Graduado en Ingeniería Informática

Universidad Politécnica de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

TRABAJO FIN DE GRADO

Modelización de *Web Components* reutilizables para simplificar el proceso de paso de información en aplicaciones web

Autor: Javier Ruiz Calle

Director: Julio García Martín

MADRID, 6 de Junio de 2018

AGRADECIMIENTOS

A mis padres, por solo preguntar que qué tal iba, y no intentar entender de qué iba el trabajo, porque no lo iban a conseguir igualmente. A Revi, que nunca será ingeniero, por dar la idea, ayudarme cada vez que no sabía como seguir, y obligarme a hacer la mejor memoria que podía hacer. Y más que a nadie, a mi novia ***Bea***, por interesarse e intentar entender lo que estaba haciendo, y por aguantarme durante estos meses, espero que no haya sufrido mucho.

ABSTRACT

This end-of-degree project is a study on why people should be developing new web applications using highly reusable components. it will focus on the use of these web components inside the HTML's form tag, as a way to limit the scope of this study.

This study is done through the creation of a framework. It begins explaining the basic knowledge required to understand how and why web components are the future, and even the present, of web development.

The reason this study is done basing it on a project is because it allows us to make it using the tools that a regular developer would use to create these web components. It allows us to get in the mindset of said developer, and thus think what would be the best way to implement them. As a consequence, the user experience is one of the main focuses while developing this project, and it is referenced throughout this paper.

By the end of this project, people who read this paper should have a solid understanding of how they should design their web components, and why they should not be afraid to use those that have been created by other developers.

RESUMEN

Este trabajo de fin de grado es un estudio sobre por qué los desarrolladores deberían pensar en crear nuevas aplicaciones web usando componentes web altamente reutilizables. El proyecto se centrará en el uso de estos componentes dentro de la etiqueta HTML *form*, para limitar el alcance de dicho proyecto.

El estudio se ha realizado mediante la creación de un *framework* que incorpora estos componentes web. Se empieza explicando los conocimientos básicos requeridos para entender por qué los componentes web son el futuro, y el presente, del desarrollo web.

La razón por la que este estudio se ha realizado a través de la creación de un *framework* es porque permite el uso de las mismas herramientas que usaría un desarrollador para crear los componentes web. De esta manera, se consigue pensar en como un desarrollador procedería a implementarlos. Esto tiene como consecuencia el hecho de que la experiencia de usuario sea uno de las partes que más se han tenido en cuenta a la hora de desarrollar el proyecto, y será referenciado a lo largo de este trabajo.

Al finalizar este proyecto, se espera que quien lea este trabajo esté en posesión de los conocimientos, y sobre todo las razones, por las cuales deberían diseñar las aplicaciones web mediante componentes, e incluso usar aquellos creados por otros desarrolladores.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE CÓDIGO

1. INTRODUCCIÓN

Se encuentran hoy en día en la web dos grandes tipos de páginas: estáticas y dinámicas. Las páginas estáticas sirven para mostrar contenido que no va a cambiar, por lo que no se puede interactuar con ellas. El lenguaje utilizado para crearlas es el mismo que para un página dinámica, es decir, HTML (*Hyper Text Markup Language*), CSS (*Cascading Style Sheets*) y Javascript, pero no están hechas para comunicarse con un servidor o modificar su contenido, siempre tendrán el mismo aspecto se haga lo que se haga. Una página estática puede ser un blog, o un artículo de periódico, donde se sabe que la información no va a cambiar dependiendo del usuario, sino que siempre se mostrará la misma información. Cuanto más se mira al pasado de la web, más páginas estáticas se encuentran, ya que la web era más simple y el navigador lo único que hacía era traer del servidor la página, el diseño y los archivos, por lo que no había interacción alguna.

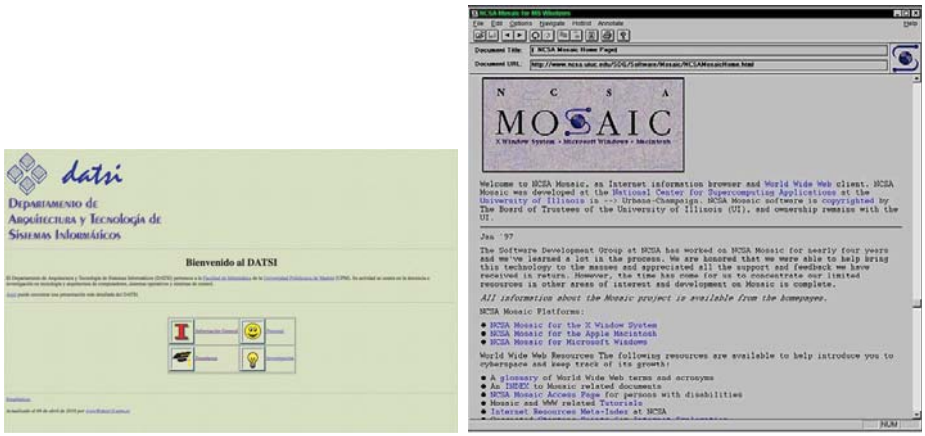

Figura 1: Ejemplos de web estáticas

Por otro lado, una página dinámica es aquella con la que sí se puede interactuar, es decir, que se puede cambiar lo que se está viendo en función de lo que el usuario haga en la página, ya sea a través de botones, formularios, *chats*, galerías, etc. Se programan estas webs con los mismos lenguajes que las estáticas, a diferencia de que en este caso lo que el usuario haga en la página tendrá repercusiones en lo que se le mostrará. Un ejemplo claro y que a mucha gente le será conocido es el carrito de la compra de Amazon, o de cualquier *e-commerce*. Cada usuario verá un carrito diferente, dependiendo de lo que haya seleccionado para comprar. Nótese que no es necesario un *backend*, o servidor, que procese la información para que una página sea dinámica, pero este trabajo asume que hay un servidor detrás que procesa las peticiones de usuario, o que haya alguna manera por la cual el usuario pueda definir información y pasarla a los componentes.

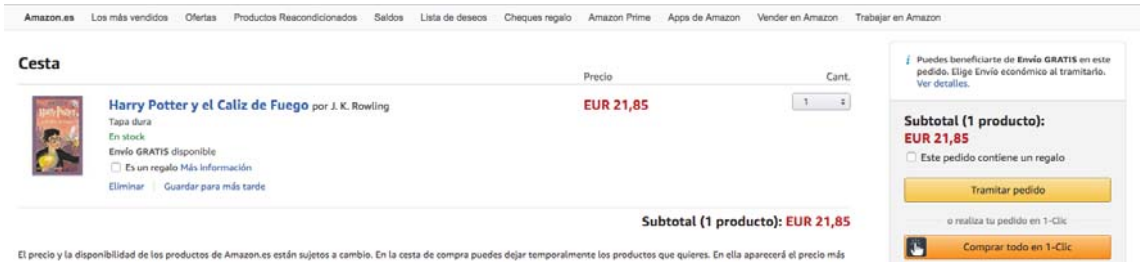

Figura 2: Carrito de la compra de Amazon, ejemplo de web dinámica

Las páginas dinámicas usan los formularios (etiqueta *form* en HTML) para enviar información al servidor y que esta sea procesada. Un ejemplo claro de formulario es el de un formulario de compra, donde el usuario rellena su nombre, apellidos, dirección de envío, número de tarjeta y CVV para realizar el pago. El formulario se encarga entonces de enviar esta información al servidor para que pueda ser tratada, y de esta manera realizar la compra.

La creación de formularios web es un conocimiento básico para un desarrollador web, ya que hoy en día en la mayoría de las páginas se tiene alguna forma de formulario. Se podría decir que el paso de información en el modelo de web actual son formularios, o formularios que intentan no parecerlo, o bien otras opciones que pueden ser implementadas como tal [1].

Sin embargo, a pesar de ser usada por tanta gente, cada formulario debe ser desarrollado desde cero, es decir, pensar en los campos que se necesitan, implementarlos, darles la lógica de negocio y realizar el diseño, que seguirá o no una entidad corporativa. Esto lleva a tiempo perdido de desarrollo, ya que la mayoría de los campos que se utilizan en los formularios suelen ser siempre los mismos. Siguiendo el ejemplo del formulario de compra, en HTML serían cinco etiquetas *input*, con sus correspondientes validaciones.

Este proyecto trata de mejorar este último punto, haciendo que el proceso de creación de formularios sea mucho más fácil, eficiente y rápido. Esto se consigue a través de una serie de patrones de diseño para hacer *Web Components*, o componentes web, patrones que se usan en el *framework* Angular. Esto permite la reutilización de código escrito por el desarrollador, y le permite así mismo usar componentes que haya podido escribir otra persona. De esta manera, un desarrollador puede usar los componentes que se realizan en este trabajo, luego él solo tiene que proveer la información necesario para crear su formulario, sin necesidad de crear y dar diseño a un problema ya resuelto.

El objetivo de este proyecto es la creación de dichos componentes, de manera que sean altamente reutilizables, con un diseño basado en el *Material Design*

de Google. Se proveerá una *API* (*Application Programming Interface*, o interfaz de programación de aplicaciones) para que los desarrolladores puedan tener acceso a las diferentes propiedades de cada uno de ellos. Se consigue de esta forma que cada persona tenga la máxima libertad posible a la hora de realizar sus formularios.

Al finalizar este estudio, se pretende concluir en una serie de normas, buenas prácticas, o simplemente consejos que permitan a los usuarios centrarse en desarrollar su lógica de negocio en lugar de tener que dedicar tiempo a crear elementos de la web que ya están hechos y que no hace falta recrear.

2. ESTADO DEL ARTE

2.1. FORMULARIOS WEB

En HTML, la etiqueta *form* es aquella que define el formulario dentro de una página web. Dentro de esta etiqueta es donde se escribe el formulario, es decir, donde se recopilará toda la información que el usuario quiere mandar al servidor. Puede tener en su interior todos los elementos que se necesiten para crear el diseño, pero los que impactan en su funcionamiento y en la información que se envía al servidor son:

- form
- label
- select
- button
- input
- fieldset
- optgroup
- datalist
- textarea
- legend
- option
- output

Así mismo, posee una serie de atributos, que pueden ser considerados como una *API* para esta etiqueta:

- accept
- autocomplete
- name
- accept-charset
- enctype
- novalidate
- action
- method
- target

Esta es la *API* oficial de la etiqueta *form* [2], luego cada usuario puede definir sus propios atributos para cambiar su funcionamiento. Un ejemplo de esto sería el atributo *ngForm* de Angular, del cual se hablará más adelante.

Generalmente, lo mínimo que se tiene que hacer en los formularios tradicionales es definir el *method* y la *action*, es decir, qué método http se va a utilizar (en general será un *POST*), y a donde se quiere enviar el formulario.

Esta manera de crear formularios presenta un gran problema para los *frameworks* modernos: la mayoría de estos *frameworks* están implementados para que la aplicación web sea una SPA (de sus siglas en inglés *Single Page Application*, o aplicación de página única en español). Esto quiere decir que la aplicación, aunque pueda cambiar de ruta, no recarga nunca la página, que es exactamente lo que ocurre cuando se envía un formulario mediante un *POST* a una URL.

Debido a esto, y para que un formulario funcione completamente con el *framework* que se va a utilizar en este proyecto, la primera acción que se suele hacer en un formulario es añadir dos cosas a la etiqueta *form*:

```
1    <form autocomplete="off" novalidate>
2        ...
3    </form>
```

Código 1: Ejemplo básico etiqueta *form*

De esta manera se consiguen tres cosas: en primer lugar, el auto completado no va a impedir el correcto funcionamiento de los componentes. En segundo lugar, el *novalidate* permite que sea el desarrollador quien esté al cargo de realizar sus propias validaciones. Esto hace que el desarrollador tenga un control mucho más granular en el funcionamiento del formulario, encargándose de los diferentes errores y sus correspondientes mensajes. Por último, al no especificar ninguna acción, la página no se recargará cuando se envié el formulario. Le corresponderá entonces al *framework* que se esté utilizando y al desarrollador de ver como se tiene que comportar la página una vez se haya enviado.

Estos *frameworks* no solo permiten tratar el estado de una aplicación, y en particular de un formulario, sino que además ofrecen métodos para poder hacerlo eficazmente. Por esto, la inclusión del *novalidate* dentro del *form* es de especial importancia, ya que permite al usuario tratar cómodamente el estado de su formulario, así como de gestionar los diferentes errores que van saltando a medida que se va rellenando dicho formulario. Estos errores, al estar definidos por los desarrolladores, permiten un mayor nivel de control sobre los datos que llegan al servidor en comparación con las comprobaciones estándar HTML [3]. Además, las comprobaciones HTML tienen muchas veces problemas de compatibilidad entre diferentes navegadores, por lo que muchas veces se desaconseja su uso.

Resumiendo, los formularios han evolucionado desde ser simplemente una etiqueta *form* que envía directamente información a un servidor, hasta un formulario donde el desarrollador tiene el control total sobre lo que está ocurriendo mientras el usuario va rellenándolo. Y sin embargo, se ha conseguido ir incluso más allá con la llegada de los formularios reactivos. Estos nuevos tipos de formulario permiten una mejor separación de responsabilidades a la hora de programarlos. En el caso del *framework* Angular, la vista o *template* sólo contiene lo que se le muestra al usuario, mientras que toda la lógica está en el controlador, teniendo ahí todas las validaciones y manejo de errores necesarios. Una de las mayores ventajas que tiene este tipo de formularios es el poder realizar *unit testing*, o pruebas unitarias, ya que al estar toda la lógica en el controlador, las variables pueden ser inicializadas por código, y por lo tanto se puede automatizar la creación de pruebas que verifiquen el correcto funcionamiento de dicho formulario.

Una vez que se tiene un conocimiento básico sobre cómo funcionan los

formularios web, se puede pasar a comprender cual es el estado actual de lo que se conoce como *Web Components*, o componentes web.

2.2. COMPONENTES WEB

El concepto de componentes web no es algo nuevo. En 1998, Microsoft propuso la creación de los *HTML Components*, los cuales funcionaban para Internet Explorer en su versión 5.5 [4]. En 2001, Mozilla intentaba crear sus propios componentes web con XBL [5] (de sus siglas en inglés *eXtensible Markup Language*). A pesar de haber tenido una segunda versión de XBL (XBL2 [6]), el proyecto se abandonó en 2012, al igual que los componentes de Microsoft se quedaron obsoletos en 2011 con la salida de la nueva versión de Windows, Windows 10.

Las ideas no eran malas, pero las implementaciones no llegaban al nivel que era necesario para crear una nueva manera de hacer aplicaciones web. Por esta razón, Google, y más adelante Facebook, con sus respectivos *frameworks* Angular y React, decidieron hacer su propia implementación. Viendo lo bien que funcionaba y cómo iba evolucionando, la W3C [7] decide crear una especificación estándar de *Web Components* [8]. A día de hoy, se sigue trabajando en esta especificación [9].

Los componentes web están basados en cuatro estándares, las cuales pueden ser usadas en conjunto o por separado:

- Elementos personalizados (*custom elements*)
- Sombra DOM (*shadow DOM*)
- Importaciones HTML
- Plantillas HTML

Los elementos personalizados son la parte básica de un componente web. Permiten encapsular la funcionalidad de una parte específica de la aplicación, contrariamente a lo que se venía haciendo que era escribir todo en un archivo muy grande con elementos anidados. La mejora de legibilidad es solo uno de los puntos fuertes que tiene este nuevo estándar. Esta mejora no es solo a nivel del árbol de archivos dentro del proyecto, sino que también se corresponde con una mejora de legibilidad del código HTML, ya que los nombres de las etiquetas serán mucho más significativos que si solo se usan las estándares.

Este nuevo estándar permite a los desarrolladores la creación de su propia etiqueta HTML a partir de una serie de APIs Javascript que define su nombre, el método asociado que proporciona la lógica, y de lo que puede extender (si fuera

necesario). Esto permite al desarrollador crear una nueva etiqueta HTML, en la cual la lógica entera está en las manos del creador. Javascript permite además la posibilidad de hacer que el componente reaccione a los diferentes eventos por los que pasa (llamados *lifecycle callbacks*), proporcionando un nivel adicional de control sobre dichos componentes.

El hecho de tener los componentes separados en diferentes archivos mejora el nivel de reutilización de estos, ya que es más cómodo llamarlos sólo cuando se les necesite, y que por lo tanto no interfieran en el tiempo de carga de la aplicación.

En Angular, aunque se verá más en detalle más adelante, los elementos personalizados corresponden a un componente, formado por su plantilla HTML, su hoja de estilos, un archivo para realizar pruebas, y su correspondiente archivo Typescript, donde se guardará toda la lógica de dicho componente. A partir de estos archivos, el desarrollador puede crear la funcionalidad necesaria para su propia etiqueta HTML.

Por otro lado, como parte de los estándares que forman un componente web, se encuentra la sombra DOM (*shadow DOM*). Esta es una solución al gran problema que presentan actualmente las páginas web: al estar todo en los mismos archivos, o al tener que renderizar todos los elementos al mismo tiempo, el alcance de las variables es global, lo que puede dar muchos problemas. Además, no se asegura que cada elemento esté bien encapsulado, es decir, que pueden haber conflictos entre partes de la aplicación que no deberían estar conectadas, lo cual puede hacer que la aplicación sea más lenta, o que produzca errores inesperados.

Para entender la utilidad de la sombra DOM, hay que hablar antes del DOM. El DOM (*Document Object Model*, o modelo de objetos del documento en español) es una interfaz de programación para documentos HTML y XML. Se trata de una representación del documento web que se renderiza para permitir su manipulación, ya sea cambiar su estructura, estilo o contenido. Representa en forma de árbol los diferentes elementos que contiene el documento, dividiéndose en nodos y objetos. De esta manera, se puede acceder a los contenidos de la página desde un lenguaje de *scripting* tal como Javascript.

La sombra DOM se encargará entonces de aislar un nuevo árbol DOM y asociarlo a un elemento (que sería el *root element*, o elemento raíz), haciendo que este se renderice por separado del árbol DOM principal. De esta manera, se consigue que el CSS del componente esté encapsulado, tal que no pueda salir o afectar al resto de la aplicación. Esto simplifica automáticamente las hojas de estilo que se tiene que escribir para cada elemento, ya que no se tiene que tener en cuenta el resto de la aplicación. Por último, la sombra DOM produce un aumento de la productividad de

la aplicación web, ya que esta no tendrá que cargar toda de una sola vez, sino que irá cargándolo trozo a trozo conforme sea necesario. La figura 3 [10] muestra cómo es el funcionamiento básico de la sombra DOM, poniendo en evidencia la encapsulación que conlleva su uso.

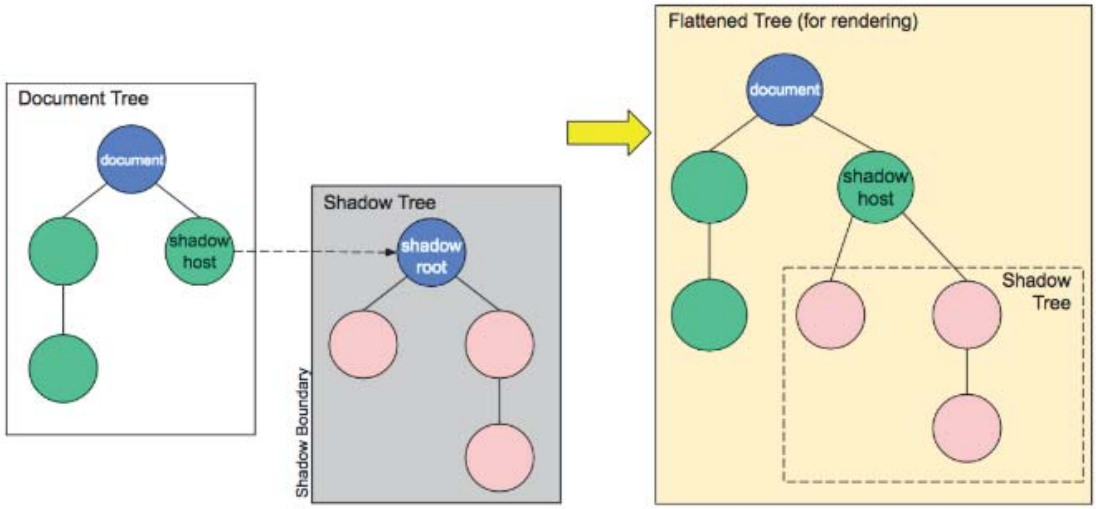

Figura 3: Funcionamiento de la sombra DOM

Uno de los puntos positivos del uso de la sombra DOM es el de poder reemplazar un *iframe*. El *iframe* es una etiqueta HTML que se encarga básicamente de lo mismo que el *shadow DOM*: encapsular componentes para mejorar su reutilización a lo largo de una o varias aplicaciones. La sombra DOM se prefiere al uso de un *iframe* ya que esta última presenta problemas de seguridad, así como el hecho de que es más complicado e ineficaz que escribir un componente para que se renderice en la sombra DOM.

El siguiente estándar en la creación de componentes web son las importaciones HTML [11]. La mejor manera de reutilizar los componentes web creados por los desarrolladores es tener su definición en una archivo separado para luego poder importarlo en las páginas donde realmente se vaya a utilizar.

Para poder usar estos componentes en la página que se necesita, se usa la etiqueta HTML *link*, que especifica la relación entre el documento y un fuente externa (en este caso el componente que se quiere utilizar). Debido a este nuevo estándar, se crea un nuevo tipo de relación llamado *import*, y se usa de la siguiente manera:

```
<link rel="import" href="./my-awesome-component.html">
...
<my-awesome-component>...</my-awesome-component>
```

Código 2: Ejemplo de uso de un componente en una plantilla HTML

El uso que se le daba hasta antes de los componentes web a esta etiqueta HTML era sobretodo para la importación de hojas de estilo. Sin embargo, con este nuevo tipo de relación, se pueden importar diferentes componentes a una página sin problema, ya sea dentro del mismo proyecto, o componentes web realizados por otros desarrolladores, para la cual se especificaría una URL.

Poco a poco, se va notando la importancia de los componentes web y su relación con este proyecto. Este proyecto se basa en la creación de componentes web, y permite a los usuarios de estos poder importarlos y utilizarlos sin tener que preocuparse de su creación. La encapsulación del componente viene dada por la sombra DOM que ya se ha comentado, y la importación con este último estándar que se ha mencionado. Más adelante se verá como hace Angular para encargarse de toda esta parte y que sea todavía más sencillo llamar a componentes externos.

El último estándar que queda por especificar es el estándar de plantillas HTML [12]. Este estándar permite el uso de las etiquetas HTML *template* y *slot*.

La etiqueta *template* es una nueva etiqueta de la *World Wide Web Consortium* que permite guardar código HTML para que sea leído por el navegador, pero que no sea ni interpretado ni renderizado. De esta manera, el usuario final no se da cuenta que ese componente está ahí hasta que se instancia durante el tiempo de ejecución con Javascript. Esto presenta una ventaja muy grande, y es que como no se interpreta este código aunque sea leído, no va a influir en el tiempo de carga de la página.

La etiqueta *slot* es un *placeholder* dentro de un componente web que puede ser rellenado con código más adelante, permitiendo la creación de árboles DOM separados para luego presentarlos juntos cuando se renderice el documento.

Estos estándares fijan como tiene que proceder un desarrollador para crear sus propios componentes web. Sin embargo, la mejor manera de usarlos es mediante un *framework* que los implemente. Dentro de estos *frameworks* encontramos a React [13] (creado por Facebook), VueJS [14], o Angular [15], el cual se usará en este proyecto.

Los *frameworks* que implementan los componentes web ofrecen más libertad a la hora de diseñar una aplicación, ofreciendo facilidades además en el contexto

de patrones de diseño (*singleton*, MVC, etc). Proveen métodos de *scaffolding* (creación del esqueleto de la aplicación) para poder empezar a diseñar rápidamente un proyecto, proporcionan utilidades a la hora de realizar pruebas unitarias, y herramientas para la construcción de la aplicación y su correspondiente despliegue en producción, entre otras muchas ventajas. Consecuentemente, estos son la manera preferida a la hora de hacer una aplicación web hoy en día.

2.3. *FRAMEWORKS* DE COMPONENTES WEB

Una vez que se ha decidido que se quiere trabajar con *web components*, llega la hora de decidir cómo se van a implementar: a mano gracias a los cuatro estándares definidos en el punto anterior, o con la ayuda de un *framework* que facilite su creación, así como dar herramientas adicionales.

El punto bueno de crear los componentes web a mano, es decir, encargarse de todos los puntos, es que se tiene absoluto control sobre lo que va a hacer el componente. Por otro lado, el hecho de tener que crear un componente desde cero implica que se pierde eficiencia en su creación y utilización. Uno de estos problemas sería el paso a producción: encargarse de que todo esté correctamente hecho y tener en cuenta los posibles problemas que puede tener este paso en el desarrollo hace que sea ineficiente encargase de hacer los componentes "a mano".

Por esta razón, lo que los desarrolladores prefieren hacer a lo hora de empezar un nuevo proyecto es utilizar un *framework* que se encargue de realizar la creación del esqueleto del proyecto, así como de proporcionar herramientas para el correcto desarrollo de la aplicación. Además, los *frameworks* proveen una manera predefinida de hacer las cosas, lo cual puede ser de mucha ayuda en caso de que la aplicación siga creciendo con el paso del tiempo, ya que todo seguirá teniendo un esquema definido y útil para el desarrollador que está trabajando en ese momento, así como todos aquellos que puedan trabajar en ese proyecto en el futuro.

2.3.1. React

Una de las librerías más conocidas es React [13], proyecto hecho público por Facebook en 2013. Se trata de una librería que permite una programación declarativa y basada en componentes, enfocada a la creación de interfaces de usuario. Usa Javascript en su versión ES6, aunque acepta el uso de la versión ES5. Técnicamente no es un *framework* ya que no tiene soporte nativo para *routing* o comunicación con servidores, si bien es verdad que puede adquirir esta funcionalidad mediante paquetes NPM.

React permite la creación de componentes web como si fueran funciones Javascript. Estas funciones aceptan elementos de entrada (llamados *props*) y devuelven un elemento de React (*React element*) para que se renderice en el navegador. Este elemento de React no es más que HTML con una sintaxis llamada JSX [16]. Se trata de un pre-procesador HTML que hará que sea compilado a Javascript. Es básicamente azúcar sintáctico para la creación y posterior renderizado del componente. A continuación se puede ver un ejemplo *Hello World* en React:

```
1    class HelloMessage extends React.Component {
2        render() {
3            return (
4                <div>
5                    Hello {this.props.name}
6                </div>
7            );
8        }
9    }
10
11   ReactDOM.render(
12       <HelloMessage name="Taylor" />,
13       mountNode
14   );
```

Código 3: Hello World en React

Se pueden observar casi inmediatamente tres cosas: la primera es que los componentes no están en un fichero separado, la segunda es la utilización casi directa del DOM a través del *ReactDOM*, y la tercera es la programación declarativa que se usa con este *framework*.

El hecho de que la definición del componente no esté en un archivo separado es una decisión tomada por React. Para el equipo de React, esta manera de trabajar es mucho más cómoda debido a la programación declarativa que usan. Además, opinan que la lógica de la interfaz de usuario está fuertemente acoplada a la lógica del componente, por lo que es mejor tenerlo todo en el mismo sitio. Esto no impide el poder importar componentes externos.

2.3.2. VueJS

Por otro lado, tenemos el *framework* de VueJS [14]. Es uno de los últimos *frameworks* que han salido (se publicó en 2014), y ha crecido enormemente debido a su simplicidad a la hora de implementar MVVM (*Model-View-ViewModel*). Fue creado por un ex-empleado de Google, y sigue siendo mantenido por él y un equipo

de 12 personas, lo cual es impresionante teniendo en cuenta que está intentado competir contra React de Facebook y Angular de Google. Usa Javascript en sus versiones ES5 o ES6.

Al igual que React, se trata de un *framework* que está orientado a la creación de interfaces de usuario. La parte principal del proyecto es simple y se puede incorporar fácilmente en otras librerías, por lo que se suele incluso mezclarlo con React o otros *frameworks* para la creación de pequeños *widgets* autosuficientes.

```
<div id="app">
    {{ message }}
</div>
...
<script>
    var app = new Vue({
        el: '#app',
        data: {
            message: 'Hello Vue!'
        }
    })
</script>
```

Código 4: Hello World en VueJS

Toda aplicación de VueJS contiene una instancia de Vue raíz. Esto es importante porque todos los componentes que se creen (que a su vez son instancias de Vue) van a estar conectados a esta instancia principal. Los componentes en Vue son una implementación del estándar de *Web Components* que se han explicado en el punto 2.2, la cual todavía sigue en estado de borrador, por lo que VueJS se permite tomar ciertas decisiones de diseño para mejorar la usabilidad de sus componentes. La instancia raíz se suele poner dentro de la etiqueta *script* de la página principal, para asegurar que esté disponible a toda la aplicación. Luego los componentes pueden estar en archivos separados.

Al estar separado el HTML de la lógica, contrariamente a como lo hace React, VueJS tiene directivas para poder controlar más cómodamente lo que se quiere renderizar en la plantilla. Estas directivas tienen como prefijo "*v-*" y pueden ser por ejemplo: *v-if*, crea condicionales en la plantilla, para decidir si se tiene que renderizar un elemento u otro, o *v-for*, crea bucles dentro de la plantilla, ideal para recorrer listas de elementos, entre muchos otros. Esto es así por temas de simplicidad, la mayoría de los desarrolladores están acostumbrados a tener una separación clara entre las plantillas y el Javascript, y además es más cómodo para importar y reutilizar los componentes que se van creando, siendo más fácil de manipular a medida que la aplicación va creciendo.

Finalmente, tenemos el último de los tres grandes *frameworks* de componentes web: Angular.

2.3.3. AngularJS

En su primera versión, comúnmente conocida como AngularJS [17], fue uno de los primeros *frameworks* en facilitar la implementación de las conocidas SPAs (*Single Page Application*). Así mismo, fue uno de los primeros en implementar el paradigma de componentes web.

Contrariamente a React, AngularJS ponía Javascript en el HTML, para poder tener la ventaja de usar sus directivas propias como *ng-if*, *ng-for*, *ng-class*, entre muchas otras, de manera similar a como lo hace VueJS. Todas las directivas propias de Angular tienen como prefijo el "*ng-*", y esto sigue así hasta su versión actual, siendo uno de los elementos más potentes que tiene. Solamente importando la versión de AngularJS que se quisiera usar, ya se tenía acceso a todas sus utilidades. En el siguiente ejemplo se puede observar esto, así como la facilidad de usar AngularJS en una aplicación web cualquiera:

```
<!doctype html>
<html ng-app>
  <head>
    <script src="angularJS version 1.x"></script>
  </head>
  <body>
    <div>
      <label>Name:</label>
      <input type="text"
             ng-model="yourName"
             placeholder="Enter a name here">
      <hr>
      <h1>Hello {{yourName}}!</h1>
    </div>
  </body>
</html>
```

Código 5: Hello World en AngularJS

Los componentes web se creaban a partir de directivas. La noción de directivas también aparece en versiones más recientes de Angular, aunque se le da un uso diferente. En su primera versión, las directivas se definían en un archivo separado, ocultando de esta manera estructuras complejas del DOM hasta que sea el momento de ser renderizado, tal y como se ha comentado en el punto 2.2.

AngularJS tiene 5 puntos clave a entender, que implementan el conocido patrón de MVC (*Model-View-Controller*):

- Controlador (*controller*): Manipula los datos detrás de la interfaz de usuario.
- Directiva (*directive*): Manipula y prepara el DOM para poder comunicarse con el controlador.
- Plantillas (*view*): Asigna directivas a elementos del DOM.
- Alcance (*scope*): Transporte de los datos entre todas las partes del sistema.
- Servicios (*service*): Usados para dependencias, y comunicación hacia fuera de la aplicación.

Esta primera versión de Angular puede utilizarse actualmente, pero el equipo de Google, viendo como iba evolucionando el desarrollo de aplicaciones web, decidió hacer un rediseño completo del *framework*. Con esto en mente, en 2016, aparece la segunda versión de Angular: Angular 2. Actualmente, Angular se encuentra en su versión 6, siendo el cambio más grande de la primera versión a la segunda. Se describirá por lo tanto la versión 6 a partir de aquí.

2.3.4. Angular

Esta nueva versión de Angular, conocida simplemente como Angular, pasa de usar Javascript a TypeScript [18]. TypeScript es un superconjunto tipado de Javascript, el cual compila a Javascript. Permite seguir usando la sintaxis conocida de Javascript, teniendo como funcionalidad añadida el tipado de variables. Angular está escrito, y es utilizado, con este lenguaje. Además, aporta una cantidad de herramientas tal que el autocompletado, navegación y refactorización. Con este lenguaje, la legibilidad del código aumenta, obligando además a los desarrolladores a pensar más en el alcance de las diferentes variables que se usan, contrariamente a lo que pasa en Javascript, un lenguaje no tipado, y con problemas a causa de ello.

Angular introduce una nueva herramienta: la *angular-cli*. Se trata de una interfaz de línea de comandos, con herramientas para crear un proyecto, añadir archivos, y realizar una gran variedad de tareas de desarrollo como puede ser la realización de pruebas, construir el proyecto para producción, o el despliegue de la aplicación.

Angular tiene una arquitectura mucho más definida que la que puede tener React o VueJS, o incluso AngularJS, y esto es evidente desde el momento en el que se inicializa un proyecto nuevo en Angular (*scaffolding*).

La arquitectura de Angular está basada en los siguientes componentes principales:

- *Modules*
- *Components*
- Servicios

Debido a que este proyecto se va a realizar con Angular, se entrará en detalle en estos componentes.

Los módulos (*NgModule*) son el principal elemento de una aplicación. Estos son los que proveen el contexto para la compilación de los componentes. Se puede decir que una aplicación en Angular es la recopilación de sus módulos. Posee un módulo raíz (comúnmente el *AppModule*), que se encarga de lanzar la aplicación, y una serie de módulos funcionales, creados por el desarrollador. Gran parte de las librerías externas que se pueden utilizar con Angular están disponibles como *NgModule*s, de manera que solamente con declararlos en el módulo principal se tienen disponibles en toda la aplicación.

Cada *NgModule* define qué componentes, directivas y *pipes* pertenecen al módulo (*declarations*), se encarga de que estas sean públicas para que puedan ser usadas por otros módulos (*exports*), importa los elementos de otros módulos para que lo usen componentes de su módulo (*imports*), y provee servicios para que lo usen otros componentes de la aplicación (*providers*). Solo el módulo raíz se encargará de definir el elemento *bootstrap*, que se encarga de compilar la aplicación entera e insertarla en la página *index.html*.

Por otro lado tenemos los componentes. Cada componente está formado (suponiendo que se genere con la *angular-cli*) por básicamente cuatro archivos: la plantilla HTML (*.component.html*), el estilo del componente (*.component.css*), el archivo de *testing* (*.component.spec.ts*) y el archivo de la lógica del componente (*.component.ts*). Si se ha creado con la *angular-cli*, esta se encargará de añadir automáticamente al módulo que le corresponda.

Cada componente controla una parte de la pantalla que ve el usuario, llamado la vista. Se define la lógica de cada componente dentro de la clase, la cual interactúa con la vista mediante una serie de APIs. Los datos que definen cada componente, como pueden ser el nombre de la etiqueta HTML que lo contiene, o las plantillas asociadas, se encuentran en el decorador *@Component*. Define así mismo los servicios asociados al componente.

Los componentes creados forman un árbol de componentes, dependiendo los unos de los otros, y la comunicación entre los distintos componentes se hace a través de eventos. Todos los componentes, al igual que los módulos, cuelgan de un componente raíz, generalmente el *app.component.html*. Al estar separados la vista de la lógica, Angular provee una sintaxis capaz de controlar estructuras de datos o modificar la vista dependiendo del estado del componente (*ng-for*, *ng-if*, *ng-model*, entre otros).

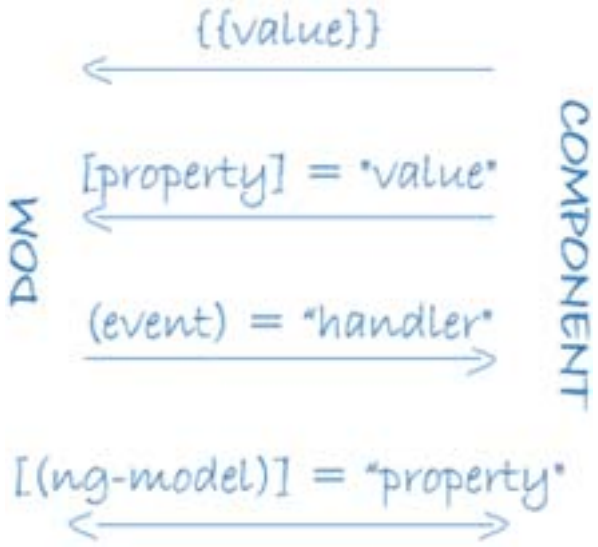

Figura 4: Maneras de enlazar datos entre componente y vista

De esta manera, se tienen dos tipos de enlace de datos: el enlace de propiedades (dirección componente a plantilla), y enlace por eventos (dirección plantilla componente). La figura 4 [19] muestra las direcciones que puede tomar este intercambio de datos. Esta manera de enlazar los datos funciona igual cuando se intenta enlazar dos componentes. Esto es de vital importancia para este proyecto, ya que la comunicación entre los componentes que se crean en este proyecto, y los componentes que son creados por los desarrolladores que los usen se van a comunicar de esta manera. Se explicará esta parte con más detalle cuando se defina cada uno de los componentes que se han creado a lo largo del proyecto.

Existe un último tipo de componente, con una función distinta a la de los componentes normales que se han definido: las directivas. Una directiva se define con la ayuda del decorador *@Directive*, y tiene dos tipos de uso: estructural, encargadas de modificar la forma del árbol DOM, ya sea añadiendo, quitando o manipulando elementos, y por atributos, las cuales cambian la apariencia o el comportamiento del DOM. En este proyecto, algunos de los componentes están definidos como directivas

para facilitar su uso. Estas directivas no se usan como etiquetas HTML, sino que se usan dentro de etiquetas ya existen para proveer una funcionalidad adicional.

Por último, tenemos los servicios. Angular distingue los componentes de los servicios para aumentar la modularidad y reusabilidad de la aplicación. Es importante notar que Angular no obliga a los desarrolladores a seguir este tipo de diferenciación entre componentes y servicios, pero sí proporciona las herramientas necesarias para hacerlo de manera correcta.

Mientras que el componente se encarga de definir la funcionalidad necesaria para la vista, el servicio es el encargado de definir la comunicación con el exterior (servidor, APIs, etc), de manera que se pueda reutilizar este código en distintos componentes. Esto presenta un valor añadido: los servicios pueden funcionar como si fueran un *singleton*. Esto quiere decir que, si los servicios se usan como Angular invita a usarlos, se puede mejorar mucho el rendimiento de la aplicación. Como con los componentes y los módulos, los servicios también pueden ser creados y añadidos automáticamente al módulo correspondiente con la ayuda de la *angular-cli*.

Todos los servicios están definidos con la etiqueta *@Injectable* [20]. Esta etiqueta permite que el servicio pueda ser "inyectado" en otros componentes o servicios. Sin esta etiqueta, un servicio no sería nada más que otra clase TypeScript. Angular llama a esto la inyección de dependencias. Cada servicio se registra en un *provider*, ya sea en un módulo para que sea útil a todos los componentes dentro de ese módulo, o dentro de un componente, donde se tendrá una instancia nueva de ese servicio por cada instancia del componente. Si el servicio ha sido definido a nivel del módulo, y se usa en un componente, Angular se encarga de comprobar si ya existe una instancia de ese servicio para devolverla, si no hay, la crea automáticamente.

Uno de los ejemplos que mejor ilustran los servicios es el servicio de autenticación de un usuario. Este servicio estará definido en el módulo raíz, de manera que todos los componentes puedan usarlo, y así poder saber en todo momento si un usuario se ha registrado o no. El código estará escrito solo en el servicio, y habrá una instancia de este servicio por aplicación. Otro gran ejemplo es el de un servicio que trae datos de una API. En este caso, no se necesita que los datos estén repetidos por cada componente que los necesite, por lo que el servicio se define como *provider* en el módulo raíz.

En la figura 5 [21] se puede observar un esquema donde se representa la arquitectura que sigue Angular. Se aprecia la separación de responsabilidades de cada parte, con el servicio encargándose de "hablar" con el exterior, el componente definiendo y reaccionando a lo que pasa en la vista, las directivas modificando el DOM, y los módulos definiendo todo.

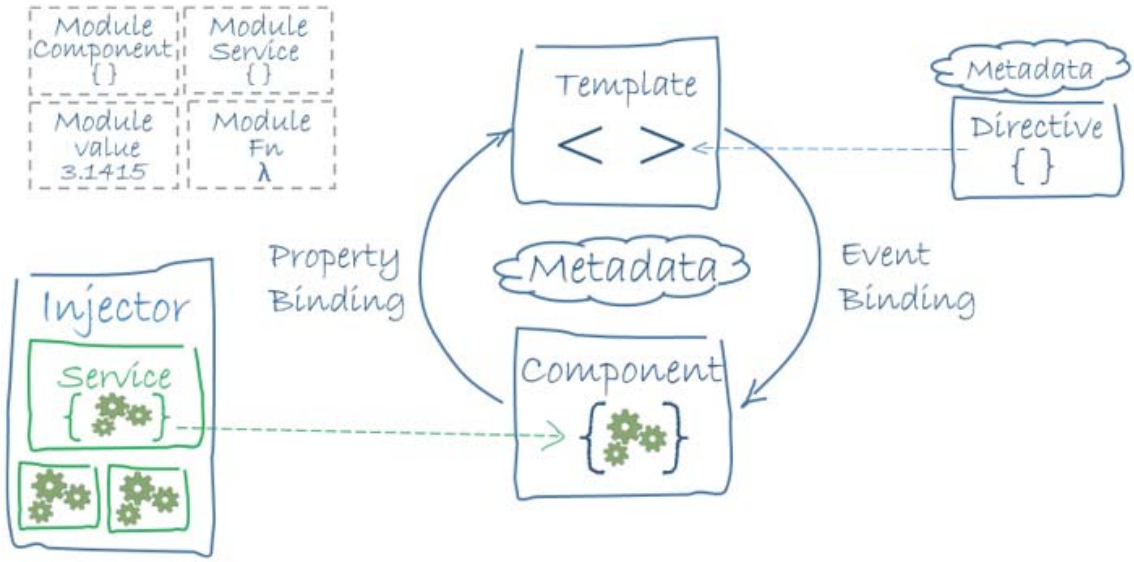

Figura 5: Esquema simple de la arquitectura de Angular

Otra de las cosas que nos proporciona Angular es un conjunto más granular sobre los *lifecycle callbacks* que la especificación oficial de los componentes web, de la que se habló en el punto 2.2.

Cada componente tiene un ciclo de vida que es manejado por Angular, que se puede resumir en:

1. Creación del componente
2. Renderizado de dicho componente
3. Creación y renderización de los componentes hijo
4. Verificación de cambios en su estado
5. Destrucción cuando se quita del DOM

Aunque se hablará más en detalle sobre estos *lifecycle hooks* [22] cuando se estén describiendo los diferentes componentes que se han realizado, se puede adelantar los que son más interesantes para este proyecto:

- ***OnInit()***: Creación del componente, a partir de este momento se puede comprobar los valores que recibe el componente de los componentes padre. Solo se la llama una vez, sirve para inicializar el componente. Este método es llamado después de la llamada al constructor de la clase.

- ***AfterViewInit()***: Una vez que se ha renderizado el componente, se tiene acceso a los hijos de este. Útil cuando se tiene que tener acceso a información proporcionada por el usuario del componente, ya que no se tiene disponible hasta este punto. Es en este punto donde se tiene acceso a los componentes que tienen directivas.

- ***OnDestroy()***: Destrucción del componente, es en este punto donde se deberían destruir todas las subscripciones que se han realizado durante el ciclo de vida del componente. Sirve también para notificar a otros componentes de la destrucción de este.

Angular propone una manera simple de trabajar con estos métodos: la clase que quiera utilizarlos deberá extender de ellos, y tener un método que los implemente (el método se llama igual que el *lifecycle hook* con el prefijo "*ng*"). Angular se encargará entonces de ejecutar cada uno de estos métodos cuando sea el momento apropiado.

La razón por la que se escoge Angular como *framework* para realizar este proyecto es su más que evidente orientación a componentes web, así como la facilidad para escalar proyectos grandes. La separación entre plantilla y lógica facilita el uso de componentes, con todavía más razón si se tiene en cuenta que Angular ofrece, como VueJS, directivas para la manipulación de estructuras de datos y del DOM dentro de la plantilla. Además, es uno de los proyectos con mejor y más amplia documentación, resultando más simple encontrar respuestas a las preguntas que van surgiendo durante el desarrollo de este proyecto. Por último, la razón por la que se va a trabajar con Angular es su facilidad de implementación del patrón de diseño MVC (*Model-View-Controller*). En comparación, React solo aporta la "V", es decir, la vista, el controlador y el modelo es tarea del desarrollador de implementarlo.

2.4. ESTILO Y DISEÑO

El diseño, hasta hace cuatro o cinco años, no era la parte más importante a la hora de crear una nueva aplicación nueva. Sin embargo, hoy en día se considera como una parte fundamental de cada empresa. No solamente se transmite lo que hace o a lo que se dedica, sino que también intenta expresar su mentalidad y como hace que el usuario final se sienta con respecta a la empresa.

Esto presenta un problema para las empresas que no se interesan en el diseño de sus aplicaciones. Generalmente, los usuarios finales no se dan cuenta de lo bonito o funcional que es un diseño, hasta que se encuentran con un diseño que es malo. Por ejemplo, si un usuario está rellenando un formulario de compra, y el diseño de la aplicación no le indica cómo tiene que seguir, lo más probable es que este

usuario decida no comprar nada e irse a otra tienda. Por esta razón, el diseño tiene además consecuencias en los ingresos de una tienda o empresa. Como consecuencia, el *marketing* ya no se hace al final cuando la aplicación ya está completamente construida, sino que se diseña desde antes, orientando muchas veces el desarrollo de una aplicación hacia como se quiere que sea el producto final, ya que finalmente el hecho de que una aplicación sea simplemente funcional puede dañar la reputación o imagen de la empresa.

Poniendo todo esto en el contexto de este proyecto, el usuario o desarrollador que vaya a utilizar estos componentes tiene que tener la certeza de que el diseño es correcto y va a ser apreciado por los usuarios finales. A continuación se describirán tres grandes proyectos, o guías de estilo, que fueron consideradas a la hora de realizar este proyecto:

2.4.1. Atlaskit

Atlassian es la empresa encargada de productos tan conocidos como pueden ser Trello [23], Bitbucket [24] o Jira [25].

Como muchas de las grandes empresas tecnológicas están haciendo en estos últimos años, Atlassian hizo público su guía de estilo y sus componentes a finales del año 2017. Es importante esta distinción entre guía de estilo y componentes, ya que lo que resulta interesante para este proyecto es la guía de estilo. Los componentes que tiene creados son componentes hechos para ser utilizados directamente en un proyecto con React, de la misma manera que se pretende que los componentes de este proyecto sean utilizados directamente en proyectos Angular.

Cada componente que Atlassian ha creado viene definido con un código de ejemplo, una pequeña definición de lo que hace, los elementos (*props*) que puede recibir como argumentos, y las diferentes maneras de usarlos.

Su guía de estilo se basa en la definición de los diferentes componentes que ofrecen, como pueden ser un *button*, o indicadores de progreso, entre otros. La idea que siguen es la de definir las diferentes interacciones con los componentes una sola vez, para luego poder reutilizarla en múltiples componentes distintos y que el usuario solo se tenga que preocupar de aprender una sola manera de hacer las cosas.

Como con todas las guías de diseño, Atlassian define meticulosamente los siguientes elementos:

- Colores: define la paleta primaria y secundaria principalmente, teniendo cuidado de cumplir siempre las relaciones de contraste estándar [26]. También especifica un rango de colores que los usuarios pueden utilizar para modificar y adecuar el diseño a cada desarrollador o empresa.

- Iconografía: Los iconos es una manera de representar comandos, acciones, directorios, etc. Por lo tanto, la manera de representarlos tiene una vital importancia, ya que se tiene que conseguir que el usuario entienda el significado sin ninguna ayuda. Atlassian provee referencias y medidas para ayudar a esto.
- Tipografía: Define los tipos de letra a usar en las diferentes plataformas, y como tienen que ser usadas. De esta manera, el usuario que siga esta guía se asegura que el texto que escriba será siempre legible.

2.4.2. Bootstrap

Bootstrap [27] es uno de los *frameworks* más conocidos para dar diseño a una página web. Es un *framework* que tiene predefinido el CSS de los diferentes elementos que forman una página web, y permite a los usuarios usarlos con la ayuda de clases en las etiquetas HTML. Bootstrap fue creado por un diseñador y un desarrollador cuando trabajaban en Twitter, y desde su creación en 2010 y posterior liberación del código, Bootstrap se ha convertido en un referente en temas de diseño de aplicaciones web.

Al ser un *framework* de CSS, Bootstrap no tiene una guía de diseño como tal, sino que permite usar su diseño directamente en cualquier aplicación web con solo importar sus hojas de estilo y el Javascript que usa:

```
1     <head>
2         <link rel="stylesheet" href="bootstrap_version">
3     </head>
4     <body>
5         ...
6         <script src="jQuery_version"></script>
7         <script src="popperjs_version"></script>
8         <script src="bootstrapjs_version"></script>
9     </body>
```

Código 6: Ejemplo básico de importaciones para usar Bootstrap

Debido a que este *framework* no permite seguir unas guías de diseño, no será utilizado para realizar este proyecto, donde se quiere generar el diseño de los componentes a mano. Sin embargo, Bootstrap permite la descarga y utilización de sus diferentes componentes por separado, entre otras razones para permitir la customización de componentes específicos por parte de los desarrolladores. En el caso de este proyecto, se usa la *grid* (cuadrícula) que proporciona Bootstrap. Ningún componente hace uso del *grid* como tal, pero sí se usa a la hora de desarrollarlos, para facilitar el desarrollo y las pruebas que se realizan.

2.4.3. *Material Design*

No se puede hablar de guías de diseño sin hablar de *Material Design* [28]. *Material Design* fue creado por Google en 2014, y se popularizó con la salida de la versión 5.0 de Android: *Lollipop*. Se trata de un estilo que intenta imitar el mundo físico y sus texturas, basándose en como los diferentes elementos reflejan la luz y crean sombras, sin tener las limitaciones que se pueden encontrar en el mundo real. Esto quiere decir que se pone un enfoque bastante grande en lo que son las animaciones, haciendo un diseño que "responda" a las accciones del usuario, es decir, que la interacción entre el usuario y la aplicación tenga consecuencias visuales que ayuden a entender el funcionamiento de cada botón, tarjeta, o lo que sea que requiera interacción.

Desde entonces, un gran número de aplicaciones han empezado a salir al mercado, e incluso algunas han rediseñado su estilo, para cumplir con *Material Design*. Particularmente, Google, como es de esperar, ha hecho que todas sus aplicaciones sigan este modelo, lo cual les proporciona un estilo distintivo al resto.

Al igual que Atlassian, Google provee una serie de componentes ya hechos para que puedan ser importados en los proyectos de desarrolladores. Estos componentes vienen con la funcionalidad ya predefinida, y Google permite su customización a través de APIs para modificar el estilo, y poder definir que se quiere realizar con cada uno de sus componentes. Lo bueno de estos componentes es que no están atados a ningún *framework* en particular, sino que están escritos en Javascript ES2015, por lo que cualquier desarrollador los puede utilizar, ya sea desde un proyecto en Angular o React, hasta una web minimalista con solo Javascript, HTML y CSS.

La idea de Google con *Material Design* es crear un diseño intuitivo y fácil de usar y entender, por lo que ha desarrollado su propia guía de diseño para ayudar a los desarrolladores a seguir su estilo. Va a ser con la ayuda de esta guía que se van a realizar los distintos componentes que se van a crear en este proyecto. Se intentará crear cada componente siendo lo más fiel posible a la especificación dada por Google.

A pesar de todos los componentes que tiene definidos Google, en este trabajo se realizarán solo los componentes básicos para poder definir un formulario. Estos componentes permitirán cambiar hasta cierto punto el estilo, intentando dar la máxima libertad a los desarrolladores que quisieran usarlo a través de APIs.

La guía de diseño, aunque vaya a servir a este trabajo para crear los componentes, no solo se queda en ese paso, sino que va más allá definiendo lo que se puede o no se puede hacer para conseguir tener una aplicación que se pueda decir que sigue el *Material Design*.

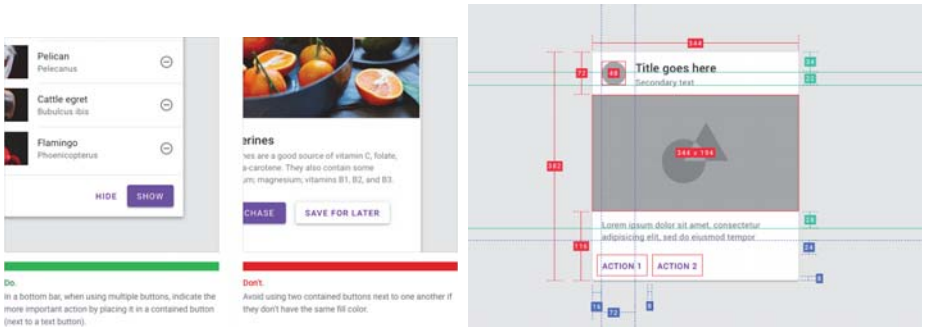

Figura 6: Documentación de una tarjeta y botones en *Material Design*

Finalmente, la razón por la que se va a seguir la guía de estilo de *Material Design* en este proyecto es debido a su extensa documentación, así como a la simplicidad que aporta a los componentes que la implementan. Además, Google ha conseguido que este diseño sea altamente reconocido por la gente, debido a su despliegue masivo en todas las aplicaciones que crea, tanto como para Android como para iOS.

3. *FRAMEWORK* DE COMPONENTES WEB

3.1. DESCRIPCIÓN

En este punto se describirán los diferentes componentes creados durante la realización de este proyecto. Por cada componente, se explicará su función y utilidad dentro de un formulario web, así como una explicación detallada de las decisiones de diseño tomadas para realizar cada uno de ellos. Después de esto, se pasará a explicar como funciona la *API* de cada componente, es decir, qué *inputs* o entradas recibe, y qué *outputs* o salidas tiene para que cada desarrollador configure el componente de la manera que más le convenga.

Al igual que Angular con el prefijo "*ng-*" o VueJS con "*v-*", en este proyecto se ha usado el prefijo "*tfg-*" para definir los componentes que se han creado. Por ejemplo, si se desea utilizar el componente botón realizado en este proyecto, se tendrá que llamar a la etiqueta *tfg-button*.

3.2. *BUTTON*

3.2.1. DESCRIPCIÓN

La etiqueta *button* en HTML es una etiqueta fundamental a la hora de crear un formulario web. Es la que nos permitirá enviar el formulario una vez que se haya terminado de realizar. Existen dos maneras de crear un botón en HTML: la primera manera es crearlo con la ayuda de la etiqueta HTML *input*:

```
1    <input type="button" value="Click Me!">
```

Código 7: Botón con etiqueta *input*

La segunda, y la manera que se utilizará en este proyecto, es con la propia etiqueta *button*:

```
1    <button type="button">
2        <img src="thumbs_up.png" alt="">
3        Click Me!
4    </button>
```

Código 8: Botón con etiqueta *button*

Esta última forma suele ser la preferida por los desarrolladores, ya que no solamente mejora la legibilidad del código y la accesibilidad, sino que además permite insertar contenido dentro de la etiqueta, es decir, que entre las etiquetas se pueden insertar texto, imágenes o incluso HTML adicional. Esto no ocurre con la etiqueta *input*, donde solo se puede definir lo que va a mostrar el botón, en el ejemplo anterior, *Click Me!*.

Nótese que aunque se use la etiqueta *button*, se especifica el tipo al igual que se hace para la etiqueta *input*. Esto es porque cada navegador tiene un tipo por defecto, así que lo más cómodo es definir desde el principio el tipo de botón que se está creando.

Este es el primer componente que se ha realizado, por lo que es el que define como el estilo en el que se van a programar la mayoría de ellos.

En un primer momento, se pensó que la mejor manera de crear este componente sería cogiendo todos los argumentos que puede recibir la etiqueta *button*, y pasándolos como *inputs*, o argumentos, al componente. Estos argumentos serían:

- *autofocus*
- *disabled*
- *form*
- *formaction*
- *formenctype*
- *formmethod*
- *formnovalidate*
- *formtarget*
- *name*
- *type*
- *value*

Esto presenta varios problemas. El primero es que se tendría que comprobar antes que nada, cuales son los argumentos que están presentes y cuales no durante el *onInit*. El segundo problema, el más importante y la razón por la que no se puede proceder de esta manera, es que no se le deja al usuario la libertad de usar sus propios métodos dentro del componente botón que se ha creado. Para empezar, no se han definido ningún tipo de eventos de los que puede recibir el botón, como puede ser el *onFocus*, *onKeyup*, entre otros. Si se añadieran como argumentos al componente, se estaría empeorando el primero de los componentes que se ha comentado anteriormente.

Aunque se siguiera haciendo de esta manera, y se tuvieran en cuenta todos los eventos y propiedades que tiene un botón, existe un problema aún mayor. Si se continuara de esta forma, el usuario no podría usar métodos propios o eventos creados para su aplicación, lo cual limitaría enormemente la usabilidad del componente que se está intentando crear.

Básicamente, se está intentando recrear una etiqueta ya existente sin dar ningún otro valor añadido que el de proveer un estilo definido, llegando incluso a limitar el uso de la etiqueta. Por esto, se recurre a lo que se conoce como *content projection* [29].

Content projection, o proyección de contenido, es el hecho de permitir que el usuario defina dentro del componente lo que quiere hacer. La plantilla entonces del componente botón pasará de ser un listado increíblemente grande de argumentos que puede recibir, a una simple línea:

```
<ng-content></ng-content>
```

Código 9: Plantilla del componente *tfg-button*

Con la etiqueta *ng-content*, se consigue "proyectar" lo que el usuario escriba dentro del componente, teniéndolo disponible en el punto de *AfterViewInit* del ciclo de vida de Angular (visto en el punto 2.3.4). Como el usuario va a poder definir cosas dentro del componente, hay que crear una manera de saber cuales son los botones a los que se les tiene que dar la funcionalidad del componente. Para esto se usan las directivas de Angular. El uso por lo tanto de este componente será de la siguiente manera:

```
<tfg-button>
  <button type="button" tfg-button>
    Click Me!
  </button>
</tfg-button>
```

Código 10: Uso del componente *tfg-button*

La plantilla del componente, una vez que se rendiriza, se traduce por lo que el usuario a puesto, en este caso una etiqueta *button* dentro del componente *tfg-button*. De esta manera, se consigue que el usuario siga teniendo toda la funcionalidad necesaria para crear su propio botón, teniendo el valor añadido que es el estilo y alguna funcionalidad adicional. Además, el usuario puede definir el HTML que necesite dentro del componente, ya que solo se tendrá en cuenta aquellas etiquetas *button* que tengan las directivas que se han definido. Uno de los mejores usos para esta funcionalidad es si el usuario que está desarrollando el formulario quiere poner, al lado del botón, un icono, y hacer que parezca que son un solo componente.

Definir el estilo del contenido proyectado no es complicado gracias a las herramientas que proporciona Angular. En el archivo CSS, donde se definen los estilos del componente, a todos los elementos que queremos seleccionar se les añaden dos elementos: *:host* y *::ng-deep*. El selector *:host* va a permitir que los elementos se apliquen dentro del componente, y el selector *::ng-deep* permite aplicar estilo a todos los elementos hijo de este componente. En el caso de este tipo de componente, lo que se conseguirá será entonces poder dar estilo al contenido proyectado del componente.

La directiva, como se ha dicho, va a ser la encargada de proporcionar el estilo. Siguiendo la guía de estilo de *Material Design* para los componentes *button* [30], se han creado cuatro tipos de directivas:

1. *tfg-button*

2. *tfg-raised-button*
3. *tfg-floating-action-button*
4. *disabled*

La primera directiva, *tfg-button*, crea un botón básico sin contorno, simplemente el texto, lo que *Material Design* define como un *Text button*. Cada vez que el usuario pulsa en el botón, un efecto de onda se produce dentro del botón, poniendo en evidencia su borde y el área que ocupa. Este efecto es conocido como *ripple*, y es ampliamente usado en *Material Design*, ya que permite dar una respuesta visual a las acciones que realiza el usuario.

Figura 7: Componente *tfg-button* con directiva *tfg-button*

Figura 8: Componente *tfg-button* con directiva *tfg-button* en estado *onHover*

Esta directiva añadirá las siguientes clases al botón que la contiene: *tfg-button*, *ripple*, y añadirá una etiqueta *span* con la clase *opacity*, que permitirá delimitar los bordes y la opacidad del botón. Además, esta última clase tendrá como tarea resaltar el botón cuando el usuario pasa por encima (evento *onHover*). De esta manera, aunque sea un botón sin bordes y solo con texto, se le puede delimitar y saber dónde está, y reconocer automáticamente que es un botón.

Este tipo de botones se usa para pequeñas acciones, incluyendo las que aparecen en tarjetas, *snackbars* (pequeñas barras con texto que proporcionan información puntual, como puede ser un aviso de *cookies*), o imágenes. En resumen, este botón no debería quitar protagonismo del texto que hay alrededor, ya que debería solo ofrecer acciones adicionales a lo que se está viendo.

La segunda directiva, *tfg-raised-button*, crea un botón con elevación, lo que en *Material Design* se conoce como un *contained button*. Al igual que para la directiva anterior, se va a añadir la clase *ripple*, y para dar un estilo diferente, la clase *tfg-raised-button*. En este caso, la etiqueta *span* no hace falta añadirla, ya que este tipo de botón sí que tiene bordes y color de fondo, por lo que no hay que esconderlo.

Figura 9: Componente *tfg-button* con directiva *tfg-raised-button*

Este tipo de botones están diseñados para elementos a los que se le quiera dar un énfasis más grande, distinguibles por su elevación y su color que inunda el botón entero, llamando la atención del usuario. Deberían contener acciones que son importantes para la aplicación.

La tercera directiva, *tfg-floating-action-button*, añade al botón del usuario las siguientes clases: *tfg-floating-action-button* y *ripple*. Al igual que la directiva anterior, este tipo de botón no tiene necesidad de la etiqueta *span*, ya que tiene que mantener el fondo y la forma que lo define.

Figura 10: Componente *tfg-button* con directiva *tfg-floating-action-button*

Siguiendo la guía de *Material Design*, este tipo de botones están diseñados para representar la acción primaria, o la más común, de una vista. El ejemplo más común es el botón de crear un nuevo email en la aplicación de *gmail*. Se trata de un botón que está siempre presente (*floating*), ya que define la tarea principal que el usuario puede realizar. Típicamente tiene una forma circular, y es así como se ha definido en el componente. Además, en la mayoría de los casos, no contienen texto, sino un icono significativo de la acción que realizan. Por esta razón, el diseño que se ha seguido para crear este componente botón es el acertado. El usuario tiene total libertad a la hora de decidir si quiere usar un icono o un texto, sin preocuparse como el componente que se ha hecho en este proyecto va a reaccionar ante esta eventualidad.

Por último, se tiene la directiva *disabled*. Esta tiene varias funciones. Para empezar, añade al botón del usuario la clase *disabled*. Esta directiva necesita que alguna de las tres anteriores estén definidas, ya que son ellas las que van a proporcionar el estilo al botón. Esta directiva se encargará entonces de quitar la clase *ripple*, haciendo que no haya respuesta visual a la acción del usuario, y además hace que, con la clase *disabled*, el botón pierda el color y se quede gris, de manera a que solamente de un vistazo se pueda reconocer que el botón está desactivado.

Figura 11: Componente *tfg-button* en modo *disabled*

Este componente no tiene como tal definida una *API*. En este trabajo, se hablará de la *API* como la lista de *inputs* o argumentos que puede recibir un componente. Por esta razón, el componente *tfg-button* no tiene una *API*, sino que tiene una serie de directivas que permiten modificar su funcionamiento y estilo.

3.3. *BUTTON TOGGLE*

3.3.1. DESCRIPCIÓN

Ya se han discutido los principales modelos de botones que se pueden utilizar en un formulario. Sin embargo, queda un tipo adicional que merece ser su propio componente, debido a su funcionamiento, a la lógica que lo acompaña, y al HTML que utiliza. Este tipo de botón es el *button toggle*, o botón interruptor. Se llama así porque solo tiene 2 estados posibles: activado o desactivado.

El HTML necesario para crear un interruptor es completamente diferente al de un botón. Para empezar, no se usa la etiqueta *button*, sino una combinación de la etiqueta *input*, *span* y *label*. A continuación se puede observar cual es la plantilla de dicho componente:

```
<label>
  <input type="radio"
         class="hide-checkbox"
         [ngClass]="{'disabled': disabled}"
         (click)="toggle()">
  <span class="content"
        [ngClass]="{'disabled': disabled}">
    <ng-content></ng-content>
  </span>
</label>
```

Código 11: Plantilla del componente *tfg-button-toggle*

Se puede observar el hecho de que en ningún momento, como se comentaba, aparece la etiqueta *button*. Esto es porque el botón de tipo interruptor en realidad hace la función de un *checkbox*, o casilla de verificación, pero con funciones más específicas. El *checkbox* sirve solo para decir si se selecciona algo o no, mientras que este interruptor se usa para activar diferentes funciones. Como ejemplo significativo se puede coger la barra de herramientas del *Google Docs*. Cada icono tiene una función particular, y tenerlo activado o desactivado implica un cambio importante en la manera en la que se trabaja.

Figura 12: Ejemplo de *toggle buttons* en Google Docs

Todos los *toggle buttons* que se encuentran en la web están creados de la misma manera en la que se ha realizado la plantilla para este componente: un *label* que envuelve todo, con *input* y un *span* en su interior. El *input* indica que el usuario puede introducir datos, en este caso un simple *click*, y el *span* es lo que mostrará el contenido del botón.

En el código de la plantilla de este componente se puede ver que se vuelve a hacer uso de la etiqueta de Angular *ng-content*. En este caso, se utiliza para permitir que el usuario decida qué contenido quiere tener dentro del botón, es decir, que el usuario es libre de poner texto, icono o una imagen. Para el componente, eso será simplemente contenido proyectado que no le interesa.

Se aprecia así mismo el funcionamiento básico que tiene que llevar a cabo: cogiendo el evento *click*, es decir, cada vez que el usuario pulse el botón, se llamará a la función *toggle()*.

Para entender cómo funciona el método *toggle()*, hay que entender antes como se comunican los componentes entre ellos. Como ya se comentó anteriormente (punto 2.3.4), los componentes reciben *inputs*, o argumentos, de sus componentes padre, y los componentes hijos se comunican con los padres a través de eventos (ver figura 4). Esto quiere decir que, cuando un usuario pulsa el botón, se le tiene que comunicar el hecho de que haya sido pulsado.

En este punto es donde entra el método *toggle()*. Este método se encarga de emitir un evento con el valor del componente en ese momento. De esta manera, el usuario podrá capturar el evento de la misma manera que por ejemplo se captura en este componente el evento *click*. Cada evento lleva en su interior el valor que tiene en ese momento el componente (*true* o *false*), y su identificador. Este identificador es muy importante, pero se discutirá más adelante su uso.

El valor que se emite al usuario es el valor del componente entero. Se ha decidido hacerlo de esta manera para mejorar la experiencia de desarrollo del usuario: con un solo evento tiene acceso a todo el estado del botón, ya sea si está seleccionado o no, su identificador, y el valor del botón, si ha decido especificar uno.

A diferencia del componente *tfg-button*, este componente no define su estilo a través de directivas. Por lo tanto, en este componente el estilo se define gracias al decorador *HostBinding* y la directiva *ngClass*.

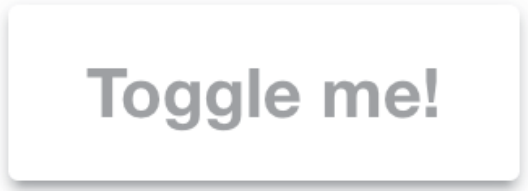

Figura 13: Componente *tfg-button-toggle*

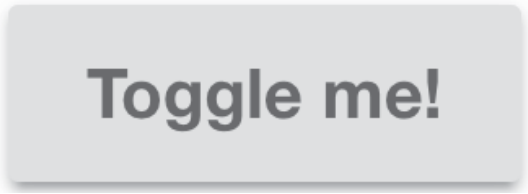

Figura 14: Componente *tfg-button-toggle* seleccionado

HostBinding se encarga de añadir una clase al *host*, en este caso, la etiqueta del propio componente: *tfg-button-toggle*. Este decorador permite añadir o quitar una

clase dependiendo de una variable booleana, siendo esta variable la que indica si el botón está pulsado o no, o si está desactivado (*disabled*). Luego en la hoja de estilos se utiliza el selector *:host* para referirse a ese elemento, y de esta manera poder dar diseño al componente entero.

3.3.2. *API*

Este componente sí tiene lo que se podría considerar como una *API*. En este caso, la *API* es una serie de argumentos o *inputs* que puede recibir el componente y que altera su diseño y/o funcionamiento. Los argumentos que puede recibir son los siguientes:

1. *value*
2. *id*
3. *disabled*

El argumento *value* es un argumento opcional, es decir, que si el usuario no lo proporciona no pasa nada. En este caso, es el que va a definir el valor que va a tener el botón cuando se pulse. Puede tener un sentido tan simple como un *on/off*, o tener un significado más grande y significativo para la aplicación. En cualquiera de los casos, el usuario tiene la posibilidad de añadirlo, y cada vez que se pulse el botón, este valor será emitido junto con el resto de variables que definen el botón en un evento.

El argumento *id* es también opcional, aunque no de la misma manera. El identificador es una parte muy importante del componente, como se verá en el punto siguiente, ya que lo ayuda a diferenciarse del resto de *toggle buttons* que pueda haber en la aplicación. Es por esto que, si el usuario quiere definirlo, se utiliza ese valor y sin problema, pero en el caso de que no le interesa un identificador en especial, se crea uno automáticamente:

```
1     this.id = 'tfg-button-toggle-' +
2         Math.random().toString(36).substr(2, 3);
```

Código 12: Construcción del identificador único de cada *tfg-button-toggle*

Esta manera de crear los identificadores puede parecer ineficiente ya que no se consigue que todos los identificadores sean únicos. Sin embargo, lo único que se necesita es que los identificadores sean distintos dentro de una misma aplicación, lo cual con esta manera se consigue de sobra. Aún así, en caso que el usuario esté en desacuerdo con esto, siempre puede utilizar el argumento *id* para crear el mismo sus propios identificadores con algún método más seguro.

El último argumento es *disabled*. Este argumento es el que indica si el usuario quiere que el botón esté desactivado o no. Lo bueno de ponerlo como argumento es que el usuario que use este componente puede cambiar este argumento mediante el código de su aplicación. Este argumento se corresponde con una variable booleana dentro de la lógica del componente, y está asociada al decorador *HostBiding*. De esta manera, una vez que la variable tenga valor de *true*, automáticamente se le añadirá una clase a la etiqueta del componente (*tfg-toggle-button*), y se procederá a cambiar no solamente el diseño, sino también el funcionamiento del botón, ya que antes de emitir el evento en caso de que se pulse, se comprueba si está desactivado o no.

Estos son los cuatro argumentos que puede recibir el componente. Cada uno de ellos modifica y/o altera el funcionamiento del botón, proporcionando al usuario la mayor libertad posible a la hora de trabajar con el *tfg-button-component*.

3.4. *GROUP TOGGLE*

3.4.1. DESCRIPCIÓN

La idea de crear un interruptor (*toggle button*) se ha hecho mediante el componente que hemos descrito en el punto anterior (3.3). Sin embargo, la funcionalidad de un interruptor puede ir mucho más allá.

Lo que se ha conseguido por ahora tiene sentido en casos simples, opciones que tengan dos opciones: *true* o *false*, encendido o apagado. Este puede ser el caso por ejemplo de un botón que active el *dark mode*, o modo oscuro, de una aplicación, cambiando su estilo. En este caso, solo existen dos opciones, o se ha pulsado el botón o no.

Pero la funcionalidad de un interruptor puede ser mucho mayor si se empiezan a combinar en grupos de interruptores. De esta manera, se pueden llevar a cabo listas de opciones, como puede ser la selección de preferencias de usuario de cualquier aplicación. Si esto se enfoca al objetivo de este proyecto, el cual es facilitar el paso de información de una aplicación web a través de formularios con componentes reutilizables, se puede ver como una oportunidad para definir opciones simples, pero agrupadas.

Todo esto que se ha comentado se podría hacer técnicamente a través del componente anterior (punto 3.3), dejando que el usuario que use ese componente se encargue de luego dar toda la lógica que necesite para crear su aplicación. Sin embargo, existe un uso de la lista de interruptores que complica la lógica de una aplicación: crear una lista que permitan escoger solo una opción.

Por esta razón, se ha decidido crear un componente adicional que se encargue de ello. Este componente es el *tfg-toggle-group*. No es un componente indispensable como lo puede ser el que remplaza a la etiqueta HTML *button* o el *input*, pero añade el suficiente valor como para que se considere utilizar el *framework* para mejorar o crear una nueva aplicación web.

Se ha intuido por los ejemplos anteriores, pero la realidad es que este componente es en realidad dos distintos, apoyándose los dos en la misma plantilla HTML:

```
1     <ng-content></ng-content>
```

Código 13: Plantilla del componente *tfg-toggle-group*

Se observa en la plantilla que en realidad este componente no es más que un *wrapper* de un listado de interruptores. El contenido que se espera que sea proyectado son los diferentes interruptores que el usuario quiere listar.

```
1     <tfg-single-selection-group
2         (selected)="handleEvent($event)">
3         <tfg-button-toggle value="foo" id="my_id">
4             Toggle Me!
5         </tfg-button-toggle>
6         <tfg-button-toggle value="bar">
7             Or Me!
8         </tfg-button-toggle>
9
10        <span>Selected value: {{ value }}</span>
11    </tfg-single-selection-group>
```

Código 14: Uso del componente *tfg-single-selection-group*

Con este código de ejemplo se aprecia claramente cual es el contenido proyectado para el componente *tfg-single-selection-group*: los dos componentes *tfg-button-toggle*. Esto es útil sobre todo porque no se limita el número de interruptores que se pueden colocar dentro del componente. Además, se coge del contenido proyectado solamente el componente *tfg-button-toggle*, por lo que si el usuario quiere introducir más cosas, como ocurre en este caso con la etiqueta *span*, puede hacerlo. Esto no tiene ninguna ventaja adicional excepto por el hecho de que mejora un poco la legibilidad del código, permitiendo que se sepa de un simple vistazo cual es la función que sirve el *span*, y a quien pertenece.

El código anterior pone en evidencia una de las grandes ventajas de usar este componente: en vez de tener que recibir un evento por cada *tfg-button-toggle* que se utilice, solo se recibe uno por cada grupo de interruptores que se cree. En este

caso, el evento es *selected*, y su contenido va a ser una lista de objetos Javascript, con las claves siendo los valores de cada uno de los interruptores, y el valor de cada clave siendo un *boolean* a *true* o *false*.

Esto presenta un gran cambio en comparación con el uso de un *tfg-toggle-button* simple. Como se comentó en la descripción del componente *tfg-toggle-button* (punto 3.3.2), el *value* de cada uno de ellos es totalmente opcional, al igual que el id. Sin embargo, en el caso de que se quisiera utilizar el componente *tfg-toggle-group*, el *value* se convierte en un argumento obligatorio, ya que es el que va a diferenciar cada uno de los interruptores dentro de un mismo grupo. Esto está hecho de esta manera ya que el valor de cada interruptor, dentro de un grupo, es más significativo y proporciona un dato inmediato al usuario. Si el valor que el evento emitiera fueran los identificadores, se estaría obligando al usuario, en cierto modo, a mantener una relación entre cada *id* y su valor asociado. Esto no resulta un problema cuando solo se usa uno, pero en un grupo se ha preferido proceder de esta manera.

Como ya se ha comentado anteriormente, este componente se encarga de emitir un solo evento, recopilando todos los estados de los distintos *tfg-toggle-button* que se encuentran en su interior. Para llevar a cabo esto, hace falta hablar de una parte fundamental de Angular: los servicios.

Los servicios ya fueron comentados en el punto 2.3.4. El servicio que se ha creado para dar apoyo a este componente sirve como si fuera un enrutador (*router*). Esto quiere decir que la manera en la que se van a comunicar dos componentes va a ser a través de este servicio. El servicio como tal solo tiene dos métodos: uno que se encarga de emitir un evento, y otro que se encarga de devolver la última situación del evento. Se entenderá mejor como funciona a continuación, donde se explicará su uso.

El primer caso de uso de este componente es el de crear un listado de interruptores para proceder a una selección múltiple. Para este caso se utiliza la etiqueta HTML *tfg-multiple-selection-group*. Lo primero que realiza este componente es agrupar todos los elementos *tfg-toggle-button*, los cuales los tiene disponible en el *AfterInit*, del ciclo de vida de Angular (ver punto 2.3.4). Una vez que se tienen todos los interruptores, el componente se subscribe a todos sus eventos. Esto permite que cada vez que cada vez que se pulse o active uno de los interruptores, este componente recibirá automáticamente su valor, y podrá lanzar inmediatamente un evento (*selected*), para que lo coja el usuario y proceda como crea adecuado. Todas estas subscripciones serán eliminadas una vez el componente se destruya. Angular facilita esto a través del *lifecycle hook OnDestroy*, donde nos encargaremos de que no quede nada en memoria una vez que se deje de utilizar este componente.

Figura 15: Componente *tfg-toggle-group* con etiqueta *tfg-multiple-selection-group*

Figura 16: Componente *tfg-toggle-group* con etiqueta *tfg-multiple-selection-group*, con elementos seleccionados

El segundo caso de uso, y la razón por la que hace falta un servicio, es la de crear un listado de interruptores en la que solo se acepte una opción. En este caso, se utilizaría la etiqueta *tfg-single-selection-group*. El funcionamiento de este componente es el siguiente: se tiene un listado de interruptores, y solo se puede tener pulsado uno al mismo tiempo. Por lo tanto, después de la primera vez, cada vez que se pulse otra opción, se tiene que activar esa y desactivar la anterior, y el componente debe emitir el evento indicando que todos están desactivados excepto el último que se ha pulsado.

Para llevar a cabo este componente, se ha tenido que recurrir a un servicio externo. Esto es debido a que Angular no permite que desde un componente se cambie el estado de otro, y hacerlo mediante *inputs* y eventos sería complicar demasiado el componente *tfg-toggle-button*, y conseguiría que su uso por si solo resultara poco eficaz, ya que el valor añadido del botón no conseguiría ganar a las desventajas y complicaciones que implicaría.

Por esta razón, se recurre a un servicio. Con el servicio, cada vez que ocurre un cambio en el componente *tfg-single-selection-group*, cada uno de los interruptores del grupo llamará al servicio, el cual se encargará de emitir un evento. A este evento

están subscritos todos los botones de la aplicación que se estén mostrando en ese momento, y por lo tanto, todos aquellos que hayan recibido el evento se desactivarán automáticamente, si coincide el valor emitido con su propio valor. Mientras tanto, el componente reconocerá cual es el valor del botón que ha sido pulsado, y por lo tanto emitirá un evento indicando este cambio.

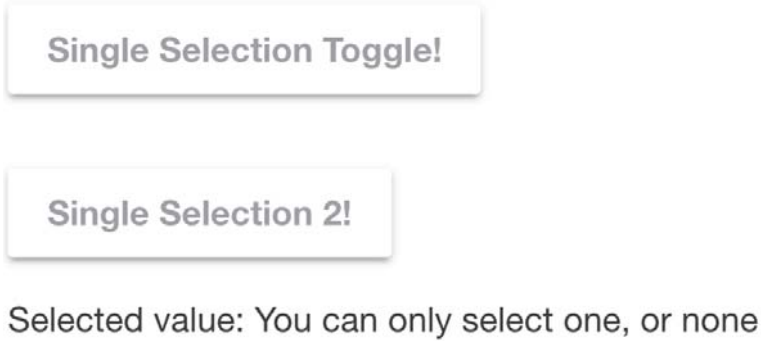

Figura 17: Componente *tfg-toggle-group* con etiqueta *tfg-single-selection-group*

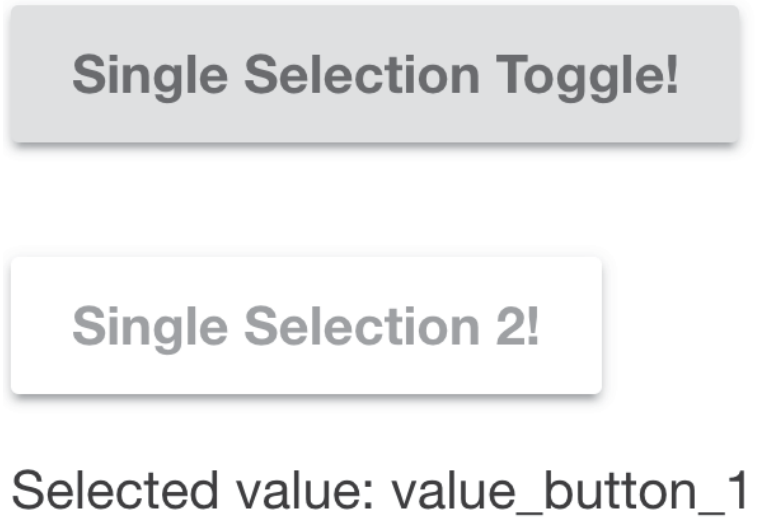

Figura 18: Componente *tfg-toggle-group* con etiqueta *tfg-single-selection-group*, con un elemento seleccionado

Al igual que para el componente de selección múltiple, cuando llegue el momento de destruir el componente (porque se cambia de página o porque el usuario así lo decide), Angular se encarga de llamar al método *ngOnDestroy*, donde se tiene indicado que todas las subscripciones se borren, para no entorpecer y permitir a la aplicación ser todo lo eficaz que pueda ser.

Para mejorar la usabilidad de estos componentes, se ha pensado bastante en cómo deberían ser los eventos que emiten cada uno de ellos. El primero, al ser una selección múltiple, pretende reducir al mínimo el trabajo que tiene que hacer el usuario que lo use, por lo que solo emite un objeto con la clave siendo el *value*, y el valor siendo un *boolean*. El segundo, como se trata de una selección simple, mantiene el mismo tipo de evento que si se usara solo un botón, es decir, que cada vez que se selecciona un nuevo interruptor, el evento emitido es simplemente el valor del componente entero, sin importar el resto de opciones, ya que lo que se supone que le interesa al usuario es esa selección.

Como conclusión de este componente, se puede pensar en él como un componente comodín. Este componente no es indispensable para la realización de formularios, pero sí facilita enormemente la tarea del desarrollador que los utilice. Además, como se ha intentado hacer en todos los componentes que se han desarrollado a lo largo de este proyecto, no se obliga al usuario a sacrificar funcionamiento o rendimiento de la aplicación por usar estos componentes. En caso de que el desarrollador que los use quiera crear su propia lista de interruptores, lo puede seguir haciendo perfectamente con la ayuda de los componentes *tfg-toggle-button*.

3.5. *INPUT*

3.5.1. DESCRIPCIÓN

La etiqueta *input*, al igual que la etiqueta *button*, es una de las etiquetas HTML más utilizadas en los formularios. Esta etiqueta es la que permite al usuario introducir información para que luego pueda ser enviada al servidor para que se valide, o simplemente para guardarlo en una base de datos.

Los ejemplos de uso de la etiqueta *input* van desde simples formularios para hacer *login* en una aplicación, donde se tienen dos *inputs*, uno para usuario, y el otro para la contraseña, hasta algunos más complicados como pueden ser formularios de compra, donde el usuario tendrá que introducir sus datos de envío, sus datos personales, y los datos de su tarjeta para realizar el pago.

La etiqueta *input* cobra sentido solo cuando se encuentra dentro de la etiqueta *form*, es decir, que forma parte de un formulario. De esta manera, la información introducida por el usuario cobra sentido ya que tiene un uso en el servidor. Obviamente, los desarrolladores pueden usar esta etiqueta fuera de un formulario, pero no es lo recomendado. Un *input* suele tener la siguiente forma [31]:

```
1     <form method="POST">
2         User: <input type="text"
```

```
                    name="fname"
                    placeholder="Enter your username">
        Password: <input type="password"
                    name="passwd"
                    placeholder="Enter your password">
        ...
        <button type="submit">Submit</button>
    </form>
```

Código 15: Ejemplo etiqueta *input* en un formulario

En este ejemplo, los *input* sirven para que el usuario pueda introducir su usuario y contraseña, para luego pulsar un botón y que se envíe esta información al servidor para que sea verificada.

Para realizar el componente *input*, se intentó realizar primero de la misma manera que se hizo el *button*, es decir, cogiendo todos los atributos obligatorios del *input*, y poniéndolos como argumentos no obligatorios para el componente. La lista sería la siguiente [31]:

- accept
- align
- alt
- autocomplete
- autofocus
- checked
- dirname
- disabled
- form
- formaction
- formenctype
- ...

Se ha empezado la lista, pero hay treinta y un atributos distintos para la etiqueta *input*. Si cada vez que se crea un nuevo componente *tfg-input*, se tiene que comprobar si existe o no uno de esos argumentos, haría que la creación de este fuera mucho más lenta, y por lo tanto, perjudicaría su uso el usuario. Además, si se tiene que hacer por cada argumento una función específica que modifique el estado del *input*, puede complicar mucho el funcionamiento del componente. Todo esto teniendo en cuenta que además tiene el mismo problema que tenía el componente *button* (punto 3.2.1), es decir, que en el caso de que el usuario quisiera crear un evento propio, o argumento propio, no podría usarlo con este componente ya que no estaría definido.

A raíz de esto, se decide proceder de manera similar al componente *tfg-button*, es decir, haciendo uso de la proyección de contenido de Angular. En este caso, la plantilla será un poco más complicada, ya que el *input*, siguiendo la guía de diseño de *Material Design* [32], tiene más cosas que definir:

```
<div class="input-wrapper">
    <ng-content></ng-content>
    <span class="highlight"></span>
    <span class="underline"></span>
    <label>{{ label }}</label>
</div>
```

Código 16: Plantilla del componente *tfg-input*

Como bien se puede observar en la plantilla de este componente, en ningún momento se llama directamente a la etiqueta *input*. Esta tarea se le deja al usuario, para que pueda sacarle partido tanto a la etiqueta que ya existe, como al componente que se ha definido en este proyecto. Para ilustrar este punto, a continuación se escribe un código de ejemplo de cómo se debería usar el componente *tfg-input*:

```
<tfg-input>
    <input tfgInput
           placeholder="User"
           label="Enter your username"
           type="text">
</tfg-input>
<tfg-input>
    <textarea tfgInput
              placeholder="Text"
              label="Write your text here">
    </textarea>
</tfg-input>
```

Código 17: Ejemplo de uso del componente *tfg-input*

En el ejemplo de uso se puede apreciar cual es el contenido proyectado que recibe el componente: la etiqueta *input* o la etiqueta *textarea*. *Textarea* [33] es una etiqueta HTML que permite introducir varias líneas de texto dentro de una caja, sin necesidad de limitar la cantidad de texto que el usuario quiera escribir, a menos que así lo haya decidido el desarrollador de la aplicación. Tiene un funcionamiento muy parecido al de la etiqueta *input*, y *Material Design* indica que las dos etiquetas tienen básicamente el mismo tipo de estilo [32]. Es por esta razón por la que se ha creado solo un tipo de componente para englobar estas dos etiquetas.

En temas de estilo, los *inputs* y *textarea* que siguen la guía de diseño de *Material Design* tienen un estilo muy distintivo del resto. Se empieza por el *container*, el cual permite reconocer a la etiqueta *input* como un botón sin necesidad de tener que interactuar con ella. A continuación tiene el *label text*, lo que en la plantilla de este componente es el argumento *label*, que es el que define el botón y su uso, lo cual indica al usuario qué tipo de texto tiene que introducir. Lo siguiente

es el *placeholder*. Este elemento aparece una vez que el usuario ha pulsado sobre el componente *tfg-input*, y entonces el label "sube", y aparece el *placeholder*. De esta manera, no solo se tiene un indicativo de lo que se tiene que introducir en el *input*, sino que además se tiene un ejemplo de lo que se espera por parte del usuario. Por último, se tiene una pequeña animación cada vez que el usuario pulsa el *input*. En un primer momento, el *input* es de color y fondo gris, indicando que nada ha sido introducido, pero una vez que el usuario pulsa *input* para introducir texto, la barra inferior y el *label* se ponen de color azul (*#2196F3*), dando al usuario una confirmación de que se ha pulsado el elemento. Una vez el usuario ha terminado de introducir texto, el color de la línea inferior volverá a ponerse en gris, indicando que ese componente no está activo, pero el *label* guardará su color azul, indicando que ya ha recibido contenido.

Figura 19: Componente *tfg-input* de tipo *text*

Figura 20: Componente *tfg-input* de tipo *text*, en estado *focus*

Para llevar a cabo y conseguir formalizar este estilo, se envuelve el contenido proyectado del usuario en una etiqueta *div*, se le añaden distintos elementos para conseguir replicar el estilo de *Material Design*. Al igual que con el componente *tfg-button*, se añaden clases al contenido proyectado mediante una directiva que tiene que añadir el usuario a su etiqueta *input* y *textarea*. Estas clases, con la ayuda de los selectores *:host* y *::ng-deep*, permiten al componente *tfg-input* dar estilo al contenido proyectado.

Es la etiqueta *span* con la clase *highlight* la que permite al componente cambiar de color mediante una animación cada vez que es pulsado. Esta animación está hecha para funcionar en la mayoría de los navegadores, ya que ha sido definida en la hoja de CSS con las extensiones necesarias para que funcione en Firefox (*@-moz-keyframes*), y en Safari y Chrome (*@-webkit-keyframes*). La etiqueta *span* con la clase *underline* crea una línea transparente encima del borde inferior del *input*,

por lo que a primera vista no se ve. Sin embargo, cuando el usuario pulsa el botón, se activa esta línea siguiendo una animación, quedándose del color azul mencionado. Por último, la etiqueta *label* es la que define el *label* del *input*, el cual deberá moverse a su posición encima del texto introducido cada vez que el usuario pulse el *input*.

Este elemento *tfg-input* no recibe argumentos como tal, sino que al usar una directiva, tiene acceso a los atributos del contenido proyectado del usuario, por lo que puede efectuar cambios en el estilo y funcionamiento del campo de texto.

3.6. *SELECT*

3.6.1. DESCRIPCIÓN

La etiqueta *select* [34] es utilizada en los formularios para crear listas desplegables. Dentro de ese desplegable, se tendrá una serie de opciones que el usuario podrá seleccionar. Su funcionamiento es similar al del componente *tfg-single-selection-group*, del cual se habló en el punto 3.4.1. La gran diferencia es que con la etiqueta *select*, las opciones están en un primer momento escondidas, lo que facilita la selección dentro de grandes listas de elementos, y mejora así mismo la legibilidad de un formulario, permitiendo que de un vistazo rápido se pueda apreciar toda la información que se requiere del usuario.

Un ejemplo de su uso puede ser, en un formulario de compra, cuando se tiene que escribir la dirección de destino, un desplegable que tenga todos los países del mundo, para que el usuario pueda escoger. Otro ejemplo sería el de una comanda a un restaurante, donde el usuario tiene que escoger qué *toppings* quiere que tenga su plato. Estos ejemplos muestran el valor añadido de la etiqueta *select* en comparación con una simple lista de botones: el usuario sabe que tiene que seleccionar una opción, pero lo hace poco a poco, y sin que ocupe todo su campo de visión, quitándole importancia al resto del pedido o de la aplicación.

Al igual que con muchos de los componentes que se han definido hasta ahora, la etiqueta *select* cobra sentido cuando se usa en el contexto de un envío de formulario, ya que requiere una acción por parte del usuario, la cual luego será evaluada en el servidor o donde sea necesario. La etiqueta *select* suele tener el siguiente aspecto [35]:

```
1     <select name="text">
2       <option value="first">First Value</option>
3       <option value="second" selected>Second Value</option>
4       <option value="third">Third Value</option>
5     </select>
```

Código 18: Ejemplo etiqueta *select*

Cada *select* está compuesto por una lista de etiquetas *option*, las cuales tienen dos elementos fundamentales: el *value*, que va a ser el valor que es tratado por el formulario, y entre las etiquetas, el texto que será leído por el usuario. Con esta información, el usuario escoge la opción que mejor le convenga, y el desarrollador de la aplicación tendrá entonces acceso a un *value* que podrá tratar y que será el que será enviado por el formulario.

Para llevar a cabo la creación de este componente, la primera opción fue la de proceder como se hizo con el *input* y el *button*, es decir, un componente que proyectara el contenido del usuario, para permitir a este que le sacara todo el partido posible. Sin embargo, se ha decidido crear un componente propio, aportando suficiente valor añadido como para que sea rentable su uso en una aplicación. A continuación se puede ver la plantilla:

```
<div class="select">
    <select class="tfg-select"
            #select
            (change)="removePlaceholder(select.value)"
            [ngClass]="{'active': disablePlaceholder}"
            (focus)="userDefault()"
            (blur)="emptyDefault()">
        <option *ngIf="disableDefaultOption"
                value=""
                [disabled]="userResetValue"
                [selected]="userResetValue"
                [hidden]="userResetValue">
            {{ showDefault ? resetValue : '' }}
        </option>
        <option *ngFor="let userValue of userValues"
                [selected]="userValue.value === defaultValue"
                value="{{ userValue.value }}">
            {{ userValue.viewValue }}
        </option>
    </select>
    <span class="highlight"></span>
    <span class="underline"></span>
    <label>{{ placeholder }}</label>
</div>
```

Código 19: Plantilla del componente *tfg-select*

Se puede observar inmediatamente el cambio en comparación con las plantillas de otros componentes. En este caso, la plantilla se complica porque hay que tener en cuenta varios detalles que se irán detallando a continuación.

Para empezar, debido al hecho que el usuario no va a tener control directo sobre el *select*, sino sobre el componente *tfg-select*, la primera tarea del usuario es definir un objeto que utilizará para pasar la información al componente. Este objeto es lo suficientemente simple como para que no sea laborioso tener que lidiar con el paso de información hacia el componente. En realidad se trata de una lista de objetos, cada uno de ellos con dos claves: *value* y *viewValue*. Como se comentó anteriormente, estos dos valores son los más básicos a la hora de definir las opciones de un *select*. El valor de la clave *value* se corresponde directamente con el atributo *value* de la etiqueta *option*, y el valor de la clave *viewValue* será el texto que se le mostrará al usuario.

Esta manera de definir las opciones supone una gran ventaja para el usuario que use este componente. No solamente le permitirá trabajar con un formato cómodo, sino que además, lo podrá definir desde la lógica de su componente propio, separando mejor las responsabilidades entre la plantilla y el modelo (ver punto 2.2). Además, al estar definido como *inputs* al componente *tfg-input*, el usuario podrá cambiar las acciones dinámicamente y podrá tener la certeza de que los valores correctos serán mostrados en todo momento, gracias entre otros a los *lifecycle hooks* de Angular.

Esta lista de objetos será pasada al componente como un argumento, y se renderizará con la ayuda de la directiva *ngFor*, la cual se encargará de recorrer la lista, y de poner los valores donde correspondan dentro de la plantilla.

La manera de comunicar el elemento seleccionado al usuario se hace de la misma manera que se ha venido describiendo hasta ahora: cada vez que el usuario realiza una selección, nueva o por primera vez, el componente emitirá que deberá ser captado por el usuario para que lo pueda tratar. Al igual que pasaba con el componente *tfg-single-selection-group* (ver punto 3.4.1), el hecho de que sea una selección simple permite que el valor emitido sea bastante simple. En este caso, el valor que se emite es simplemente el del *value* correspondiente a la opción seleccionada por el usuario. El hecho de que el valor emitido sea tan simple (un simple *string*) funciona también a favor del usuario: cuanto más simple sea el evento, más fácil le resultará al usuario tratarlo, por lo que aumenta el valor añadido que puede proporcionar este componente.

En temas de estilo, este componente resulta muy parecido al estilo que se le ha proporcionado al componente *tfg-input* que engloba las etiquetas HTML *input* y *textarea*. No se volverá entrar en detalle, ya que se ha explicado en el punto 3.5.1. Básicamente, se encuentra una etiqueta *span* con la clase *highlight* que se encargará de realizar la animación cada vez que el usuario pulse sobre el componente, una etiqueta *span* con la clase *underline* que resaltará la línea inferior del componente, y

una etiqueta *label* donde aparecerá el texto que indica para que sirve el componente sin que el usuario tenga que haberlo pulsado. El resto del estilo está hecho de tal manera que se esconda todo el estilo por defecto que le dan los diferentes navegadores, ya que ninguno de estos estilos se asemejan ni son comparables al estilo final si se sigue correctamente la guía de estilo de *Material Design*.

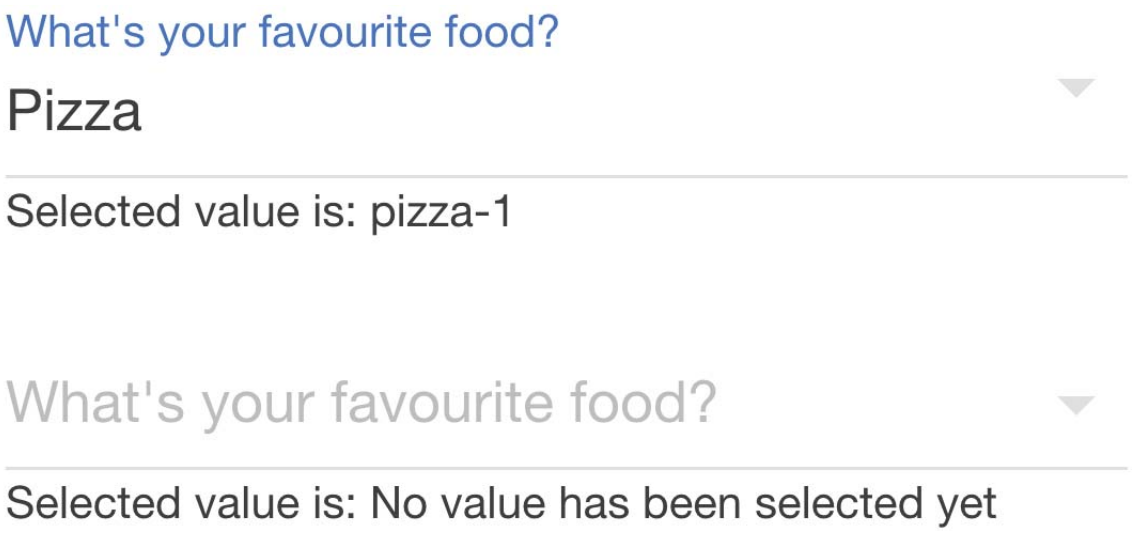

Figura 21: Componentes *tfg-select*, uno con opción por defecto y el otro sin

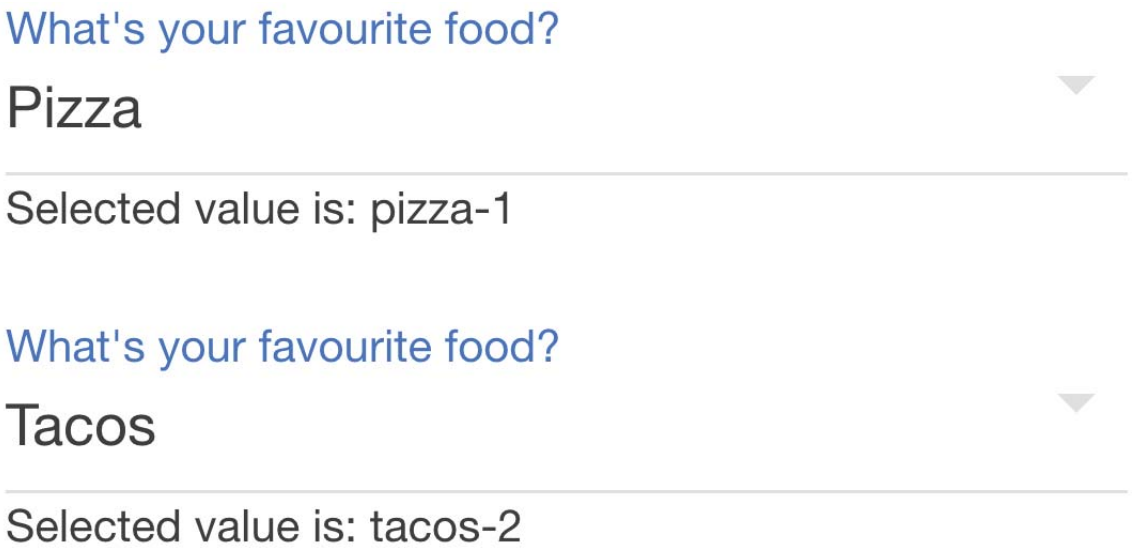

Figura 22: Componentes *tfg-select*, con opción seleccionada en los dos

3.6.2. *API*

En la plantilla del componente se puede apreciar una gran cantidad de variables y elementos que no han sido discutidas hasta ahora. Estas variables dependen de los diferentes *inputs* o argumentos que puede recibir este componente para alterar su funcionamiento y/o estilo. A continuación se listan todos los argumentos que puede recibir:

1. *placeholder*
2. *values*
3. *setSelectedValue*
4. *resetValue*

El primer argumento es el *placeholder*. Es el argumento más simple de tratar, ya que simplemente se comprueba si existe o no, y se pone en la plantilla, gracias a la interpolación de variables que proporciona Angular: *{{ placeholder }}*, tendrá el valor que el usuario le asigne, si no le ha asignado nada, se le asigna una *string* vacía.

El segundo argumento es el *values*. El valor de esta variable es en realidad el listado de objetos que define el usuario que quiera usar este componente. Tiene que ser de la siguiente forma:

```
1     foods: Object[] = [
2         { value: "steak-0", viewValue: "Steak" },
3         { value: "pizza-1", viewValue: "Pizza" },
4         { value: "tacos-2", viewValue: "Tacos" }
5     ];
```

Código 20: Ejemplo de valores que debe recibir el *tfg-select*

Esta lista es obligatoria que sea proporcionada por el usuario, ya que sino el componente da un error diciendo que no encuentra el objeto. Esto es lógico, ya que sin estas opciones, no tiene sentido el uso de este componente. Una vez que el componente las tiene disponibles, la plantilla se encargará de renderizarlas gracias a la directiva de Angular *ngFor*, como se puede observar en la plantilla del componente que se ha puesto en el punto anterior.

El siguiente argumento es el *setSelectedValue*. Este argumento es opcional, y permite definir cual es el valor por defecto que se quiere que esté seleccionado al inicializar el componente.

Lo primero que se hace es comprobar que el valor que ha recibido el componente se corresponde con uno de los valores existentes del componente. Si no se corresponde con ninguno, el componente procede como si no hubiera recibido dicho argumento. Para no dar ninguna confusión al usuario, se emite el valor del elemento seleccionado, que en este caso sería una *string* vacía, indicando al desarrollador que no se ha encontrado una correspondencia entre su valor y su lista de valores. En caso de que exista una correspondencia, se activa el atributo *selected* de la etiqueta *option*, y se emite un evento nada más inicializar el componente permitiendo al usuario tener directamente el valor por defecto seleccionado. La activación de este atributo se puede apreciar en la plantilla del punto anterior 3.6.1.

El último argumento es el *resetValue*. Al igual que el argumento anterior, este también es opcional. Este argumento permite al usuario crear una opción adicional que, cuando sea seleccionada, haga que el componente vuelva a su estado inicial. Este estado inicial, en caso de que el usuario haya definido un valor por defecto, será esta valor. En caso contrario, será una *string* vacía, indicando que no se ha seleccionado nada. La lógica a seguir para este argumento depende totalmente del usuario que lo use, ya que este argumento solo tiene sentido en el caso en el que la selección de un valor no sea obligatoria, o exista un valor por defecto.

Este último argumento es el que justifica la existencia de una etiqueta *option* en la plantilla de este componente. Esta opción solo aparece cuando el componente recibe este argumento, gracias a la directiva de Angular *ngIf*. Todos los atributos adiciones que se observan en esta opción sirven a ocultarla en dos momentos específicos: cuando el argumento no es llamado, y cuando el usuario selecciona esa opción. El problema es que si se selecciona esta opción, el *label* bajará a su estado inicial, puesto que hemos devuelto el componente a su estado inicial. Por lo tanto, cada vez que se seleccione esta opción, automáticamente desaparecerá de las opciones disponibles (las cuales no vemos en ese momento porque el *select* está cerrado), y así se evita que choquen los dos textos.

Todos estos argumentos, opcionales o no, sirven para darle un valor añadido a lo que puede ser una simple etiqueta *select*. Se podría haber procedido como para el *tfg-input* o el *tfg-button*, pero se ha preferido hacer de esta manera ya que el funcionamiento y la lógica de un *select* se puede ir rápidamente de las manos, y quitar al usuario esa preocupación, además de proveer un estilo que sigue la guía de *Material Design* es razón suficiente para pensar en utilizar este componente *tfg-select* en vez de la simple etiqueta HTML.

3.7. *BREADCRUMBS*

3.7.1. DESCRIPCIÓN

El componente *tfg-breadcrumbs* se puede traducir como "migas de pan". No tiene una correspondencia específica con un componente HTML, como lo tenían hasta ahora la mayoría de los componentes. En la figura siguiente (23) se puede ver un ejemplo de su uso en el *framework Materialize* [36]:

Figura 23: Ejemplo de un componente *breadcrumbs* del *framework Materialize*

Este componente puede ser utilizado tanto en formularios como fuera de ellos, aunque se ha ideado con la intención de que se utilice dentro de los formularios. Más que ayudar a pedir o recibir información sobre el usuario, se trata de una ayuda visual para que el usuario sepa en qué paso se encuentra. Un ejemplo de uso puede ser dentro de un formulario de compra, se empieza pidiendo la información del usuario, a continuación la dirección, y por último la información del método de pago. Muchas veces, estos pasos se encuentran en vistas diferentes, por lo que este componente ayuda a saber el paso en el qué se encuentra, viendo directamente el paso en el que se encuentra, por donde a pasado, y que le falta para completar su compra. La plantilla de este componente es la siguiente:

```
1     <nav class="breadcrumb-nav">
2         <ol class="tfg-breadcrumb"
3             [style.background-color]="changeBckgrdColor()">
4             <li class="tfg-breadcrumb-item"
5                 [ngClass]="{'active': i === userValues.length - 1}"
6                 *ngFor="let userValue of userValues; let i = index">
7                 <a [style.color]="changeLinkColor()"
8                    *ngIf="i !== userValues.length - 1"
9                    href="{{ userValue.url }}">
10                    {{ userValue.viewValue }}
11                </a>
12                <span *ngIf="i === userValues.length - 1">
13                    {{ userValue.viewValue }}
14                </span>
15            </li>
16        </ol>
17    </nav>
```

Código 21: Plantilla del componente *tfg-breadcrumb*

El componente *tfg-breadcrumb* está definido por una etiqueta HTML *nav* [37], la cual sirve para definir un listado de enlaces de navegación. En su interior, se encuentra una lista ordenada (etiqueta HTML *ol*), donde se definirá, por cada una de las opciones que el usuario desee mostrar, el elemento que corresponda, siendo separado cada uno de los enlaces por el carácter "/".

Este componente está definido casi en su totalidad gracias a los argumentos que recibe por parte del usuario, por lo que se pasa directamente a la descripción de su *API*, donde se describirá más en profundidad su funcionamiento.

3.7.2. *API*

Al ser un componente simple y que puede resultar de gran ayuda para el usuario, se ha intentado que la *API* de este componente pueda cambiar la mayoría de elementos que forman este componente. De esta manera, se tienen los siguientes argumentos:

1. *values*
2. *backgroundColor*
3. *linkColor*

En primer lugar se tiene el argumento más importante de todos. Su uso es obligatorio, ya que si el usuario no especifica este argumento, el componente lanzará un error. Este argumento recibe los valores que el usuario quiere que se muestren en el componente.

El componente recibe una lista de objetos con dos claves cada uno: la primera es la *url*, la cual definirá a donde se quiere mandar al usuario cuando pinche en el enlace, y la segunda es *viewValue*, que indica el valor que se le mostrará al usuario dentro del *tfg-breadcrumb*. Al igual que para el componente *tfg-select* (ver punto 3.6.1), se ha intentado que la creación de este argumento sea lo más simple posible para el usuario, dejando solamente las partes fundamentales para el correcto funcionamiento de este componente. Al ser recibido como argumento, el valor de este objeto puede ser cambiado dinámicamente por el usuario, y Angular se encargará de que el valor mostrado sea siempre el correcto.

```
1     breadcrumbs: Object[] = [
2         { url: "/option1", viewValue: "Option 1" },
3         { url: "/option2", viewValue: "Option 2" }
4     ];
```

Código 22: Ejemplo del argumento *values* del componente *tfg-breadcrumb*

En la plantilla que se encuentra en el punto 3.7.1, se puede apreciar como la directiva de Angular *ngFor* recorre cada uno de los elementos. Se tiene además una directiva que no se ha visto anteriormente, *ngClass*, que añadirá una clase cada vez que se cumpla una condición. En este caso, se añadirá la clase *active* al último elemento de la lista, resaltando el paso en el que el usuario se encuentra actualmente.

El segundo argumento que puede recibir este componente es el *backgroundColor*. Este argumento es opcional, y define el color del fondo del componente. De esta manera, el usuario puede ajustar el color del componente al estilo de la aplicación que esté diseñando el usuario. Por defecto, el color del fondo del componente es gris (#e9ecef), siendo este un tono neutro que quedará bien con cualquier tipo de aplicación.

Figura 24: Componente *tfg-breadcrumb*, en el tercer paso de un formulario

Se observa en la plantilla del punto 3.7.1 como se asocia el estilo del *background-color* a un método que se ha definido en el componente: *changeBackgrdColor()*. Este componente confirmará si se ha recibido o no el argumento, y pondrá devolverá el color que tenga que tener el fondo del componente.

El tercer y último argumento es opcional, y sirve para cambiar el color de los enlaces. Está pensado para que el usuario pueda crear el componente con colores que queden bien juntos, y no tener que conformarse con los colores por defecto que ofrece. En la plantilla se asocia el color de la etiqueta *a* con un método definido en el componente que comprobará si se ha recibido o no el argumento, devolviendo el color que sea conveniente.

La *API* de este componente es bastante simple, y en general es un componente simple, que se usará no para el correcto funcionamiento de un formulario, sino para mejorar la usabilidad de este.

3.8. *CHECKBOX*

3.8.1. DESCRIPCIÓN

El componente *tfg-checkbox* no tiene una correspondencia directa con una etiqueta HTML, aunque sí hace alusión a una manera muy común de utilizar la etiqueta *input*. Como ya se comentó en el punto 3.5, donde se hablaba de la etiqueta

input, esta etiqueta puede recibir distintos valores en su atributo *type*. Para no limitar el funcionamiento de esta etiqueta, se hizo de tal manera a que el usuario pudiera definir su etiqueta *input* sin problemas, y para el componente *tfg-input* eso sería contenido proyectado.

Uno de los casos particulares donde sí vale la pena crear un nuevo componente es cuando el usuario quiere utilizar un *checkbox*:

```
1     <label class="container">One
2         <input type="checkbox" checked="checked">
3         <span class="checkmark"></span>
4     </label>
```

Código 23: Ejemplo de un *input* de tipo *checkbox*

El ejemplo más conocido de un *checkbox*, y que todo el mundo se ha visto obligado a usar alguna vez, es el de utilizarlo para que el usuario pueda aceptar los términos y condiciones de una aplicación o página web.

Figura 25: Componente *tfg-checkbox*

Figura 26: Componente *tfg-checkbox*, seleccionado

La función de un *checkbox* es de tener dos opciones: verdadero o falso, pulsado o no pulsado, uno o cero. Este funcionamiento es el mismo que el del componente *tfg-button-toggle*, visto en el punto 3.3. Lo que cambia entre uno y otro es el caso de uso, pero esto ya depende del usuario, y de como quiera implementar su formulario.

La gran diferencia que existe entre los dos componentes es la posibilidad del *checkbox* de tener un estado adicional, accionable solo mediante Javascript, es decir, que no se puede definir este estado mediante HTML. Este estado es el estado *indeterminate*.

El estado *indeterminate* es solo un estado visual, es decir, que el *checkbox*, cuando esté en ese estado, estará activado. El uso de este estado tiene sentido cuando se está haciendo una lista de *checkboxes*, lista al estilo del componente *group toggle* (ver punto 3.4). El *checkbox* de mayor nivel, el cual agrupa una lista de *checkboxes*, solo estará activo completamente cuando se hayan seleccionado todos los *checkbox* en su interior. Hasta entonces, estará desactivado, en el estado *indeterminate*.

Para crear el componente *tfg-checkbox*, se ha creado la siguiente plantilla:

```
1     <div class="tfg-checkbox">
2         <input type="checkbox"
3                id="tfg-checkbox-{{ id }}"
4                class="input-checkbox"
5                [ngClass]="{'indeterminate' : indeterminate,
6                            'disabled': disabled}"
7                [checked]="userCheck"
8                (change)="emitCheck($event.target.checked)">
9         <label for="tfg-checkbox-{{ id }}"
10               class="label-checkbox">
11            <ng-content></ng-content>
12        </label>
13    </div>
```

Código 24: Plantilla del componente *tfg-checkbox*

En esta plantilla se puede observar cuales son las etiquetas que forman un *checkbox*: una etiqueta *div* que define el tamaño y ocupación del componente, un *input* de tipo *checkbox*, del cual se hablará más adelante, y una etiqueta *label* que va a ser la que mostrará el mensaje que el usuario quiere que se renderize al lado derecho de la caja. Este contenido lo decide el usuario, y será contenido proyectado para el componente, el cual lo capturará con la etiqueta *ng-content*.

Cada *input* de tipo *checkbox* emite un evento cada vez que un usuario pulsa en él. De esta manera, y para facilitar la tarea al usuario, el componente *tfg-checkbox* captura este evento, para luego emitir su propio evento dirigido al usuario. Este nuevo evento está compuesto por dos valores: el valor del *checkbox*, un *boolean* a verdadero o falso, y una *string* con el identificador de dicho *checkbox*. Este identificador es de vital importancia, ya que permite usar diferentes *checkbox* dentro de una misma vista. Este identificador se encarga de unir la etiqueta *input* con el *label*, de tal manera que los dos están asociados.

A diferencia del componente *tfg-toggle-group* (punto 3.4), el componente *tfg-checkbox* no tendrá un componente propio para hacer grupos. Esta tarea se le deja al usuario, para que tenga la libertad de escoger cómo prefiere hacerlo. Aún

así, viendo lo parecido que es el funcionamiento entre el *tfg-checkbox* y el *tfg-button-toggle*, el usuario tiene amplias opciones para llevar a cabo su formulario con este *framework*.

3.8.2. *API*

El funcionamiento del componente *tfg-checkbox* está en gran parte definido por su *API*. Se consigue con esto darle total control al usuario que quiera utilizar este componente.

Los argumentos que puede recibir este componente son:

1. *id*
2. *disabled*
3. *indeterminate*
4. *initialValue*
5. *userCheck*

El primer argumento, *id*, es el identificador del *checkbox*. Como se comentó en el punto anterior (3.8.1), el modo de funcionamiento de los *checkbox* hace que sea obligatorio tener un identificador único en cada uno de ellos. Esto es debido a la forma en la que se define un *checkbox*: la etiqueta *input* y la etiqueta *label* están unidas. El identificador se define en el atributo *id* del *input*, y luego desde el *label* se usa el atributo *for* para indicar a quién corresponde. Esto hace que el este atributo sea obligatorio, y para diferenciarlo de otros identificadores que pueda haber en la aplicación, se le añade un prefijo: *tfg-checkbox-*. Este modo de funcionamiento se puede apreciar en la plantilla del componente (ver punto 3.8.1).

El segundo argumento es el *disabled*. Como en otros componentes, este argumento es opcional, y sirve solamente para que el usuario pueda decidir qué *checkboxes* quiere que estén activos y cuales no. En el caso de que el usuario decida utilizar este argumento, se añadirá automáticamente una clase al componente para dar el estilo y la funcionalidad necesaria. Esto se hará a través de la directiva *ngClass*. Lo bueno de este argumento, y su uso con la directiva, es que el usuario podrá cambiar su valor dinámicamente, durante la ejecución de su aplicación, y el componente responderá conforme al valor que reciba.

El tercer argumento es *indeterminate*. Este argumento es opcional, y va a permitir al usuario comunicar al componente si desea utilizar o no el tercer estado del *checkbox*. El único cambio será en el estilo del botón, y cuando esté en este estado, el componente emitirá un evento con su valor de activado a *true*. Como solo se trata de un cambio en el estilo, cuando el usuario use este argumento, se añadirá una clase *indeterminate* a la etiqueta *input* del componente mediante la directiva *ngClass*.

El cuarto y quinto argumento trabajan de manera conjunta: *initialValue* y *userCheck*. Estos argumentos sirven para tratar el estado inicial del componente. De esta manera, el usuario puede definir acciones por defecto en sus *checkboxes*, facilitando la usabilidad del componente y de la aplicación en general.

La manera de funcionar de este componente ha sido ideada de tal manera que el usuario no tenga ninguna dificultad añadida para juntarlo con otros componentes. En el siguiente punto se podrá ver uno de los muchos usos que se le puede dar a este componente *tfg-checkbox*.

3.9. TABLE

3.9.1. DESCRIPCIÓN

El componente *tfg-table* tiene una correspondencia con una etiqueta existente HTML: la etiqueta *table* [38]. Una tabla está diseñada para mostrar información o datos. Esta información será mostrada en una serie de filas y columnas, definidas por el usuario, que permitirán mostrar información de manera ordenada al usuario.

El HTML de una tabla es bastante simple, aunque se puede complicar muy rápido:

```
1     <table>
2         <tr>
3             <th>Month</th>
4             <th>Savings</th>
5         </tr>
6         <tr>
7             <td>January</td>
8             <td>$100</td>
9         </tr>
10        ...
11        <tr>
12            <td>December</td>
13            <td>$200</td>
```

```
        </tr>
    </table>
```

Código 25: Ejemplo etiqueta *table*

En este ejemplo, se aprecian las partes más importantes de la etiqueta *table*. La tabla está compuesta por filas, y dentro de cada fila se define las columnas que se desea que hayan. En este caso, hay dos columnas, y en la primera fila se define la *table head* con la etiqueta *th*. El resto de filas contienen datos, por lo que se definen con *table data* o la etiqueta *td*.

El uso de una tabla en un formulario no suele ser muy habitual, pero existen casos de uso. El componente *tfg-table* intenta facilitar uno de estos casos, con la ayuda del componente *tfg-checkbox* (ver punto 3.8). Su plantilla demuestra hasta qué punto el diseño de una tabla se puede complicar, y por qué es necesario ofrecer un componente que se encargue de ello:

```
    <table class="my-table table-hover">
        <thead class="table-head">
            <tr>
                <th class="checkbox" *ngIf="checkboxes">
                    <tfg-checkbox
                        [id]="'master'"
                        (checked)="handleMasterCheckbox($event)"
                        [userCheck]="selectAll && !this.unselected">
                    </tfg-checkbox>
                </th>
                <th class="row-header truncate {{ column.key }}"
                    *ngFor="let column of columns | keys">
                    {{ column.value }}
                </th>
            </tr>
        </thead>
        <tbody>
            <tr class="element" *ngFor="let row of rows; index as i;">
                <td class="checkbox {{ element.key }}"
                    *ngIf="checkboxes">
                    <tfg-checkbox
                        [id]="i"
                        (checked)="handleCheckbox($event)"
                        [userCheck]="selectAll"
                        class="single-checkbox">
                    </tfg-checkbox>
                </td>
                <td class="element-box"
                    *ngFor="let element of row | keys">
                    {{ element.value }}
```

```
31                  </td>
32              </tr>
33          </tbody>
34      </table>
```

Código 26: Plantilla del componente *tfg-table*

A diferencia del ejemplo de uso de la etiqueta *table*, en la plantilla del componente *tfg-table* se define explícitamente cual es la cabeza y qué es lo que forma el cuerpo de la tabla. El diseño de la tabla está basado, como todos los componentes, en la guía de estilo de *Material Design*. Para esto, se ha tenido que quitar todo el estilo por defecto de los navegadores, añadir las distintas clases que se observan en la plantilla, para conseguir que se asemeje a una tabla de Google.

No.	Name	Weight	Symbol
1	Hydrogen	1.0079	H
2	Helium	4.0026	He
3	Lithium	6.941	Li

Figura 27: Componente *tfg-table*, en su modo más simple

3.9.2. *API*

El componente *tfg-table* está definido en su totalidad a partir de los argumentos que recibe por parte del usuario. Estos argumentos son los siguientes:

1. *rows*
2. *columns*
3. *checkboxes*
4. *name*

El primer elemento es el que define los datos que serán renderizados en la tabla. Para facilitar el uso de este componente, la forma en la que estos datos deben ser definidos es mediante una lista de objetos. De esta manera, la gran parte de componentes que necesitan datos del usuario para poder ser renderizados correctamente es siempre la misma, mejorando la usabilidad del *framewrok*.

Esta lista de objetos tendrá los valores que el usuario desee poner. Es decir, el usuario tiene total libertad a la hora de definir la información que quiere enseñar en su formulario. La única limitación es que tendrá que estar relacionado este objeto con el objeto que define las columnas. Este argumento será renderizado en el cuerpo de la tabla, gracias a la directiva de Angular *ngFor*. Las claves de los objetos servirán para añadir una clase a cada casilla de la tabla, para que el usuario, en caso de que lo quisiera, tenga acceso, y el valor será lo que el usuario verá renderizada.

El segundo argumento representa las columnas que tendrá la tabla. Este argumento es obligatorio, y está formado por un objeto clave-valor. Tendrá todos los elementos que el usuario quiera poner como columnas, tal que la tabla sea tan grande como el usuario desee. Las claves del objeto serán utilizadas como clases para la casilla donde se encuentran, y el valor de cada clave será la información que verá el usuario como título de la columna. Tiene que haber el mismo numero de columnas en este objeto, como claves dentro de cada objeto del argumento anterior, para que el componente pueda renderizar correctamente la información.

El tercer argumento es *checkboxes*. Este argumento es un *boolean* que indicará al componente si el usuario quiere usar o no una columna adicional. En esta columna, que será la primera columna por la izquierda, se encontrará, por cada fila, un componente *tfg-checkbox*.

	No.	Name	Weight	Symbol
☐	1	Hydrogen	1.0079	H
☐	2	Helium	4.0026	He
☐	3	Lithium	6.941	Li

Figura 28: Componente *tfg-table*, junto con el componente *tfg-checkbox*

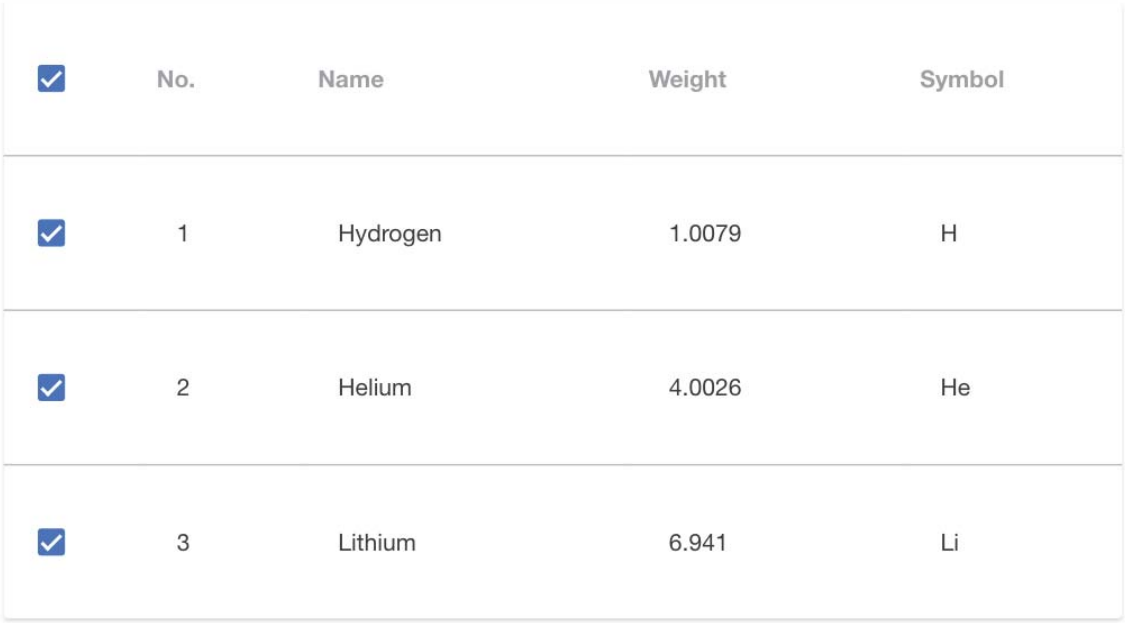

Figura 29: Componente *tfg-table*, junto con el componente *tfg-checkbox*, todos ellos seleccionados

Estos *checkboxes* tienen dos tipos distintos. Los que se encuentran en la parte del cuerpo de la tabla, cada vez que sean pulsados, emitirán un evento para que sea recogido por el usuario. Este evento estará compuesto por dos valores: el primero es un *boolean* que indica si el *checkbox* está pulsado o no, y el segundo es el número de fila que ha sido pulsada. Para el usuario, al tener en el mismo sitio el objeto que define el cuerpo de la tabla, y el método que maneja este evento, emitir el número de fila que ha sido seleccionada es cómodo, ya que no hay información repetida, y tiene acceso directo a su propio objeto.

El segundo tipo de *checkbox* es el que se encuentra en la primera fila, donde se definen los títulos de las columnas. Este tiene un funcionamiento diferente al resto: cada vez que sea pulsado, se permitirá seleccionar todas o ninguna de las filas, cambiando el estado de los *checkboxes*. Este *tfg-checkbox* emite un evento con dos elementos: el nombre de la tabla, indicando al usuario que toda la tabla ha sido seleccionada, y un *boolean* indicando si está seleccionado o no.

El cuarto y último argumento es *name*. Este argumento es obligatorio si se quieren usar las *checkboxes*, ya que, como se ha comentado, en caso de seleccionar todas las casillas, el evento que se emite es este argumento.

El funcionamiento de este componente, cuando recibe el argumento *checkboxes*, resulta parecido al componente *tfg-toggle-group* (ver punto 3.4). La diferencia de estilo entre los dos, además de la funcionalidad específica que tiene cada uno,

permite al usuario tener un gran abanico de posibilidades a la hora de crear listados de opciones en su formulario.

3.10. COMPILACIÓN Y DESPLIEGUE

No se ha comentado nada al respecto hasta este punto del trabajo, pero Angular posee herramientas que permiten la creación de un artefacto para su posterior despliegue.

Este artefacto puede ser creado para trabajar en *development*, o un entorno de pruebas y desarrollo, y para *production*, que sería para un despliegue real de la aplicación. No se entrará mucho en detalle sobre el procedimiento que sigue Angular para crear estos artefactos, pero sí se comentará lo que se usa para compilar este proyecto:

```
1    ng build --prod
```

Código 27: Llamada a la *angular-cli* para compilar el proyecto

En este caso, la línea de comandos de Angular, cuando se llama al comando *build*, crea un artefacto para su posterior despliegue. Especificando la opción *–prod*, Angular se encarga de crear el artefacto con las siguientes optimizaciones [39]. Estos elementos hacen que el artefacto final que es creado esté lo más optimizado posible:

- Compilación *Ahead-of-Time*: Pre-compila las plantillas de los componentes.
- Despliegue del entorno de compilación activando el modo de producción.
- *Bundling*: Concatena las aplicaciones y las librerías en paquetes.
- *Minification*: Borra espacios en blanco, comentarios y otras cosas que no intervengan en el funcionamiento de la aplicación.
- *Uglification*: Modifica el código para usar nombres de variables y funciones más cortas y sin sentido, reduciendo el tamaño del código y mejorando el rendimiento.
- *Dead code elimination*: Elimina módulos y código no usado.

El despliegue de la aplicación se hace con la ayuda de la herramienta Travis CI [40]. Travis es una herramienta de integración continua, y es de gran uso en este proyecto debido a la facilidad de usarlo junto a proyectos alojados en Github [41]. Este proyecto está alojado en Github, de manera privada hasta que se entregue, momento donde se pensará en liberar el código.

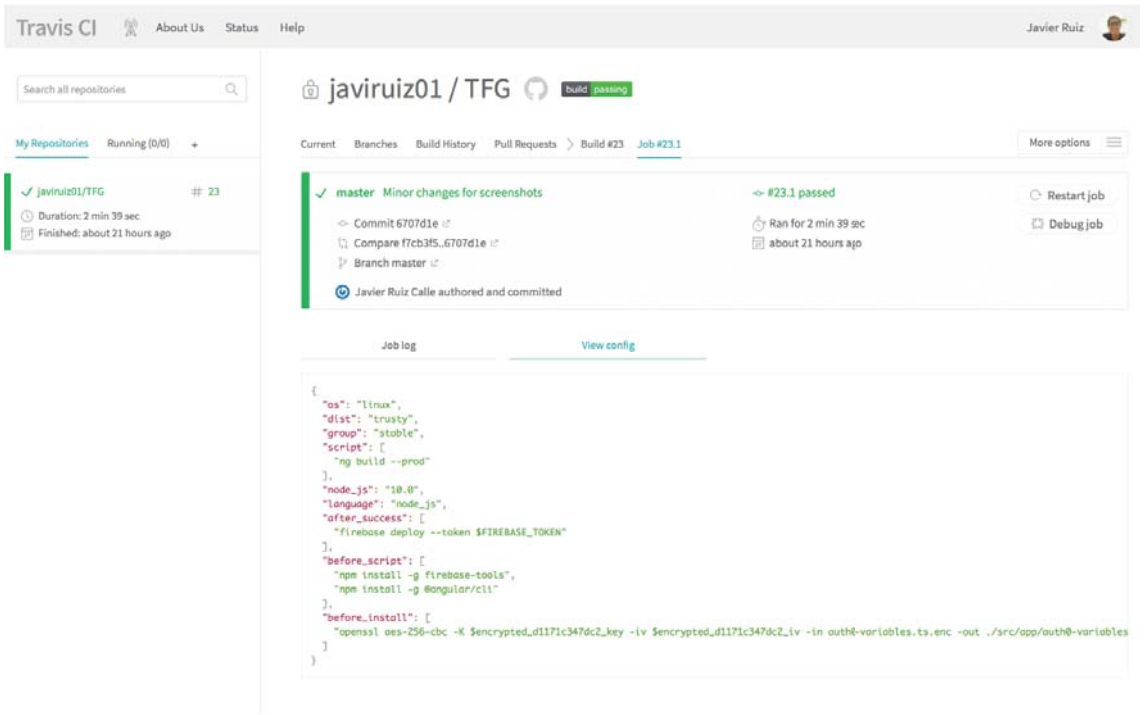

Figura 30: Panel con configuración para la herramienta de integración continua Travis CI

Cada vez que se realiza un *commit* nuevo en el proyecto, y se envía a Github, Travis-CI se encarga de realizar una serie de *scripts* para probar que todo está correcto. En el caso de este proyecto, lo único que va a hacer es compilar el proyecto para un entorno de producción, permitiendo saber con cada *commit* que el proyecto está listo para ser desplegado.

Además de esto, Travis-CI permite también automatizar el despliegue de una aplicación. En este caso, este despliegue se hará sobre Firebase de Google [42]. Lo que ofrece Firebase es un servicio de *hosting* en su capa gratuita, por lo que es perfecto para desplegar el proyecto. Este despliegue se hace a través de Travis-CI con un simple comando en el *script* de Travis. Firebase ofrece un dominio propio para que los usuarios puedan ver su proyecto desplegado, pero también permite poner un dominio propio, facilitando el tipo de DNS que se tiene que añadir en el dominio, y su contenido para que todo funcione correctamente. De esta manera, a la hora de escribir esta memoria, el proyecto está desplegado en la siguiente dirección: https://tfg.javi.engineering.

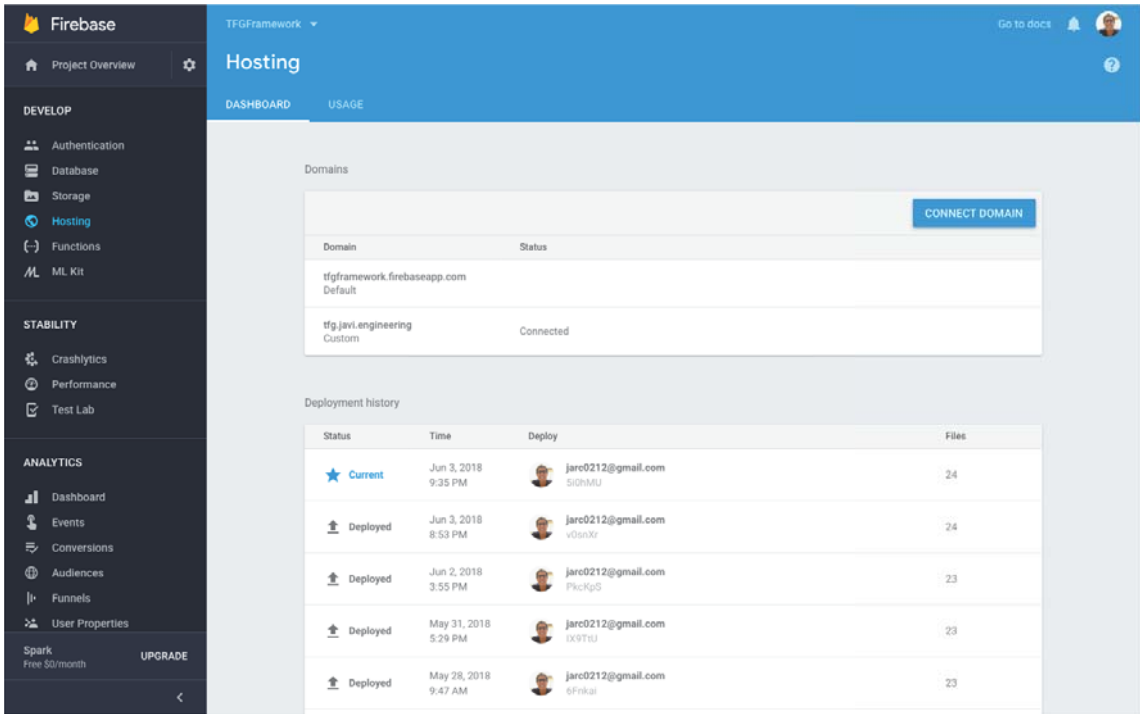

Figura 31: Panel de Firebase con el histórico de despliegue del proyecto

4. *TESTING*

El *testing* es una de las partes más importantes a la hora de desarrollar una aplicación nueva. Para este proyecto, se ha llevado a cabo de dos maneras distintas, las cuales se explicarán a continuación.

La primera manera de probar los diferentes componentes ha sido realizada durante su creación. Debido a que se ha intentado que los componentes sigan todo lo que puedan la guía de estilo de *Material Design*, se han empezado todos ellos por definir su lógica primero.

Una vez que la lógica de los componentes ha sido creada, llega el momento de dar el estilo al componente. Durante este paso, los pequeños fallos que van surgiendo en la lógica se hacen evidentes. Todos los componentes se ha intentado que funcionen dentro de una etiqueta *form*, aunque no se ha llegado a probar su uso directamente en un formulario funcional.

Esto es solo el primer paso para probar los componentes creados. El diseño no se ha probado, sino que se ha intentado que quede lo más parecido posible a los componentes que define Google. La lógica de los componentes, además, se ha probado con pequeñas pruebas unitarias, sobre todo para los componentes que reciben listas de objetos como argumentos. De esta manera, se comprueba que todos renderizan bien su contenido, siempre y cuando el contenido de los argumentos sea correcto.

Los componentes que lanzan eventos al usuario han sido probados de manera simple, definiéndolos en un componente externo que capturaba esos eventos e imprimía sus valores. Una vez que se tiene el valor del evento, se intenta ver qué tan complicado sería manejar el objeto que se recibe. Esto se ha hecho para mejorar la usabilidad de los componentes, permitiendo a los usuarios que trabajen de la forma más cómoda posible cuando usen los componentes de este proyecto.

La segunda manera en la que se ha creado ha sido, una vez se acabó de definir todos los componentes de este proyecto, de crear un formulario que existiera ya, y recrearlo con los componentes realizados durante el proyecto. Esto permite ver problemas en el diseño de los componentes, ya sea porque el diseño que se le ha dado es demasiado restrictivo, o porque no funciona como se esperaba.

De esta manera, se ha cogido en primer lugar la página de *login* de Stripe [43]. Se trata de un formulario simple de acceso a la aplicación, con dos campos para que el usuario ponga su usuario y contraseña, un *checkbox* para recordar al usuario, y un botón para enviar le formulario.

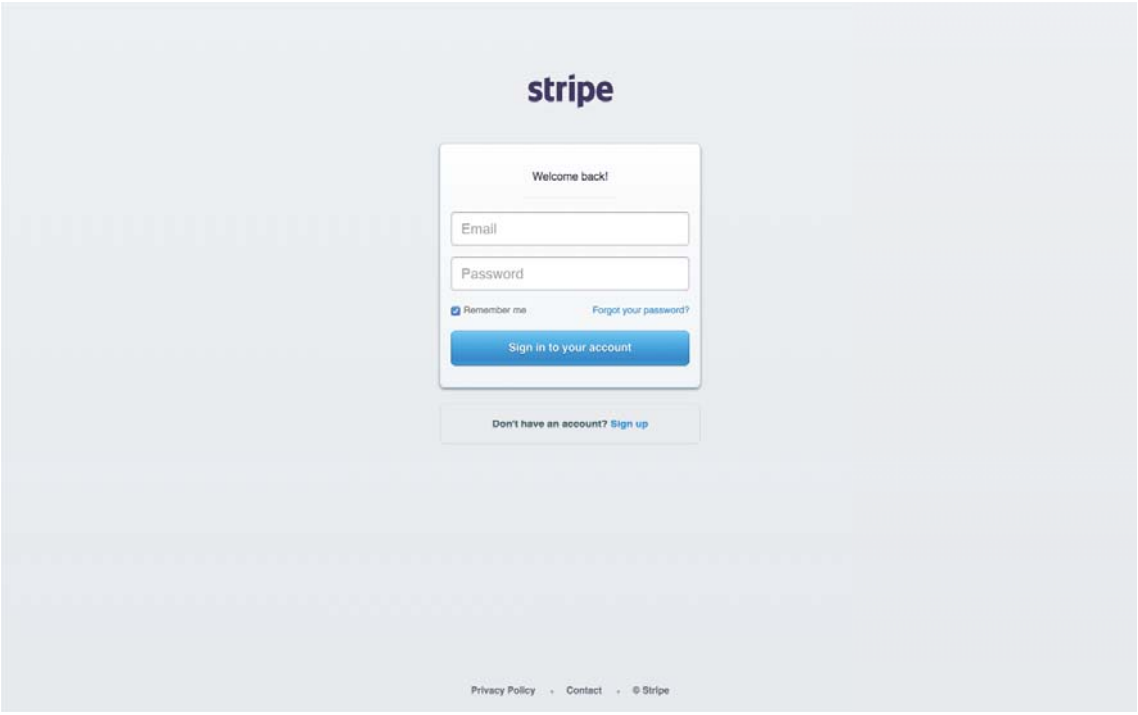

Figura 32: Portal de *login* de Stripe

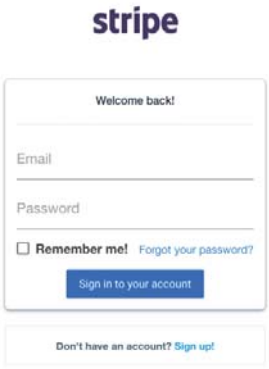

Figura 33: Portal de Stripe con componentes propios

Sin contar con el formulario, se ha intentado que los dos estilos de páginas se parezcan en todo lo posible, aunque con un estilo un poco más *Material Design*. Esta prueba ha servido para poder probar que los componentes estaban bien diseñados, que su uso era fácil, y que cada uno de los componentes funciona correctamente.

El otro gran ejemplo que se ha realizado ha sido el proceso de creación de un usuario en Airbnb [44].

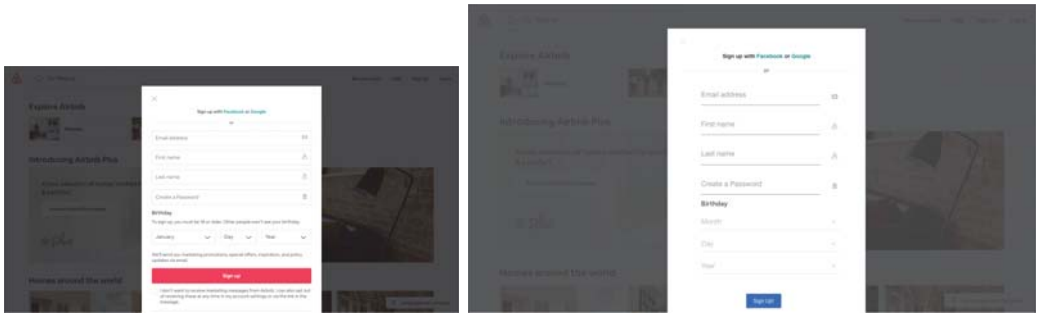

Figura 34: Página de creación de usuario de Airbnb

Esta prueba pone en evidencia uno de los mayores cambios que se pueden hacer al *framework* para mejorarlo enormemente: proporcionar un mayor control al usuario para que cambie el estilo más libremente. Se habla de estos en el punto sobre líneas futuras. A parte de este pequeño detalle, la lógica de los componentes funciona correctamente, por lo que por lo menos se puede asegurar que el *framework* funciona y está listo para ser usado.

5. *LÍNEAS FUTURAS*

El trabajo realizado para este trabajo de fin de grado (TFG) es la primera versión de algo que podría ser mucho más grande. Para este trabajo, se ha enfocado como un estudio hacia cómo se debería llevar a cabo la construcción de unos componentes web altamente reutilizables, pero en un futuro podría llegar a ser un *framework* útil y que a la gente le parezca que aporta el suficiente valor añadido como para que salga rentable usarlo.

El proyecto ha intentado ser realizado teniendo el mayor cuidado a la hora de definir los componentes, pensando siempre en la usabilidad de estos y cual sería la mejor manera de que sus usuarios estuvieran de acuerdo con las decisiones tomadas. Se ha dedicado tiempo desde cosas que pueden parecer tan pequeñas como los objetos que reciben los componentes, que mantengan una estructura similar a lo largo de todo el *framework*, los eventos que emiten cada uno de los componentes, pensando siempre en como el usuario tiene que tratarlos, hasta cosas más grandes como que el estilo sea lo suficientemente estándar para que una empresa o *start-up* pueda usar los componentes sin tener que comprometer su identidad corporativa.

Aún así, quedan cosas que pueden mejorar este proyecto enormemente. A continuación se plantean algunas mejoras que puede recibir esta aplicación:

- **Definición del estilo**: Actualmente, el estilo de los componentes está definido con CSS. Para un proyecto "pequeño" como el que se ha realizado, esto puede no tener mucho efecto, pero esto cambia si se desea seguir trabajando en este proyecto. Lo mejor sería utilizar un pre-procesador de CSS como puede ser *less* [45] o *sass* [46].

  Estas dos opciones son básicamente un lenguaje de *scripting* que extienden de CSS y luego son compiladas a CSS, por lo que no dan problemas a la hora de crear aplicaciones para navegadores antiguos. Tienen la ventaja de que permiten crear código menos repetitivo, y por lo tanto más legible, y permiten añadir lógica y cálculos en las hojas de estilo. Además, crean un código que es más fácil de mantener con el tiempo.

- **Definición por módulos**: El proyecto está definido de tal manera que si un usuario quisiera utilizar estos componentes, tendría que importar la totalidad del proyecto, aunque solo fuera para utilizar uno de esos componentes.

  Para mejorar esto, lo que se podría hacer sería definir cada uno de los componentes en su propio módulo (ver punto 2.3.4). Cada uno de estos módulos tendrían entonces encapsulados todos los servicios, lógica, hojas de estilo y

plantillas necesarias para que el usuario pudiera importar el *framework* componente a componente. Todos estos módulos "colgarían" del módulo raíz (*AppModule*), el cual definiría todo el proyecto.

- **Temas**: El estilo del proyecto, pese a que intenta seguir la guía de estilo de *Material Design*, no da mucha libertad al usuario de definir sus propios estilos. Se ha intentado que sea así, haciendo que los componentes puedan ser definidos por el usuario gracias a la proyección de contenido de Angular, pero sigue quedando trabajo en esta parte.

  A partir de aquí se pueden tomar varias decisiones. Una de ellas sería crear temas pre-definidos en el proyecto, y dejar que el usuario definiera cual de esos temas es el que prefiere usar para su aplicación. Estos temas podrían ser: *Dark mode*, modo oscuro, muy de moda últimamente en las aplicaciones para móviles, modo de alto contraste, para mejorar la accesibilidad de los componentes, entre otros. Otra idea darle al usuario la responsabilidad de definir una serie de clases que luego fueran a ser utilizadas por los componentes, de manera a que el usuario sabría en todo momento cual es el estilo que está siguiendo la aplicación, y puede mantener su imagen corporativa.

- **Más componentes**: Este proyecto ha sido orientado a realizar un estudio sobre como se puede mejorar el paso de información en una aplicación web gracias a los componentes web. Este paso de información se hace a través de formularios, por lo que los componentes creados se han centrado en proporcionar una utilidad dentro de la etiqueta *form*, aprovechándose de las ventajas que proporciona Angular para este tipo de proyectos.

  Sin embargo, todavía quedan componentes que no se han definido. Uno de ellos puede ser un componente que se encargue de manejar los errores de manera genérica, es decir, que se pudiera utilizar con cualquiera de los otros componentes ya creados para indicar al usuario cuando se ha equivocado, si le faltan cosas por rellenar, etc.

- **Componentes más genéricos**: Como se ha comentado en el punto anterior, los componentes que se han creado para este proyecto se han centrado en su uso para dentro de formularios web.

  Si se quisiera llevar más allá, se podrían definir componentes más genéricos, que sean de uso para una aplicación web en general. De esta manera, se podría definir una *navbar*, la barra de navegación superior, una *sidebar*, barra de navegación lateral, tarjetas [47] al estilo de Google, entre muchos otros componentes.

  Si se deseara llevar más allá este punto, lo mejor sería hacer *open source* este

proyecto, de manera que la gente pudiera ayudar y recomendar maneras en las que pueda seguir adelante este proyecto.

- ***Snapshot testing***: El *snapshot testing* se trata de un tipo de pruebas que se realizan para probar que la aplicación entera funciona correctamente. Específicamente, se trataría de realizar pruebas automatizadas que prueben si los componentes se renderizan correctamente. Usar este tipo de *tests* junto con Travis-CI mejoraría enormemente la integración continua de este proyecto.

- **Fragmentación de la documentación**: Este proyecto ha sido realizado para llevar a cabo el trabajo de fin de grado. Después de que haya sido entregado, una de las mejores opciones sería publicar el proyecto en GitHub [48], para hacerlo *open-source*, y poder seguir trabajando en el. El problema de esto es que esta memoria es demasiado grande y no está orientada a servir como manual de este tipo de aplicaciones. Por lo tanto, lo mejor sería "fragmentar" el proyecto, para crear una documentación adecuada a su publicación en GitHub.

- **Centralización del estado**: Mientras que el resto de puntos que se han comentado tratan sobre mejora del trabajo ya realizado, este punto trata sobre algo que no se ha tocado en el proyecto.

  Uno de los mayores problemas que tiene Angular es el manejo del estado dentro de aplicaciones grandes. Debido a como los componentes se comunican entre ellos, *inputs* en la dirección padre-hijo y eventos en la dirección hijo-padre (ver punto 2.3.4), esto puede provocar que algunos componentes tengan que tratar un estado que no les corresponde. Es aquí donde entra el patrón de diseño del estado centralizado.

  De manera simplificada, este patrón de diseño mantiene el estado de la aplicación entera en un solo componente que se encarga de ello. De esta manera, la renderización de componentes es más simple, y depende del estado de la aplicación entera en todo momento. Se simplifica el funcionamiento de la aplicación, ya que, para un mismo estado, siempre se renderizarán los mismos elementos.

  Este tema no se ha trabajado en este proyecto, ya que no forma parte de alcance de este proyecto. El proyecto está centrado en la usabilidad de los componentes web por el usuario, y la utilización y lógica del estado sigue estando completamente bajo su responsabilidad. Además, el debate sobre la gestión del estado sigue estando abierto, donde empresas tan grandes como Netflix siguen reinventando su lógica cada ciertos meses.

Estas son solo algunas de las direcciones que se podrían tomar una vez se haya entregado y presentado este proyecto. Cualquiera de estas opciones es correcta,

y lo mejor sería llevar a cabo todas ellas, cerrando el proyecto más grande que he podido realizar durante mi trayectoria en la Universidad.

6. *CONCLUSIÓN*

El propósito original de este estudio erradica en el debate abierto sobre la estandarización del proceso de desarrollo de componentes web. Para ello, a lo largo del estudio se recogen y se justifican diversas buenas prácticas, consejos, ejemplos y referencias que puedan ayudar a tomar las decisiones de diseño más adecuadas a la hora de crear aplicaciones web altamente escalables desde el punto de vista arquitectónico.

Los conceptos discutidos a lo largo de este documento pertenecen a un ámbito muy específico a la par que amplio: desarrollo de aplicaciones web. El ejercicio de abstracción requerido para la comprensión del texto requiere, en algunas ocasiones, de un conocimiento exhausto de tecnologías disruptivas, muchas de ellas continúan todavía en desarrollo.

Tras analizar las distintas vías posibles para exponer la problemática y poder extraer así conclusiones, se entiende como necesaria la ejemplificación como hilo conductor del estudio. Es así como toma forma el denominado framework de componentes.

Esta memoria no es más que el mero reflejo de la evolución del citado framework. Todas las decisiones de diseño tratadas a lo largo de este estudio son fruto de un largo proceso de evaluación. Resulta evidente considerar el proceso de investigación seguido en este estudio como la más relevante de las conclusiones, ya que este revela la falta de visión global a la hora de dar soluciones por parte de la industria.

Las soluciones abordadas en este proyecto intentan o bien ser flexibles de cara al desarrollador último (consumidor de componentes de terceros) o, por otro lado, proporcionar la libertad estrictamente necesaria para su correcto uso.

Tanto el desarrollo de nuevos frameworks de componentes web como las sucesivas iteraciones sobre los ya existentes no debe abordarse sin contemplar la dualidad crítica que se recoge en este estudio; aportando siempre la flexibilidad que permita la máxima customización y el uso avanzado de los componentes sin olvidar la simplicidad necesaria para elaborar una documentación eficiente y favorecer la facilidad de uso de los mismos.

La diversidad temática del plan de estudios de la Universidad ha favorecido la elección de un tema tan alejado del contenido actual del propio plan. Las competencias adquiridas durante mis estudios del grado me han permitido concluir

este trabajo haciendo uso de una planificación en el tiempo, una colección de recursos y referencias, que desde luego no hubiera afrontar previamente a mi paso por la Universidad.

La versión final del proyecto está alojada en GitHub, donde se ha ido dejando constancia de todo el trabajo realizado durante estos seis meses. Al hacerlo público, se va a conseguir que este trabajo llegue a más personas, sobre todo a aquellas interesadas en ver cuales son las razones por las cuales se deberían desarrollar las aplicaciones web con la ayuda de *frameworks* como Angular, React, y este propio trabajo. El proyecto está disponible en la dirección:

`https://github.com/javiruiz01/tfg`

Referencias

[1] "Javier revillas on twitter: "paso de información en webapps: formularios, formularios disfrazados de cosas, cosas oscuras que podrían ser formularios."." https://twitter.com/javirevillas/status/1003975485544718341. (Accessed on 06/05/2018).

[2] "Html form elements." https://www.w3schools.com/html/html_form_elements.asp. (Accessed on 05/07/2018).

[3] "Javascript form validation." https://www.w3schools.com/js/js_validation.asp. (Accessed on 05/09/2018).

[4] "Html components." https://www.w3.org/TR/1998/NOTE-HTMLComponents-19981023. (Accessed on 05/09/2018).

[5] "Xbl 1.0 reference - mozilla | mdn." https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XBL/XBL_1.0_Reference. (Accessed on 05/09/2018).

[6] "Xml binding language 2.0." https://www-archive.mozilla.org/projects/xbl/xbl2.html. (Accessed on 05/09/2018).

[7] "World wide web consortium (w3c)." https://www.w3.org/. (Accessed on 05/09/2018).

[8] "Introduction to web components." https://www.w3.org/TR/2012/WD-components-intro-20120522/. (Accessed on 05/09/2018).

[9] "Web components current status - w3c." https://www.w3.org/standards/techs/components#w3c_all. (Accessed on 05/09/2018).

[10] "Using shadow dom - web components | mdn." https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM. (Accessed on 05/10/2018).

[11] "Html imports." https://w3c.github.io/webcomponents/spec/imports/. (Accessed on 05/10/2018).

[12] "Html standard." https://html.spec.whatwg.org/multipage/scripting.html#the-template-element/. (Accessed on 05/10/2018).

[13] "React - a javascript library for building user interfaces." https://reactjs.org/. (Accessed on 05/10/2018).

[14] "Vue.js." https://vuejs.org/. (Accessed on 05/10/2018).

[15] "Angular." https://angular.io/. (Accessed on 06/05/2018).

[16] "Jsx in depth - react." https://reactjs.org/docs/jsx-in-depth.html. (Accessed on 05/11/2018).

[17] "Angularjs — superheroic javascript mvw framework." https://angularjs.org/. (Accessed on 05/12/2018).

[18] "Typescript - javascript that scales.." https://www.typescriptlang.org/. (Accessed on 05/12/2018).

[19] "Angular - introduction to components." https://angular.io/guide/architecture-components. (Accessed on 05/14/2018).

[20] "Mastering angular dependency injection with @inject, @injectable, tokens and providers." https://toddmotto.com/angular-dependency-injection. (Accessed on 05/14/2018).

[21] "Angular - architecture overview." https://angular.io/guide/architecture. (Accessed on 05/15/2018).

[22] "Angular - lifecycle hooks." https://angular.io/guide/lifecycle-hooks. (Accessed on 05/14/2018).

[23] "Trello." https://trello.com/. (Accessed on 05/15/2018).

[24] "Bitbucket | the git solution for professional teams." https://bitbucket.org/. (Accessed on 05/15/2018).

[25] "Jira | issue & project tracking software | atlassian." https://www.atlassian.com/software/jira. (Accessed on 05/15/2018).

[26] "Understanding success criterion 1.4.3 | understanding wcag 2.0." https://www.w3.org/TR/UNDERSTANDING-WCAG20/visual-audio-contrast-contrast.html. (Accessed on 05/15/2018).

[27] "Bootstrap · the most popular html, css, and js library in the world.." http://getbootstrap.com/. (Accessed on 05/15/2018).

[28] "Homepage - material design." https://material.io/. (Accessed on 05/15/2018).

[29] "Transclusion in angular 2 with ng-content." https://toddmotto.com/transclusion-in-angular-2-with-ng-content. (Accessed on 05/24/2018).

[30] "Buttons - material design." https://material.io/design/components/buttons.html. (Accessed on 05/25/2018).

[31] "Html input tag." https://www.w3schools.com/tags/tag_input.asp. (Accessed on 05/27/2018).

[32] "Text fields - material design." https://material.io/design/components/text-fields.html#text-fields-layout. (Accessed on 05/27/2018).

[33] "Html textarea tag." https://www.w3schools.com/tags/tag_textarea.asp. (Accessed on 05/27/2018).

[34] "Html select tag." https://www.w3schools.com/tags/tag_select.asp. (Accessed on 05/27/2018).

[35] "<select>- html | mdn." https://developer.mozilla.org/en-US/docs/Web/HTML/Element/select. (Accessed on 05/27/2018).

[36] "Breadcrumbs - materialize." https://materializecss.com/breadcrumbs.html#! (Accessed on 05/28/2018).

[37] "Html nav tag." https://www.w3schools.com/tags/tag_nav.asp. (Accessed on 05/28/2018).

[38] "Html table tag." https://www.w3schools.com/tags/tag_table.asp. (Accessed on 05/31/2018).

[39] "Angular - deployment." https://angular.io/guide/deployment. (Accessed on 06/02/2018).

[40] "Travis ci - test and deploy with confidence." https://travis-ci.com/. (Accessed on 06/02/2018).

[41] "Github." https://github.com/. (Accessed on 06/02/2018).

[42] "Firebase." https://firebase.google.com/. (Accessed on 06/02/2018).

[43] "Stripe: Login." https://dashboard.stripe.com/login. (Accessed on 06/03/2018).

[44] "Vacation rentals, homes, experiences & places - airbnb." https://www.airbnb.com/. (Accessed on 06/03/2018).

[45] "Getting started | less.js." http://lesscss.org/. (Accessed on 06/02/2018).

[46] "Sass: Syntactically awesome style sheets." https://sass-lang.com/. (Accessed on 06/02/2018).

[47] "Cards - material design." https://material.io/design/components/cards.html. (Accessed on 06/02/2018).

[48] "Github." https://github.com/. (Accessed on 06/02/2018).

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Tue Jun 05 23:09:44 CEST 2018
	Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
	Numero de Serie	630
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)