con cualquier tipo de módulo con que se tope.

Como solución a este problema, en este proyecto se opta por el uso de **WebPack**, el empaquetador rey en estos momentos. *WebPack* no es quizás la mejor opción para todo tipo de casos puesto que es famoso por ser relativamente complicado de configurar y poner en marcha. Aún así, el factor de ser el más popular juega claramente a su favor puesto que esto significa que es aquel que cuenta con el mayor número de compatibilidades y ayudas adicionales.

Afortunadamente, dentro de *VueJS* se encuentra una de estas ayudas adicionales que va a facilitar en gran medida el uso del empaquetador: **Vue-CLI-Service**. Esta herramienta proporciona una capa que cubre buena parte de los entresijos de la configuración del ensamblado, e integra al *framework* de forma transparente dentro de este. Además, este instrumento da por entendido el uso de los componentes más habituales y, por ello, los "pre-instala" dentro de la configuración con tal de que solo sea necesario activarlos.

Figura 2.8: Logo de *WebPack*

Así con todo, este proyecto no ahonda sobre el funcionamiento interno de *WebPack* puesto que esto ha sido ocultado en buena parte por *VueJS*. Por otro lado, el resultado que surge del proceso de empaquetamiento sí es de interés. Al final de la línea lo que se obtiene es un fichero grande que contiene toda la lógica de nuestra aplicación. A este fichero se lo conoce como **paquete** o *bundle*, y en él se encuentra todo el código puro que un navegador necesita para hacer su trabajo.

Para cada uno de los lenguajes pilares se obtendrá entonces uno de estos paquetes, que pueden ser entregados por el servidor de una sola vez para que el cliente tenga una aplicación completa. Adicionalmente, también se añaden aquellos recursos que sean estáticos: como imágenes y fuentes, puesto que también forman parte visual de la *Web*.

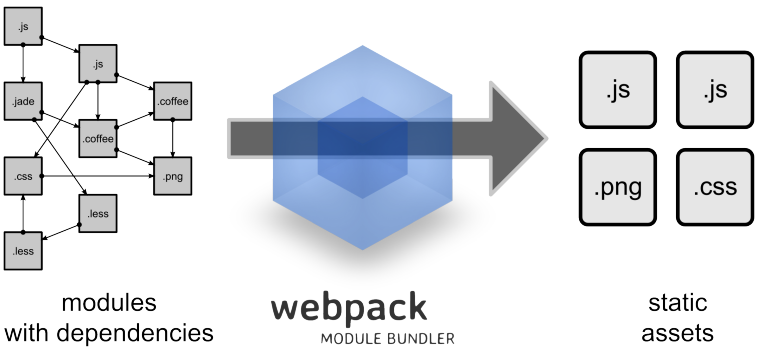


Figura 2.9: Uso de WebPack

3 El desarrollo del proyecto

Visto una vez todo el preámbulo requerido para entrar en el desarrollo *Web*, es el momento al fin de atacar al problema que aquí atañe: el desarrollo de la aplicación. En este capítulo se van a aplicar los conocimientos aprendidos en los apartados anteriores para crear y producir una página *Web* que pueda ser presentada a un cliente con orgullo.

El primer paso para la correcta realización de cualquier proyecto consiste en la definición de los pasos a seguir y los objetivos a cumplir a lo largo del trabajo. En el caso de este documento, en el apartado 1.1 ya se presentó una lista concisa de los apartados que se busca satisfacer. Por contraste, mientras que se ha dado un gran marco de contexto a la solución que se pretende llegar nunca se ha presentado como es ella en sí misma.

Bien pues, como una imagen vale más que mil palabras a continuación su enseña un boceto de la página *Web* de referencia:

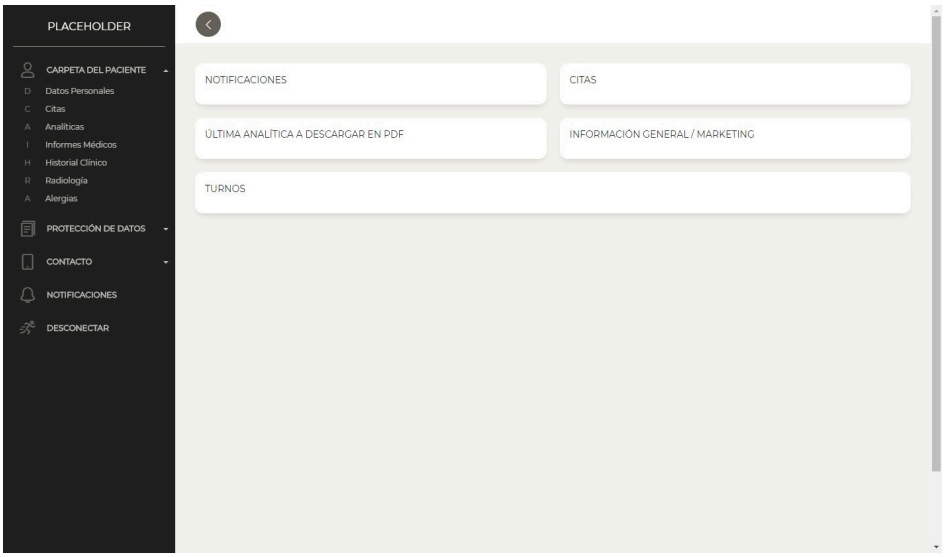


Figura 3.1: Boceto de la aplicación

La *Web* no es más que el típico panel de usuario en el que se presentan un montón de opciones para que el usuario pueda interacturar con el servicio. En este caso concreto estas opciones están pensadas para proporcionar al paciente de una determinada clínica médica acceso a una variedad de información de su perfil que previamente necesitaría de presencia física para su obtención.

Otra vez, en el apartado 1.1 se hizo hincapié sobre las opciones que la *Web* ha de asumir, que por lo general son bastante de esperar cuando se habla del entorno en el que se mueve.

Con respecto a esto último, hay que tener en cuenta que en el entorno de la medicina los datos que se manejan dentro de él son de una vital importancia. Los datos de carácter médico se pueden usar para hacer daño, por lo que otro punto de inquietud a tener presente durante el desarrollo de la aplicación recae sobre la protección y el cumplimiento de la normativa relacionada con este campo.

Por todo ello, nadie excepto el doctor y el paciente debe tener la oportunidad de ver estos datos por medio de la aplicación, ni siquiera durante la fase de desarrollo y pruebas. Con tal de garantizar esta opacidad se comienza presentando dos primeras medidas de seguridad:

1. Hasta su finalización y correcta puesta en marcha, la aplicación solo puede acceder a una base de datos de prueba similar a la real pero rellena con información falsa.

2. La aplicación solo es accesible a través de un usuario administrador definido y controlado por la clínica.

A estos dos puntos se les sumará otros pendientes aún por mencionar que garantizarán en su conjunto la seguridad de los datos aquí tratados.

3.1 Esbozando la aplicación

Continuando con la planificación del proyecto, no es de gran utilidad comenzar a trabajar sin tener una buena dirección sobre los pasos a seguir. La labor ahora radica en reducir las tareas a su mínima expresión de forma que se dé una visión concreta de como van a ser implementadas. Por esta razón, a continuación se presenta una abstracción sobre lo que se aspira a crear:

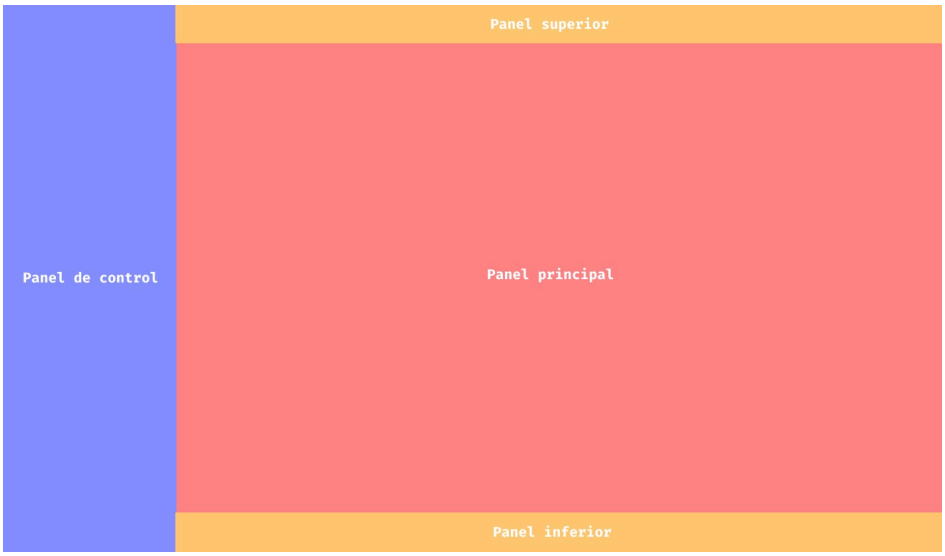


Figura 3.2: Boceto de la aplicación

La vista está formada por cuatro paneles principales:

1. **Panel principal**: Presenta la vista asociada a la herramienta seleccionada.

2. **Panel de control**: Menú desplegable que proporciona acceso a las distintas herramientas de la aplicación. El panel puede ser escondido con tal de ahorrar espacio en pantalla.

3. **Panel superior**: Se dibuja siempre por encima del panel principal. Presenta opciones de importancia que siempre deben estar visibles. En el momento inicial del proyecto, este panel solo contiene un botón para el despliegue del panel de control.

4. **Panel inferior**: Similar al superior pero en este caso no se dibuja por encima del principal y contiene opciones de menor importancia. Aquí se pueden ubicar enlaces para acceso a *Webs* adicionales.

Si se comprueba este mismo boceto desde un punto de vista más cercano a la implementación, el resultado acaba en un árbol de componentes como el siguiente:

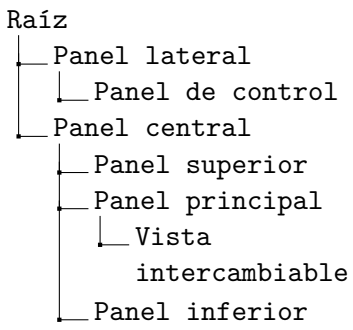


Figura 3.3: Árbol de componentes de la aplicación

Afortunadamente, esta estructura en árbol es el tipo de organización en donde *VueJS* es más fuerte. Tal como se vio en el apartado 2.2.5.1, mediante el uso de componentes anidados es posible generar una relación entre los miembros de la *Web* para que sigan un modelo muy similar al planteado. Junto a ello, el *framework* también cuenta con un medio por el cual emular la vista intercambiable en donde se va a presentar la herramienta.

Las siguientes secciones de este capítulo están dedicadas entonces al análisis y descripción de todos estos componentes que forman la página *Web*. La finalidad objetivo es pues terminar este documento con una idea clara tanto sobre el uso de cada componente como de la implementación que se le ha dado.

3.2 La raíz

Ninguna casa se empieza a construir primero por su tejado, y una página *Web* no es un caso excepcional. Para poder levantar un sistema competente primero en necesario contar con una buena base sobre la que trabajar. En este caso, esta base corresponde con un *framework* propiamente instalado y configurado.

VueJS es ante todo una librería monolítica cuya definición pertenece por completa al mundo de *JavaScript*. A partir de este lenguaje, el *framework* es capaz de leer la página y reaccionar ante eventos con tal de generar un código *HTML* correspondiente a un estado de la *Web* puntual. Teóricamente esto es sencillo de entender, pero a la hora de ponerlo en práctica resulta un tanto confuso. *HTML* es por su propia naturaleza un lenguaje estático, ¿cómo es posible que *Vue* se relacione con él para aportar dinamismo?.

Toda página *Web* en *VueJS* parte de dos elementos: una página *HTML* y un *script JavaScript*; siendo el nombre más habitual para estos dos ficheros el de **index.html** y **main.js** respectivamente. Tal como ocurre en un navegador cualquiera, todo comienza por el primero de los dos elementos. En él, se deben incluir dos partes fundamentales: en primer lugar, se debe importar y ejecutar el *script* de instalación; en segundo, se debe indicar una etiqueta sobre la que colgará todo el código generado por *Vue*. Así pues, un ejemplo de *HTML* básico resulta parecido al siguiente:

Código 3.1: Punto de entrada HTML en VueJS

```
<html>
  <head>
    <!-- Ejecución de script de instalación -->
    <script src="main.js" />
  </head>
  <body>
    <!-- Punto de anclaje -->
    <div id="app"/>
  </body>
</html>
```

Por otro lado, en el proceso de instalación se debe como mínimo crear una instancia de un proceso *VueJS* y relacionarlo con el *HTML* visto anteriormente. Con esto, un *script* básico de instalación puede presentarse como el siguiente:

Código 3.2: Instalación básica de VueJS

```
import Vue from 'vue'

// Componente raíz de la aplicación
import App from '@/App.vue'

new Vue({
  el: '#app', // ID de la etiqueta de anclaje
  render: h => h(App) // La Web parte de App como punto inicial del árbol
})
```

Volviendo al apartado 2.2.5.1, en él se mencionó como hay un par de módulos que son de interés para esta aplicación: **Vue-Router** y **Vuex**. Si se integra la inclusión de estos dos módulos en la instalación, el *script* final queda como el siguiente:

Código 3.3: Instalación completa de VueJS

```
import Vue from 'vue'
import Vuex from 'vuex'
import VueRouter from 'vue-router'

import App from '@/App.vue'

// Configuración de los módulos
import { store } from '@/store/store.instance.js'
import { router } from '@/router/router.instance.js'

Vue.use(Vuex)
Vue.use(VueRouter)

new Vue({
  el: '#app',
  store: store,
    router: router,
  render: h => h(App)
})
```

Y finalmente, el componente raíz se presenta tal como:

Código 3.4: Componente raíz de la aplicación

```
<template>
    <router-view />
</template>
```

Realmente, este componente resulta un tanto decepcionante puesto que después de todo los pasos que se han dado está un tanto vacío y seco. Sin embargo, la magia de esto resulta sobre la vista enrutada que se muestra en la etiqueta central. Esta función proviene del módulo *Vue-Router* instalado previamente y es a través de ella de donde nace el árbol que surge de la raíz. En la sección posterior se va a profundizar con mayor ahínco en el quehacer de esta funcionalidad.

Para terminar, es posible ir completando el árbol de componentes visto al principio con aquellos que se van integrando:

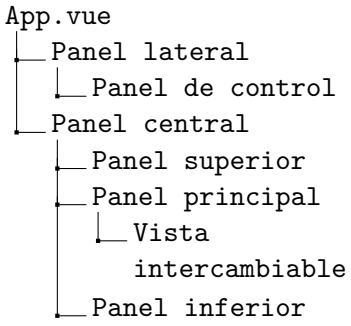


Figura 3.4: Árbol de componentes de la aplicación: Raíz

3.3 Enrutamiento de vistas con Vue-Router

Volviéndolo a introducir una vez más, *Vue-Router* es un módulo de *VueJS* que permite al *framework* la creación de **Single Page Websites**. Con tal de poder entender en qué mejora esto con respecto al uso habitual de la plataforma, primero es necesario comprender el funcionamiento estándar de una servicio *Web*.

A nivel técnico, una página *Web* no es más que un recurso como otro cualquiera que se encuentra almacenado en un servidor y al que se puede acceder a través de una dirección *URL*. Para toda vista que se presente en una aplicación, el cliente debe conocer la ubicación del recurso y generar una petición al servidor con tal de que este se lo entregue.

La gran debilidad con la que este mecanismo cuenta atiende al tiempo entre petición y respuesta que el cliente debe esperar. Dependiendo de la conexión del usuario y la carga con la que cuente el servidor, la progresión natural de la página *Web* se puede ver afectada por estos momentos en los que se rompe la inmersión del usuario.

Con tal de solucionar esto, una *Single Page Website* trata de reducir al máximo el número de ocasiones que se contacta con el servidor pidiéndole a este la totalidad o gran mayoría de las vistas, de tal manera que recae sobre el propio cliente decidir en que momento se muestra cada una.

A nivel de usuario, la existencia de esta tecnología es completamente transparente puesto que es la *URL* la que sigue decidiendo el contenido que se muestra en cada momento dentro la aplicación. En este caso, la diferencia resulta en que la *URL* no conlleva una petición al servidor si no que produce lo que se llama un enrutamiento de la vista actual.

El encargado de controlar todo este proceso es el enrutador, un miembro del contexto *VueJS* que a la hora de la verdad no va mucho más allá de ser un simple mapa que relaciona *URLs* con componentes del *framework*.

Con tal de ver un caso práctico de todo esto, a continuación se muestra un esbozo de como es el enrutador con el que la aplicación cuenta:

Código 3.5: Ejemplo de enrutador en VueJS

```
export const routes = [{
  path: '/',
  component: DashBoard,
  children: [{
    path: '/home',
    component: Home
  }, {
    path: '/profile',
    component: Profile
  }
}

const instance = new VueRouter({
    routes: routes
})
```

Observando un poco el ejemplo, es posible ver como la dirección principal de la *Web*: "/" indica al enrutador que en ese caso debe mostrar el componente **DashBoard**. El efecto que esto conlleva es que la etiqueta *router-view* que se indicó en la raíz anteriormente va a ser ahora sustituida por ese componente en concreto. A su vez, esa ruta tiene una serie de sub-rutas definidas dentro de ella, esto es debido a que un componente enrutado puede tener a su vez otros componentes de este tipo. La *URL* que apunta a una de estas sub-rutas indica que la *Web* tiene que mostrar el componente padre y dentro de él el hijo que corresponda.

Esta última función va a resultar de especial utilidad pues a través de ella se permite el intercambio entre las herramientas que se van a presentar al usuario. Por ende, a través de la ruta se va a decidir aquello que se muestra en el **panel principal** del boceto. Dando un paso adicional, también es posible visualizar el miembro que va a producir el cambio de rutas: **el panel de control**. Este componente almacena un conjunto de botones que en su acción cambia la *URL* del servicio tal que se muestre la vista correspondiente para la selección.

A nivel interno, un ejemplo de como estos botones producen un cambio en el enrutador es tal y como se muestra a continuación:

Código 3.6: Ejemplo de enrutamiento

```
this.$router.push(URL)
```

En donde simplemente se ejecuta una función de un miembro del contexto de *VueJS*, tal como se aprecia por el carácter: **$**, que viene proporcionado por el módulo instalado anteriormente.

Para terminar, es posible seguir rellenando los huecos del boceto con nuevos contenidos aquí explicados. En concreto, ya se cuenta con la función e implementación del panel principal, así como de la barra lateral. En el siguiente esquema se introducen los nombres de los componentes que se ha proporcionado a estas partes, así como una ligera reestrucura que lo asemeja más a la realidad.

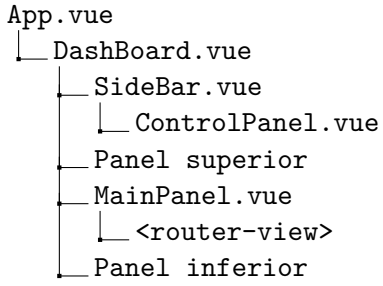


Figura 3.5: Árbol de componentes de la aplicación: Enrutador

3.4 Terminando el boceto: Los últimos paneles

Inspeccionando el árbol de componentes del boceto, a estas alturas ya es posible hacerse una idea concreta del funcionamiento interno de la *Web*. A modo de recolección, *VueJS* se engancha al código *HTML* y mediante su módulo de enrutamiento altera el significado de las *URLs* de tal modo que por ellas se decide el contenido mostrado al usuario. Realmente lo último que falta por introducir pues son los dos paneles decorativos que rodean al principal y las propias herramientas en sí mismo.

Comenzando por el primer caso, estos dos paneles se añaden a la interfaz como meros espacios en blanco cuyo propósito real es la reserva de un hueco en pantalla en donde se pueda poner información y derivados que afecten a la *Web* en su conjunto. Estos dos espacios se encuentran en los dos extremos verticales de la pantalla: en lo alto, mayor prioridad; y en lo bajo, menor prioridad.

El único punto a resaltar en estos dos componentes es el uso de *CSS* para la presentación del panel superior como un componente fijo que siempre es dibujado por encima del resto. Para ello, simplemente se ha hecho uso del concepto de pila de dibujado tal que este componente en concreto se encuentre en una capa superior al resto de sus compañeros.

A nivel de código, esto último se expresa sencillamente mediante la siguiente línea:

Código 3.7: Alterando la pila de dibujado

```
.on-top {
  z-index: 1;
}
```

En donde la capa por defecto en la que se encuentra el resto de los estilos es la número 0.

Así pues, es posible terminar el boceto añadiendo los dos últimos miembros al árbol de componentes:

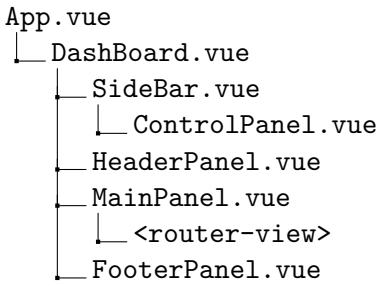


Figura 3.6: Árbol de componentes de la aplicación: Vista principal

Y a parte de esto, es posible mencionar también las posibles vistas / herramientas que van a poder ser presentadas en el panel principal de la aplicación:

1. **Allergies.vue**: Presenta un informe con las alergias que padece el paciente.

2. **Analytics.vue**: Presenta una lista en donde el paciente puede descargarse sus analíticas en el formato que desee.

3. **Appointments.vue**: Permite la consulta y petición de citas con el centro médico.

4. **Centers.vue**: Muestra un mapa con los centros con los que cuenta la clínica.

5. **Consents.vue**: Permite la firma electrónica de documentos de consentimiento.

6. **Contact.vue**: Presenta diversos medios con los que contactar con la clínica: E-Mail, Tlf, etc.

7. **History.vue**: Muestra el historial médico del paciente.

8. **Home.vue**: Punto de entrada de la aplicación, da acceso a las otras herramientas.

9. **Noficitacions.vue**: Avisos de interés para el usuario.

10. **Profile.vue**: Presenta los datos de perfil del usuario registrado.

11. **Radiology.vue**: Similar a las analíticas, pero para la descarga de radiografías y demás imágenes.

12. **Reports.vue**: Proporciona acceso a informes médicos del paciente.

3.5 Dotando de vida a la aplicación: Las herramientas

Toda la planificación y el montaje que se ha acumulado hasta el momento no tiene sentido alguno si la aplicación no tiene función alguna. En este caso, la funcionalidad de la *Web* viene dada por las vistas que se han listado en el apartado anterior. En su gran mayoría, todas ellas comparten un sentido común a la hora de actuar, y esto es que están fundamentadas en la consulta de datos.

A modo general, la aplicación está dispuesta para que cada una de estas herramientas cuente con un espacio personalizado en el que presente tanta información como crea conveniente. Siguiente el boceto de la aplicación, un ejemplo cualquiera de estas vistas puede ser como el siguiente:

Figura 3.7: Boceto de una herramienta

Pero realmente como se exhibe la información es la parte más trivial del proceso. Lo que aquí verdaderamente importa es conocer tanto cómo va la conversación con el servidor para conseguir esta información, como el proceso de tratamiento de la misma.

Antes de nada, es necesario volver y dar un repaso al apartado 2.2.6 pues es ello el cimiento en el que se basa todo lo que se va a revelar a continuación. Teniendo esto en mente, el primer paso a dar gira una vez más en torno al enrutamiento que se ha ido desgranando a lo largo de los apartados anteriores. Una de las funciones no mencionadas del enrutador hasta este punto consiste en su capacidad de anunciar cambios en él a través del sistema de eventos. Específicamente, esto se resume en que el enrutador tiene la habilidad de informar al componente al que este va a saltar de que el salto se va a producir y que debe actuar en consecuencia a ello.

A nivel de aplicación, este evento es utilizado por las herramientas como un aviso de que deben estar preparadas pues su momento llega. Debido a motivos de seguridad, el cliente está indicado para que almacene la menor cantidad de información del cliente como sea posible. Por ende, la información mostrada en pantalla solo existe en la *Web* durante el tiempo en el que el usuario permanece en esa pantalla. Todo esto conlleva la necesidad de que cada vez que una herramienta es enrutada, esta necesita contactar con el servidor para reobtener la información y presentarla al usuario.

A modo de ejemplo y para tener algo más concreto de este proceso, en la siguiente figura se muestra un fragmento de código que corresponde con la inicialización de una herramienta:

Código 3.8: Reaccionando a cambios en la ruta

```
import axios from 'axios'

export default {
  async beforeRouteEnter () { // 1
    await axios.get(URL, { // 2
      'Authorization': .... // 3
    })
    .then (function (response) { // 4
      processData(response.data)
    })
    .catch(function (error) { // 5
      processError(response.error)
    })
  }
}
```

Estudiando esto desde arriba hasta abajo, es posible sacar los siguientes detalles:

1. **beforeRouteEnter**: Es el método predefinido al que el enrutador llamará antes de producir el cambio de vista. Dicho de otra forma, es en este lugar en donde se produce la respuesta al evento.

2. **axios.get**: Se hace uso de la librería *Axios* para la comunicación con el servidor. En la gran mayoría de los casos, el verbo a utilizar será el **GET**.

3. **Authorization**: Cada petición debe contener en su interior el **token** de autenticación. Esto es un código que le permite al servidor saber que el usuario es quién dice ser. Mas sobre esto se verá posteriormente en el apartado sobre el acceso a la plataforma.

4. **then(function (response))**: Evento similar al visto en el enrutamiento al que se llama cuando la respuesta del servidor está lista. La información de respuesta viene almacenada en el campo **data**.

5. **catch (function (error))**: Similar al anterior pero en este caso es llamado cuando el servidor ha fallado en responder la petición.

Como es inevitable, entre la petición y la respuesta pasa un cierto tiempo en el que el usuario debe esperar. Sin embargo, el enrutador no está dispuesto ha hacer esta espera y por defecto la vista es presentada antes de que contenga la información correcta. Con tal de evitar este caso, todas las herramientas han sido protegidas por un sistema que envuelven a la vista en una pantalla de carga que será reemplazada una vez el componente lo indique. Así mismo, existe una pantalla adicional de error que hace la misma función que la anterior pero para el caso en que no haya habido repuesta.

Una vez la petición haya sido respondida con éxito, es el momento de desempaquetar los datos e interpretar los contenidos. Como norma de diseño, se establece que toda la información transmitida entre cliente y servidor, tanto la que va en una dirección como en la otra, debe seguir el formato **JSON**. Esto es un formato de texto estándar que permite una representación inequívoca de las estructuras de datos presentes en el lenguaje *JavaScript*. Un ejemplo de este modelo es tal como el siguiente:

Código 3.9: Ejemplo de objeto en JS

```
var object = {
  item1: 'value1',
  item2: 1000,
  item3: ['a', 'b', 'c'],
  item4: [1, 2, 3],
  item5: {
    foo: 'bar'
  }
};
```

Código 3.10: Ejemplo en JSON

```
{
   "item1": "value1",
   "item2": 1000,
   "item3": [
      "a",
      "b",
      "c"
   ],
   "item4": [
      1,
      2,
      3
   ],
   "item5": {
      "foo":"bar"
   }
}
```

Y para el cerrar el círculo, a esto último se le acompaña del **JSON Schema**. Un formato de texto adicional que permite definir aquello que se espera de un fichero *JSON*, comparando y proporcionando una validez en caso de que el esquema sea cumplido. Un cara a cara es posible de ver en el siguiente cuadro:

Código 3.11: Ejemplo de JSON

```
{
   "item1": "value1",
   "item2": 1000,
   "item3": [
      "a",
      "b",
      "c"
   ],
   "item4": [
      1,
      2,
      3
   ],
   "item5": {
      "foo":"bar"
   }
}
```

Código 3.12: JSON Schema equivalente

```
{
  "$schema": "http://↩
      ↪ json-schema.org/↩
      ↪ draft-07/schema#",
  "type": "object",
  "properties": {
    "item1": {
      "type": "string"
    },
    "item2": {
      "type": "integer",
      "examples": [
        1000
      ]
    }
  }
}
```

A modo de recolección, en cada enrutamiento que se produzca la nueva vista presentada ha de conectarse con el servidor para conseguir la información que debe mostrar. Mientras esto se produce, una ventana de carga hace esperar al usuario mientras el servidor trabaja. La información contenida en la respuesta viene codificada en el estándar *JSON* que es enfrentado a un esquema para comprobar que lo recibido es igual a lo que se espera. A partir de entonces, la pantalla de carga es eliminada y la vista muestra los valores correspondientes.

3.6 Accediendo a la aplicación: El login

Una vez recorridos todos los apartados anteriores es actualmente posible darse una idea de como son los entresijos de esta página. Queda sin embargo todavía un último punto que completa todo este conjunto y que todavía no se ha detallado: el acceso. La aplicación es desde un principio un servicio cuya orientación se basa en la personalización de contenidos para el usuario y la restricción de los datos que cada uno puede ver. Se hace imperioso entonces contar con un medio mediante el cual un usuario pueda identificarse como quién es para que la aplicación pueda adaptarse al pequeño ambiente que le concierne.

Bien pues, para esta necesidad es inevitable contar con una nueva vista, lo más separada de la *Web* posible, en donde el usuario pueda introducir sus credenciales. Afortunadamente, el diseño de la página está ya de antemano pensado para la aceptación de un caso así. Volviendo al apartado 3.2, la raíz de la aplicación cuenta con un *router-view* maestro que permite el cambio completo de la vista que se está presentando. Si se añade un nuevo camino a esta ruta, es posible reescribir el árbol de componentes tal que se presente como algo así:

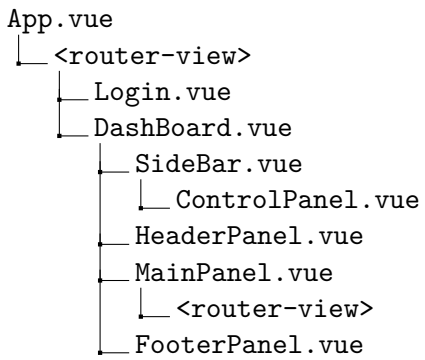


Figura 3.8: Árbol de componentes de la aplicación: Login

Siendo esta nueva vista de *login* algo como se puede ver en la siguiente imagen:

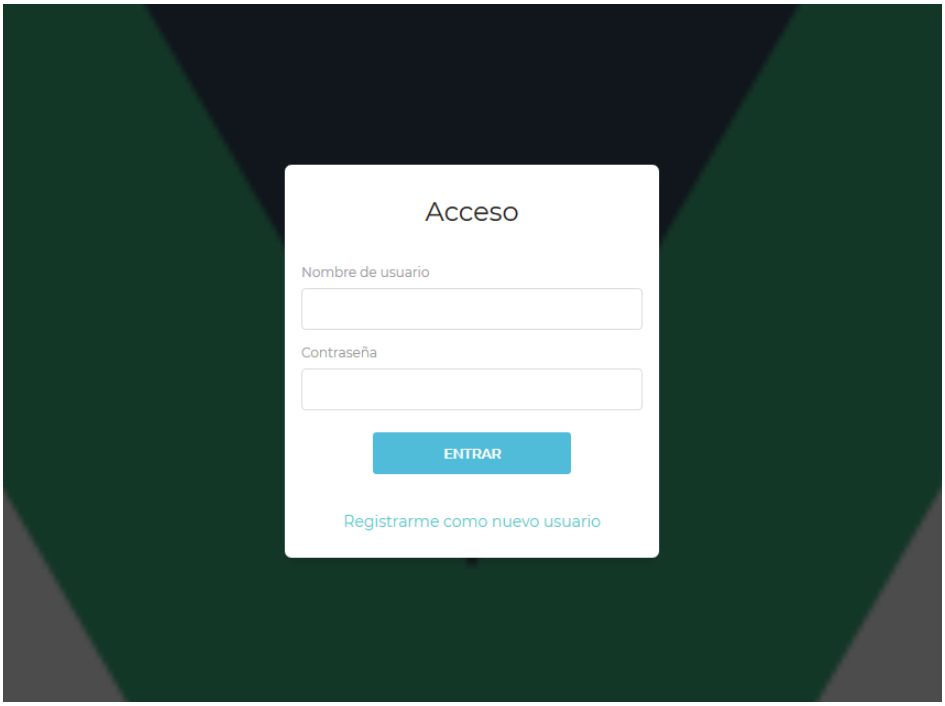


Figura 3.9: Boceto de la página de acceso

A modo fundamental, el proyecto fue escrito para que la identificación del usuario sea realizada mediante un simple nombre de usuario y contraseña. Sin embargo, métodos adicionales de seguridad que no entran dentro de esta fase del proyecto están pensados para un futuro. Ejemplos de ellos pueden ser: identificación a dos pasos, código *pin* desde la clínica, etc.

A nivel interno, el botón de "Entrar" genera una llamada al servidor en donde se le pide a este la confirmación de los datos del usuario. A diferencia del resto de llamadas, este es uno de esos casos en los que se hace uso del verbo *POST*, puesto que esta está encaminada a producir una acción en el servidor. Si los datos enviados son erróneos, un simple marco con un mensaje de error es presentado al usuario para que este lo vuelva a intentar. Si por el contrario los datos son correctos, entonces entra en juego el último módulo de *VueJS* que queda por utilizar: **Vuex**.

Vuex fue inicialmente introducido junto con *Vue-Router* en el apartado 2.2.5.1. En él, se mencionaba como este módulo proporciona un estado global a la aplicación en donde es factible almacenar información de carácter general para la aplicación. El gran uso que se le da aquí es justamente la conservación de la confirmación que un servidor proporciona frente a un acceso válido. Este uso se debe a que los datos contenidos en *Vuex* cuenta con dos propiedades de interés:

1. Los datos son accesibles por medio del contexto de *VueJS* desde cualquier componente de la aplicación.

2. La modificación de los datos es rigurosa, y por lo tanto presenta cierta resistencia a que alguien indeseado la cambie.

Para mostrar esto a nivel de código, el espacio de almacenamiento de *VueJS* es organizado en lo que se conoce como un **store**, el cual es modificado únicamente a través de las **mutaciones**. Si un dato no se puede modificar a través de una de estas últimas, entonces es considerado como de solo lectura. En el siguiente ejemplo es posible ver un caso que aplica esto anterior:

Código 3.13: Ejemplo de store en Vuex

```
export const store = {
  state: () => {
    return {
      loggedIn: false,
      /**
       * User credentials
       */
      user: {
        id: undefined,
        token: undefined
      }
    }
  },
  mutations: {
    login (state, user) {
      ...
    }
  }
}
```

Donde el campo **state** guarda la información cruda y la mutación la modifica a través de la referencia que *Vuex* le proporciona. La gran ventaja de esto es pues que por medio de este método es posible exigir una identificación y limitar el número de componentes que pueden modificar el estado. En concreto, en el caso del acceso de la aplicación solo el propio *login* debe ser capaz de alterar estos datos mientras que los demás deben limitarse a leerlos.

Para dotar de un poco más de visión, la forma en la que un componente accede al estado es tal como se muestra en el siguiente recuadro:

Código 3.14: Ejemplo de acceso al estado global

```
this.$store.state[Campo]
```

Finalmente, únicamente queda por definir la información que va a ser almacenada en este estado. Volviendo a la respuesta del servidor, esta contiene exclusivamente dos campos: **id** y **token**. El primero es un identificador unívoco que el servidor proporciona para que el cliente pueda referirse al usuario a partir de este punto. El segundo es un código que permite al servidor entender que los mensajes de petición que recibe están relacionados con el usuario al que validó. De esta forma, descifrar el identificador de la persona no es suficiente puesto que es necesario el código adicional para que el servidor se crea las peticiones que está recibiendo.

Si se vuelve atrás, más en concreto al apartado 3.5, es el *token* aquí mencionado el que se va a incluir como un campo de cada mensaje de *Axios* con tal de dotar a estos mensajes de autenticidad a nivel de aplicación.

3.7 Levantando el servicio: La puesta en marcha

Llegado este punto, la aplicación ya se encuentra en un estado en el que cuenta con una forma definida y unas funciones claras. El servicio está en una situación en la que ya tiene algo que ofrecer al mundo. El tiempo de desarrollo ha terminado y con ello la hora de producción ha llegado. El problema es, ¿Cómo se pone esto en marcha?.

La puesta en producción de un servicio no va más allá de la simple instalación de este dentro de un servidor. El servicio se presenta como un recurso más del servidor al que cualquier cliente tiene en un principio la potestad de exigir. Como ya se ha visto anteriormente, todo recurso presente en un servidor viene definido por una dirección con la cual referirse al mismo. Por esto, el primer paso para el despliegue de la aplicación conlleva el elegir una *URL* específica para la misma.

En este caso, un ejemplo de *URL* puede ser como el siguiente:

Código 3.15: Ejemplo de URL para el servicio

```
1 www.upct.medical-service.com/#/home
```

El símbolo # aquí presente desvela el hecho de que esto es una *Single Page Website* debido a que este formato de dirección suele ser particular de los enrutadores como *Vue-Router*. Hay mecanismos mediante los cuales es posible ocultar este último hecho mediante otro tipo de *URLs*, pero realmente no hay ninguna desventaja en que esto se sepa.

Una vez el recurso está dispuesto en el servidor, es tentador coger el código fuente con el que se ha estado trabajando en la fase de desarrollo y sencillamente ponerlo para ser servido. Sin embargo, esto es un gran error. Por un lado, el código usado para depurar suele contener grandes cantidades de datos adicionales que se utilizan para la búsqueda de errores en el código, esto da por un lado más información a un atacante de como está formada la página y por otro aumenta el tamaño de la misma. Junto a esto, el desarrollo puede tener una serie de condicionantes en los compiladores por el cual el código obtenido puede no estar siquiera en una estructura que permita su fácil entrega a un cliente. Por todo esto, un compilado especial para la puesta en producción es necesario. Afortunadamente, el empaquetador ya tiene esto presente y por ello tiene un modo de funcionamiento especial para estos casos.

El comando para producir el código que ha de ser instalado en el servidor es el siguiente:

Código 3.16: Comando para el compilado de la aplicación

```
1 yarn run build
```

Finalmente, también queda la cuestión de elegir un servidor sobre el que hacer funcionar esto. En la gran mayoría de los casos se suele utilizar el conocido *Apache* corriendo bajo una distribución de *Linux*. Pese a ello, con tal de no crear trabajo adicional en la administración del sistema se optó porque la aplicación vaya bajo un servidor **Laragon** en el sistema operativo **Windows Server**.

3.8 Futuros avances: Lo que aún queda en el tíntero

En este documento se ha descrito lo que se podría considerar como la primero fase de desarrollo de este proyecto. Aquí, se han definido las tecnologías que perfilan el trabajo, la manera en la que estas trabajan las unas con las otras y el uso que se ha dado de ellas para conseguir lo que se puede considerar un prototipo de la aplicación. Como es de esperar, mucho más trabajo y refinamiento queda aún pendiente para que esto pueda ser visto algún día por un paciente. A continuación, se va a indagar en partes del proyecto que están propuestos y pensados pero que no han tenido cabida por el momento a falta de recursos:

1. **Reactividad**: Muy al principio se destacó como las tecnologías *Web* fueron elegidas sobre las móviles debido a que las primeras ofrecen un mayor grado de accesibilidad. Esto es cierto y se mantiene, pero no es algo que ocurra por arte de magia. Permitir que una *Web* sea accedida desde un dispositivo móvil conlleva la implementación de una vista específica adaptada para ello. La reactividad se refiere al hecho de que la página sea capaz de reaccionar al medio en el que se encuentre y optimizar su visualización.

2. **La interfaz**: Los bocetos que se han presentado no son más que un mero utensilio que el desarrollador tiene para ir poniendo material sobre la pantalla. El diseño y definición de como debe ser realmente la interfaz es un trabajo habitualmente ignorado que también requiere de atención y tiempo.

3. **El registro**: El mínimo necesario para entrar en la aplicación es contar con una vista que permita dicho acto. Esto último ya se ha implementado anteriormente, aunque no es de gran utilidad si no se cuenta con una forma de añadir identidades al sistema. Una vista de registro está todavía pendiente de ser instalada tal que un nuevo usuario pueda definirse y formar parte del sistema.

4. **Notificaciones**: Tanto a nivel de escritorio como de móvil, los navegadores modernos tienen la posibilidad de enviar notificaciones al sistema operativo del usuario con la intención de indicarle cosas que van ocurriendo en la *Web*. Esto se presenta de gran utilidad para informar al usuario de que tareas como una analítica se encuentran ya disponibles para su consulta.

Y esto es únicamente aquello de lo que se podría considerar como la siguiente fase de trabajo, cargos posteriores como el mantenimiento o el añadido de más funcionalidades queda en estos momentos aún por definir.

4 Conclusiones

Una de las intenciones que ha tenido este documento es hacer ver como la entrada en el mundo del *Front-End*, especialmente para alguien novato, es terriblemente caótica y complicada. Si se comparan los requerimientos mínimos para la puesta en marcha de un proyecto viable en *Java* y luego en *JavaScript*, se nota como la diferencia va de un simple compilador con algún gestor de tareas a una pila de compilación formada por un transpilador, un pre-procesador, un empaquetador, etc.

Ya se hizo incapié en esto anteriormente, el gran desarrollo de la tecnologías *Web* pillaron completamente desprevenido a los medios con los que se contaba en el momento, y no ha habido más remedio que ir parcheando más y más con tal de tapar fallos que son tan profundos que algunos se pueden considerar fundamentales. No hace falta siquiera conocer la historia de estas tecnologías para darte cuenta de que hay algo que no funciona del todo bien en ellas.

Dando una ligera vuelta a lo largo de *Internet*, las peculiaridades de *JavaScript* como su comportamiento inesperado en cálculos aritméticos, su falta de librería estándar o la complejidad de sus medios de ensamblado han dado lugar a que este frente sea tomado como un motivo de burla dentro de la comunidad de desarrolladores. El *Front-End* es habitualmente visto como un campo turbio y complejo al que pocos quieren acercarse.

Hay cierta verdad en las burlas que se hacen sobre el *Front-End*, pero bien es cierto que si a día de hoy está tecnología es tan utilizada es porque simplemente no tiene rival alguno. *Internet* es el gran titán de las telecomunicaciones modernas, y desde sus inicios no ha existido una mejor manera de acceder a él que a través de un navegador. Proporcionar un servicio desde un punto remoto evitando instalaciones, actualizaciones, errores, etc; es simplemente tan cómodo tanto para el usuario como para el desarrollador que incluso con la existencia de alternativas la *Web* sigue siendo el rey.

El *Front-End* está aquí para quedarse, los últimos tiempos han visto grandes avances que intentan introducir cada vez más y más las tecnologías *Web* en todo tipo de aplicaciones. Si bien es cierto que tiene problemas, el gran uso y las necesidades que se exigen de esto produce que siempre haya alguien dispuesto a trabajar e intentar dar soluciones. Los tiempos se mueven muy rápido y en cuestión de años herramientas tan populares como la conocida *Adobe Flash* han quedado en desuso simplemente porque algo mejor aparece para cubrir su puesto. ¿Quién dice que de aquí a unos años no se vean sustituidos los famosos *HTML*, *CSS* y *JavaScript*?.

Bibliografía

Imsirovic, A. (2018). *Vue.js quick start guide*. Packt.

Schwarzmüller, M. (2018, Dec). *Academind*. `https://academind.com`.

Simpson, K. (2018). *You don't know js*. O'Reilly.

Street, M. (2017). *Vue.js 2.x by example*. Packt.