Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Ingeniería de pruebas: aplicación, alcance y rendimiento en un banco de test para una aplicación cliente-servidor

Autor: Mario Jiménez Calderón
Tutor: Francisco José Fernández Jiménez

Dpto. Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Trabajo Fin de Grado
Grado en Ingeniería de Telecomunicación

Ingeniería de pruebas: aplicación, alcance y rendimiento en un banco de test para una aplicación cliente-servidor

Autor:
Mario Jiménez Calderón

Tutor:
Francisco José Fernández Jiménez

Dpto. Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2020

Trabajo Fin de Grado: Ingeniería de pruebas: aplicación, alcance y rendimiento en un banco de test para una aplicación cliente-servidor

Autor: Mario Jiménez Calderón

Tutor: Francisco José Fernández Jiménez

El tribunal nombrado para juzgar el Trabajo Fin de Grado arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2020

El Secretario del Tribunal

A mi familia

A mis amigos

A mis maestros

Agradecimientos

Ahora que llevo más de dos años en el mundo laboral y lejos de casa, me doy cuenta de la importancia de cada detalle visto a lo largo de mis estudios universitarios.

En el ecuador de la carrera, cuando ya tenía cierta experiencia, maldecía ciertas asignaturas por su dificultad y, a veces, incluso culpaba al profesor por su metodología a la hora de enseñar. Ahora me doy cuenta de que, gracias a esas asignaturas, soy autosuficiente y que no importa no entender algo, lo que es realmente importante son los recursos, la habilidad para buscarnos la vida y el no tenerle miedo a algo desconocido.

Por esto y por mucho más, mi primer gracias va para vosotros, los profesores.

No es por hacernos aprobar asignaturas, sino por los conocimientos y habilidades que nos habéis ido enseñando, pues gracias a esto ahora puedo afrontar el examen de verdad, el de la vida fuera de la Escuela.

Quiero agradecerle a mi madre el poder costearme la carrera y un Erasmus, que, pese a que reconozco que el nivel en la universidad de destino era ligeramente inferior, me hizo independiente y capaz de ver que se puede vivir fuera de casa sin las comodidades que eso conlleva.

Y, por último, gracias a todos mis compañeros y amigos que de una manera u otra siempre han estado ahí, para ser compañeros de prácticas, para trabajar en algún proyecto, para ayudarnos o para salir a tomar algo.

Mario Jiménez Calderón

Sevilla, 2020

Resumen

En este proyecto se ha creado una aplicación web del tipo single-page application (SPA), la cual está escrita en JavaScript y basada en el framework React JS.

Por otro lado, la aplicación se apoya en datos obtenidos de un servidor simple creado con la librería JSON Server.

Una vez que se ha tenido la aplicación finalizada y en funcionamiento, se han escrito tests end-to-end, tests de integración y tests unitarios para cubrir, en la medida de lo posible, toda la interfaz de la aplicación creada.

Para ello se ha utilizado la librería Cypress para los tests end-to-end y las librerías Jest y Enzyme para los tests de integración y unitarios.

Tras detallar los tests utilizados y analizar los resultados, teniendo especial interés en el tiempo de ejecución de cada categoría de tests, se ha procedido a una optimización de estos.

La optimización se ha empezado por los tests end-to-end comparando que zonas de la aplicación puede ser testeada con tests de integración, buscando siempre no perder alcance. Tras este análisis se han eliminado tests, lo cual ha repercutido en una mejora notable en el tiempo de ejecución de esta categoría al comparar los resultados con los obtenidos en la fase pre-optimización.

Respecto la mejora en los tests de integración, se ha realizado un proceso similar al anterior. Se ha buscado mejorar el tiempo de ejecución reemplazando algunos tests que cubren partes de la aplicación que pueden testearse con tests unitarios. En este caso la comparación también muestra una mejora en el tiempo de ejecución total. Adicionalmente, se ha comprobado de forma aislada que al añadir tests unitarios el tiempo de ejecución se incrementa levemente, en cambio, tener tests de integración innecesarios supone una diferencia de tiempo mayor.

Con estas optimizaciones se ha demostrado que seguir una estrategia de testeo influye en gran medida al tiempo de ejecución de los tests.

Finalmente, se ha hecho una inspección manual de la aplicación para detectar errores que no han sido cubiertos por los tests debido a fallos en la validación de la interfaz. Estos errores o comportamientos no esperados han sido analizados y se han propuesto soluciones.

Abstract

For this project there has been created a single-page application (SPA) which is written in JavaScript and based on the React JS framework.

On the other hand, the application relies on data obtained from a simple server created with JSON Server library.

Once the application is finished and running the following kind of test have been written: end-to-end tests, integration tests and unit tests. This is in order to cover the application interface as deep as possible.

Cypress library has been used for end-to-end tests then Jest and Enzyme libraries for integration and unit tests.

After detailing the tests used and analyzing the result paying attention to the execution time of each category of tests the next step was to optimize those tests.

Optimization started with end-to-end tests comparing which areas of the application can be tested with integration tests while trying to keep the same level of coverage. After this analysis, tests have been removed which has resulted in a notable improvement in the execution time of this category when comparing the results with those obtained in the pre-optimization phase.

Regarding the improvement in the integration tests a similar process to the previous one has been carried out. We have sought to improve the execution time by replacing some tests that cover parts of the application that can be tested with unit tests. In this case the comparison also shows an improvement in the total execution time. Additionally, it has been verified in isolation that when adding unit tests the execution time is slightly increased. On the other hand, having unnecessary integration tests implies a greater time difference.

With these optimizations it has been shown that following a testing strategy greatly influences the execution time of the tests.

Finally, a manual inspection of the application has been done to detect errors that have not been covered by the tests due to errors in the validation of the interface. These errors or unexpected behaviors have been analyzed and solutions have been proposed.

Índice

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

"Sólo porque hayas contado todos los árboles no significa que hayas visto el bosque."

Anónimo

La fase de pruebas o testeo de una aplicación de cualquier índole se podría definir brevemente como un proceso para verificar si los resultados reales coinciden con los esperados. Esto significa, comprobar que un sistema software cumple con las especificaciones y objetivos previstos.

Aunque esto pueda sonar a algo sencillo, el testeo de una aplicación puede ser algo casi tan complejo como el desarrollo de la misma aplicación. Estas pruebas no sólo te ayudan a identificar errores, sino que te describen la aplicación y crecen junto a ella. Este crecimiento implica el tener que cubrir más funciones de esta y, por consiguiente, aumentará el tiempo de ejecución de nuestros test.

Por este motivo, es de vital importancia seguir una estrategia de testing en la que haya un equilibrio lo más perfecto posible entre rendimiento y alcance.

1.1 Motivación y objetivos

En el transcurso de mis estudios, en lo que concierne a la programación, la parte del testing siempre se ha explicado muy por encima por lo que siempre lo he considerado como algo complementario al código base.

Pero en cuanto he afrontado trabajos de programación más serios, me he dado cuenta de que es una parte vital de cualquier proyecto. Esto se magnifica en cuanto entran aspectos relacionados con la comunicación, como la que se da entre un cliente y un servidor.

Trabajar en una aplicación expuesta a cambios externos conlleva un riesgo extra, pues no todo depende del lado del cliente, y tener un banco de pruebas que te alerten de un fallo antes de ejecutarla es un seguro para evitar catástrofes de menor o mayor importancia.

Un fallo en una aplicación puede parecer algo leve, pero, aunque suene alarmista, ese fallo puede conllevar pérdidas monetarias o incluso pérdidas humanas. Como ejemplo de ambos casos, se puede mencionar lo ocurrido con la reciente remesa de aviones Boeing 737 MAX [1] donde pese a ser testeado y aprobado por la Administración Federal de Aviación (FAA), tras los dos accidentes fatales ocurridos con el modelo de avión que se menciona, la misma Boeing reconoció que existían fallos de software en el controlador de vuelo. Este error se ha llevado la vida de más de 300 personas [2][3] y costes millonarios a la compañía que, hoy en día, sigue tratando de solucionar los fallos y sacar el modelo a la venta [4].

Sin embargo, añadir test no conlleva que el software esté libre de fallos. El objetivo principal es cubrir todo lo posible; desde la función más sencilla, hasta la aplicación en ejecución y su interacción con el servidor. Pero a su vez, cuantos más tests sean añadidos, más tardarán en ejecutarse, por lo que si queremos ahorrar tiempo de ejecución también hay que tener en cuenta el factor rendimiento.

En resumen, si la fase de pruebas de una aplicación es vital, la ingeniería detrás de un banco de test es lo que hace que esas pruebas tengan mayor o menor relevancia.

El objetivo de este documento es dejar a un lado la teoría relacionada con el testing, abordar de lleno la ingeniería de pruebas y comprobar a través de experimentos con diferentes tipos de test, cual es la estructura final de los test con los que se consigue más fiabilidad en el menor tiempo de ejecución posible.

Estos test se realizarán sobre una aplicación web, centrándonos en los casos de uso que surgen desde el cliente, es decir, lo que ve el usuario. Es por ello que no se tratarán test internos del servidor, pues eso sigue una metodología diferente.

1.2 Descripción de la aplicación a testear

Para este proyecto, la solución es algo costosa ya que necesitamos partir de una aplicación. Para ahorrar tiempo en el desarrollo de esta, la idea es crear algo simple, pero con las acciones suficientes como para construir un buen banco de pruebas.

Se va a crear una aplicación web basada en JavaScript y React JS donde no se discutirán detalles de su implementación o diseño, pues no afecta al objetivo del trabajo, pero si sobre la estructura de la aplicación, ya que los test unitarios y de integración van a ejecutar pruebas sobre esos ficheros. Este proyecto no trata de enseñar a desarrollar una aplicación o a como escribir tests.

Para la parte del servidor, dado que no será testeada, se utilizará la librería JSON Server [5] que proporciona un servicio REST con una configuración muy simple e intuitiva. Con esto se generarán los endpoints que necesitamos, o en otras palabras, URLs que reciben o devuelven información de un servidor en forma de respuestas HTTP.

La aplicación consistiría en un panel de información y registro de trabajadores para una empresa genérica.

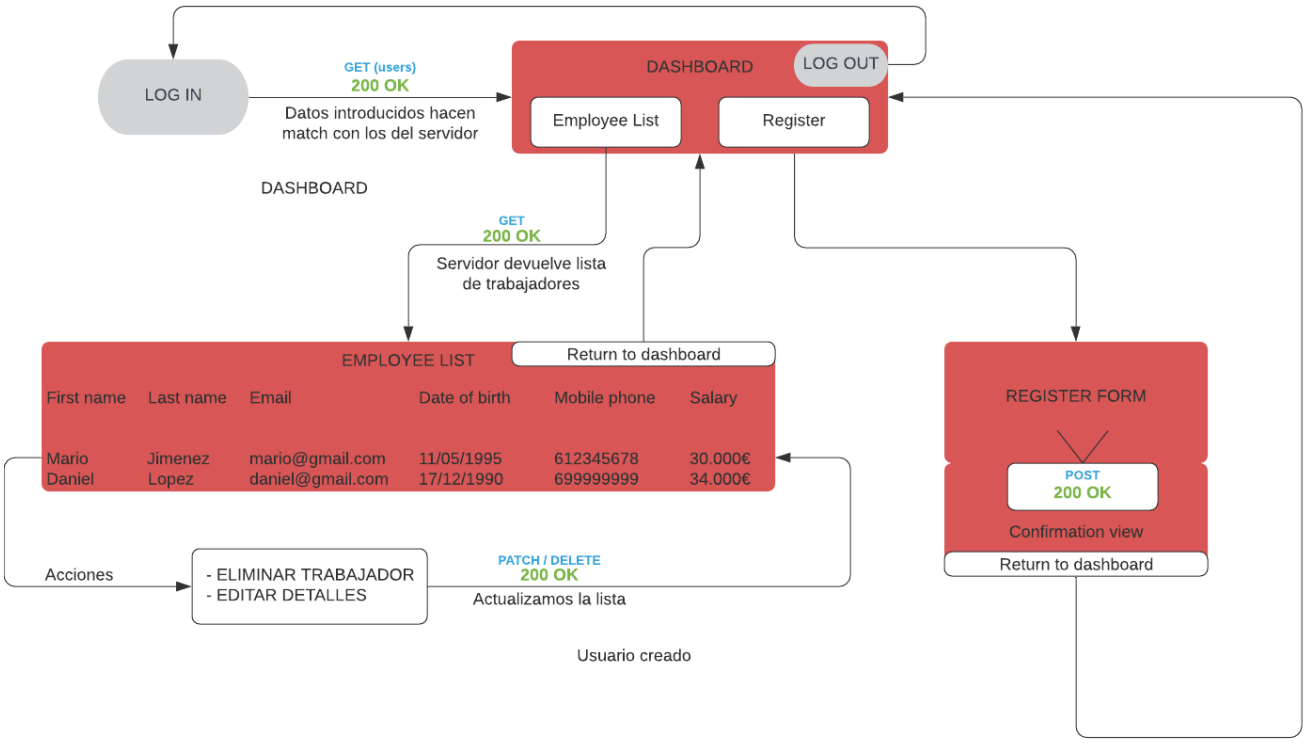

Figura 1 - Diagrama de la aplicación

En la figura 1 se vería representado un esquema de la aplicación mostrando todas las acciones posibles.

De aquí podemos destacar las cinco posibles comunicaciones con el servidor:

- Obtener lista de usuarios para iniciar la sesión
- Obtener lista de trabajadores
- Eliminar trabajador

- Editar detalles del trabajador
- Registrar nuevo trabajador

Adicionalmente, tendremos tres tipos de roles:

- Recursos Humanos (administrador)
- Recursos Humanos Asociado (permisos de escritura limitados)
- Trabajador particular o usuario (solo lectura)

Existirá un rol de super usuario que puede hacer absolutamente cualquier acción. Después un rol de solo lectura que únicamente podrá visualizar la lista de trabajadores y algunos de sus detalles, por ejemplo, no podrán ver el salario de cada trabajador. Por otro lado, se tendría un rol con poderes limitados que podría ver y editar algunos detalles de los trabajadores, pero no podrán registrar a uno nuevo.

1.3 Descripción de la solución

Las formas de realizar pruebas de la aplicación mencionada en el punto anterior son infinitas, por lo que se definirán dos posibilidades iniciales basadas en si son tests que se ejecutan a nivel de código o de aplicación.

Estos casos serán los siguientes:

- Sólo tests End-to-End: este es el caso más común en cualquier proyecto web en el que se quiere tener una seguridad fiable sin dedicar mucho tiempo al aprendizaje o desarrollo de test. En la teoría tienen cierto parecido con los test de integración ya que prueba todo el flujo del software desde el punto de vista del usuario final. La principal diferencia es que testea la aplicación desde la misma interfaz ya iniciada y no desde el código. El resultado son test muy fáciles de crear ya que se hacen de manera más visual, de hecho, hay herramientas para no tener que escribir ningún tipo de código. Además, son los más fiables ya que hacen uso del servidor real. Por el contrario, y por ese mismo motivo el tiempo de ejecución de estos test es exponencialmente superior a los que no requieren de servidor ya que en integración la comunicación con el servidor simulado es inmediata. El tiempo destinado para investigar errores es, en ocasiones, bastante superior al resto ya que tenemos que incluir posibles fallos en el servidor.
- Sólo test unitarios y de integración:
    - Tests unitarios: este tipo de test destaca por ser muy rápido en la fase de ejecución ya que no tiene ningún tipo de dependencia más allá que los parámetros introducidos para la prueba, es decir, no hace uso de ningún servicio. Otra ventaja que tienen estos test frente a los de integración es que, al analizar pequeñas partes del código, es muy fácil de detectar el foco del problema y son realmente sencillos de escribir. Por ejemplo, si se tiene una función asociada a un botón, lo que se prueba aquí es que, dados unos parámetros, al llamar a esa función se espera tal resultado. Por el contrario, aquí no estamos analizando cómo funcionan todas esas pequeñas partes de código en conjunto, por lo que no podríamos saber si falla algo en la interacción de unos componentes con otros.
    - Tests de integración: en estas pruebas, al contrario que en el caso anterior, si se analizarían los componentes desde la perspectiva del usuario. Esto significa que se tendría que montar un simulacro de una parte de la aplicación, configurando las dependencias entre componentes y simulando el comportamiento de la parte del servidor, mediante el uso respuestas y datos reales. Todo esto hace que este tipo de test sea el más costoso de crear, pues en muchas ocasiones el tiempo necesario para escribir estas pruebas es superior al tiempo dedicado en desarrollar esa parte de la aplicación, haciendo que en muchos proyectos se descarten por falta de tiempo o recursos. Partiendo del ejemplo anterior, en estos test no analizamos una función en concreto, comprobaríamos que si hacemos clic en ese botón esperamos tal cambio en la interfaz de la aplicación. Esto conlleva un comportamiento más fiel a la experiencia del usuario, aunque perdemos en eficacia a la hora de localizar el error, pues si algo falla, pese a indicar el error, ese resultado no es suficiente para saber de dónde viene el problema.

- Caso óptimo: tras analizar las conclusiones sacadas de los dos casos anteriores, se propondría un modelo que incluya todo tipo de test de manera que haya un balance entre tiempo de ejecución y fiabilidad de los test.

1.4 Estructura de la memoria

1.4.1 Introducción

Presentación de las motivaciones por las que realizar este proyecto junto al objetivo de este. También se describe la funcionalidad de la aplicación a realizar y la estrategia de testeo a desarrollar para concluir con la finalidad del trabajo.

1.4.2 Tecnologías utilizadas

Descripción del código de programación utilizado para desarrollar el proyecto, así como librerías relacionadas con el desarrollo y testeo. La finalidad es ofrecer un conocimiento mínimo para comprender lo que se va a realizar en los capítulos sucesivos.

Se introducirán los siguientes conceptos:

- Single-page application
- JavaScript
- React JS
- JSON Server
- Jest
- Enzyme
- Cypress

1.4.3 Análisis de la aplicación

Introducción detallada a la aplicación sobre la que se realizarán los tests. Se mostrarán todos los estados, eventos y llamadas al servidor externo involucrados en la aplicación.

La estructura es en base a las páginas o rutas disponibles en la aplicación, la cual está escrita en inglés por motivos de consistencia con el lenguaje de programación:

- Login
- Dashboard
- Employees
- Register

1.4.4 Pruebas end-to-end

Recorrido por todos los bloques de tests end-to-end definidos para testear la aplicación. Se trata de presentar los tests que se han realizado, para su posterior mejora. También se mostrará el resultado de la ejecución de estos tests.

Los casos de uso vendrán definidos según el rol del usuario que ha iniciado sesión:

- Admin
- Associate
- User

1.4.5 Pruebas unitarias y de integración

Recorrido por los ficheros de tests unitarios y de integración sólo mencionando los descriptores de cada test. No se mostrará código. El objetivo es el mismo que en el caso anterior, introducir los tests que después buscarán mejorarse. La ejecución de estos tests se mostrará al final con su resultado.

1.4.6 Ingeniería de pruebas

Mejora de tests end-to-end por un lado, y tests de integración por otro, para lograr un tiempo de ejecución menor al conseguido en las pruebas anteriores. Se buscará no perder alcance de testeo. Al final se mostrarán los nuevos resultados y se hará una comparativa con los resultados pre-optimización.

1.4.7 Pruebas de manuales

Se introducirá un error aparentemente no cubierto por los tests, para su posterior análisis manual. Se discutirá el alcance de los tests end-to-end.

1.4.8 Planificación

Desglose del proceso llevado a cabo para desarrollar el proyecto, destacando las partes en las que se ha destinado un mayor esfuerzo.

1.4.9 Conclusiones y líneas futuras

Final del documento, donde se argumentará las afirmaciones y deducciones sostenidas durante el proyecto junto con ideas que pueden mejorar el proyecto.

2 TECNOLOGÍAS UTILIZADAS

En este capítulo se describirán las tecnologías utilizadas para la implementación de la aplicación y su banco de tests.

2.1. Single-page application

Una single-page application (SPA), o aplicación de página única, como su propio nombre indica, es una aplicación donde toda la información se queda siempre en la misma página, renderizando sólo nuevos componentes en zonas que se requiera. Es la forma en la que se escriben las aplicaciones web modernas como puede ser el cliente de Gmail, Google Maps, PayPal, Reddit y muchas más.

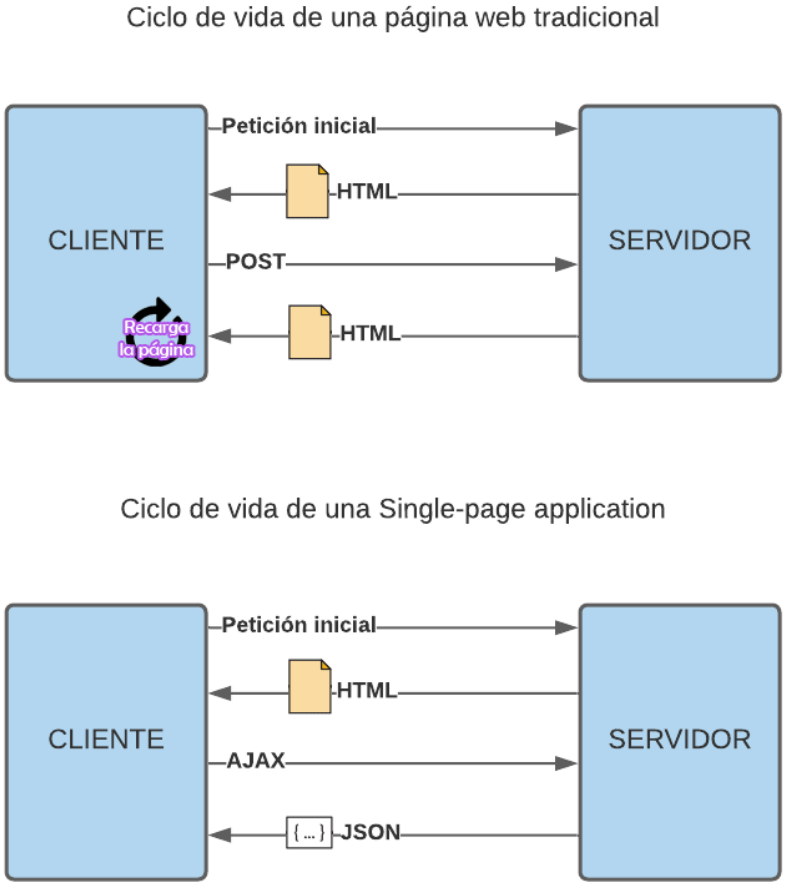

Figura 2 – Comparación entre página tradicional y SPA

En la figura 2 se puede ver las como es el ciclo de una SPA en comparación con una página tradicional donde al hacer alguna petición, vuelve a cargar toda la información desde cero.

2.1.1 JavaScript (JS)

Se trata de un lenguaje de programación orientado a objetos usado para crear y controlar páginas web con contenido dinámico, es decir, contenido que cambia sin la necesidad de que el usuario refresque la página. No necesita ser compilado, son los navegadores los encargados de interpretar el código.

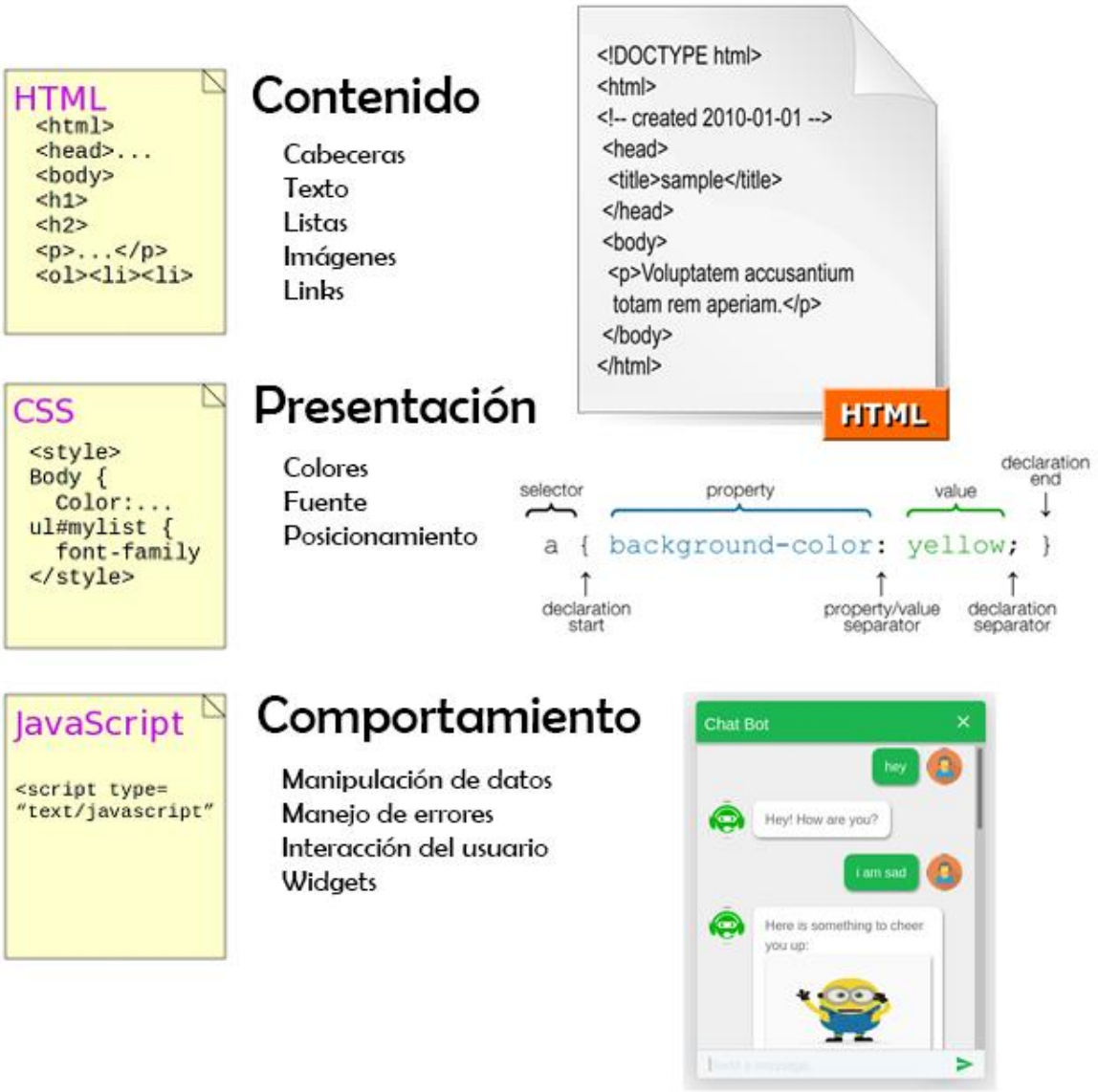

Figura 3 – HTML, CSS y JavaScript

Para entender mejor la función de JavaScript en la figura 3 puede verse los elementos que participan en la creación de una página:

- **HTML** es la estructura de la página.
- **CSS** controla como se visualiza la página.
- **JavaScript** hace que la página sea dinámica para obtener algo más que una página de texto plano.

Una característica que suele llamar mucho la atención cuando se viene de C o Java es que el tipado es dinámico, lo que significa que el tipo está asociado al valor y no a la variable.

2.1.2 React JS

React JS es el framework más popular para los desarrolladores front-end. Es una librería de JavaScript y de código abierto cuyo propietario es Facebook. Concretamente, se utiliza para construir interfaces de usuario para aplicaciones de página única.

El propósito principal de React es ser rápido, escalable y simple.

Las características principales de este framework son:

- **JSX**

  En vez de utilizar JavaScript puro para modelar, en React se utiliza JSX que se trata de una extensión de JavaScript con la opción de que permite utilizar etiquetas HTML para renderizar componentes.

  Es por ello que los navegadores no pueden leer JSX directamente ya que no se trata de JS puro. JSX tiene que ser interpretado, lo que significa que antes de que el código llegue al navegador, un compilador de JSX traducirá el código a JS puro.

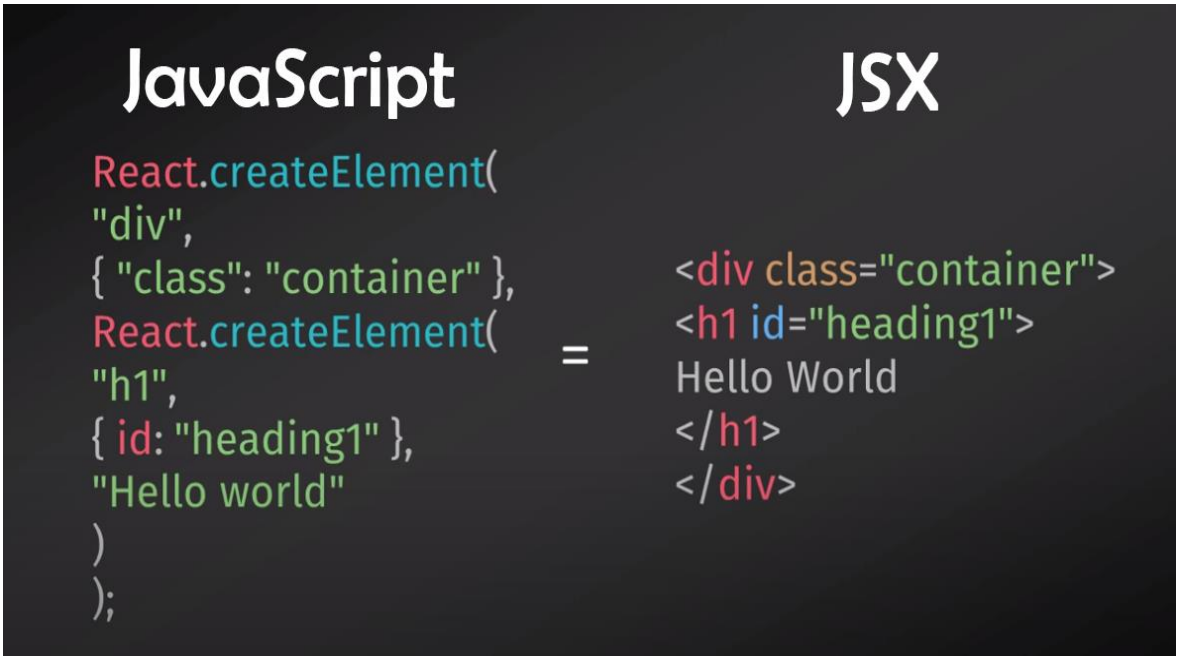

Figura 4 – JavaScript y JSX

En la figura 4 se puede ver un ejemplo escrito en JS y al otro lado su equivalente en JSX, donde podría decirse que este es una forma abreviada de llamar a *React.createElement()*.

El objetivo principal de JSX es el de simplificar y ahorrar código aportando un aspecto más visual e intuitivo a los componentes descritos en el código. Permite que personas menos familiarizadas con JavaScript puedan entender y modificar algunas partes del código, lo cual es muy importante cuando se involucran diseñadores o programadores de CSS en tu proyecto.

Pese a su similitud, conviene no confundir JSX con HTML puro dado que, aunque las etiquetas sean las mismas, la forma en la que JSX maneja estas etiquetas puede diferir en algunos aspectos. Por ejemplo, en JSX se utiliza una nomenclatura *camelCase* para los nombres de los atributos mientras que en HTML se utiliza *snake-case*. Además, existe la convención de que los componentes de React serán siempre definidos con la **primera letra en mayúscula**. Por ejemplo: < *Button* />

HTML → `<input style='border: 1px solid black; border-radius: 4px'/>`

JSX → `<input style={{border: '1px solid black', borderRadius: '4px'}}/>`

- **Flujo de datos en sentido único**

  En React, el flujo de datos desciende en forma de valores inmutables por los componentes que necesiten consumirlos.

  Estos valores o propiedades que reciben los componentes no pueden ser modificados directamente desde los hijos, pero si pueden modificarse a través de funciones 'callback' que no son más que funciones que llegan del componente padre, el mismo desde el cual se originaron los datos.

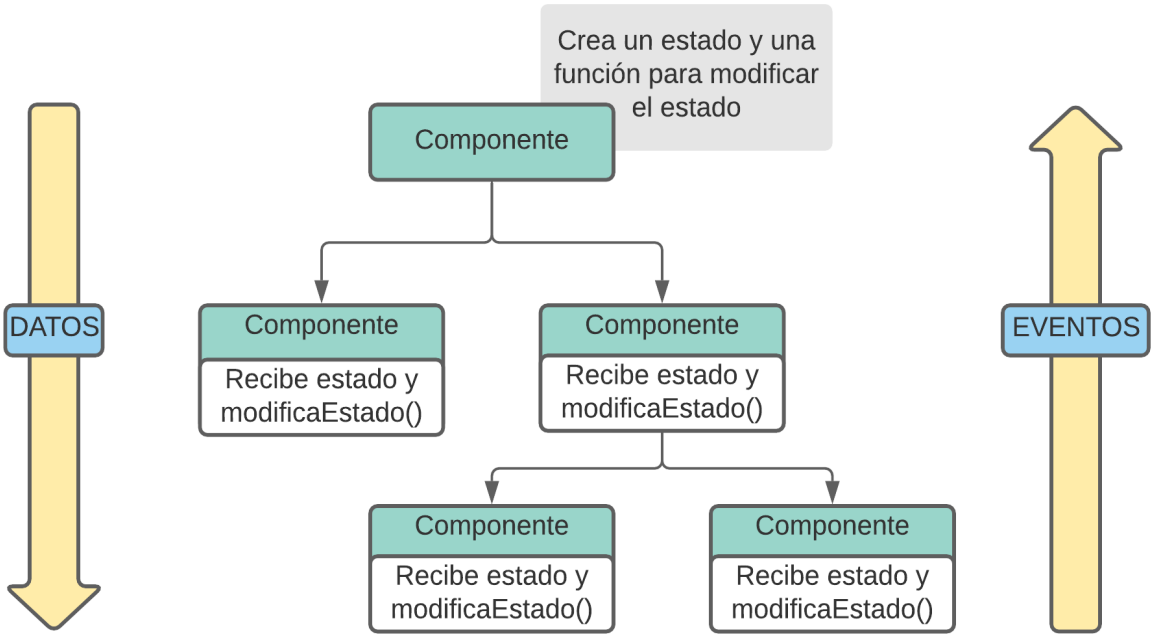

Figura 5 – Flujo de datos en React

Tal y como se ha comentado anteriormente, en la figura 5 puede verse este proceso en el que el componente padre crea un estado y lo comparte hacia sus hijos. El estado es sólo modificable desde su componente creador, es por ello que en el flujo de datos descendiente viaja tanto el estado como una función *modificaEstado()* para manejar o modificar ese estado.

Este estado llega al hijo, el cual puede compartirlo de nuevo hacia abajo, consumirlo o "modificarlo" mediante la función indicada previamente. Realmente este componente no modifica el estado, pero si envía un evento o acción de forma ascendente para que el componente padre modifique el estado y lo comparta una vez actualizado.

React re-renderiza aquellos componentes en los que hay un estado involucrado. En el ejemplo de la figura 5, no importa desde donde se llame a la función *modificaEstado()* para confirmar que el resultado será que todos los componentes de la figura volverán a ser renderizados con el valor nuevo del estado.

En conclusión, puede decirse que cuando hablamos de flujo de datos en sentido único, la forma completa de decirlo sería "las propiedades fluyen hacia abajo; las acciones fluyen hacia arriba".

- **Virtual Document Object Model (DOM)**

  Para entender lo que es el virtual DOM, conviene conocer que es el DOM.

  En español se podría traducir literalmente como Modelo de Objeto de Documento y es la estructura que genera el navegador al cargar una página web. Los objetos del DOM modelan el documento y todos los elementos que pueda tener. A estos elementos se pueden acceder a través de React (JS) para poder manipularlos.

  El virtual DOM es una representación del DOM, es como una copia más ligera que tiene las mismas propiedades que el DOM real pero no tiene el poder necesario como para poder cambiar lo que se muestra en la página web.

  React crea esta estructura de datos que registra los cambios en los componentes y los actualiza. Esto permite que el programador trabaje como si la totalidad de la página fuera renderizada en cada cambio, cuando realmente en el navegador solo cambiará el componente o componentes afectados con el cambio.

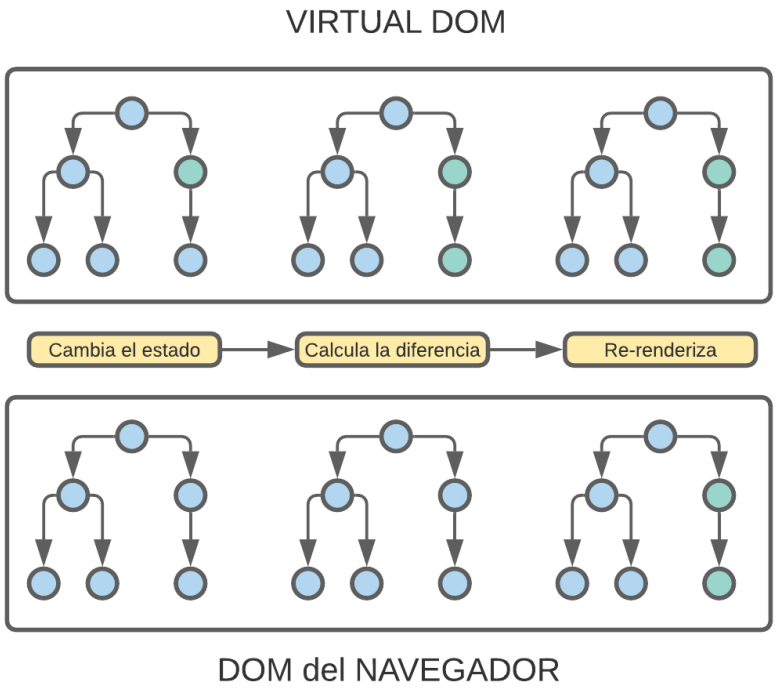

Figura 6 – Cambios en virtual DOM y DOM real

Tal y como se aprecia en la figura 6, los nodos verdes representan los nodos que han cambiado. Estos nodos son los elementos que se han visto afectados por un cambio de estado.

La diferencia entre la versión previa del virtual DOM y el virtual DOM actual se calcula en el paso intermedio. Entonces, toda la rama que sale del padre se re-renderiza dando la interfaz actualizada. Esta rama actualizada es también aplicada al DOM del navegador o DOM real donde ya si se podrían ver los cambios en el navegador.

Básicamente se le está diciendo a React que estado queremos en la interfaz y este se asegura de que el DOM real tenga ese estado, sin tener que conocer cómo funciona esta manipulación de datos que hay detrás del proceso.

Aunque parezca un proceso costoso, manipular un virtual DOM es muy rápido, al contrario que hacerlo con el DOM real. Para entenderlo mejor, manipular un virtual DOM es como trazar el plano de una habitación, mientras que manipular el DOM real sería como construir esa misma habitación en una casa.

Finalmente conviene mencionar el ciclo de vida de los estados en React.

En la versión que se va a utilizar en el proyecto disponemos de los React Hooks que simplifican todo el ciclo, de hecho, solo se mencionarán dos hooks ya que serán los utilizados en la aplicación propuesta. La traducción de 'hook' en español sería 'gancho', y como su propio nombre indica, son funciones que permiten 'enganchar' estados y el ciclo de vida.

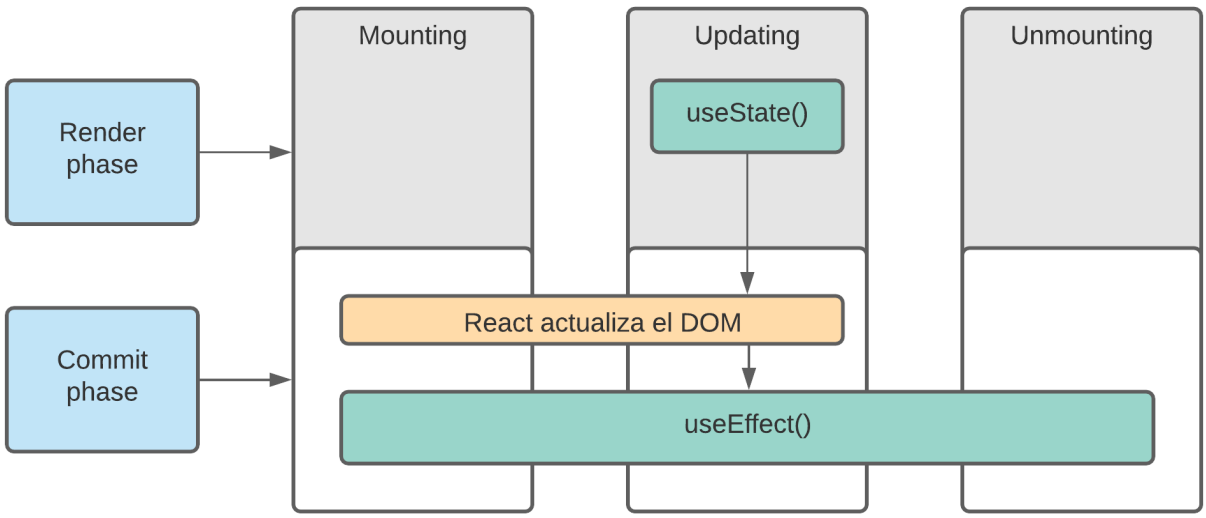

Figura 7 – Ciclo de vida de React

Como ya se ha mencionado existen más hooks pero en la figura 7 solo se ven representados los dos que vamos a utilizar: *useEffect()* y *useState()*.

También pueden reconocerse tres ciclos: montaje, actualización y desmontaje (del DOM). Estos ciclos están divididos en dos fases cada uno: fase de renderizado, que es cuando se comparan la versión previa y nueva del DOM y la fase de 'asignación' que es cuando React finaliza de asignar los cambios en el DOM.

Entonces, la función *useState()* nos permite modificar estados una vez tenemos el DOM montado. Por ejemplo, el estado de un botón sería manejado con este hook, pues se quiere que, una vez montada la interfaz, hacer click en el botón y que realice una función que actualizará su estado.

Por último con *useEffect()* se puede actualizar un estado justo después de tener el DOM actualizado. En concreto, por la naturaleza del proyecto, solo se utilizará durante el ciclo de montaje. Este hook es de gran utilidad a la hora de realizar llamadas a un servidor externo de cara a obtener datos que actualizarán el estado, pues se llamaría a esta API justo después de montar el componente de React. Por ejemplo, para rellenar una tabla con datos obtenidos desde un servidor.

Siguiendo con el ejemplo anterior, suponemos que los datos recibidos por el servidor para rellenar la tabla, son guardados en un objeto, el cual es nuestro estado y además se le indica a *useEffect()* que repita el proceso de llamar al servidor cada vez que esos datos cambien. En este caso, inintencionadamente se estaría creando un bucle, pues aunque los datos que nos llegan del servidor en la segunda llamada sean los mismos, la referencia del objeto recibido es diferente a la más antigua, por lo que React tomará esos datos como distintos y se volvería a realizar una llamada al servidor.

Debido a la naturaleza de los estados, un error común es crear bucles de renderizado donde un estado se actualiza automáticamente después de cada renderizado y esto se repite de manera indefinida. Es por ello que, para evitar bucles, la fase de asignación sólo se ejecuta una vez por ciclo.

2.2. Servidor

2.2.1 JSON Server

Para realizar de forma realista las conexiones de la aplicación con un servidor, se hace uso de JSON Server que no es más que una librería escrita en JavaScript que proporciona un servidor con funciones limitadas y con un mantenimiento que no requiere de ningún conocimiento adicional.

Para preparar la base de datos hay que crear un fichero de tipo *.json* y rellenarla con las entidades que se requieran:

```
{
  "posts": [
    { "id": 1, "title": "json-server", "author": "typicode" }
  ],
  "comments": [
    { "id": 1, "body": "some comment", "postId": 1 }
  ],
  "profile": { "name": "typicode" }
}
```

Una vez iniciado el servidor se podrán realizar llamadas GET, POST, PUT, PATCH y DELETE que actualizarán la base de datos en concordancia con los parámetros utilizados.

2.3. Testing

Para la fase de pruebas, existen librerías que proporcionan los recursos necesarios para describir y realizar tests. Debido a que los tests unitarios y de integración son totalmente diferentes a los end-to-end, se elegirán varias librerías para poder abarcar los diferentes tipos de tests que serán utilizados para el banco de pruebas del proyecto.

2.3.1 Jest y Enzyme

Jest es una librería de testeo escrita en JavaScript y creada por los mismos desarrolladores de React, aunque eso no significa que su uso esté destinado únicamente a React.

Por otro lado, Enzyme es otra librería que está diseñada específicamente para testear componentes de React. Esta librería fue creada por AirBnB y añade herramientas muy útiles para renderizar componentes, localizar elementos e interactuar con ellos.

Jest es una librería es bastante simple. Dispone de dos funciones globales: *describe* para describir lo que va a testearse y por otro lado *it* para definir el test en sí.

Dentro de los *it* esperaremos un resultado para indicar que el test se ha ejecutado con éxito, para ello se utiliza *expect* junto con un 'matcher' que no es más que una función para afirmar algo sobre el valor esperado. Hay una gran variedad de matchers definidos en la documentación de Jest [6].

Otra utilidad es el objeto Jest [6] que contiene funciones para crear 'mocks', es decir, objetos simulados que copian el comportamiento de un objeto real de forma controlada. Los mocks son de bastante utilidad a la hora de simular la respuesta de un servidor externo. En los tests unitarios y de integración nunca se debe llamar al servidor real, pues esto implicaría un rendimiento muy inferior al esperado por estos tests, al incrementar enormemente el tiempo de ejecución de los tests.

Para comprender mejor las capacidades de Jest se plantea el siguiente ejemplo en el que se va a testear una función de suma y otra de resta.

```
describe('Suma y resta', () => {

const sum = (a,b) => a+b
const minus = (a,b) => a-b

   it('devuelve la suma',() => {
    expect(sum(2,2)).toBe(4)
 })
   it('devuelve la resta',() => {
    expect(minus(5,2)).toBe(3)
 })
})
```

En este ejemplo se puede identificar qué es lo que va a testearse dentro del *describe* (Suma y resta), junto a las dos funciones a comprobar. Después dentro de los *it* se comprueba que sumar 2+2 es 4 y que 5-2 es 3. Para ello se hace uso del matcher *.toBe*

En cuanto a Enzyme, proporciona utilidades para definir lo que hay entre el *expect* y el matcher, es decir, identificar el elemento de React que vamos a testear.

Tiene tres métodos principales: *mount(), shallow() y render()*

- **Mount** renderiza todo el DOM del componente indicado junto a sus hijos. Es especialmente útil para tests de integración donde tenemos que hacer pruebas a una porción del código, donde se quiere interactuar con el DOM o hacer uso del ciclo de vida de React para comprobar el componente al completo. Permite acceder a las propiedades del componente padre y a las de sus hijos.
- **Shallow** sólo renderiza el componente que se le indica, sin incluir los hijos. Esto permite aislar el componente para poder realizar tests unitarios ya que protege al componente de cambios o errores en los hijos, lo cual puede alterar el resultado del test.
- **Render** es parecido a *mount()*, ya que renderiza tanto al componente padre como a los hijos, pero con la diferencia de que *render()* no renderiza el DOM sino que te devuelve un HTML estático. No tiene acceso a los ciclos de vida de React y tiene menos funcionalidades por lo que no es especialmente útil para las pruebas del proyecto.

En resumen, en el proyecto se utilizará *mount()* para tests de integración y *shallow()* para tests unitarios. La función *render()* no va a ser utilizada, debido a lo descrito anteriormente.

La librería de Enzyme es bastante amplia [8] pero se van a comentar las funciones más relevantes que dispone el objeto devuelto por *mount()* y *shallow()* con el siguiente ejemplo, donde se va a hacer un test a un contador.

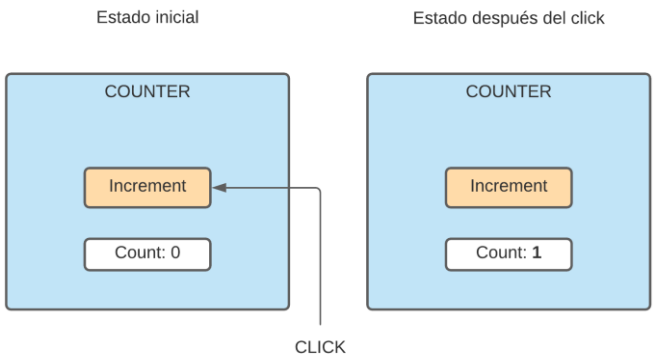

Figura 8 – Ejemplo de un contador

```
import React from 'react'
import { shallow } from 'enzyme'
import Counter from './index'

describe('Funcionalidad del componente Counter', () => {
  it('aumenta el contador cuando se hace click en el botón', ()
=> {
   const wrapper = shallow(<Counter/>)
   wrapper.find('button').simulate('click')
   expect(wrapper.find('p').text()).toBe('Count: 1')
  })
})
```

El primer método a destacar es el *.find()* disponible en el objeto devuelto por el *shallow(<Counter/>).* Este método nos permite identificar el elemento *button* del componente. En este caso *button* es la etiqueta HTML del botón, pero Enzyme permite identificar elementos de otras formas, por ejemplo, por medio de un identificador tal y como se verá en el proyecto, pues en este caso solo existe un botón pero si existiera más de uno, conviene tener un nivel más de identificación.

La siguiente función que puede observarse es *.simulate('click')* que simulará un click en el DOM del elemento.

Finalmente se ejecuta la línea que valida el test, donde nuevamente por medio del método *find()* identificamos el texto que muestra el estado del contador y se comprueba que el resultado es 1.

2.3.2 Cypress

Para realizar tests end-to-end se va a utilizar otra librería de testeo denominada Cypress debido a que utiliza JavaScript para escribir sus tests, lo cual facilita su aprendizaje cuando se tiene experiencia con este lenguaje. Además la estructura de los tests es bastante similar a los descritos con Jest y se instala como cualquier otra librería de JavaScript.

Una característica fundamental que tiene Cypress respecto a la competencia es que espera automáticamente a que se ejecuten las funciones o llamadas al servidor antes de seguir ejecutándose, evitando problemas con llamadas asíncronas. Esto es, en gran medida, debido a que Cypress no está basado en Selenium (otra herramienta de testeo end-to-end), sino que parte de una arquitectura totalmente nueva, que se centra exclusivamente en tests de esta índole.

Aunque vaya a utilizarse para un proyecto basado en React, es compatible con cualquier framework de JavaScript y permite ejecutar sus tests en navegadores basados en Chrome y Firefox.

Al contrario que Jest, que reporta el resultado de los tests por el terminal, Cypress dispone de una interfaz gráfica donde muestra en tiempo real la ejecución de los tests y su resultado.

Para mostrar las utilidades de Cypress se va a recorrer un ejemplo mostrado en sus guías [9]

Se trata de una página con varias categorías de las cuales se le hará click a una para redirigir el navegador a otra ruta donde aparece un recuadro en el que se puede introducir texto.

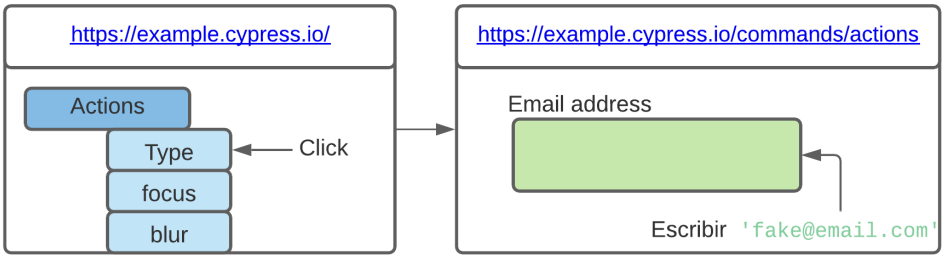

Figura 9 – Ejemplo Cypress

```
describe('My First Test', () => {
  it('Gets, types and asserts', () => {
    cy.visit('https://example.cypress.io')

    cy.contains('type').click()

    // Should be on a new URL which includes '/commands/actions'
    cy.url().should('include', '/commands/actions')

    // Get an input, type into it and verify that the value has been updated
    cy.get('.action-email')
      .type('fake@email.com')
      .should('have.value', 'fake@email.com')
  })
})
```

En este ejemplo puede comprobarse como la nomenclatura para describir los tests es idéntica a la mostrada en Jest, pues tenemos un *describe* para especificar qué es lo que va a testearse y un *it* donde se ejecuta el test.

Lo primero que destaca en este código es el objeto global *cy* que es donde se incluyen todos los métodos de Cypress [10].

Como ocurre en Jest y Enzyme, la nomenclatura de los métodos suele ser bastante autodescriptiva, para ayudar al programador a entender que está ocurriendo.

El primer método que aparece es *.visit()* que simplemente abre la URL pasada en el primer argumento de la función. En Cypress, al contrario que en Jest que solo valida algo cuando se utiliza *expect()*, todas las líneas que utilicen algún método de Cypress son validaciones, por lo que si en este caso al abrir la página, esta no cargara, el test será marcado como no exitoso.

En la siguiente línea se puede ver el método *.contains()* que devuelve el elemento del DOM que contenga el texto introducido en el primer argumento, que en este caso es 'type'. Acto seguido se utiliza la función *.click()* para accionar el botón.

Después se comprueba que el navegador se encuentra en la ruta esperada tras hacer el click. Para ello se utiliza *.url()* para obtener la URL actual y el método *.should()* para comprobar que es la esperada.

Finalmente se accede al recuadro donde se puede introducir texto mediante el método *.get()* donde se le pasa la clase del elemento a encontrar por el primer argumento de la función ('.action-email'). Acto seguido escribe una dirección de email con *.type()* y se comprueba con *.should()* que el campo de texto contiene lo introducido.

3 ANÁLISIS DE LA APLICACIÓN

Recordando el diagrama estructural de la aplicación que aparece en la figura 1, se procede a realizar un análisis de la interfaz de usuario de la aplicación real. Este análisis se centrará en identificar las páginas y componentes que posteriormente serán testeados.

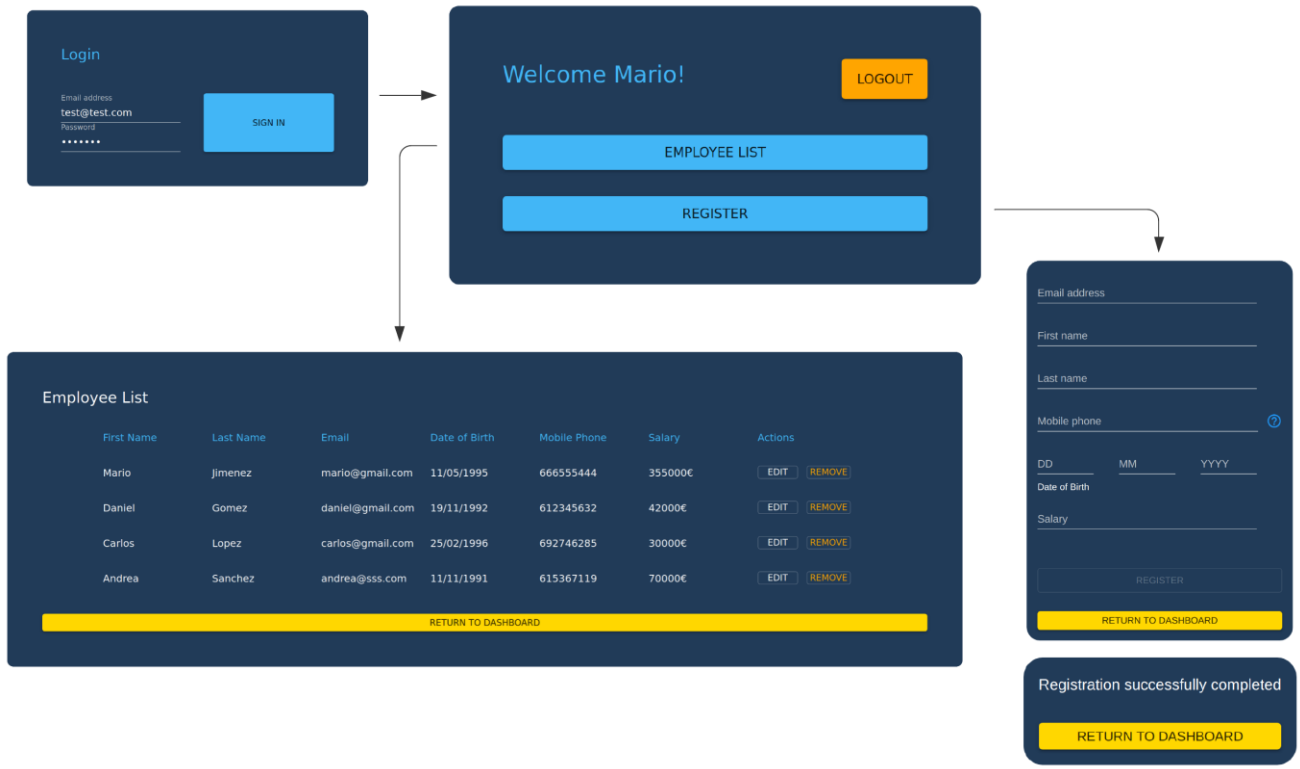

Figura 10 – Estructura real de la aplicación vista desde un administrador

Tal y como puede observarse en la figura 10, disponemos de cuatro grandes componentes que a su vez son rutas independientes.

Estas rutas son:

- **/login:** página de inicio donde el usuario introducirá sus credenciales para identificar su usuario.
- **/dashboard:** menú principal donde el usuario tendrá, según su rol, la opción de visitar la lista de empleados, el formulario de registro de empleados o volver al login.
- **/employees:** lista de empleados donde según el rol del usuario podrá visualizar todos o algunos detalles así como editar o eliminar trabajadores.
- **/register:** ruta solo accesible por administradores donde se podrán introducir los datos de un nuevo empleado. Al clicar en el botón para registrar un nuevo empleado, se podrá visualizar un componente de confirmación.

Durante la navegación en la aplicación, al cambiar de ruta, se entraría en el ciclo de desmontaje donde React desmontaría la ruta previa y pasaría al ciclo del montaje de la ruta nueva.

Al visitar cualquiera de estas rutas, por seguridad, el usuario siempre será redirigido a la primera ruta o **/login** para que pueda introducir sus credenciales de usuario. Sólo habrá un caso en el que el usuario no sea redirigido y es al refrescar la página.

```
> window.history
History {length: 6, scrollRestoration: "auto", state: {…}}
    length: 6
    scrollRestoration: "auto"
  ▼ state:
      key: "vca114"
    ▶ state: {email: "test@test.com", id: 1, role: "admin", name: "Mario"}
    ▶ __proto__: Object
  ▶ __proto__: History
```

Esto se debe a que el objeto *history* del navegador se guarda en la caché y al actualizar sigue siendo posible acceder al estado en el que se encuentra el objeto con los detalles del usuario.

Antes de pasar a analizar estas rutas de un modo más aislado conviene mencionar que en esta aplicación suponemos el caso ideal en el que el servidor externo usado es perfecto, esto significa que cualquier llamada que hagamos a este será devuelta con éxito, nunca devolverá un error. Se trata de un hecho intencionado para poder introducir reglas de validación en la interfaz, en vez de en el servidor, con el objetivo de realizar más pruebas y plantear.

El código se ha estructurado de forma que cada ruta representa un módulo y un módulo, por lo general, son dos componentes de React: un contenedor y un componente con los elementos a renderizar.

En el contenedor estará toda la lógica relacionada con los estados, es decir, la inicialización de los estados y las funciones callback que los modifican. Los componentes serán los encargados de consumir y realizar acciones a los estados del contenedor. Por otro lado, dentro de los componentes estarán descritos los elementos necesarios para construir la interfaz del módulo.

La elección de esta estructura es principalmente por simplificar los componentes de cara al testing. Al tener la lógica separada del componente, se puede aislar el renderizado de elementos de sus acciones, lo cual permite enfocar los tests unitarios al análisis de los componentes sin funciones que alteren los resultados. Es decir, se tendría un componente que recibe unas propiedades y se espera que esas propiedades se muestren en el componente tal y como han llegado o que se renderice un elemento u otro dependiendo del valor de estas.

Para el análisis de la aplicación consideraremos ambos componentes como uno, es por ello que se mencionará la existencia de **módulos** en vez de componentes.

3.1 Login

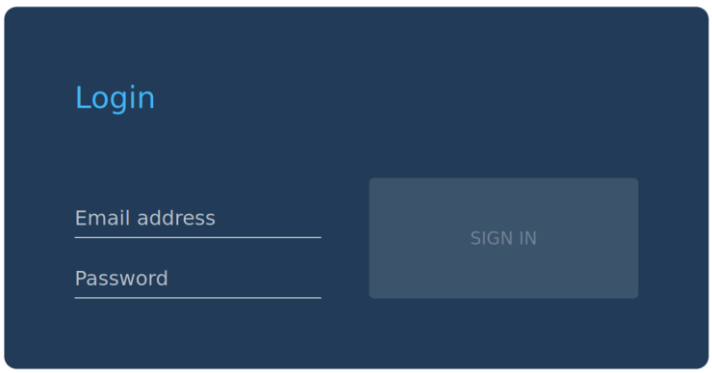

Figura 11 –Estado inicial del módulo Login

En la figura 11 puede contemplarse el estado inicial del módulo Login que contiene dos campos de texto y un botón, que por defecto está desactivado debido a que los campos de texto están vacíos.

Para describir los estados involucrados en este módulo conviene comprender el funcionamiento del inicio de sesión, dado que por motivos de simplicidad, no se ha implementado un inicio sesión real donde al introducir los datos del usuario y realizar una llamada POST exitosa al servidor, este devolvería el objeto con la sesión del usuario.

Volviendo a la figura 11, de manera totalmente transparente al usuario, se está realizando una llamada GET al servidor para obtener la lista de usuarios registrados disponibles, lo cual es una práctica que se salta cualquier principio de seguridad y protección de datos, dado que estamos recibiendo datos de usuarios ajenos al actual, incluyendo contraseñas. Como ya se ha comentado, todo es intencionado por motivos de simplicidad y de darle protagonismo al front-end o vista de la aplicación.

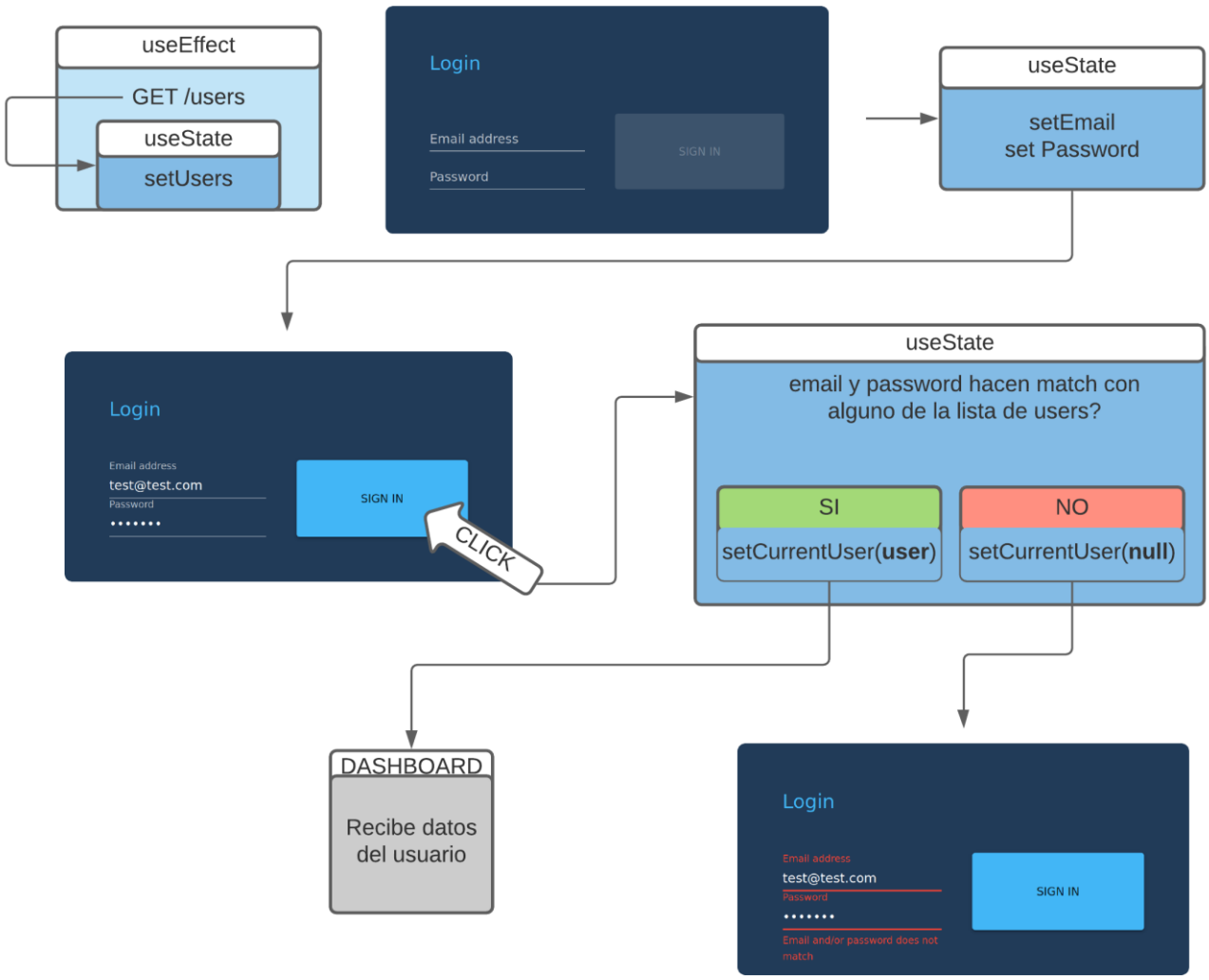

Figura 12 –Ciclo de vida del módulo Login

Se utilizará lo representado en la figura 12 para explicar paso a paso el módulo Login.

En primer lugar como ya se ha mencionado, se realizaría una llamada GET al endpoint */users* para obtener la lista de usuarios disponibles así como sus detalles. Cada usuario tendrá un identificador, email, contraseña, nombre y rol. Esta respuesta del servidor se almacena localmente en un estado de React.

El siguiente paso sería introducir el email y contraseña, los cuales se almacenan en otros dos estados.

Una vez completada la introducción de las credenciales se haría click en el botón *SIGN IN* que llamaría a una función en la que se comparara el email y contraseña introducidos con la lista de usuarios buscando alguna coincidencia.

En el caso de que tanto el email como la contraseña encuentren una coincidencia, se guardaría en otro estado el usuario con su información asociada, redirigiendo la interfaz a la ruta */dashboard* cuyo modulo recibiría esos datos de usuario quitando la contraseña que es irrelevante para el resto de componentes. Si no hubiera ninguna coincidencia, se guardaría el estado con un valor nulo, lo cual hace que la interfaz nos muestre un error en la parte inferior de los campos de texto.

3.2 Dashboard

El Dashboard forma parte de la ruta a la que se aterriza inmediatamente después de iniciar sesión.

Su función es la de un enrutador hacia el resto de secciones de la aplicación por lo que es en el único modulo en el que no se realiza ninguna llamada al servidor.

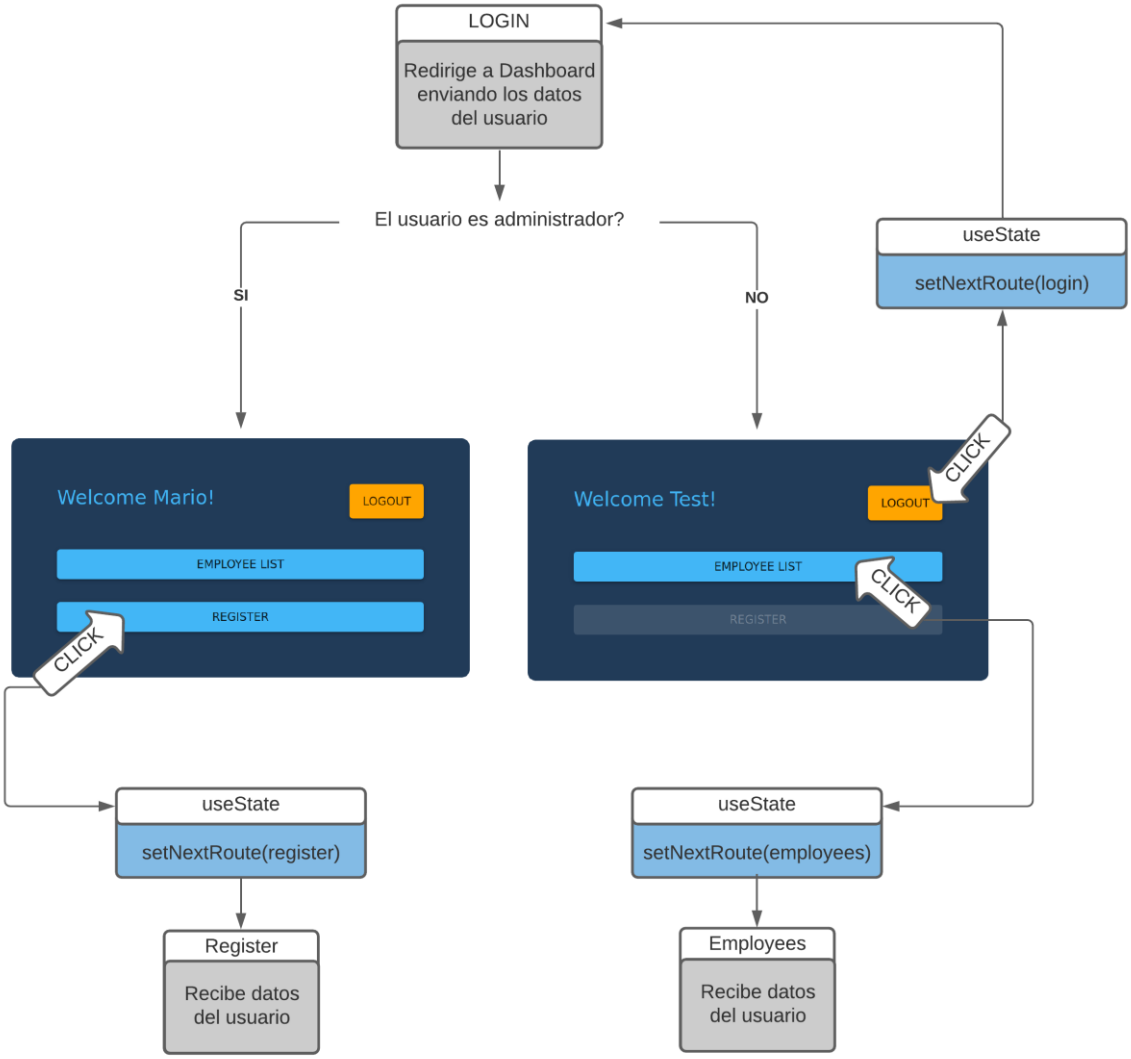

Figura 13 – Ciclo de vida del módulo Dashboard

El módulo Dashboard comprueba el rol del usuario que llega desde el Login para decidir si debe permitir al usuario el acceso al formulario de registro de nuevos empleados (administrador) o si, por el contrario, debe desactivar el botón (asociado o usuario).

El único estado de React presente en este módulo, es el encargado de guardar la siguiente ruta. Entonces, si el usuario hace click en alguno de los botones presentes, la función callback *setNextRoute()* cambiará el estado a la ruta elegida, redirigiendo la interfaz y enviando los datos del usuario.

3.3 Register

El módulo Register tiene la peculiaridad de que es el usuario con rol de administrador el único que tiene acceso a él.

Este módulo con reglas de validación para cada campo de texto, por lo que se analizará primero el caso en el que la validación de todos los campos devuelve un resultado no exitoso

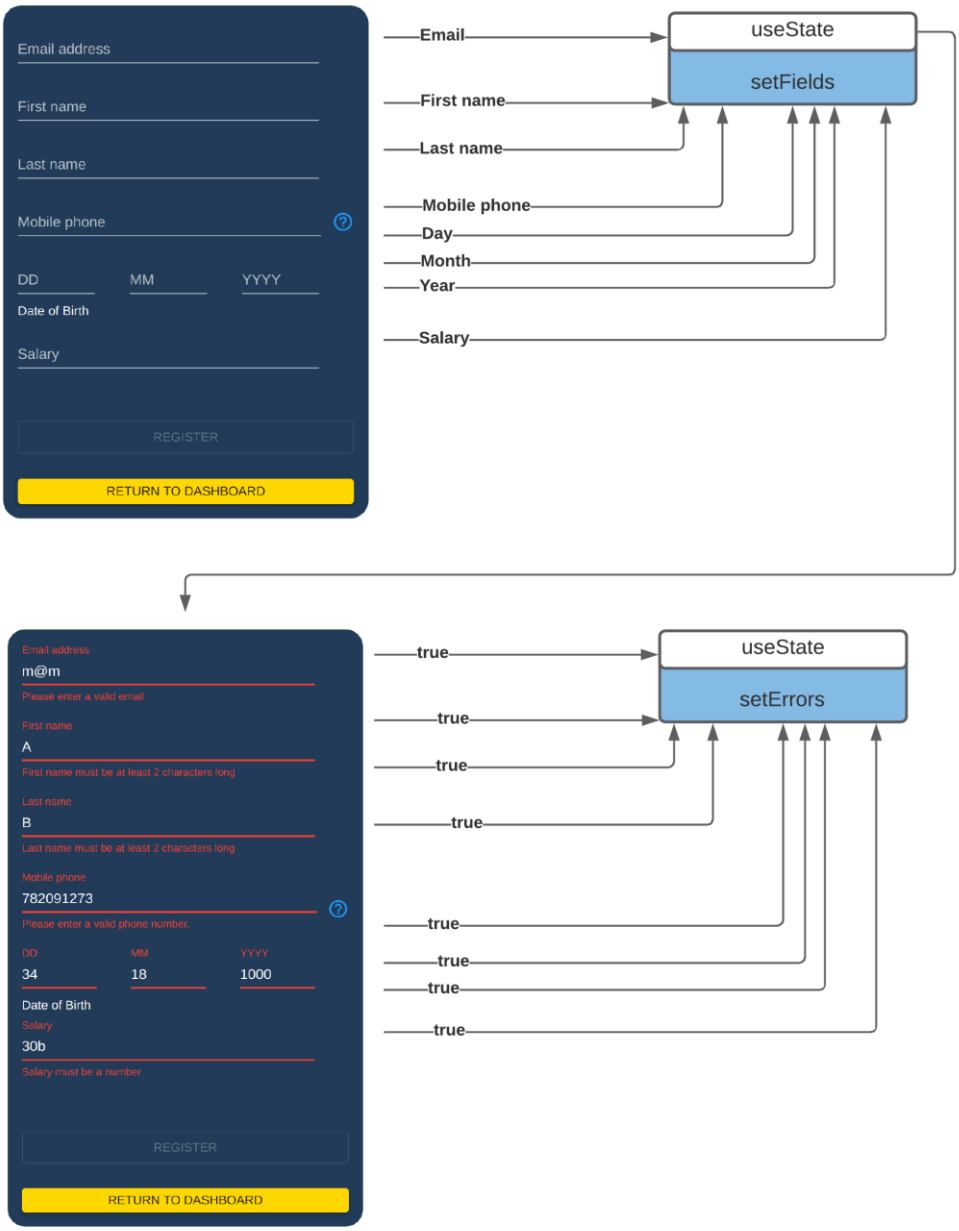

Figura 14 – Ciclo de vida del módulo Register cuando lo introducido es incorrecto

Al contrario que en el módulo Login, donde existía un estado para cada campo de texto, en este caso se tiene un único estado que contiene un objeto que se actualiza cada vez que se introduce un nuevo valor en cualquier campo de texto del módulo. Esto puede observarse en la figura 14 y se realiza con el objetivo tener un código más limpio y tener todos los valores en el mismo estado.

La primera acción a realizar es introducir un valor que, dependiendo del campo que estamos modificando, pasará unas reglas de validación u otras:

- **Email address:** contiene una expresión regular que se asegura de que el formato del email es [nombre]@[servidor].[dominio]
- **First name y Last name:** comprueba que el valor es de al menos dos caracteres.
- **Mobile phone:** comprueba que el valor introducido es un número de nueve caracteres y que empieza por 6.
- **DD:** comprueba que el valor introducido es un número de dos caracteres y entre el 1 y el 31.
- **MM:** comprueba que el valor introducido es un número de dos caracteres y entre el 1 y el 12.
- **YYYY:** comprueba que el valor introducido es un número de cuatro caracteres y que no es menor que 1900.
- **Salary:** comprueba que el valor introducido es un número.

Entonces, en este caso se introduce un valor no valido en cada campo de texto lo que modifica el estado que almacena los errores de cada categoría, para marcar que hay un error.

El botón *REGISTER* está configurado de manera que se desactive en caso de que algún valor del estado de errores esté fijado a *true*, por lo que en este caso no podremos seguir salvo que arreglemos los valores no válidos.

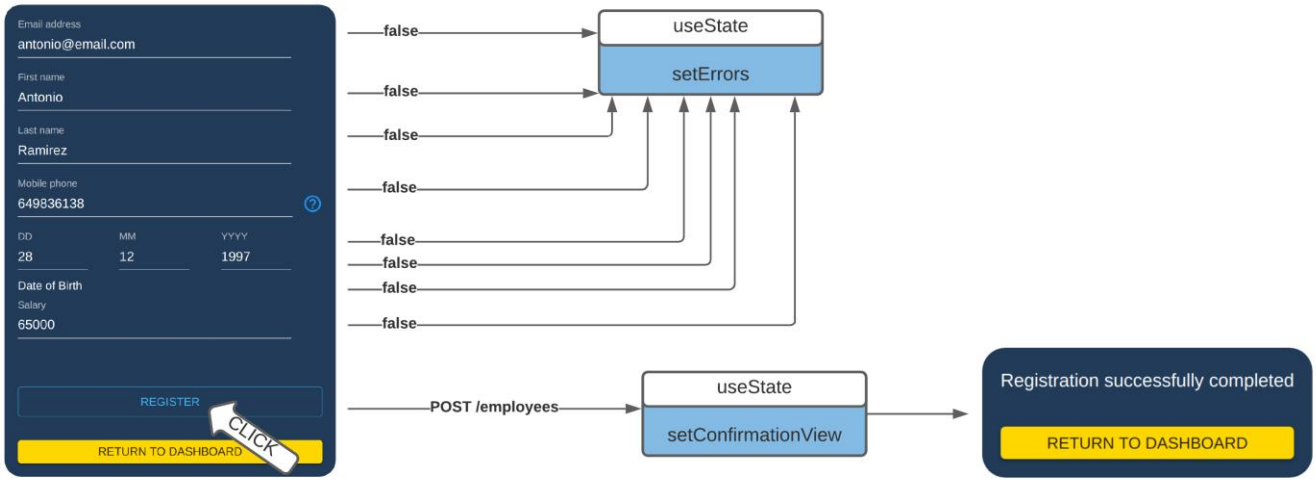

Figura 15 – Ciclo de vida del módulo Register cuando lo introducido es correcto

En el caso de una validación exitosa, el valor asignado a los errores es *false* y esto permite accionar el botón *REGISTER* tal y como se indica en la figura 15.

Una vez clicado el botón, se haría una llamada POST al servidor para escribir el nuevo empleado en la base de datos. Inmediatamente después de finalizar la llamada a la API, se asignaría el valor *true* al estado *confirmationView* para renderizar el mensaje de confirmación, que sustituiría el formulario de registro.

3.4 Employees

El módulo Employees destaca por el hecho de que se trata de una lista de elementos a los cuales se le pueden aplicar acciones individualmente. Este tipo de componentes pueden resolverse de varias formas, pero la más limpia y óptima, es decir, la que menos merma el rendimiento de la aplicación, es tratar la lista como dos módulos independientes.

Figura 16 – Subdivisión del módulo Employees

En la figura 16 aparece el módulo Employees desde la vista de un administrador. Es importante destacar el rol, dado que en este módulo, la interfaz cambia según el rol, pero se analizará primero el rol con todos los poderes para comprobar todas las acciones posibles.

Este módulo tiene una cabecera con el título, la lista de empleados y un botón para volver al Dashboard.

En esta misma figura puede verse dos bordes que delimitan esos dos módulos mencionados anteriormente. El borde rosa hace referencia al módulo de la lista de empleados y el borde azulado al módulo de un elemento de la lista.

En resumen, la lista de empleados es un módulo 'lista' (*EmployeesList)* que renderiza uno o más módulos elemento (*EmployeesListItem)*.

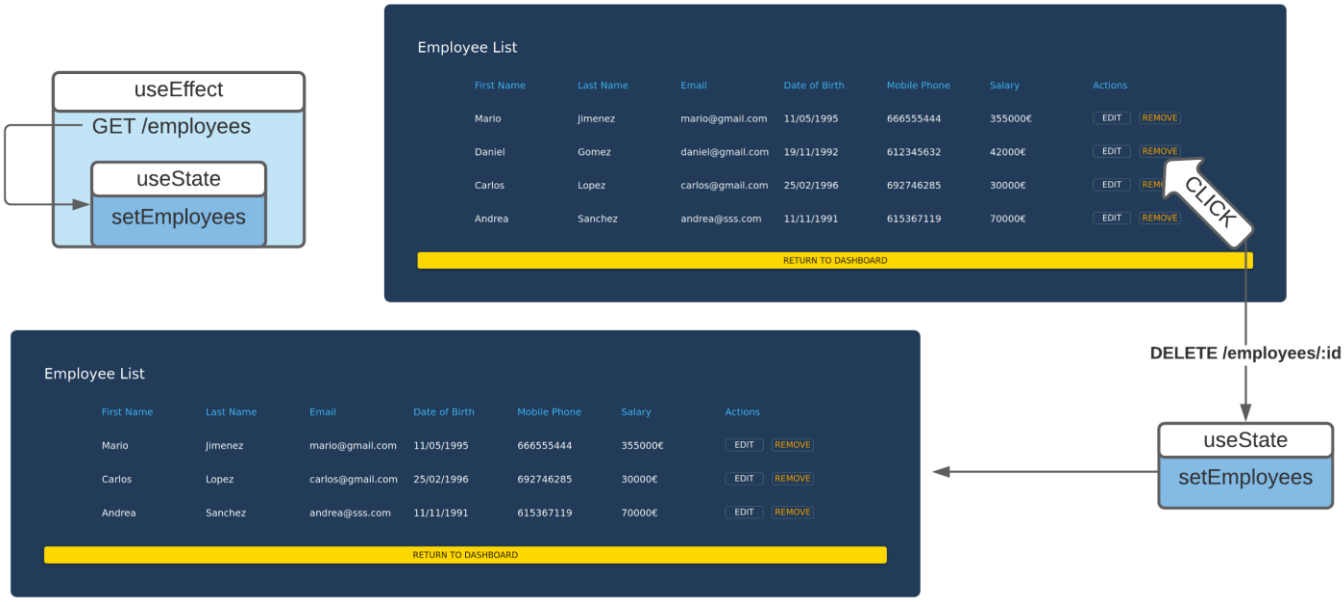

Figura 17– Ciclo de vida del módulo EmployeesList

Empezando por la lista, representada en la figura 17, se realizaría una llamada GET al endpoint */employees* para obtener la lista de empleados con sus detalles. Estos detalles son los mostrados en el título de cada columna. Posteriormente se almacenarían estos datos es un estado.

Una vez que se actualiza el estado, se renderiza la lista.

La otra acción que afecta al estado de *EmployeesList* es cuando el usuario acciona el botón *REMOVE*. Este botón llama al servidor con el método DELETE al mismo endpoint anterior pero especificando el identificador del empleado a eliminar. Acto seguido, elimina al empleado del estado, volviendo a renderizar la lista sin ese *EmployeesListItem* que lo representaba.

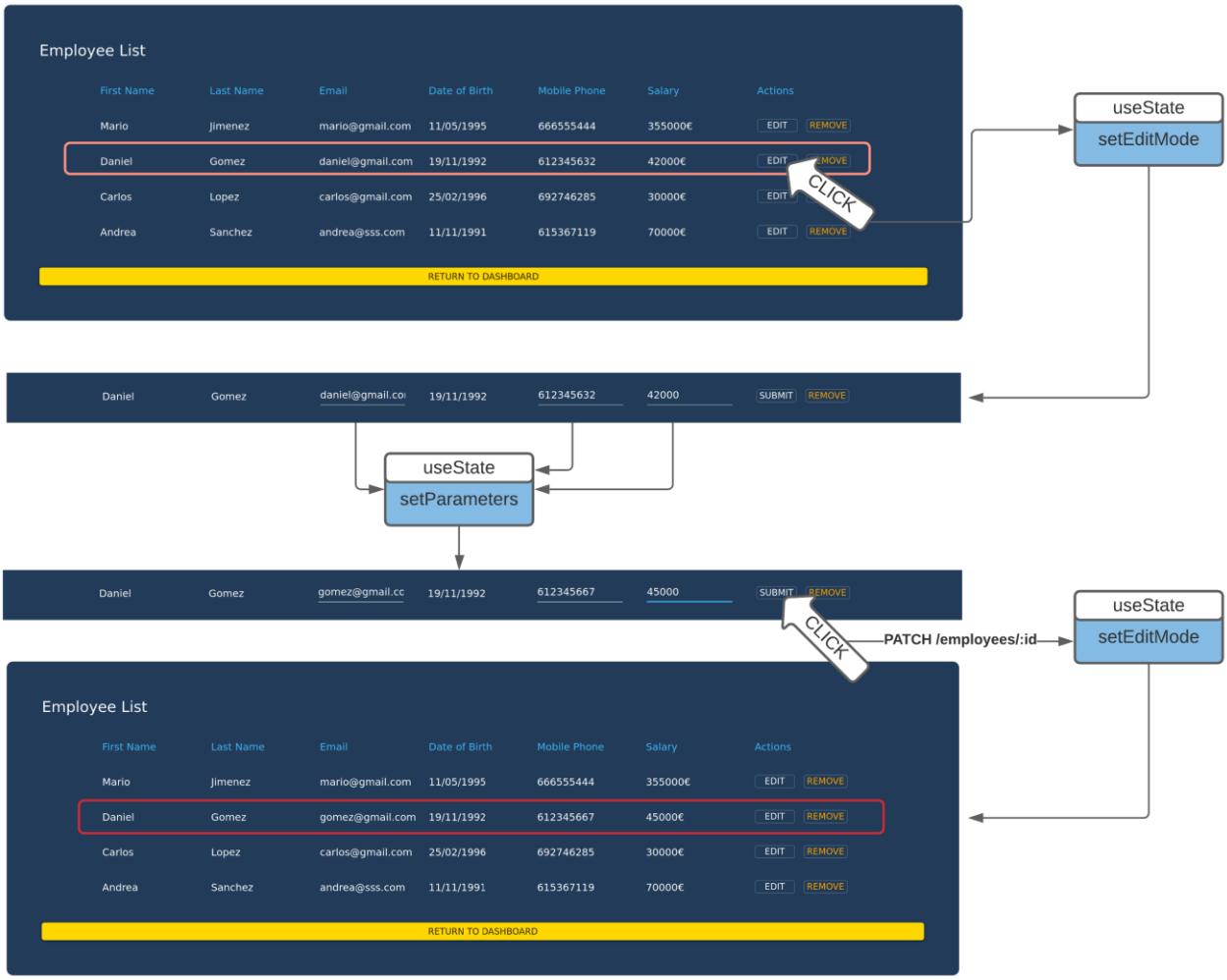

Figura 18– Ciclo de vida del módulo EmployeesListItem

Ahora, en la figura 18, se muestra el módulo elemento de la lista junto sus estados y acciones.

Se tiene un estado que controla si el elemento está en modo edición o no, este estado se modifica al hacer click en el botón *EDIT* que renderiza tres campos de texto que permite al usuario modificar el email, número de móvil y salario. En este caso, a diferencia del módulo Register, no hay validación dado que no aportaría nada a los tests, pues sería testear exactamente lo mismo.

Estos parámetros son controlados por un estado similar al que se tenía en el módulo Register, pues un objeto almacena el valor de todos los campos.

Una vez modificados los campos, la siguiente acción disponible es enviar estos cambios al servidor por medio de un click en el botón *SUBMIT*, realizándose una llamada PATCH a */employees/:id* donde *:id* es el identificador del empleado a modificar. Posteriormente se volvería a cambiar el estado que controla el modo edición para asignar que la aplicación vuelve a estar en modo lectura.

Aunque el botón *REMOVE* se encuentra renderizado dentro de este módulo, el efecto que tiene en el elemento es inocuo dado que al eliminarlo de la lista, este elemento no va a renderizarse.

Ahora se tiene la suficiente información para esclarecer el motivo por el que esta subdivisión de módulos es la idónea. Utilizando la lógica mostrada, como tenemos estados independientes entre los elementos de la lista, si un estado se modifica, este no afecta al resto de la lista. En cambio, si solo se tuviera un módulo lista que controlara todos los elementos, cualquier modificación en la lista resultaría en un re-renderizado completo de todos los elementos de la lista, pues conviene recordar que React re-renderiza cualquier componente que esté involucrado con un estado que acaba de ser modificado.

A continuación se van a mostrar la vista del módulo Employees cuando el rol no es administrador.

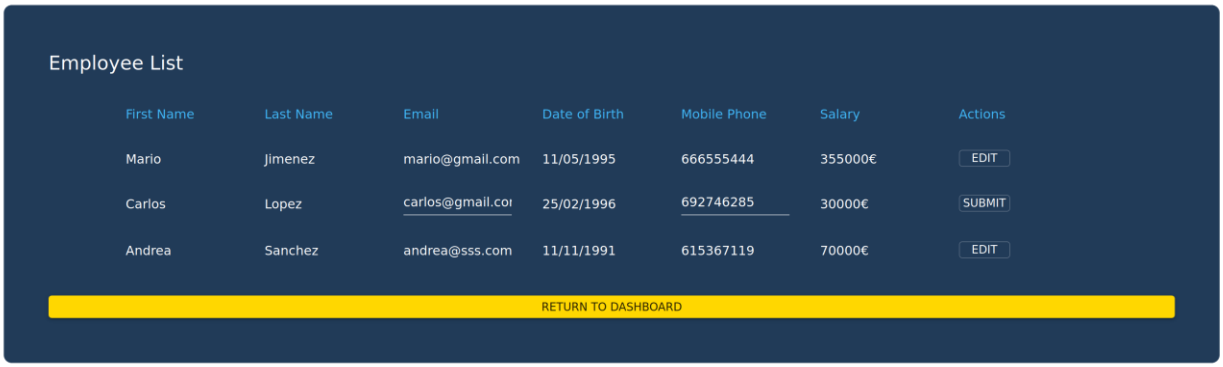

Figura 19– Vista del módulo Employess cuando el rol es de asociado

El caso de que se tenga un rol asignado de asociado, como en la figura 19, el usuario podría ver todas las columnas disponibles y podría editar el email y el número de teléfono, pero no el salario, dado que se trata de información más sensible. Además, puede observarse la ausencia del botón *REMOVE*, pues este rol tampoco tiene poderes para hacerlo.

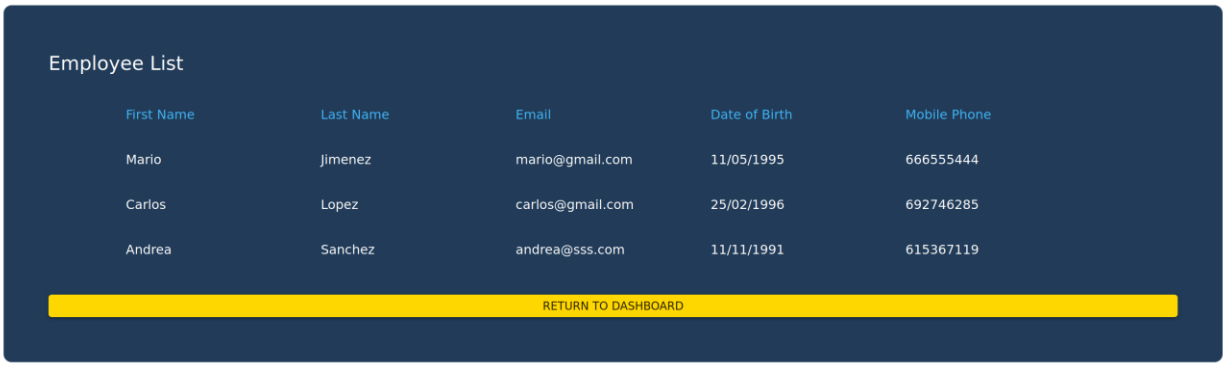

Figura 20– Vista del módulo Employess cuando el rol es de usuario

Para el rol usuario mostrado en la figura 20, se elimina por completo el poder de escritura y, además, no puede visualizar la columna que hace referencia al salario de cada empleado.

4 PRUEBAS END-TO-END

En este capítulo se va a abordar el primer tipo de testing en la aplicación planteada. Se trata de los tests end-to-end y el motivo de empezar por estos es, que en proyectos web suele ser la única opción existente, especialmente en start-ups donde no hay recursos para destinar a otros tipos de tests, pues como se verá a continuación, escribir estos tests es muy sencillo.

Aunque a priori podría sugerirse un orden similar al usado en el capítulo anterior, donde se analizaba cada ruta o modulo, la estructura principal dependerá del rol del usuario.

La decisión de definir una estructura por rol es debido a que en estos tests se trata de emular la experiencia de un usuario real, y este debe siempre pasar por el módulo Login, iniciar sesión y, dependiendo del rol, tendrá disponible unas funciones u otras.

En cambio, dentro de cada rol, si se definirán los tests dependiendo del módulo que se quiera testear. Para ello, se definirán unas rutas bypass que permitirán a Cypress acceder directamente a rutas protegidas, sin necesidad de pasar por el Login. En entornos de producción estas rutas no estarán disponibles, su uso es exclusivo a entornos de desarrollo o testeo.

```
{process.env.NODE_ENV !== 'production' &&
<>
    <Route
        path="/testing-bypass-dashboard/:role"
        exact
        component={({match}) =>
            <Redirect to={{ pathname: '/dashboard', state: {name: 'Test', role: match.params.role} }}/>}
    />
    <Route
        path="/testing-bypass-employees/:role"
        exact
        component={({match}) =>
            <Redirect to={{ pathname: '/employees', state: {name: 'Test', role: match.params.role} }} />}
    />
    <Route
        path="/testing-bypass-register/:role"
        exact
        component={({match}) =>
            <Redirect to={{ pathname: '/register', state: {name: 'Test', role: match.params.role} }} />}
    />
</>
}
```

Para activar o desactivar el acceso a estar rutas se utiliza la variable de entorno *NODE_ENV* que React tiene por defecto. Su valor puede ser *production* o *development* y son asignadas dependiendo de como se lance la aplicación. Esto será explicado más adelante en el anexo donde se explica como lanzar la aplicación y ejecutar los tests.

Crear tests que validen las rutas bypass en tests de integración no sería muy fiable dado que la variable de entorno tendría que ser simulada, pues en estos tests la aplicación no se ha iniciado. Como esta variable es la única condición para activar o desactivar las rutas bypass, se descarta este tipo de test.

Por otro lado, si es posible hacer un test end-to-end donde en vez de acceder al servidor local en el que está lanzada la aplicación en desarrollo, se acceda a otro en el que sesté lanzada en producción. En desarrollo no es necesario crear tests específicos para ello, pues estas rutas serán accedidas para realizar otros tests y de fallar, se podría identificar desde el principio que algo no funciona al visitar la ruta.

```
describe( title: 'Bypass routes on production', fn: () => {
    describe( title: 'Dashboard', fn: () => {
        it( title: 'can not access the dashboard page as admin', fn: () => {
            cy.visit('http://localhost:5000/testing-bypass-dashboard/admin')
            cy.url().should( chainer: 'include', value: '/login')
        })
        it( title: 'can not access the dashboard page as associate', fn: () => {
            cy.visit('http://localhost:5000/testing-bypass-dashboard/associate')
            cy.url().should( chainer: 'include', value: '/login')
        })
        it( title: 'can not access the dashboard page as user', fn: () => {
            cy.visit('http://localhost:5000/testing-bypass-dashboard/user')
            cy.url().should( chainer: 'include', value: '/login')
        })
    })
    describe( title: 'Employees', fn: () => {
        it( title: 'can not access the employees page as admin', fn: () => {
            cy.visit('http://localhost:5000/testing-bypass-employees/admin')
            cy.url().should( chainer: 'include', value: '/login')
        })
        it( title: 'can not access the employees page as associate', fn: () => {
            cy.visit('http://localhost:5000/testing-bypass-employees/associate')
            cy.url().should( chainer: 'include', value: '/login')
        })
        it( title: 'can not access the employees page as user', fn: () => {
            cy.visit('http://localhost:5000/testing-bypass-employees/user')
            cy.url().should( chainer: 'include', value: '/login')
        })
    })
    describe( title: 'Register', fn: () => {
        it( title: 'can not access the employees page as admin', fn: () => {
            cy.visit('http://localhost:5000/testing-bypass-register/admin')
            cy.url().should( chainer: 'include', value: '/login')
        })
    })
})
```

Estos tests simplemente visitan cada ruta bypass probando todos los roles posibles y comprueban que el navegador siempre es redirigido a la ruta Login, pues es lo que se espera cuando no se tiene acceso a estas rutas.

Debido a que muchos tests serán compartidos entre los roles, por ejemplo, los tests del Login son idénticos, se empezará por el rol de administrador y en el resto de roles solo se mencionaran los tests que cambian respecto a este.

Aunque suene redundante, desde la perspectiva del usuario no se puede asegurar que, por ejemplo, si se puede iniciar sesión con un usuario de rol administrador, también se podrá hacer con un rol de asociado, dado que puede haber algún fallo en la lógica interna de la interfaz o del servidor que haga que ese rol en concreto no tenga acceso a la aplicación.

Respecto la identificación de cada componente se verán varias formas de hacerlo, pero la que más se repite y la que será la única utilizada en tests de integración y unitarios, es mediante un *data-test-id*

```
<div style={{ display: 'flex' }}>
    <form data-test-id="login-form" noValidate onSubmit={() => null}>
        <FormControl error={!currentUser}  style={{ paddingRight: '38px' }}...>
        <FormControl error={!currentUser}  style={{ paddingRight: '38px' }}...>
    </form>
    <Button
        data-test-id="login-form-sign-in-button"
        fullWidth
        disabled={!email.length || !password.length}
        variant="contained"
        color="primary"
        onClick={handleOnSubmit}
        style={{ maxHeight: '96px' }}
    >
        Sign in
    </Button>
</div>
```

Este identificador se define como atributo de los componentes que posteriormente serán testeados con el objetivo de que su identificación sea lo más específica posible.

```
▼<div style="display: flex;">
  ▶<form data-test-id="login-form" novalidate>…</form>
  ▶<button tabindex="0" type="button" data-test-id="login-form-sign-in-button" style="max-height: 96px;">…</button>
  </div>
```

Así se vería el código HTML, donde los elementos aparecen con ese *data-test-id* para identificar los componentes que se quieren testear.

Por último queda mencionar que estos tests modifican la base de datos del servidor, por lo que se usará una base de datos preparada para estos tests que volverá a su estado inicial después de ejecutar los tests de cada caso de uso. Para ello se utiliza el siguiente comando:

```
beforeEach( fn: () => {
    cy.exec( command: 'cp testing-db-data.json testing-db.json')
})
```

Este comando básicamente copia una base de datos que no va a mutar en la base de dato preparada para el banco de tests, reescribiéndola si existiera.

4.1 Administrador (admin)

4.1.1 Login

Para testear el módulo Login asignaremos dos objetos: unas credenciales de usuario válidos y otros no válidos. Por otro lado, usaremos la función *beforeEach* para ejecutar la función de Cypress que abre la ruta del módulo en el navegador cada vez que ejecutemos un 'it'.

```
describe( title: 'Login', fn: () => {
    const validUser = {
        email: 'test@admin.com',
        password: 'test123',
    }
    const invalidUser = {
        email: 'invalid@email.com',
        password: 'invalid123'
    }

    beforeEach( fn: () => {
        cy.visit('http://localhost:3000/login')
    })
```

4.1.1.1 Renderiza los elementos del módulo Login

```
it( title: 'renders the Login elements', fn: () => {
    cy.get('div[data-test-id=login-form-error]').should( chainer: 'not.exist')
    cy.get('h5[data-test-id=login-form-header]').contains( content: 'Login')
    cy.get('label').contains( content: 'Email address')
    cy.get('label').contains( content: 'Password')
    cy.get('button[data-test-id=login-form-sign-in-button]').contains( content: 'Sign in')
})
```

1. El elemento que muestra un error en los campos de texto, identificado por un *data-test-id,* no debe existir.
2. La cabecera del módulo, , identificada por un *data-test-id,* contiene el texto 'Login'
3. Hay una etiqueta que contiene el texto 'Email address'
4. Hay una etiqueta que contiene el texto 'Password'
5. El botón para iniciar sesión, identificado por un *data-test-id* contiene el texto 'Sign in'

Comentarios: en estos tests se muestran dos formas de identificar elementos. Por lo general, conviene ser lo más específico posible para evitar falsos positivos. Por otro lado, el test 5 no solo comprueba el texto del botón, también comprueba que existe ese botón pues en caso contrario también fallaría al no encontrar el elemento. Esta peculiaridad se repite en cualquier test.

4.1.1.2 No puede enviar un formulario no válido (email y contraseña incorrectos)

```
it( title: 'can not submit an invalid form (both invalid)', fn: () => {
    cy.get('div[data-test-id=login-form-email-input]').type(invalidUser.email)
    cy.get('div[data-test-id=login-form-password-input]').type(invalidUser.password)

    cy.get('button[data-test-id=login-form-sign-in-button]').click()
    cy.contains('Email and/or password does not match')
})
```

1. En el campo de texto del email, identificado por un *data-test-id,* introduce un email no valido.
2. En el campo de texto de la contraseña, identificado por un *data-test-id,* introduce una contraseña no valida.
3. En el botón de iniciar sesión, identificado por un *data-test-id,* realiza un click
4. Hay un elemento que contiene el texto 'Email and/or password does not match'

4.1.1.3 No puede enviar un formulario no válido (email incorrecto)

```
it( title: 'can not submit an invalid form (email invalid)', fn: () => {
    cy.get('div[data-test-id=login-form-email-input]').type(invalidUser.email)
    cy.get('div[data-test-id=login-form-password-input]').type(validUser.password)

    cy.get('button[data-test-id=login-form-sign-in-button]').click()
    cy.contains('Email and/or password does not match')
})
```

1. En el campo de texto del email, identificado por un *data-test-id,* introduce un email no valido.
2. En el campo de texto de la contraseña, identificado por un *data-test-id,* introduce una contraseña valida.
3. En el botón de iniciar sesión, identificado por un *data-test-id,* realiza un click
4. Hay un elemento que contiene el texto 'Email and/or password does not match

4.1.1.4 No puede enviar un formulario no válido (contraseña incorrecta)

```
it( title: 'can not submit an invalid form (email invalid)', fn: () => {
    cy.get('div[data-test-id=login-form-email-input]').type(invalidUser.email)
    cy.get('div[data-test-id=login-form-password-input]').type(validUser.password)

    cy.get('button[data-test-id=login-form-sign-in-button]').click()
    cy.contains('Email and/or password does not match')
})
```

1. En el campo de texto del email, identificado por un *data-test-id,* introduce un email valido.
2. En el campo de texto de la contraseña, identificado por un *data-test-id,* introduce una contraseña no valida.
3. En el botón de iniciar sesión, identificado por un *data-test-id,* realiza un click
4. Hay un elemento que contiene el texto 'Email and/or password does not match

4.1.1.5 Puede enviar un formulario válido

```
it( title: 'can submit a valid form', fn: () => {
    cy.get('div[data-test-id=login-form-email-input]').type(validUser.email)
    cy.get('div[data-test-id=login-form-password-input]').type(validUser.password)

    cy.get('button[data-test-id=login-form-sign-in-button]').click()
    cy.url().should( chainer: 'include', value: '/dashboard')
})
```

1. En el campo de texto del email, identificado por un *data-test-id,* introduce un email valido.
2. En el campo de texto de la contraseña, identificado por un *data-test-id,* introduce una contraseña valida.
3. En el botón de iniciar sesión, identificado por un *data-test-id,* realiza un click
4. La URL actual debe incluir '/dashboard'

4.1.2 Dashboard

El módulo Dashboard es el más simple de analizar, solo asignaremos un objeto con unas credenciales de usuario válidos. También usaremos la función *beforeEach* para, en este caso, abrir la ruta bypass que nos da acceso al Dashboard sin necesidad de pasar por el Login.

```
describe( title: 'Dashboard', fn: () => {
    const validUser = {
        email: 'test@admin.com',
        password: 'test123',
        name: 'Test'
    }
    beforeEach( fn: () => {
        cy.visit('http://localhost:3000/testing-bypass-dashboard/admin')
    })
```

4.1.2.1 Renderiza los elementos del módulo Dashboard

```
it( title: 'renders the Dashboard elements', fn: () => {
    cy.contains(`Welcome ${validUser.name}!`)
    cy.get('button[data-test-id=dashboard-employees-button]').contains( content: 'Employee list')
    cy.get('button[data-test-id=dashboard-register-button]').contains( content: 'Register')
    cy.get('button[data-test-id=dashboard-logout-button]').contains( content: 'Logout')
})
```

1. Hay un elemento que contiene el texto 'Welcome *validUser.name ('Test')*'.
2. El botón para navegar a la lista de trabajadores, identificado por un *data-test-id* contiene el texto 'Employee list'
3. El botón para navegar al formulario de registro, identificado por un *data-test-id* contiene el texto 'Register'
4. El botón para cerrar sesión, identificado por un *data-test-id* contiene el texto 'Logout'

4.1.2.2 Puede visitar la página de la lista de empleados

```
it( title: 'can visit the employees page', fn: () => {
    cy.get('button[data-test-id=dashboard-employees-button]').click()
    cy.url().should( chainer: 'include', value: '/employees')
})
```

1. En el botón para navegar a la lista de trabajadores, identificado por un *data-test-id,* realiza un click
2. La URL actual debe incluir '/employees'

4.1.2.3 Puede visitar la página del formulario de registro

```
it( title: 'can visit the register page', fn: () => {
    cy.get('button[data-test-id=dashboard-register-button]').click()
    cy.url().should( chainer: 'include', value: '/register')
})
```

1. En el botón para navegar al formulario de registro, identificado por un *data-test-id,* realiza un click
2. La URL actual debe incluir '/register'

4.1.2.4 Puede cerrar sesión

```
it( title: 'can logout', fn: () => {
    cy.get('button[data-test-id=dashboard-logout-button]').click()
    cy.url().should( chainer: 'include', value: '/login')
})
```

1. En el botón para cerrar sesión, identificado por un *data-test-id,* realiza un click
2. La URL actual debe incluir '/login'

Comentarios: aunque desde el punto de vista del desarrollador, se puede afirmar que el cierre de sesión no es más que redirigir al usuario a la ruta del Login, de cara al usuario, este proceso es transparente, por lo que desde su perspectiva, esta acción conlleva un cierre de sesión.

4.1.3 Employees

Como preparación previa para testear el módulo Employees, asignaremos un objeto con dos empleados y sus datos.

La razón de tener estos objetos es para tenerlos como referencia y tener tests más limpios, dado que de no usarlo, tendríamos que incrustar los datos directamente. Esto se conoce como *hard-code* y es una mala práctica en el ámbito de la programación.

Aunque en la aplicación tenemos acceso directo al fichero donde se aloja la base de datos y podrían extraerse los datos desde ahí, en casos reales la base de datos suele estar en un repositorio independiente al código de la interfaz. Es por ello que se ha decidido proceder con objetos en los que se definan los datos que estarían en el servidor.

La ruta bypass utilizada en este caso es la que nos brinda acceso a la lista de trabajadores.

```
describe( title: 'Employees', fn: () => {
    const employees = [
        {
            id: 1,
            firstName: 'Mario',
            lastName: 'Jimenez',
            email: 'mario@gmail.com',
            dateOfBirth: '11/05/1995',
            mobilePhone: '666555444',
            salary: '355000'
        },
        {
            id: 2,
            firstName: 'Test',
            lastName: 'E2E',
            email: 'e2e@testing.com',
            dateOfBirth: '11/11/1991',
            mobilePhone: '611666116',
            salary: '50000',
        }
    ]

    beforeEach( fn: () => {
        cy.visit('http://localhost:3000/testing-bypass-employees/admin')
    })
```

4.1.3.1 Renderiza los elementos del módulo Employees

```
it( title: 'renders the Employee List elements', fn: () => {
    cy.get('p[data-test-id=table-header-first-name]').contains( content: 'First Name')
    cy.get('p[data-test-id=table-header-last-name]').contains( content: 'Last Name')
    cy.get('p[data-test-id=table-header-email]').contains( content: 'Email')
    cy.get('p[data-test-id=table-header-date-of-birth]').contains( content: 'Date of Birth')
    cy.get('p[data-test-id=table-header-mobile-phone]').contains( content: 'Mobile Phone')
    cy.get('p[data-test-id=table-header-salary]').contains( content: 'Salary')
    cy.get('p[data-test-id=table-header-actions]').contains( content: 'Actions')

    cy.get(`p[data-test-id=employees-list-item-${employees[0].id}-first-name]`).contains(employees[0].firstName)
    cy.get(`p[data-test-id=employees-list-item-${employees[0].id}-last-name]`).contains(employees[0].lastName)
    cy.get(`p[data-test-id=employees-list-item-${employees[0].id}-email]`).contains(employees[0].email)
    cy.get(`p[data-test-id=employees-list-item-${employees[0].id}-date-of-birth]`).contains(employees[0].dateOfBirth)
    cy.get(`p[data-test-id=employees-list-item-${employees[0].id}-mobile-phone]`).contains(employees[0].mobilePhone)
    cy.get(`p[data-test-id=employees-list-item-${employees[0].id}-salary]`).contains( content: `${employees[0].salary}€`)
    cy.get(`button[data-test-id=employees-list-item-${employees[0].id}-edit]`).contains( content: 'Edit')
    cy.get(`button[data-test-id=employees-list-item-${employees[0].id}-remove]`).contains( content: 'Remove')

    cy.get(`p[data-test-id=employees-list-item-${employees[1].id}-first-name]`).contains(employees[1].firstName)
    cy.get(`p[data-test-id=employees-list-item-${employees[1].id}-last-name]`).contains(employees[1].lastName)
    cy.get(`p[data-test-id=employees-list-item-${employees[1].id}-email]`).contains(employees[1].email)
    cy.get(`p[data-test-id=employees-list-item-${employees[1].id}-date-of-birth]`).contains(employees[1].dateOfBirth)
    cy.get(`p[data-test-id=employees-list-item-${employees[1].id}-mobile-phone]`).contains(employees[1].mobilePhone)
    cy.get(`p[data-test-id=employees-list-item-${employees[1].id}-salary]`).contains( content: `${employees[1].salary}€`)
    cy.get(`button[data-test-id=employees-list-item-${employees[1].id}-edit]`).contains( content: 'Edit')
    cy.get(`button[data-test-id=employees-list-item-${employees[1].id}-remove]`).contains( content: 'Remove')
})
```

1. La columna del nombre, identificada por un *data-test-id,* contiene el texto ‘First Name’.
2. La columna del apellido, identificada por un *data-test-id,* contiene el texto ‘Last Name’.
3. La columna del email, identificada por un *data-test-id,* contiene el texto ‘Email’.
4. La columna de la fecha de nacimiento, identificada por un *data-test-id,* contiene el texto ‘Date of Birth’.
5. La columna del teléfono móvil, identificada por un *data-test-id,* contiene el texto ‘Mobile Phone’.
6. La columna del salario, identificada por un *data-test-id,* contiene el texto ‘Salary’.
7. La columna de las acciones, identificada por un *data-test-id,* contiene el texto ‘Actions’.
8. El nombre del primer empleado, identificado por un *data-test-id,* contiene el texto ‘Mario’.
9. El apellido del primer empleado, identificado por un *data-test-id,* contiene el texto ‘Jimenez’.
10. El email del primer empleado, identificado por un *data-test-id,* contiene el texto ‘mario@gmail.com’.
11. La fecha de nacimiento del primer empleado, identificada por un *data-test-id,* contiene el texto ‘11/05/1995’.
12. El número de móvil del primer empleado, identificado por un *data-test-id,* contiene el texto ‘666555444’.
13. El salario del primer empleado, identificado por un *data-test-id,* contiene el texto ‘355000’.
14. El botón para editar los detalles del primer empleado, identificado por un *data-test-id,* contiene el texto ‘Edit’.
15. El botón para eliminar al primer empleado, identificado por un *data-test-id,* contiene el texto ‘Remove’.
16. El nombre del segundo empleado, identificado por un *data-test-id,* contiene el texto ‘Test’.
17. El apellido del segundo empleado, identificado por un *data-test-id,* contiene el texto ‘E2E’.
18. El email del segundo empleado, identificado por un *data-test-id,* contiene el texto ‘e2e@testing.com’.
19. La fecha de nacimiento del segundo empleado, identificada por un *data-test-id,* contiene el texto ‘11/11/1991’.
20. El número de móvil del segundo empleado, identificado por un *data-test-id,* contiene el texto ‘611666116’.
21. El salario del segundo empleado, identificado por un *data-test-id,* contiene el texto ‘50000’.
22. El botón para editar el segundo empleado, identificado por un *data-test-id,* contiene el texto ‘Edit’.
23. El botón para eliminar al segundo empleado, identificado por un *data-test-id,* contiene el texto ‘Remove’.

4.1.3.2 Puede editar a un empleado

```
it( title: 'can edit an employee', fn: () => {
    cy.get(`button[data-test-id=employees-list-item-${employees[1].id}-edit]`).click()

    cy.get('input[name=email]').clear()
    cy.get(`div[data-test-id=employees-list-item-${employees[1].id}-email-input]`).type( text: 'changed@email.com')
    cy.get('input[name=mobilePhone]').clear()
    cy.get(`div[data-test-id=employees-list-item-${employees[1].id}-mobile-phone-input]`).type( text: '666666666')
    cy.get('input[name=salary]').clear()
    cy.get(`div[data-test-id=employees-list-item-${employees[1].id}-salary-input]`).type( text: '11111')

    cy.get(`button[data-test-id=employees-list-item-${employees[1].id}-submit]`).click()

    cy.get(`p[data-test-id=employees-list-item-${employees[1].id}-email]`).contains( content: 'changed@email.com')
    cy.get(`p[data-test-id=employees-list-item-${employees[1].id}-mobile-phone]`).contains( content: '666666666')
    cy.get(`p[data-test-id=employees-list-item-${employees[1].id}-salary]`).contains( content: '11111€')
    cy.get(`p[data-test-id=employees-list-item-${employees[1].id}-first-name]`).contains(employees[1].firstName)
    cy.get(`p[data-test-id=employees-list-item-${employees[1].id}-last-name]`).contains(employees[1].lastName)
    cy.get(`p[data-test-id=employees-list-item-${employees[1].id}-date-of-birth]`).contains(employees[1].dateOfBirth)
})
```

1. En el botón para editar los detalles del segundo empleado, identificado por un *data-test-id,* realiza un click
2. En el campo de texto para modificar el email del segundo empleado, identificado por el parámetro *name*, elimina el valor.
3. En el campo de texto para modificar el email del segundo empleado, identificado por un *data-test-id*, introduce 'changed@email.com'.
4. En el campo de texto para modificar el número de móvil del segundo empleado, identificado por el parámetro *name*, elimina el valor.
5. En el campo de texto para modificar el número de móvil del segundo empleado, identificado por un *data-test-id*, introduce '666666666'.
6. En el campo de texto para modificar el salario del segundo empleado, identificado por el parámetro *name*, elimina el valor.
7. En el campo de texto para modificar el salario del segundo empleado, identificado por un *data-test-id*, introduce '11111'.
8. En el botón para enviar los detalles modificados del segundo empleado al servidor, identificado por un *data-test-id,* realiza un click
9. El email del segundo empleado, identificado por un *data-test-id,* contiene el texto 'changed@email.com'.
10. El número de móvil del segundo empleado, identificado por un *data-test-id,* contiene el texto '666666666'.
11. El salario del segundo empleado, identificado por un *data-test-id,* contiene el texto '11111'.
12. El nombre del segundo empleado, identificado por un *data-test-id,* contiene el texto 'Test'.
13. El apellido del segundo empleado, identificado por un *data-test-id,* contiene el texto 'E2E'.
14. La fecha de nacimiento del segundo empleado, identificada por un *data-test-id,* contiene el texto '11/11/1991'.

Comentarios: puede observarse que al final se comprueba que los datos modificados efectivamente han cambiado en la interfaz y también se comprueba que los no modificados no se han visto afectados. Aunque a priori parezca carecer de sentido, puede ser un foco de errores, dado que un fallo en el front-end podría hacer que se enviaran parámetros al servidor que no participan en la modificación.

4.1.3.3 Puede eliminar a un empleado

```
it( title: 'can remove an employee', fn: () => {
    cy.get(`button[data-test-id=employees-list-item-${employees[1].id}-remove]`).click()

    cy.get(`div[data-test-id=employees-list-item-${employees[1].id}]`).should( chainer: 'not.exist')
})
```

1. En el botón para eliminar al segundo empleado, identificado por un *data-test-id,* realiza un click.
2. El componente del segundo empleado, identificado por un *data-test-id,* no debe existir.

4.1.3.4 Puede volver al Dashboard

```
it( title: 'can return to Dashboard', fn: () => {
    cy.get('button[data-test-id=return-to-dashboard-button]').click()
    cy.url().should( chainer: 'include', value: '/dashboard')
})
```

1. En el botón para volver al Dashboard, identificado por un *data-test-id,* realiza un click.
2. La URL actual debe incluir '/dashboard'

4.1.4 Register

Los preparativos de los tests del módulo Register son un objeto con los datos de registro de un empleado válido y la función para navegar a la ruta bypass.

```
describe( title: 'Register form', fn: () => {
    const newEmployee = {
        firstName: 'New',
        lastName: 'Employee',
        email: 'new@employee.com',
        day: '15',
        month: '12',
        year: '1992',
        mobilePhone: '666556446',
        salary: '35700'
    }

    beforeEach( fn: () => {
        cy.visit('http://localhost:3000/testing-bypass-register/admin')
    })
```

4.1.4.1 Renderiza los elementos del módulo Register

```
it( title: 'renders the Register form elements', fn: () => {
    cy.get('label').contains( content: 'Email address')
    cy.get('label').contains( content: 'First name')
    cy.get('label').contains( content: 'Last name')
    cy.get('label').contains( content: 'Mobile phone')
    cy.get('svg[data-test-id=expansion-panel-button]').click()
    cy.get('p[data-test-id=expansion-panel-content]').contains( content: 'We require a Spanish phone number for additional contact details')
    cy.get('h6').contains( content: 'Date of Birth')
    cy.get('label').contains( content: 'DD')
    cy.get('label').contains( content: 'MM')
    cy.get('label').contains( content: 'YYYY')
    cy.get('label').contains( content: 'Salary')
})
```

1. Hay una etiqueta que contiene el texto 'Email address'
2. Hay una etiqueta que contiene el texto 'First name'
3. Hay una etiqueta que contiene el texto 'Last name'
4. Hay una etiqueta que contiene el texto 'Mobile phone'
5. Hay una etiqueta que contiene el texto 'Email address'
6. En el botón para desplegar el panel de expansión, identificado por un *data-test-id,* realiza un click.
7. El mensaje de información del desplegable, identificado por un *data-test-id,* contiene el texto 'We require a Spanish phone number for additional contact details'.
8. Hay una cabecera que contiene el texto 'Mobile phone'
9. Hay una etiqueta que contiene el texto 'DD'
10. Hay una etiqueta que contiene el texto 'MM'
11. Hay una etiqueta que contiene el texto 'YYYY'
12. Hay una etiqueta que contiene el texto 'Salary'

4.1.4.2 Valida el formulario

```
it( title: 'validates the form', fn: () => {
    cy.get('div[data-test-id=registration-form-email-input]').type( text: 'incorrectEmail@test').find( selector: 'input').blur()
    cy.get('p[data-test-id=registration-form-email-error]').contains( content: 'Please enter a valid email')
    cy.get('div[data-test-id=registration-form-email-input]').find( selector: 'input').clear()
    cy.get('div[data-test-id=registration-form-email-input]').type( text: 'incorrectEmail.com').find( selector: 'input').blur()
    cy.get('p[data-test-id=registration-form-email-error]').contains( content: 'Please enter a valid email')

    cy.get('div[data-test-id=registration-form-first-name-input]').type( text: 'A').find( selector: 'input').blur()
    cy.get('p[data-test-id=registration-form-first-name-error]').contains( content: 'First name must be at least 2 characters long')

    cy.get('div[data-test-id=registration-form-last-name-input]').type( text: 'B').find( selector: 'input').blur()
    cy.get('p[data-test-id=registration-form-last-name-error]').contains( content: 'Last name must be at least 2 characters long')

    cy.get('div[data-test-id=registration-form-mobile-phone-input]').type( text: '61234532').find( selector: 'input').blur()
    cy.get('p[data-test-id=registration-form-mobile-phone-error]').contains( content: 'Please enter a valid phone number.')
    cy.get('div[data-test-id=registration-form-mobile-phone-input]').find( selector: 'input').clear()
    cy.get('div[data-test-id=registration-form-mobile-phone-input]').type( text: '312345321').find( selector: 'input').blur()
    cy.get('p[data-test-id=registration-form-mobile-phone-error]').contains( content: 'Please enter a valid phone number.')

    cy.get('div[data-test-id=date-field-day-input]').type( text: '32')
    cy.get('div[data-test-id=date-field-day-input]').find( selector: 'aria-invalid').should( chainer: 'be.true')
    cy.get('div[data-test-id=date-field-month-input]').type( text: '13')
    cy.get('div[data-test-id=date-field-month-input]').find( selector: 'aria-invalid').should( chainer: 'be.true')
    cy.get('div[data-test-id=date-field-year-input]').type( text: '1000')
    cy.get('div[data-test-id=date-field-year-input]').find( selector: 'aria-invalid').should( chainer: 'be.true')

    cy.get('div[data-test-id=registration-form-salary-input]').type( text: 'SSSSSS').find( selector: 'input').blur()
    cy.get('p[data-test-id=registration-form-salary-error]').contains( content: 'Salary must be a number')
})
```

1. En el campo de texto para introducir el email, identificado por un *data-test-id*, introduce 'incorrectEmail@test' y acciona el evento *onBlur*.
2. El mensaje de error del campo email, identificado por un *data-test-id,* contiene el texto 'Please enter a valid email'.
3. En el campo de texto para introducir el email, identificado por un *data-test-id*, elimina el valor.
4. En el campo de texto para introducir el email, identificado por un *data-test-id*, introduce 'incorrectEmail.com' y acciona el evento *onBlur*.
5. El mensaje de error del campo email, identificado por un *data-test-id,* contiene el texto 'Please enter a valid email'.
6. En el campo de texto para introducir el nombre, identificado por un *data-test-id*, introduce 'A' y acciona el evento *onBlur*.
7. El mensaje de error del campo nombre, identificado por un *data-test-id,* contiene el texto 'First name must be at least 2 characters long'.
8. En el campo de texto para introducir el apellido, identificado por un *data-test-id*, introduce 'B' y acciona el evento *onBlur*.
9. El mensaje de error del campo apellido, identificado por un *data-test-id,* contiene el texto 'Last name must be at least 2 characters long'.
10. En el campo de texto para introducir el teléfono móvil, identificado por un *data-test-id*, introduce '61234532' y acciona el evento *onBlur*.
11. El mensaje de error del campo teléfono móvil, identificado por un *data-test-id,* contiene el texto 'Please enter a valid phone number'.
12. En el campo de texto para introducir el teléfono móvil, identificado por un *data-test-id*, elimina el valor.
13. En el campo de texto para introducir el teléfono móvil, identificado por un *data-test-id*, introduce '312345321' y acciona el evento *onBlur*.
14. El mensaje de error del campo teléfono móvil, identificado por un *data-test-id,* contiene el texto 'Please enter a valid phone number'.
15. En el campo de texto para introducir el día, identificado por un *data-test-id*, introduce '32'.
16. En el campo de texto para introducir el día, identificado por un *data-test-id*, encuentra el atributo *aria-invalid* cuyo valor debe ser 'true'.
17. En el campo de texto para introducir el mes, identificado por un *data-test-id*, introduce '13'.
18. En el campo de texto para introducir el mes, identificado por un *data-test-id*, encuentra el atributo aria-*invalid* cuyo valor debe ser 'true'.
19. En el campo de texto para introducir el año, identificado por un *data-test-id*, introduce '1000'.
20. En el campo de texto para introducir el año, identificado por un *data-test-id*, encuentra el atributo *aria-invalid* cuyo valor debe ser 'true'.
21. En el campo de texto para introducir el salario, identificado por un *data-test-id*, introduce 'SSSSSS' y acciona el evento *onBlur*.
22. El mensaje de error del campo salario, identificado por un *data-test-id,* contiene el texto 'Salary must be a number'.

Comentarios: en este caso, se ha comprobado que la validación funciona, para ello se han introducido datos no válidos. No se ha creado un objeto que guarde estos detalles para mostrar cómo se visualizaría el código cuando se incrusta el contenido directamente en el test. Solo se van a definir estos datos una vez en el código, así que podría decirse que no es tan mala práctica ya que de necesitar cambiar algo, solo tendríamos que modificarlo en una línea del código, pero aun así, tenerlo todo definido en un único objeto facilita la edición y nos ofrece más escalabilidad, pues en un futuro puede que sea utilizado más de una vez.
El evento onBlur es llamado cuando el usuario sale de un campo de texto.
El atributo aria-invalid indica que el valor introducido no pasa la validación indicada en la aplicación.

4.1.4.3 Puede registrar un nuevo empleado

```
it( title: 'can register a new employee',  fn: () => {
    cy.get('div[data-test-id=registration-form-email-input]').type(newEmployee.email)
    cy.get('div[data-test-id=registration-form-first-name-input]').type(newEmployee.firstName)
    cy.get('div[data-test-id=registration-form-last-name-input]').type(newEmployee.lastName)
    cy.get('div[data-test-id=registration-form-mobile-phone-input]').type(newEmployee.mobilePhone)
    cy.get('div[data-test-id=date-field-day-input]').type(newEmployee.day)
    cy.get('div[data-test-id=date-field-month-input]').type(newEmployee.month)
    cy.get('div[data-test-id=date-field-year-input]').type(newEmployee.year)
    cy.get('div[data-test-id=registration-form-salary-input]').type(newEmployee.salary)

    cy.get('button[data-test-id=registration-form-register-button]').click()

    cy.get('p[data-test-id=registration-completed-view-title]').contains( content: 'Registration successfully completed')
    cy.get('button[data-test-id=return-to-dashboard-button]').click()
    cy.url().should( chainer: 'include',  value: '/dashboard')

    cy.get('button[data-test-id=dashboard-employees-button]').click()
    cy.url().should( chainer: 'include',  value: '/employees')

    cy.contains(newEmployee.firstName)
    cy.contains(newEmployee.lastName)
    cy.contains(newEmployee.email)
    cy.contains(`${newEmployee.day}/${newEmployee.month}/${newEmployee.year}`)
    cy.contains(newEmployee.mobilePhone)
    cy.contains(`${newEmployee.salary}€`)
})
```

1. En el campo de texto del email, identificado por un *data-test-id*, introduce ‘new@employee.com’.
2. En el campo de texto para introducir el nombre, identificado por un *data-test-id*, introduce ‘New’.
3. En el campo de texto del apellido, identificado por un *data-test-id*, introduce ‘Employee’.
4. En el campo de texto del teléfono móvil, identificado por un *data-test-id*, introduce ‘666556446’.
5. En el campo de texto para introducir el día, identificado por un *data-test-id*, introduce ‘15’.
6. En el campo de texto para introducir el mes, identificado por un *data-test-id*, introduce ‘12’.
7. En el campo de texto para introducir el año, identificado por un *data-test-id*, introduce ‘1992’.
8. En el campo de texto para introducir el salario, identificado por un *data-test-id*, introduce ‘35700’.
9. En el botón para registrar al empleado, identificado por un *data-test-id,* realiza un click.
10. El mensaje de confirmación, identificado por un *data-test-id,* contiene el texto ‘Registration successfully completed’.
11. En el botón para volver al módulo Dashboard, identificado por un *data-test-id,* realiza un click.
12. La URL actual debe incluir ‘/dashboard’
13. En el botón para navegar al módulo Employees, identificado por un *data-test-id,* realiza un click.
14. La URL actual debe incluir ‘/employees
15. Hay un elemento que contiene el texto ‘New’.
16. Hay un elemento que contiene el texto ‘Employee’.
17. Hay un elemento que contiene el texto ‘new@employee.com’.
18. Hay un elemento que contiene el texto ‘15/12/1992’.
19. Hay un elemento que contiene el texto ‘‘666556446’.
20. Hay un elemento que contiene el texto ‘35700’.

Comentarios: el evento onBlur es accionado automáticamente cuando introducimos texto en el siguiente campo, entonces, indirectamente también se estaría comprobando la validación, pues de fallar alguna regla, el botón de registrar empleado estaría desactivado y fallaría el click. Aunque visitemos dos rutas ajenas al Register, es necesario comprobar que el usuario efectivamente se ha renderizado en la lista de empleados.

4.2 Asociado (associate)

Las diferencias entre el rol de administrador y el asociado aparecen en el módulo Dashboard y Employees. De hecho, este rol no tiene acceso al módulo Register.

Como resultado, se obviarán los tests del módulo Login al ser idénticos y los del Register al no existir tests.

4.2.1 Dashboard

4.2.1.1 Muestra el botón de registrar empleado desactivado

```
it( title: 'shows the register button disabled', fn: () => {
    cy.get('button[data-test-id=dashboard-register-button]').should( chainer: 'be.disabled')
})
```

1. El botón para registrar un empleado, identificado por un *data-test-id,* debe estar desactivado.

4.2.2 Employees

4.2.2.1 Puede editar el email y teléfono móvil de un empleado

```
it( title: 'can edit the email and mobile phone of an employee', fn: () => {
    cy.get(`button[data-test-id=employees-list-item-${employees[1].id}-edit]`).click()

    cy.get('input[name=email]').clear()
    cy.get(`div[data-test-id=employees-list-item-${employees[1].id}-email-input]`).type( text: 'changed@email.com')
    cy.get('input[name=mobilePhone]').clear()
    cy.get(`div[data-test-id=employees-list-item-${employees[1].id}-mobile-phone-input]`).type( text: '666666666')

    cy.get(`div[data-test-id=employees-list-item-${employees[1].id}-salary-input]`).should( chainer: 'not.exist')

    cy.get(`button[data-test-id=employees-list-item-${employees[1].id}-submit]`).click()

    cy.get(`p[data-test-id=employees-list-item-${employees[1].id}-email]`).contains( content: 'changed@email.com')
    cy.get(`p[data-test-id=employees-list-item-${employees[1].id}-mobile-phone]`).contains( content: '666666666')
})
```

1. En el botón para editar los detalles del segundo empleado, identificado por un *data-test-id,* realiza un click
2. En el campo de texto para modificar el email del segundo empleado, identificado por el parámetro *name*, elimina el valor.
3. En el campo de texto para modificar el email del segundo empleado, identificado por un *data-test-id*, introduce 'changed@email.com'.
4. En el campo de texto para modificar el número de móvil del segundo empleado, identificado por el parámetro *name*, elimina el valor.
5. El campo de texto para modificar el número de móvil del segundo empleado, identificado por un *data-test-id*, introduce '666666666'.
6. El campo de texto para modificar el salario del segundo empleado, identificado por el parámetro *name*, no debe existir.
7. En el botón para enviar los detalles modificados del segundo empleado al servidor, identificado por un *data-test-id,* realiza un click
8. El email del segundo empleado, identificado por un *data-test-id,* contiene el texto 'changed@email.com'.
9. El número de móvil del segundo empleado, identificado por un *data-test-id,* contiene el texto '666666666'.

4.3 Usuario (user)

El rol de usuario es el más limitado, pues sólo dispone de poderes de lectura y además no puede visualizar todos los detalles de los empleados. Este caso tiene las mismas peculiaridades que el asociado en cuanto al módulo Dashboard, es por ello que se obviarán también estos tests.

4.3.1 Employees

4.3.1.1 Renderiza los elementos del módulo Employees

```
it( title: 'renders the Employee List elements', fn: () => {
    cy.get('p[data-test-id=table-header-first-name]').contains( content: 'First Name')
    cy.get('p[data-test-id=table-header-last-name]').contains( content: 'Last Name')
    cy.get('p[data-test-id=table-header-email]').contains( content: 'Email')
    cy.get('p[data-test-id=table-header-date-of-birth]').contains( content: 'Date of Birth')
    cy.get('p[data-test-id=table-header-mobile-phone]').contains( content: 'Mobile Phone')
    cy.get('p[data-test-id=table-header-salary]').should( chainer: 'not.exist')
    cy.get('p[data-test-id=table-header-actions]').should( chainer: 'not.exist')

    cy.get(`p[data-test-id=employees-list-item-${employees[0].id}-first-name]`).contains(employees[0].firstName)
    cy.get(`p[data-test-id=employees-list-item-${employees[0].id}-last-name]`).contains(employees[0].lastName)
    cy.get(`p[data-test-id=employees-list-item-${employees[0].id}-email]`).contains(employees[0].email)
    cy.get(`p[data-test-id=employees-list-item-${employees[0].id}-date-of-birth]`).contains(employees[0].dateOfBirth)
    cy.get(`p[data-test-id=employees-list-item-${employees[0].id}-mobile-phone]`).contains(employees[0].mobilePhone)
    cy.get(`p[data-test-id=employees-list-item-${employees[0].id}-salary]`).should( chainer: 'not.exist')
    cy.get(`button[data-test-id=employees-list-item-${employees[0].id}-edit]`).should( chainer: 'not.exist')
    cy.get(`button[data-test-id=employees-list-item-${employees[0].id}-remove]`).should( chainer: 'not.exist')

    cy.get(`p[data-test-id=employees-list-item-${employees[1].id}-first-name]`).contains(employees[1].firstName)
    cy.get(`p[data-test-id=employees-list-item-${employees[1].id}-last-name]`).contains(employees[1].lastName)
    cy.get(`p[data-test-id=employees-list-item-${employees[1].id}-email]`).contains(employees[1].email)
    cy.get(`p[data-test-id=employees-list-item-${employees[1].id}-date-of-birth]`).contains(employees[1].dateOfBirth)
    cy.get(`p[data-test-id=employees-list-item-${employees[1].id}-mobile-phone]`).contains(employees[1].mobilePhone)
    cy.get(`p[data-test-id=employees-list-item-${employees[1].id}-salary]`).should( chainer: 'not.exist')
    cy.get(`button[data-test-id=employees-list-item-${employees[1].id}-edit]`).should( chainer: 'not.exist')
    cy.get(`button[data-test-id=employees-list-item-${employees[1].id}-remove]`).should( chainer: 'not.exist')
})
```

6. La columna del salario, identificada por un *data-test-id,* no debe existir.
7. La columna de las acciones, identificada por un *data-test-id,* no debe existir.

13. El salario del primer empleado, identificado por un *data-test-id,* no debe existir.
14. El botón para editar los detalles del primer empleado, identificado por un *data-test-id,* no debe existir.
15. El botón para eliminar al primer empleado, identificado por un *data-test-id,* no debe existir.

21. El salario del segundo empleado, identificado por un *data-test-id,* no debe existir.
22. El botón para editar los detalles del segundo empleado, identificado por un *data-test-id,* no debe existir.
23. El botón para eliminar al segundo empleado, identificado por un *data-test-id,* no debe existir.

Comentarios: para evitar una repetición de los mismos tests que comparte con el rol de administrador y asociado, sólo se han indicado los tests que son difieren, el resto puede comprobarse en el código.

4.4 Resultados

En los apartados anteriores se ha indicado el alcance de las pruebas end-to-end. El siguiente paso sería ejecutar esos tests utilizando Cypress y comprobar el resultado.

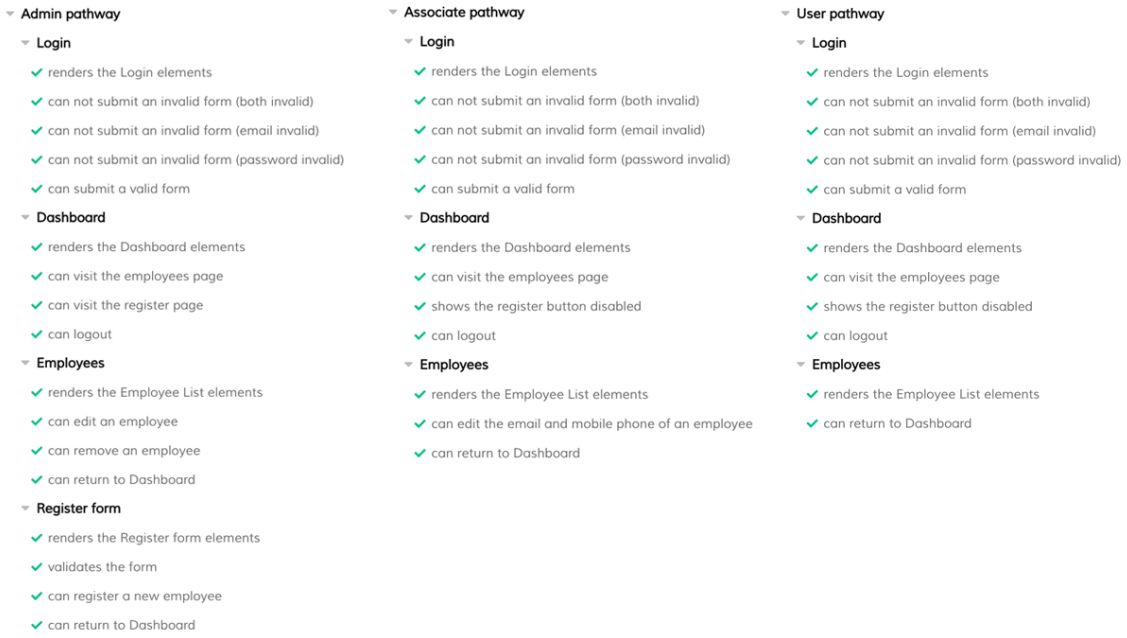

Figura 21 – Vista de los bloques de tests ejecutados en Cypress

En la figura 21 se puede observar, desde la interfaz gráfica que proporciona Cypress, que todos los tests se han ejecutado con éxito. Están separados por los roles administrador, asociado y usuario.

En total se han ejecutado 40 sentencias *it*, lo cual no significa que se hayan ejecutado 40 tests ya que realmente cada línea en la que se declara una función de Cypress es un test. Podría decirse que se han ejecutado 40 bloques de tests que se centran en testear una parte o acción de la aplicación.

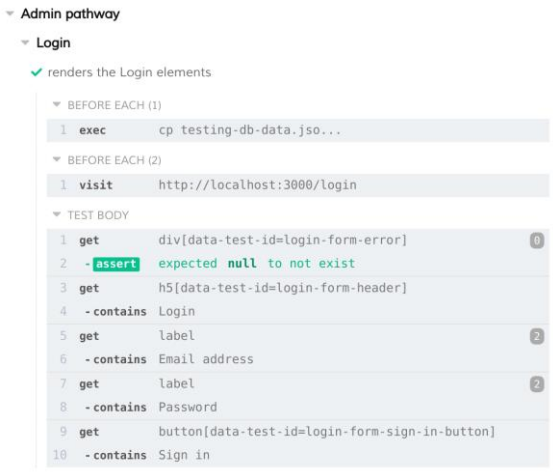

Figura 22 – Vista de los tests ejecutados en el bloque 'renders the Login elements'

Cypress permite visualizar los tests de cada bloque haciendo clic en estos tal y como puede contemplarse en la figura 22.

Respecto al tiempo total de ejecución de los tests, el cual es la única unidad de medida relevante que nos aportan los tests, también se ve representado en esta interfaz.

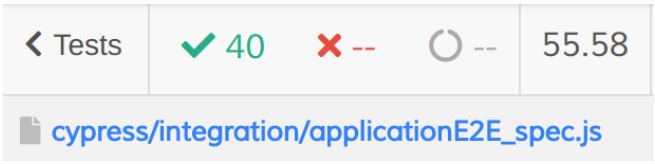

Figura 23 – Resumen del resultado de la ejecución

Como puede verse en la figura 23, los tests han tomado un total de **55.59 segundos** en ejecutarse y el fichero JavaScript con los tests es *applicationE2E_spec.js.* También indica que hay 40 bloques de tests ejecutados con éxito y que ninguno ha fallado.

El tiempo de ejecución puede variar según en qué máquina se ejecuten dado que hay muchos factores implicados, tanto software como hardware. Este dato carece de relevancia siempre y cuando todos los tests se ejecuten bajo las mismas condiciones.

Para concluir con este apartado, y también relacionado con el tiempo de ejecución de estas pruebas, hay que mencionar el hecho de que en nuestra aplicación el servidor se ejecuta en la red local por lo que la comunicación entre cliente y servidor es prácticamente instantánea. Si el servidor fuera externo el tiempo de ejecución sería superior, aunque en conexiones modernas no debería influir demasiado en el resultado.

5 PRUEBAS UNITARIAS Y DE INTEGRACIÓN

Las últimas pruebas que quedan por realizar son las unitarias y de integración, que se van a analizar en conjunto debido a que ambas son ejecutadas a nivel de código. Tal y como se ha explicado, se utilizará Jest y Enzyme. La diferencia principal a nivel de código es que los tests unitarios utilizaran la función *shallow()* que solo renderiza el componente aislado y *mount()* para los tests de integración pues esta función también renderiza los componentes hijo.

Conviene mencionar algunas diferencias respecto lo visto anteriormente en los tests end-to-end:

- A diferencia de los tests end-to-end, donde los bloques *it* contenían varias líneas que ejecutaban un test cada uno, en estos tests normalmente habrá un *expect* dentro de cada *it*, o en otras palabras, un test por *it.* Debido a este hecho, no se va a mostrar código, pues no es tan intuitivo como en Cypress y solo aportaría más complejidad a la hora de entender que se está analizando.
- Los tests que ejecutemos con Jest serán mostrados en el terminal, no se usará una interfaz gráfica como la que si existe en Cypress.
- La estructura también es diferente ya que aquí habrá un fichero de test de integración por módulo y un fichero de test unitario por componente. En este caso vamos a realizar un análisis por módulo en vez de por rol y dentro de cada módulo la división será por fichero de test. Los ficheros cuya terminación es *.test.js* hacen referencia a los tests de integración y los acabados en *.spec.js* a los tests unitarios.
- La identificación de elementos en el DOM siempre será por *data-test-id*, atributo que ya se ha utilizado para encontrar algunos elementos en los tests end-to-end. En Cypress se utilizaron funciones que identificaban elementos según un texto, con Enzyme sólo se utilizará el identificador asignado en el código de la aplicación para este fin.

Las simulaciones o 'mocks' de las llamadas al servidor en los tests de integración se obviarán, pero hay que tener en cuenta que una llamada al servidor es una llamada asíncrona que devuelve una promesa no resuelta. Esta promesa representa el estado de carga, es decir, el estado que hay entre realizar la llamada al servidor y el momento en el que se reciben los datos. Para llegar a ese estado en el que recibimos los datos, hay que resolver la promesa de forma manual.

En algunos tests se analizará el estado de la aplicación antes de cargar los datos y después de obtener los datos, aunque en otros se supondrá que los datos ya han llegado.

Recordando la estructura de ficheros, se tiene mínimo un contenedor y un componente por módulo. No existirán tests unitarios para los contenedores pues estos solo incluyen estados y lógica que consume el componente hijo y esta lógica estará cubierta por los tests de integración.

En los tests unitarios hay un test que se repite entre componentes cuya descripción es 'renders as expected'. En este test lo que se hace es comparar 'snapshots' cada vez que se ejecuta el test. Un snapshot no es mas que una copia del DOM asociado al componente que se está testeando. Este test calcula la diferencia entre el último snapshot que se tenga actualizado y el DOM que se renderiza cuando se ejecuta el test. Es una herramienta muy potente ya que básicamente realiza un *diff* de dos ficheros de texto, lo cual tiene un tiempo de ejecución mínimo.

En conclusión, en este capítulo se procede a analizar los módulos (tests de integración) y los componentes (tests unitarios) asociados a estos.

Antes de abordar la descripción de cada fichero de tests, conviene mencionar algunos ejemplos a nivel de código para comprender como se realiza cada prueba, pues en esta sección no se describirá el código de cada fichero.

```
const findSignInButton = (wrapper) => wrapper.find('[data-test-id~="login-form-sign-in-button"]').hostNodes()

const findEmailInput = (wrapper) => wrapper.find('[data-test-id~="login-form-email-input"]').find('input')

const findPasswordInput = (wrapper) => wrapper.find('[data-test-id~="login-form-password-input"]').find('input')

const findLoginError = (wrapper) => wrapper.find('[data-test-id~="login-form-error"]').hostNodes()
```

En este extracto del fichero de tests de integración del módulo Login, lo primero que se define son funciones para acceder a los elementos del DOM que se van a testear. Esta práctica es habitual especialmente en tests de integración.

Las cuatro funciones reciben el mismo argumento de nombre *wrapper*. Este elemento no es más que el objeto de Enzyme devuelto por la función *mount()*. En el caso de los tests unitarios sería el objeto recibido al llamar a la función *shallow()*.

Partiendo de la primera función definida como *findSignInButton()*, se hace uso de la función *.find()* de Enzyme para identificar al elemento por medio de su *data-test-id.* Lo siguiente que aparece es otra función de Enzyme, *.hostNodes()* que devuelve el elemento HTML en vez del componente de React.

Por ejemplo, en este caso se está identificando un botón. Tras utilizar *.find()* se identificaría el componente 'Button' de React y después con *.hostNodes()* se llegaría al elemento 'button' de HTML que es en el que se tiene interés en manipular.

La segunda función *findEmailInput()* se diferencia de la anterior en que en esta se realiza un segundo *find()* en vez de invocar a *hostNodes().* La finalidad es la misma, se quiere identificar un campo de texto, obteniendo el componente de React 'Input' en el primer *.find()* y el elemento HTML 'input' en el segundo.

Realmente ambos casos podrían resolverse con cualquiera de las funciones mencionadas y, aunque en el ejemplo se utilizan ambas para mostrar dos caminos para llegar a lo mismo, realizarlo con *.find()* va a ser siempre más específico dado que al aplicar *.hostNodes()* puede darse el caso en el que el elemento HTML devuelto contenga subelementos a los que se podría acceder con una búsqueda adicional.

Estas cuatro funciones no son estrictamente necesarias, pero por seguir buenas prácticas de escalabilidad y tener un código más limpio, es conveniente definirlas para no tener que repetir estas líneas de código en múltiples ocasiones a lo largo del fichero.

Entonces, para identificar un elemento al que se le quiere simular alguna acción o comprobar algún resultado, se utilizarán estas funciones cuyo nombre indica el elemento a encontrar.

```
it( title: 'shows an error if email/password does not match', fn: () => {
    findEmailInput(wrapper).simulate('change', {
        target: {
            value: 'mario@jimenez.com'
        },
    })

    findPasswordInput(wrapper).simulate('change', {
        target: {
            value: '123qwerty'
        },
    })

    findSignInButton(wrapper).simulate('click')

    expect(findLoginError(wrapper)).toHaveLength( expected: 1)
})
```

Siguiendo con los tests de integración del módulo Login se tienen tres acciones y un *expect()* donde se valida el resultado del test.

Empezando por la primera acción, se identificaría el campo de texto del email y con la función de Enzyme *.simulate()* se simula un cambio en el valor del campo. Se realizaría el mismo proceso para el campo contraseña y el botón para iniciar sesión, aunque en este último caso se realiza un click.

Finalmente el test comprueba que al aplicar las acciones previas, las cuales buscan forzar un error en inicio de sesión, se muestra el elemento que representa el mensaje de error. Esta comprobación se hace por medio de *toHaveLength()* que es un matcher al que da acceso *expect()* de Jest.

```
expect(wrapper.find(LoginContainer).props().location.pathname).toBe( expected: '/dashboard')
```

Cuando se quiere comprobar la ruta en la que se encuentra la interfaz dentro de los tests de integración, se accede al objeto 'location', el cual es un objeto que provee React Router, la librería utilizada para crear rutas en React.

Este objeto, entre otras cosas, tiene el atributo 'pathname' que representa la ruta en la que se encuentra la aplicación actualmente.

Entonces, en este test se comprueba que tras ciertas acciones realizadas previamente, la aplicación se encuentra en la ruta '/dashboard'.

```
it( title: 'renders as expected', fn: () => {
    expect(wrapper).toMatchSnapshot()
})
```

Este tipo de tests es el mencionado al principio del capítulo en el que el matcher *toMatchSnapshot()* de Jest comprueba que el snapshot antiguo es idéntico al calculado en el momento en el que se ejecuta el test.

```
exports[`Login Unit Tests when there are no errors renders as expected 1`] = `
<div
  className="makeStyles-loginBox-1"
>
  <WithStyles(ForwardRef(Typography))
    color="primary"
    data-test-id="login-form-header"
    style={
      Object {
        "marginBottom": "2em",
      }
    }
    variant="h5"
  >
    Login
  </WithStyles(ForwardRef(Typography))>
  <div
    style={
      Object {
        "display": "flex",
      }
    }
  >
```

Esta sería la estructura de un snapshot tomado al módulo Login donde puede verse el aspecto del DOM.

Por ejemplo, si se modificara alguna propiedad CSS como 'marginBottom' con un valor nuevo, el test lo detectaría y fallaría mostrando donde está la diferencia. Entonces, Jest daría la opción de actualizar el snapshot para incluir este cambio en el caso de que sea una modificación esperada.

```
describe( title: 'before loading', fn: () => {
    it( title: 'does not render any employee', fn: () => {
        expect(wrapper.find('[data-test-id~="employees-list-item"]').hostNodes()).toHaveLength( expected: 0)
    })
})
describe( title: 'after loading', fn: () => {
    beforeEach( fn: async () => {
        await flushPromises()
        wrapper.update()
    })
    it( title: 'renders all the employees', fn: () => {
        expect(wrapper.find('[data-test-id~="employees-list-item"]').hostNodes()).toHaveLength( expected: 3)
    })
```

Finalmente, el último ejemplo por mostrar es el de los estados de carga. Para ello se muestra un extracto de los tests de integración del módulo Employees. No se han utilizado funciones para la identificación de elementos para mostrar la diferencia respecto al caso en el que se utilizan funcionas para este fin.

Dentro de cada *it()* no aparece nada que no se haya explicado previamente, lo que sí es nuevo es lo incluido dentro del *beforeEach()*. Esta función de Jest ejecuta las líneas de código que se definan dentro de esta justo antes de que se inicie cada test o *it()*. En este caso se trata de una función asíncrona que llama a la función *flushPromises()* que lo único que hace es resolver la promesa devuelta por la simulación del servidor.

Tras resolver la promesa, se hace uso de la función de Enzyme *update()* que sincroniza el DOM con los cambios externos a los que ha sido expuesto el componente de React.

Una vez realizado esto se podría proceder con cualquier test que requiera de tener los datos del servidor cargados en la aplicación.

A continuación se muestran las pruebas que se han realizado a cada módulo. Estas pruebas están divididas en ficheros con tests de integración (*Módulo.**test**.js)* y tests unitarios (*Módulo.**spec**.js)*. Sólo se mencionará el descriptor de cada prueba, no se describirá el código de estos ficheros dado que en gran parte es código basado en los ejemplos anteriores.

5.1 Login

5.1.1 Login.test.js (integración)

- Validación
    1. De inicio, el formulario de inicio de sesión no muestra ningún error
    2. De inicio, muestra el botón para iniciar sesión está desactivado
    3. Muestra el botón para iniciar sesión desactivado si el campo para la contraseña está vacío
    4. Muestra el botón para iniciar sesión desactivado si el campo para el email está vacío
- Al enviar el formulario
    1. Muestra un error si el email o contraseña no coinciden con la lista de usuarios del servidor
    2. Redirecciona la interfaz al dashboard si las credenciales son correctas

5.1.2 Login.spec.js (unitario)

- Cuando no hay errores
    1. El componente se renderiza correctamente
    2. Los campos de texto tienen los valores que llegan por las propiedades
- Cuando hay errores
    3. El componente se renderiza correctamente

5.2 Dashboard

5.2.1 Dashboard.test.js (integración)

1. Redirecciona la interfaz a la página de inicio de sesión cuando se hace click en el botón ‘Logout’
2. Redirecciona la interfaz a la página de empleados cuando se hace click en el botón ‘Employees list’
3. Redirecciona la interfaz a la página de registro cuando se hace click en el botón ‘Register’

5.2.2 Dashboard.spec.js (unitario)

1. El componente se renderiza correctamente

- Cuando el rol es administrador
    2. El botón 'Logout' no está desactivado
    3. El botón 'Employees list' no está desactivado
    4. El botón 'Register' no está desactivado
- Cuando el rol es asociado
    5. El botón 'Logout' no está desactivado
    6. El botón 'Employees list' no está desactivado
    7. El botón 'Register' está desactivado
- Cuando el rol es usuario
    8. El botón 'Logout' no está desactivado
    9. El botón 'Employees list' no está desactivado
    10. El botón 'Register' está desactivado

5.3 Employees

5.3.1 Employees.test.js (integración)

1. Redirecciona la interfaz al dashboard cuando se hace clic en el botón 'Return to Dashboard'

- Antes de cargarse
    2. No renderiza a ningún empleado
- Después de cargarse
    3. Renderiza a todos los empleados
    4. Se rellena la lista con los datos obtenidos de la llamada a la API
    - Cuando se hace click en el botón 'Edit'
        5. Renderiza un campo de texto en los elementos del empleado seleccionado que son modificables
        - Cuando se modifican los campos y se hace click en el botón de enviar
            6. Actualiza al empleado seleccionado
    - Cuando se hace click en el botón 'Remove'
        7. Elimina al empleado seleccionado.

5.3.2 Employees.spec.js (unitario)

El componente *Employees.js* hace de módulo que renderiza tres componentes: la cabecera (elemento de texto HTML), la lista de empleados (*EmployeeList)* y el botón para volver al dashboard (*ReturnButton*) por lo que lo único que se puede comprobar es que estos tres componentes se han renderizado en el DOM.

Para poder analizar alguna funcionalidad se tendría que acceder a niveles inferiores de la lista de empleados ya que el árbol de componentes no se está renderizando con la función *shallow*.

1. El componente se renderiza correctamente

5.3.3 EmployeeList.spec.js (unitario)

En este caso ocurre algo similar al anterior, la lista recibe los datos por propiedades y la renderiza pero las acciones se encuentran en el elemento (EmployeeListItem) por lo que desde aquí solo puede comprobarse que la lista se renderiza correctamente.

1. El componente se renderiza correctamente

5.3.4 EmployeeListItem.spec.js (unitario)

- Cuando el usuario tiene el rol de admin
  - Cuando el modo edición está desactivado
    - El componente se renderiza correctamente
  - Cuando el modo edición está activado
    - El componente se renderiza correctamente
- Cuando el usuario tiene el rol de asociado
  - Cuando el modo edición está desactivado
    3. El componente se renderiza correctamente
    4. Renderiza todas las columnas disponibles para el asociado
    5. Renderiza el botón de editar empleado
    6. No renderiza el botón de eliminar empleado
  - Cuando el modo edición está activado
    7. El componente se renderiza correctamente
    8. Renderiza los campos de texto para el email y el teléfono móvil
    9. No renderiza un campo de texto para el salario
    10. Renderiza el botón de enviar
- Cuando el usuario tiene el rol de usuario
  6. El componente se renderiza correctamente
  7. Renderiza todas las columnas disponibles para el usuario
  8. No renderiza el salario
  9. No renderiza el botón de editar empleado
  10. No renderiza el botón de eliminar empleado

5.4 Register

En este módulo existen dos componentes adicionales que se van a obviar dado que su función es formalizar los campos de texto para el teléfono móvil, que tiene un mensaje desplegable y otro para la fecha que son tres campos de texto enlazados. A estos dos componentes se le han realizado los test unitarios correspondientes para ser consistente con la aplicación, pero no se van a analizar.

5.4.1 RegisterForm.test.js (integración)

- Validación
  1. De inicio, muestra el botón 'Register' desactivado
  2. Muestra el botón 'Register' desactivado si algún campo de texto está vacío
  - Email
    3. De inicio, no muestra ningún error
    4. No muestra ningún error cuando el email es válido
    5. Muestra un error cuando el email introducido es no válido
  - Nombre
    6. De inicio, no muestra ningún error
    7. No muestra ningún error cuando el nombre es válido
    8. Muestra un error cuando el nombre introducido tiene menos de 2 caracteres
  - Apellido
    9. De inicio, no muestra ningún error
    10. No muestra ningún error cuando el apellido es válido
    11. Muestra un error cuando el apellido introducido tiene menos de 2 caracteres
  - Teléfono móvil
    12. De inicio, no muestra ningún error
    13. No muestra ningún error cuando el teléfono móvil tiene un formato válido
    14. Muestra un error cuando el teléfono móvil tiene un formato no válido
    15. De inicio, no muestra el mensaje desplegable de información
    - Al hacer click en el botón de información
      16. Muestra el mensaje de información
  - Fecha
    - Día
      17. De inicio, no muestra ningún error
      18. No muestra ningún error cuando se introduce un día válido
      19. El borde del campo de texto aparece en rojo cuando el día es no válido
      20. Cambia automáticamente al campo mes cuando se introduce el día
      21. Cambia automáticamente al campo año cuando el mes ya ha sido introducido previamente

- Mes
    22. De inicio, no muestra ningún error
    23. No muestra ningún error cuando se introduce un mes válido
    24. El borde del campo de texto aparece en rojo cuando el mes es no válido
    25. Cambia automáticamente al campo año cuando se introduce el mes
- Año
    26. De inicio, no muestra ningún error
    27. No muestra ningún error cuando se introduce un año válido
    28. El borde del campo de texto aparece en rojo cuando el año es no válido
- Salario
    29. De inicio, no muestra ningún error
    30. No muestra ningún error cuando el salario es válido
    31. Muestra un error cuando el salario introducido no es válido

- Al enviar el formulario
    - Cuando todos los campos están rellenados y se hace click en el botón 'Register'
        - Antes de cargarse
            32. No muestra la vista de registro completado
            33. Muestra el botón 'Register' desactivado
        - Después de cargarse
            34. Muestra la vista de registro completado

5.4.2 RegisterForm.spec.js (unitario)

- Cuando no hay errores
    1. El componente se renderiza correctamente
    2. Los campos de texto tienen los valores que llegan por las propiedades
    3. Le pasa al componente 'InputDate' las propiedades correctas
- Cuando hay errores
    4. El componente se renderiza correctamente

5.5 Resultados

El resultado de ejecutar todos los tests descritos en este capítulo es el siguiente:

```
 PASS  src/modules/RegisterForm/InputExpansion/__tests__/InputExpansion.spec.js
 PASS  src/modules/Employees/EmployeeList/EmployeeListItem/__tests__/EmployeeListItem.spec.js
 PASS  src/modules/RegisterForm/__tests__/RegisterForm.spec.js
 PASS  src/modules/Dashboard/__tests__/Dashboard.test.js
 PASS  src/modules/Login/__tests__/Login.spec.js
 PASS  src/modules/Dashboard/__tests__/Dashboard.spec.js
 PASS  src/modules/Login/__tests__/Login.test.js
 PASS  src/modules/Employees/__tests__/Employees.spec.js
 PASS  src/modules/Employees/EmployeeList/__tests__/EmployeeList.spec.js
 PASS  src/modules/RegisterForm/InputDate/__tests__/InputDate.spec.js
 PASS  src/modules/Employees/__tests__/Employees.test.js
 PASS  src/modules/RegisterForm/__tests__/RegisterForm.test.js

Test Suites: 12 passed, 12 total
Tests:       88 passed, 88 total
Snapshots:   15 passed, 15 total
Time:        8.592s
```

Figura 24 – Resultado de la ejecución de Jest en el terminal

La figura 24 muestra el resultado de los tests unitarios y de integración. Pueden observarse todos los ficheros de test implicados en la ejecución y en la parte inferior un resumen de esta.

'Test Suites' serían los ficheros ejecutados, los 'Tests' sería el número total de sentencias 'it' y los 'Snapshots' son todos los tests que involucran una comparación de snapshots.

Por último, se tendría el tiempo de ejecución que en este caso ha sido de **8.592 segundos** lo que representa un tiempo que, si lo comparamos con los 55.59 segundos de los tests end-to-end ,sería más de seis veces más rápido en ejecutarse.

6 INGENIERÍA DE PRUEBAS

En este capítulo se va a tratar de optimizar las pruebas ya presentadas de modo que se consiga un tiempo de ejecución inferior tratando de sacrificar el mínimo alcance posible. Para ello se tendrá que fijar una metodología para analizar uno a uno los tests propuestos.

La conclusión más obvia que se puede sacar de los resultados de los tests es que los test end-to-end requieren de un tiempo de ejecución mayor que el resto. Pues el alcance de los tests de integración y unitarios es parejo al de los end-to-end con la salvedad de que estos últimos son más fieles a la realidad al utilizar el servidor real.

Por otra parte, los resultados de los tests de integración y unitarios se han dado como un conjunto y no por separado. El motivo de hacerlo así es que Jest requiere de un tiempo para inicializar la plataforma interna de testeo, por lo que si ejecutáramos los tests por separado, tendríamos un tiempo de ejecución superior que al hacerlo en conjunto.

```
Watch Usage: Press w to show more.
 PASS  src/modules/Dashboard/__tests__/Dashboard.test.js
 PASS  src/modules/Login/__tests__/Login.test.js
 PASS  src/modules/Employees/__tests__/Employees.test.js
 PASS  src/modules/RegisterForm/__tests__/RegisterForm.test.js

Test Suites: 4 passed, 4 total
Tests:       50 passed, 50 total
Snapshots:   0 total
Time:        6.126s


 PASS  src/modules/RegisterForm/InputExpansion/__tests__/InputExpansion.spec.js
 PASS  src/modules/Login/__tests__/Login.spec.js
 PASS  src/modules/Dashboard/__tests__/Dashboard.spec.js
 PASS  src/modules/Employees/EmployeeList/__tests__/EmployeeList.spec.js
 PASS  src/modules/RegisterForm/__tests__/RegisterForm.spec.js
 PASS  src/modules/Employees/__tests__/Employees.spec.js
 PASS  src/modules/Employees/EmployeeList/EmployeeListItem/__tests__/EmployeeListItem.spec.js
 PASS  src/modules/RegisterForm/InputDate/__tests__/InputDate.spec.js

Test Suites: 8 passed, 8 total
Tests:       38 passed, 38 total
Snapshots:   15 passed, 15 total
Time:        3.704s
```

Figura 25 – Resultado de los tests de integración y unitarios por separado

En la figura 25 se han ejecutado ambos tests por separado ofreciendo una duración total superior a la que obtuvimos en el capítulo anterior (8.592 segundos). Este tiempo de ejecución que requiere Jest es insignificante cuando se testean aplicaciones grandes, pero en la aplicación propuesta, representa algo más de un segundo de diferencia.

A pesar de esto, siguiendo la teoría y la naturaleza de estos tests, se va a afirmar que los tests unitarios son menos pesados que los de integración a igualdad de alcance. Es decir, si hay algo testeado en integración que pueda ser testeado con tests unitarios, los unitarios requerirán de un tiempo de ejecución inferior.

Entonces, se tendría que, ordenados de mayor tiempo de ejecución a menor, los tests end-to-end serían los primeros, después los de integración y, por último, los unitarios.

Con esta información se puede plantear una metodología para mejorar nuestros tests.

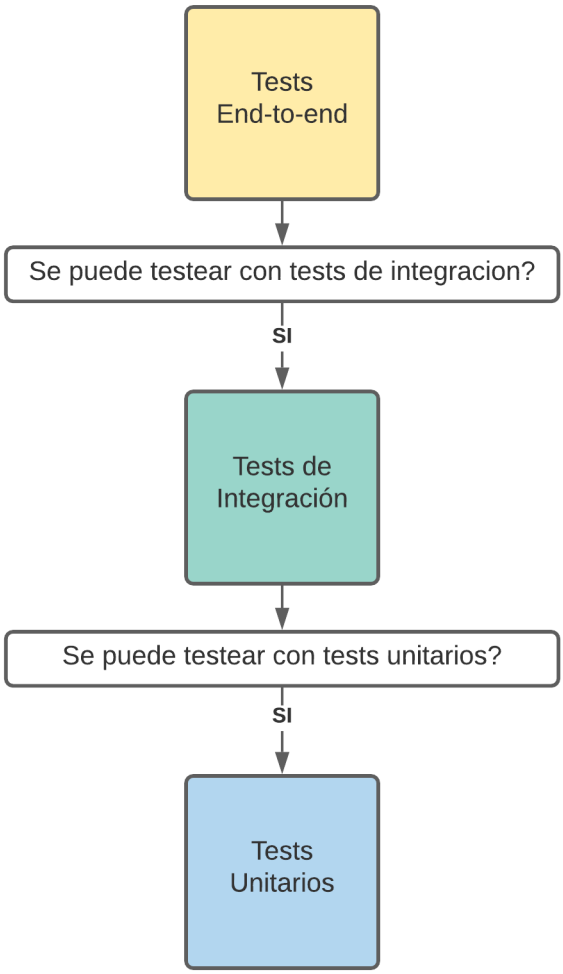

Figura 26 – Metodología de mejora en tiempo de ejecución

Siguiendo lo planteado en la figura 26, se puede afirmar que, el tiempo de ejecución va a conseguir una mejora si partiendo de un test, este puede escribirse utilizando un tipo de test inferior.

El problema que surge ahora es si este cambio provocaría una pérdida importante en el alcance de los tests, ya que la respuesta a la pregunta planteada, puede ser afirmativa en la mayoría de casos teniendo en cuenta que en integración se puede simular la respuesta de un servidor.

A continuación se va a partir de los tests end-to-end en busca de una mejora siguiendo una metodología como la mostrada en la figura 26 pero añadiendo la variable alcance en la ecuación, tratando de buscar el equilibrio más perfecto posible.

6.1 Optimización de pruebas end-to-end

Para entender mejor lo que va a realizarse en esta sección, conviene recordar o identificar a las tres entidades que participan en el desarrollo de la aplicación y los tests.

Por un lado se tiene la interfaz de usuario o front-end, donde son los tests de integración y los unitarios los que cubren la aplicación. Después estaría el servidor o back-end que tendría sus pruebas internas.

Y, por último, una entidad que se podría definir como la interacción entre la interfaz y el servidor que es donde actúan los tests end-to-end.

Llegados a este punto, se podría formular el planteamiento de mejora desde otro punto de vista:

- *'¿Depende el test propuesto de la interacción entre la interfaz y el servidor?'*

En caso afirmativo, el test está bien planteado dentro de los tests end-to-end y en caso negativo podría eliminarse y cubrirlo en integración.

Con la ayuda de los bloques de tests identificados en el capítulo 4, se van a analizar aquellos bloques que podrán ser eliminados, para su posterior testeo en integración. Se indicará la ruta de estos bloques en la documentación con la numeración que tiene entre paréntesis. También se indicará su análogo en tests de integración o unitarios, analizados en el capítulo 5.

El papel de los roles no afecta a esta optimización, pues si un bloque es redundante en el rol de administrador, su análogo en el de asociado lo será también. Es por ello que la estructura de esta sección será por módulos.

6.1.1 Login

Después de plantear la pregunta anterior, el único bloque de tests en el que hay interacción con el servidor es:

- **'Puede enviar un formulario válido' (4.1.1.5)**

Para poder iniciar sesión, debe haber una coincidencia entre las credenciales introducidas y los datos del servidor, por lo que existe una comunicación.

Por otro lado se tienen los siguientes bloques:

- **'Renderiza los elementos del módulo Login' (4.1.1.1)**
- **'No puede enviar un formulario no válido (email y contraseña incorrectos)' (4.1.1.2)**
- **'No puede enviar un formulario no válido (email incorrecto)' (4.1.1.3)**
- **'No puede enviar un formulario no válido (contraseña incorrecta)' (4.1.1.4)**

Todo sucede en el lado del front-end pues solo se comprueba que los elementos que se esperan de la interfaz están renderizados y el resto está destinado a validar los campos de texto.

Concluimos este módulo con la eliminación de esos cuatro bloques.

En este caso, el fichero de tests de integración **'Login.test.js' (5.1.1)** cubre la validación y el fichero de test unitarios **'Login.spec.js' (5.1.2)** cubriría la comprobación de un renderizado correcto.

6.1.2 Dashboard

Este módulo es bastante peculiar dado que es el único en el que no hay interacción con el servidor, por lo tanto, el bloque que cubre este módulo, puede eliminarse al completo.

Los bloques afectados serían los siguientes:

- **'Renderiza los elementos del módulo Dashboard' (4.1.2.1)**
- **'Puede visitar la página de la lista de empleados' (4.1.2.2)**
- **'Puede visitar la página del formulario de registro' (4.1.2.3)**
- **'Puede cerrar sesión' (4.1.2.4)**

La redirección a cada página es testeada en los tests de integración **'Dashboard.test.js' (5.2.1)** y la comprobación del renderizado correcto, dependiendo del rol del usuario, se realiza en los tests unitarios **'Dashboard.spec.js' (5.2.2)**.

6.1.3 Employees

El módulo Employees está involucrado en acciones provocadas por tres llamadas al servidor. Además no tiene ninguna regla de validación a la hora de introducir texto, por lo que es un objetivo claro de tests end-to-end.

Los bloques de tests en los que existe interacción con el servidor son:

- **'Renderiza los elementos del módulo Employees' (4.1.3.1)**
- **'Puede editar a un empleado' (4.1.3.2)**
- **'Puede eliminar a un empleado' (4.1.3.3)**

El primer bloque comprueba que los elementos de la lista se han renderizado correctamente. Estos elementos son las cabeceras de cada columna y los empleados de cada fila, cuyos datos provienen del servidor. En este caso podría eliminarse la comprobación de las cabeceras, pues eso está cubierto en los tests unitarios **'EmployeeList.spec.js' (5.3.3)**. Pero el resto permanecerá inalterable.

Tanto editar como eliminar a un empleado requieren de una llamada independiente al servidor, por lo que de estos no puede cambiarse nada.

El bloque **'Puede volver al Dashboard' (4.1.3.4)** si puede eliminarse y está cubierto por los tests de integración **'Employees.test.js' (5.3.1)**.

6.1.4 Register

En este módulo se realiza una llamada al servidor para registrar a un nuevo empleado y, además, se visita la lista de trabajadores para comprobar que se ha registrado correctamente.

El único bloque en el que se interactúa con el servidor es:

- **'Puede registrar un nuevo empleado' (4.1.4.3)**

Los bloques propuestos para su eliminación son:

- **'Renderiza los elementos del módulo Register' (4.1.4.1)**
- **'Valida el formulario' (4.1.4.2)**

El primero está cubierto por los tests unitarios **'RegisterForm.spec.js' (5.4.2)** y la validación que se hace en el segundo está en los de integración **'RegisterForm.test.js' (5.4.1)**.

6.2 Optimización de pruebas de integración

La optimización de estas pruebas no es tan inmediata como en el caso de los tests end-to-end. El motivo es muy simple, el esfuerzo dedicado a estas pruebas es muy costoso, por lo que es de vital importancia detectar cuando se debe implementar este tipo de test.

A causa de esto, se han introducido intencionadamente tests de integración que bajo ningún concepto deberían haberse planteado.

Esto se encuentra en el fichero de tests de integración **'RegisterForm.test.js' (5.4.1).**

En este fichero, se puede contemplar un gran bloque con el descriptor 'Validación' que representa la mayor parte de ese banco de tests.

Este bloque se encarga de validar todos los campos de texto que aparecen en el formulario de registro de un nuevo empleado. Para ello, renderiza el formulario e introduce diferentes valores en los campos para comprobar que saltan errores en el caso de introducir un valor no válido o que por el contrario, el formulario puede ser enviado al introducir valores válidos.

Una validación de algo, es una función que recibe un valor y devuelve un parámetro que representa éxito o error en la validación. Es decir, no es ni siquiera React, se trata de JavaScript puro.

Lo que está ocurriendo en esos tests de integración es que se está analizando una función por medio de simulaciones en el DOM, lo cual incrementa el tiempo de ejecución y además, se pierde una gran cantidad de tiempo en desarrollar un test así. Por supuesto el mantenimiento de estos tests puede llegar a ser incontrolable ya que en el momento en el que cambie alguna regla de validación, puede romperse más de un test por completo.

Por el contrario, como se verá a continuación, realizar tests a una función es tan sencillo como el ejemplo que aparece en la página 30 de este documento, en la explicación de Jest.

```
const validation = {
  email: (email) => !/^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(email) && 'Please enter a valid email',
  firstName: (firstName) => firstName.length < 2 && 'First name must be at least 2 characters long',
  lastName: (lastName) => lastName.length < 2 && 'Last name must be at least 2 characters long',
  mobilePhone: (mobilePhone) => (
    (!/^[0-9]+$/.test(mobilePhone)
    || !(mobilePhone.startsWith('6') && mobilePhone.length === 9))
    && 'Please enter a valid phone number.'),
  day: (day) => day.length !== 2 || day > 31 || day === '00',
  month: (month) => month.length !== 2 || month > 12 || month === '00',
  year: (year) => year.length !== 4 || year < 1900 || year === '0000',
  salary: (salary) => ((!/^[0-9]+$/.test(salary)) && 'Salary must be a number')
}

export default validation
```

Figura 27 – Código del objeto 'validation' utilizado en el módulo Register

La figura 27 muestra un objeto donde cada propiedad es una función que valida un campo de texto del formulario de registro. Estas funciones devuelven un mensaje de error cuando el valor no es válido o 'false' en el caso de ser válido. Los campos día, mes y año de la fecha de nacimiento no tienen mensaje de error, simplemente devuelven 'true' cuando el valor no es válido y 'false' en el caso contrario.

Con la explicación anterior y sin necesidad se comprender como funciona cada regla de validación ya se tiene suficiente información como para plantear los tests:

```
Validation rules Unit Tests
  Email
    ✓ returns an error message if the value is invalid (2ms)
    ✓ returns false if the value is valid
  First name
    ✓ returns an error message if the value is invalid
    ✓ returns false if the value is valid
  Last name
    ✓ returns an error message if the value is invalid (1ms)
    ✓ returns false if the value is valid
  Mobile phone
    ✓ returns an error message if the value is invalid (1ms)
    ✓ returns false if the value is valid
  Day
    ✓ returns true if the value is invalid
    ✓ returns false if the value is valid
  Month
    ✓ returns true if the value is invalid (1ms)
    ✓ returns false if the value is valid
  Year
    ✓ returns true if the value is invalid
    ✓ returns false if the value is valid
  Salary
    ✓ returns an error message if the value is invalid (1ms)
    ✓ returns false if the value is valid
```

- Email, Nombre, Apellido, Teléfono móvil, Salario
    1. Devuelve un mensaje de error si el valor no es válido
    2. Devuelve ‘false’ si el valor es válido
- Día, Mes, Año
    1. Devuelve ‘true’ si el valor no es válido
    2. Devuelve ‘false’ si el valor es válido

Estos tests están comprendidos en el fichero **‘validation.spec.js’** y pasan a formar parte del banco de pruebas.

Por el contrario, se eliminaría todo el bloque ‘Validation’ de los tests de integración del formulario de registro.

El otro fichero de tests de integración con un problema similar es **‘Login.test.js’ (5.1.1)**. En este fichero también se validan dos campos de texto. A diferencia del caso anterior, aquí no tenemos reglas de validación ya que lo único que se comprueba es que si los campos están vacíos, el botón para iniciar sesión aparece desactivado.

Para arreglar estos tests simplemente habría que eliminar el bloque de validación en integración y se implementaría un nuevo caso en los unitarios **'Login.spec.js' (5.1.2)** quedando de la siguiente manera:

- Estado inicial
    1. El componente se renderiza correctamente
- Cuando no hay errores
    2. El componente se renderiza correctamente
    3. Los campos de texto tienen los valores que llegan por las propiedades
- Cuando hay errores
    4. El componente se renderiza correctamente

Donde se ha añadido el caso de estar en el estado inicial, dado que esto implica partir de un email y contraseñas en blanco.

Respecto al resto de tests de integración que conforman la aplicación, no se identifica ninguna anomalía que sugiera una implementación adicional dentro de los tests unitarios.

6.3 Resultados

Los resultados en las pruebas que quedan tras las optimizaciones, serán necesariamente mejores que antes de la optimización en los tests end-to-end. Esto se debe a que lo único que se ha realizado es una eliminación de bloques de tests.

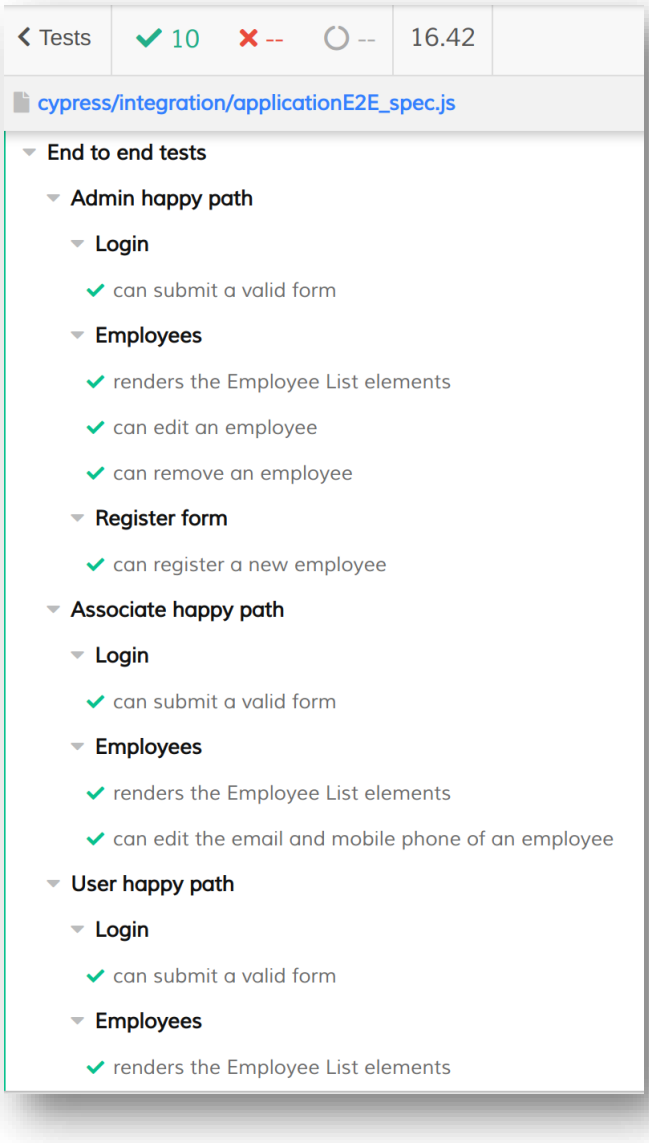

Figura 28 – Resultado de los tests end-to-end tras la optimización

Si comparamos la figura 28, con los resultados que salieron antes de la optimización, a parte de la diferencia de bloques de tests, también se ha modificado el descriptor de cada rol, siendo ahora 'Admin / Associate / User happy path' es decir, un escenario en el que no debe haber ningún tipo de error a la hora de ejecutar las acciones. Este cambio se debe a que ya no hay validaciones, por lo que las acciones deben ser lineales.

Respecto a los resultados, si los comparamos con los anteriores (figura 23), se tienen 10 bloques de tests contra 40 y respecto al tiempo de ejecución se ha reducido hasta los **16.42 segundos** frente a los 55.58 segundos que se obtuvieron en la primera ejecución. Esto implica una reducción del **70,46%** del tiempo de ejecución respecto al pre-optimizado.

En cuanto a los cambios en los tests de integración, en la teoría se espera una mejora, pero es necesario ejecutar las pruebas para confirmarlo, pues en este caso se ha eliminado un test de integración para después crear otro de tipo unitario.

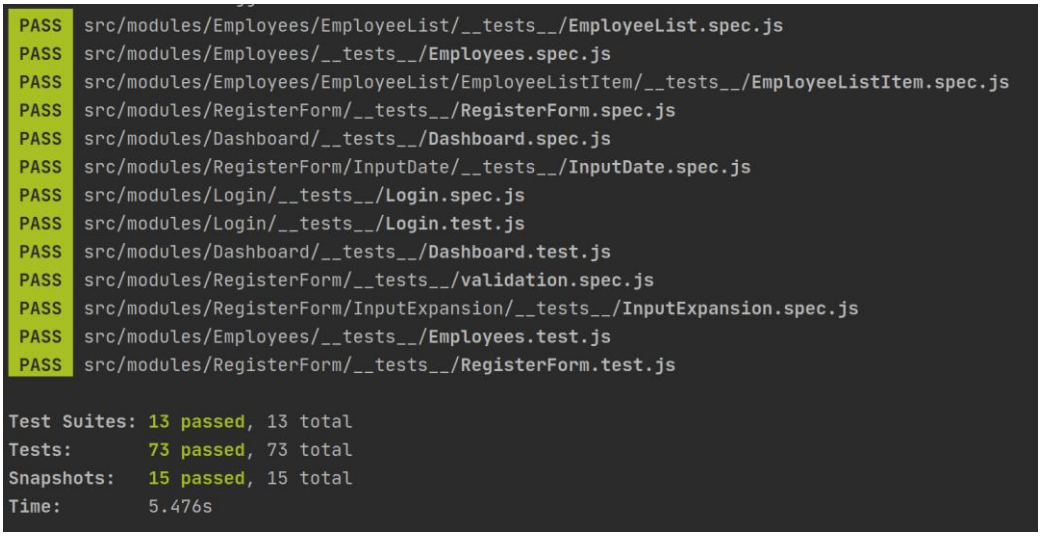

Figura 29 – Resultado de los tests de integración y unitarios tras la optimización

La figura 29 nos muestra los datos de la nueva ejecución que al ser comparados con los antiguos (figura 24), se tendría que ahora existe un fichero de test adicional, debido a que se creó **'validation.spec.js'** para cubrir lo que eliminamos del fichero con los tests de integración del módulo Register. De hecho, estos cambios junto con los que sufrió el módulo Login han hecho que el número de sentencias 'it' se vea reducido de 88 a 73.

Respecto al tiempo de ejecución de los tests se puede ver una mejora al tener ahora **5.476 segundos** frente a los 8.592 segundos de la versión previa a la optimización. En este caso el tiempo de ejecución se ha reducido en un **36.3%**

Figura 30 – Resultado de los tests de integración y unitarios por separado tras la optimización

Por otro lado se tiene una mejora evidente en el tiempo de ejecución de los tests de integración, tal y como se observa en la figura 30. El tiempo de ejecución anterior fueron 6.126 segundos frente a los **3.618 segundos** de la ejecución actual, lo cual muestra casi tres segundos de mejora. En cuanto a los unitarios el hecho de haber añadido más tests no se ha visto muy afectado en cuanto al tiempo de ejecución pues se ha pasado de 3.704 segundos a **3.814 segundos**.`

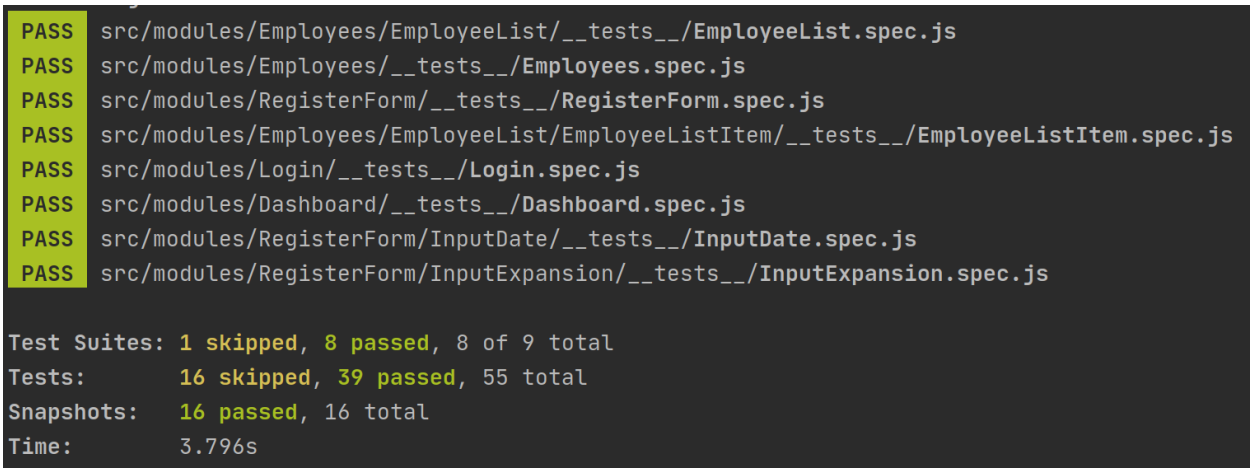

Figura 31 – Resultado de los tests unitarios sin ejecutar los de validación del módulo Register

Otro dato interesante de esta optimización, es que el resultado de ejecutar todos los tests unitarios exceptuando el que testea la validación del formulario de registro, representado en la figura 31, muestra una diferencia entre tiempos de ejecución muy pequeña respecto el resultado de ejecutar todos los tests unitarios. Se tienen 3.796 segundos frente a los 3.814 segundos que devolvía en la ejecución previa.

Esto confirma los pocos recursos que requieren los tests unitarios realizados a las funciones de validación, pues no se está renderizando nada, sólo se ejecutan funciones que reciben un argumento y realizan una comprobación simple.

En resumen, los cambios aplicados en esta optimización han sido cambiar las pruebas de validación de los dos formularios que aparecen en la aplicación (inicio de sesión y registro), de tests de integración a tests unitarios sin perder alcance.

Es por ello que, recordando la hipótesis planteada previamente: *'...se va a afirmar que los tests unitarios son menos pesados que los de integración a igualdad de alcance. Es decir, si hay algo testeado en integración que pueda ser testeado con tests unitarios, los unitarios requerirán de un tiempo de ejecución inferior.'*

Ahora si se puede confirmar que efectivamente, al invertir en tests unitarios, se consigue una mejora en el tiempo total de ejecución de los tests.

7 PRUEBAS MANUALES

Las pruebas manuales es una metodología que a veces se obvia o no se considera parte del testeo de una aplicación, pero es vital para encontrar errores que no se contemplan en los tests automáticos debido a que se encuentran fuera de su alcance o por condiciones de carrera que son más fáciles de identificar cuando se interactúa directamente con la interfaz de usuario.

En este proyecto siempre se ha considerado que se tiene un servidor ideal que siempre responde las llamadas con éxito y debido a esto, no se ha incluido validación ante fallos en el servidor.

Para manejar errores provenientes del servidor, es necesario la declaración 'try…catch' lo cual no está implementado en la aplicación. Olvidar esta declaración, provocando excepciones no controladas, es un error muy común que raramente provoca fallos de gravedad, pues si el servidor genera una excepción, lo más probable es que no se reciban los datos esperados y la aplicación falle en algún momento al no poder tener acceso a esa información.

En este capítulo se va a analizar de forma manual lo que pasaría en cada módulo cuando se hace uso de un servidor real, el cual en algún momento dado, genera un error como respuesta a la llamada realizada desde la aplicación.

Se tratará de comprobar si los tests end-to-end de los que se dispone podrían cubrir este fallo en el sistema.

7.1 Login

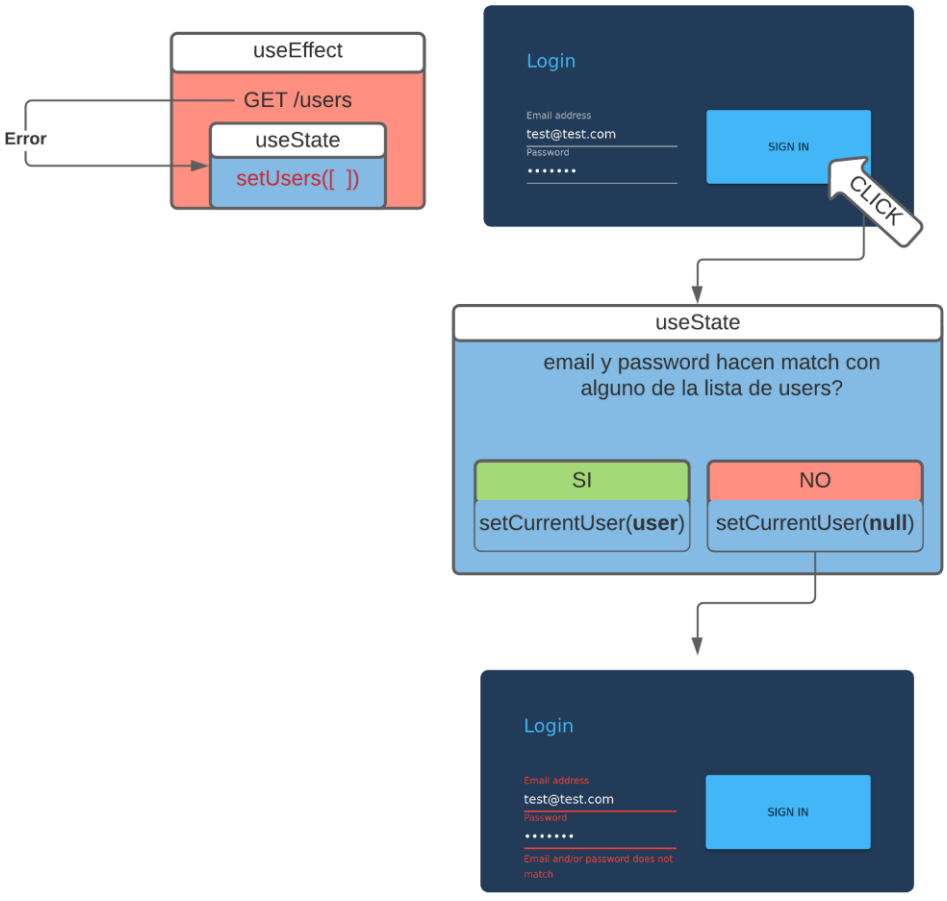

Figura 32 – Diagrama del módulo Login cuando el servidor falla

En la figura 30 se puede observar el ciclo de vida del módulo Login cuando hay un error en el servidor. Casualmente, a pesar de que no se ha previsto de validación ante errores del servidor, el usuario ve un error cuando intenta conectarse.

La razón de esto es que cuando se realiza el GET, el siguiente paso sería modificar el estado 'users' con la lista de usuarios proveniente del servidor. En este caso esa lista está vacía, por lo que al introducir cualquier email y contraseña, estos nunca harán match con algún usuario de la lista, pues no hay ninguno.

Esto está cubierto por el bloque de tests **'Puede enviar un formulario válido' (4.1.1.5)**. Pues las acciones que realiza el test son las mismas que las mostradas en la figura 30.

7.2 Employees

Figura 33 – Diagrama del módulo Employees cuando el servidor falla

En el diagrama de la figura 31 faltaría por ser representada una llamada adicional en la que se llamaría al servidor para obtener la lista de empleados.

Esta llamada se ha omitido dado que si el servidor genera un error durante la obtención de la lista, este módulo estaría vacío y el primer tests end-to-end de la lista de empleados fallaría al ejecutarse, detectando el error. El bloque de tests involucrado en detectar esto es **'Renderiza los elementos del módulo Employees' (4.1.3.1)**.

Por el contrario, si la llamada para editar o borrar a un empleado falla, no hay ningún test que notifique el error. El motivo de que los tests no tengan alcance en esta parte se puede comprobar siguiendo el diagrama. Al introducir los cambios en el empleado y hacer clic, el servidor falla haciendo que estos cambios nunca se apliquen en la base de datos. Como no se está controlando la excepción, el código ejecuta la siguiente línea que es cambiar el modo edición a desactivado. En este momento veríamos los cambios en el empleado pues esto está guardado en el estado de React, pero si el navegador refrescara la página, la información del empleado volvería a su estado original. Al eliminar a un empleado el resultado es el mismo. Localmente se borra el empleado del estado y por consiguiente, no se renderiza, pero esa información nunca llega a la base de datos. Al refrescar la página, el servidor nos devolverá la lista tal y como estaba antes del borrado.

Detectar este fallo se ha realizado de forma manual obligando al servidor a fallar y viendo cómo reacciona la interfaz. La solución a esto sería implementar una validación ante errores del servidor. Entonces, si este falla, el estado de React no se modificaría.

Otra solución provisional sería añadir una línea al bloque de tests que refrescara la página para comprobar que los cambios se han aplicado con éxito. Pero esto sería una mala práctica que aumentaría el tiempo de ejecución del test, para cubrir un error que no se ha considerado en la interfaz.

7.2.1 Llamadas simultáneas a la misma API

Existe otro caso que puede generar errores o resultados inesperados en la interfaz, y que puede ocurrir especialmente en la lista de empleados. Esto ocurre cuando varios usuarios están operando en esta página y ambos parten de la misma lista de empleados obtenida desde el servidor.

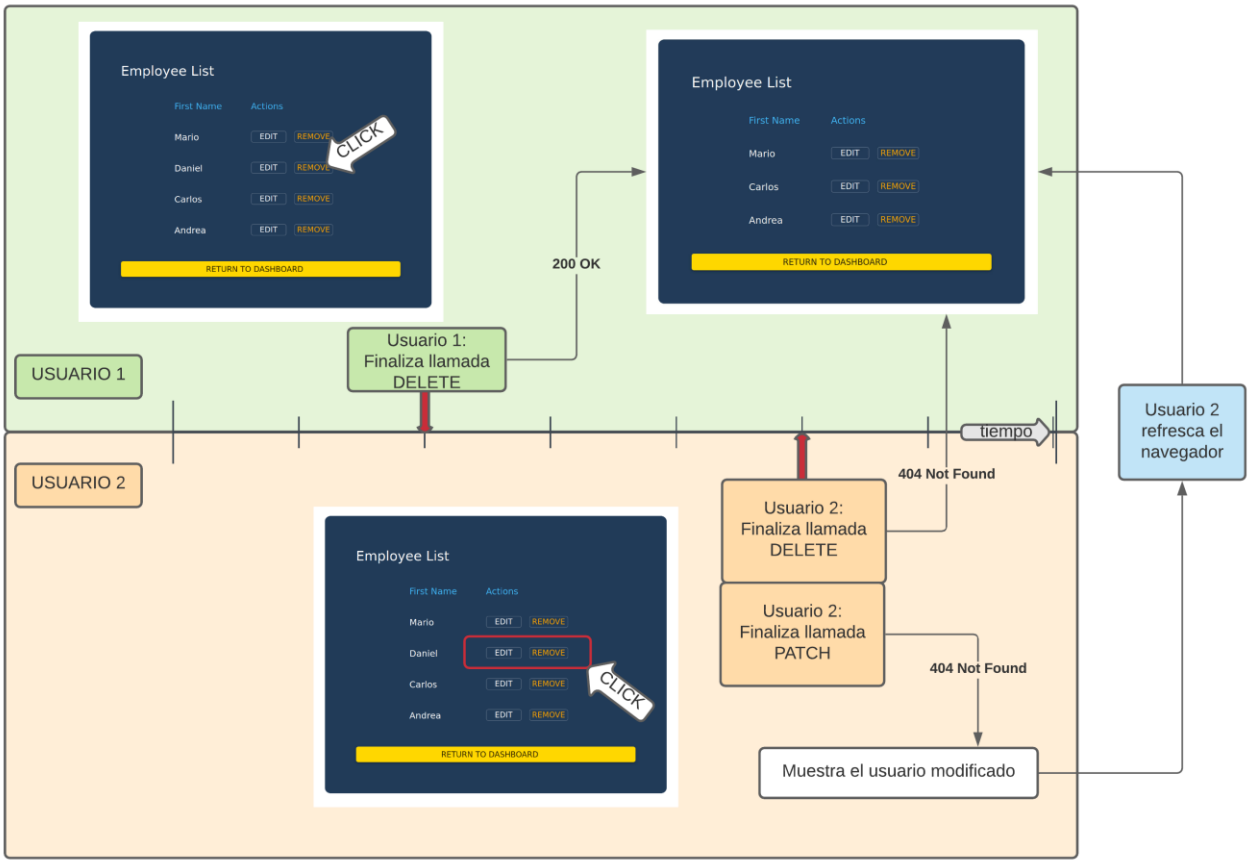

Figura 34 – Diagrama del ciclo de acciones de dos usuarios en Employees (acceso a empleado eliminado)

En la figura 32 puede observarse como dos usuarios realizan llamadas al servidor a lo largo del tiempo. Por un lado se tendría al 'USUARIO 1' que realiza una llamada al servidor para eliminar al empleado 'Daniel'.

Por otro estaría el 'USUARIO 2' que podría modificar a ese mismo empleado o borrarlo. En ambos casos, al realizarse la llamada al servidor, este respondería con un error '404 Not Found' debido a que no encuentra esa entidad en la base de datos. Como no se dispone de una validación de errores del servidor en la interfaz, este error no se detectaría y en el caso de borrar al empleado, se vería la lista final sin ese elemento de la lista produciendo el resultado esperado. Por el contrario, si se modifica al empleado, se podría ver una lista con ese elemento actualizado, lo cual no sería fiel a lo que hay representado en la base de datos.

El resultado es que si el 'USUARIO 2' refresca el navegador, vería que ese empleado no existe pues fue borrado previamente por el 'USUARIO 1'.

Este comportamiento podría solventarse de diferentes maneras. Lo más inmediato sería mostrar un error en el que se le indique al 'USUARIO 2' que ha habido un error en el servidor al tratar de ejecutar una acción en un elemento que no existe y llamar otra vez a la API que devuelve la lista de usuarios para tenerla actualizada.

Junto con esta validación, lo ideal sería implementar un sistema de notificaciones donde se tenga implementado un receptor de notificaciones que esté siempre activo esperando alguna notificación del servidor. En este caso, cuando el 'USUARIO 1' aplique la acción de eliminar a un empleado, la interfaz del 'USUARIO 2' recibiría una notificación para actualizar la lista con los datos nuevos, donde se podría implementar un sistema en el que se alerte al usuario que ha habido cambios y que compruebe de nuevo la lista.

Este último cambio no puede asegurar un correcto funcionamiento ya que puede que la acción del segundo usuario se realice antes de que la interfaz detecte la notificación, pero eso quedaría cubierto con el mensaje de error que se implementaría previamente.

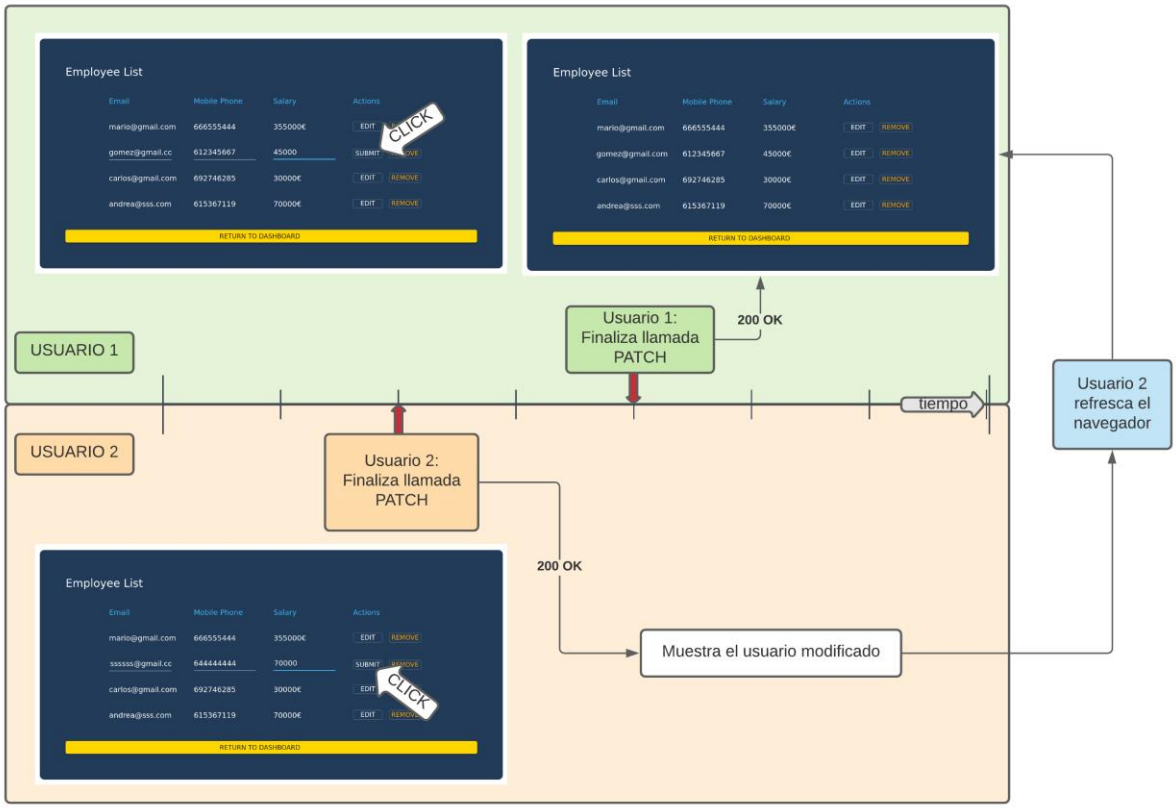

Figura 35 – Diagrama del ciclo de acciones de dos usuarios en Employees (modificación simultanea al mismo empleado)

En la figura 33 se muestra un caso similar al anterior pero con la diferencia de que aquí se modifica a un usuario que no es eliminado, por lo que el servidor nunca responderá con un error.

Se tiene que el 'USUARIO 2' es el primero en realizar una llamada para modificar a un usuario, aplicando estos cambios en la base de datos del servidor y mostrando los cambios en la interfaz.

Posteriormente el 'USUARIO 1', que desconoce los cambios producidos por el usuario anterior, pues ambos parten de la misma lista de empleados, realiza otra modificación a ese mismo empleado, obteniendo una respuesta exitosa por parte del servidor y mostrando los cambios en la interfaz.

En este punto, si el 'USUARIO 2' refresca o vuelve a visitar la página, podría ver que sus cambios no aparecen en la interfaz debido a que la base de datos se actualiza con la última llamada realizada al servidor.

Realmente esto no se puede considerar un fallo, pues tanto la interfaz como el servidor están actuando del modo en el que se espera. Es por ello que no podría implementarse un mensaje de error.

La única manera de solucionarlo sería con el mismo sistema de notificaciones mencionado anteriormente.

7.3 Register

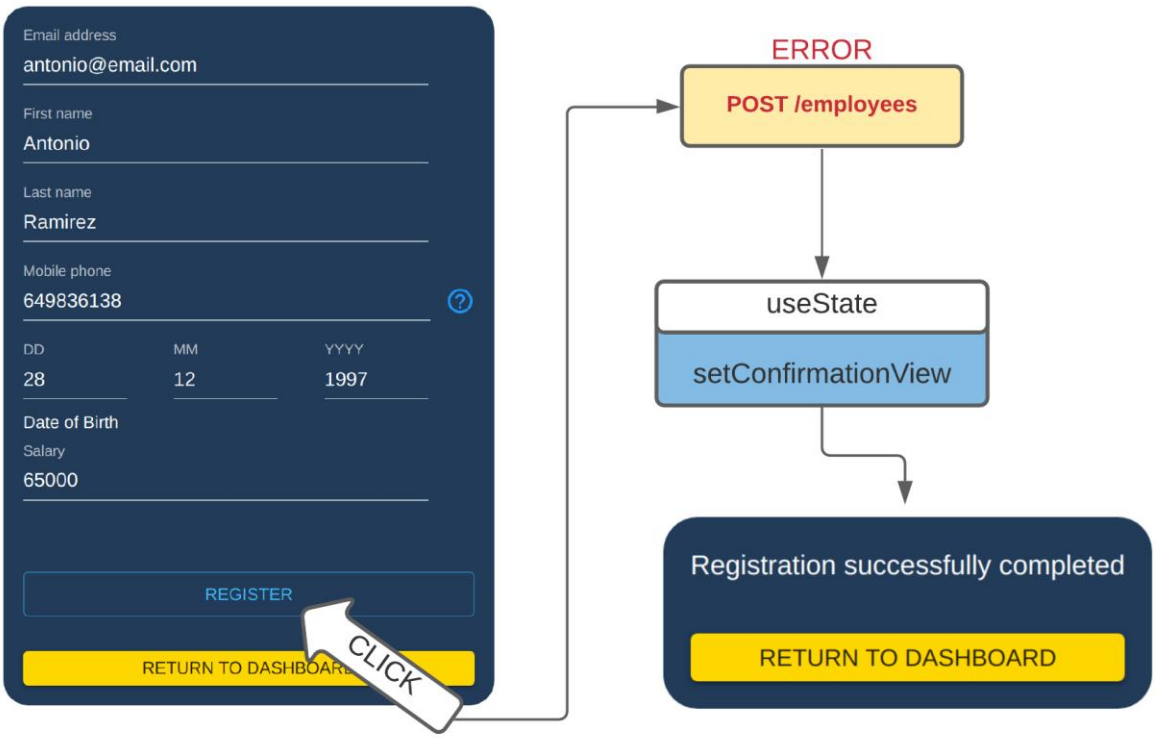

Figura 36 – Diagrama del módulo Register cuando el servidor falla

En el módulo Register, representado en la figura 32, ocurre algo similar que en el Dashboard. El usuario hace click en registrar, esto acciona el evento que realiza la llamada al servidor para escribir al nuevo empleado en la base de datos, pero la llamada es devuelta con un error. Entonces, el siguiente paso sería cambiar el estado que renderiza el mensaje de registro completado con éxito. En este punto se estaría visualizando ese mensaje pese a que el usuario no ha sido registrado en la base de datos.

A diferencia del caso anterior, donde al actualizar un empleado, no existía un test que notificara el fallo, aquí si se tendría cubierto el error. El bloque de tests **'Puede registrar un nuevo empleado' (4.1.4.3)** accede a la lista de empleados tras registrar uno nuevo para comprobar que este se renderiza correctamente. Sería en ese test cuando Cypress marcaría el test como fallido al no poder identificar los datos del empleado en la lista.

Aunque se pueda confirmar que existe un test que cubre este error. Al igual que ocurría en el Login, este error se detectaría por casualidad, pues el test que hay escrito tiene la función de comprobar de que lo introducido en el formulario de registro coincide con el usuario creado. Es por ello que la solución sería igual que en el caso anterior, implementar validación ante errores del servidor.

8 PLANIFICACIÓN

El desarrollo de este proyecto ha sido delicado y más costoso que lo esperado pues es muy fácil llegar a un alcance muy superior que al planteado.

En una primera aproximación se utilizó una aplicación más compleja lo cual provocó una cantidad de tests desmesurada para abordar el objetivo de este proyecto, de hecho, se abandonó esa idea por exceso de información.

Además, se producía una repetición de tests sin cubrir otros que sí se han planteado en esta versión final del proyecto. Aunque podría haberse optado a realizar el experimento con una parte de esa aplicación, habría quedado algo ambiguo al no conocerse en detalle el funcionamiento de toda la aplicación.

Debido a lo anterior, se empezó de cero a plantear una aplicación simple pero con la suficiente complejidad como para cubrir los tests más típicos que suelen plantearse.

Como la comunicación entre el cliente y el servidor es la parte más crítica en una aplicación, se empezó por identificar las llamadas más comunes que se realizan a un servidor que son: GET, POST, DELETE y PATCH.

Entonces sólo había que plantear una aplicación que realice todas esas llamadas de una manera en la que todo esté conectado. En este caso con GET obtenemos la lista de usuarios y de empleados, y el resto de llamadas son relacionadas con la lista de empleados: POST para registrar uno nuevo, DELETE para eliminar y PATCH para modificar a un empleado.

El desarrollo de la aplicación sufrió de ajustes continuos, como la inclusión de la librería Material-UI[12] que ofrece componentes estilizados de React para ofrecer un diseño más vistoso y moderno. Pero en general, el desarrollo no ha sido la fase a la que más tiempo se ha dedicado, de hecho, el planteamiento de esta fue más largo y más si se tiene en cuenta ese primer intento de proyecto.

Respecto a la fase de tests, sin ningún lugar a dudas el escribir tests de integración es la parte más costosa con diferencia, pero eso no debe ser motivo para desecharlos pues la recompensa merece la pena, sobre todo para los desarrolladores que trabajan en ella. Si la aplicación crece y algo falla es muy sencillo detectar el foco y por otro lado, junto con los tests unitarios, describen la aplicación. Esto es especialmente útil de cara a que un segundo desarrollador, comience a trabajar en el proyecto sin conocimientos previos del funcionamiento interno de la aplicación.

Por último, hubo un reajuste relativamente grande al finalizar los tests y fue quitar la validación del servidor. También se eliminaron los tests asociados a esta validación, pues había que comprobar que los errores se mostraban en la interfaz. El motivo de quitar esto fue que esos tests no tenían gran relevancia, pues eran similares a los que ya había y, sobre todo, para poder darle más protagonismo al capítulo de tests manuales.

9 CONCLUSIONES Y LÍNEAS FUTURAS

En este proyecto se ha propuesto un banco de test inicial, en el que se ha seguido una metodología tradicional. Esto significa potenciar la fase de test end-to-end en la que se puede cubrir la mayor parte de la aplicación a coste mínimo. Después se ha realizado lo mismo pero basado en tests de integración y unitarios.

Tras el planteamiento de los tests, se ha procedido a realizar una revisión en busca de mejoras en el tiempo de ejecución. Para ello se han propuesto estrategias para reducir este tiempo sin perder efectividad, pues siempre hay que tratar de cubrir todo lo posible de la aplicación.

Los resultados obtenidos en las pruebas, tras realizar las optimizaciones necesarias, desvelan una mejora importante en el tiempo de ejecución de los tests. Por lo tanto, quedaría demostrado que un buen planteamiento y elección de estrategia a la hora de abordar los tests de cualquier aplicación, implica lograr un banco de pruebas más óptimo en cuanto a tiempo de ejecución de las pruebas y con un alcance que cubra todas las zonas críticas de la aplicación.

Sin embargo, tras analizar todo el proceso de este proyecto, desde la creación del primer test, hasta la mejora del último, se pueden plantear conclusiones adicionales.

Cuando se realizaron los cambios en los tests end-to-end, eliminando tests duplicados en integración, se pudo llegar a la conclusión de que, teniendo en cuenta que eso que hemos eliminado ya estaba cubierto, no se ha perdido alcance de testeo.

Esto es verídico hasta cierto punto. Al quitar estas pruebas, la responsabilidad de tener esa parte de la aplicación cubierta ante posibles fallos, pasa de estar repartida entre dos observadores a caer directamente en un único observador.

Si el proyecto tratase de un caso real, que el mismo desarrollador de la interfaz tenga que mantener los tests end-to-end es algo que raramente se ve. Suelen ser roles separados y el hecho de eliminar tests que ya están presuntamente cubiertos, hace que se tenga que confiar en un único rol, lo cual no es algo negativo, dado que siempre debe haber un reparto de responsabilidades, pero también es cierto que dos observadores van a cubrir la aplicación mejor que uno.

Entonces, aunque en la aplicación propuesta, se haya decidido eliminar esos tests debido a que todo el entorno es mantenido por el mismo desarrollador, puede haber casos en los que indirectamente se pierda alcance de testeo en la aplicación.

Una vez se han eliminado los tests end-to-end que ya estaban cubiertos, si analizamos los tests de integración, puede detectarse que sigue existiendo una duplicidad entre algunos tests end-to-end y los de integración. En este punto, seguir eliminando tests end-to-end carecería de sentido, pues lo que queda son pruebas que cubren las interacciones con el servidor.

Entonces se estaría en una situación que puede llevar al desarrollador a eliminar los tests redundantes en integración. Esto se debe a que son tests que interactúan con un servidor simulado, lo cual nunca te va a aportar la misma seguridad que los tests que utilicen el servidor real. En cambio, eliminar estos tests sería un error por el simple hecho de que los tests a nivel de código permiten detectar errores al instante.

Es posible que esto no pueda apreciarse en la aplicación propuesta debido a que todo está en la red local, donde el desarrollador tiene acceso al código de la interfaz, servidor y tests end-to-end. Pero en casos reales, este código estará repartido en diferentes repositorios independientes unos de otros. Además, existirá mínimo un entorno dedicado al desarrollo y otro para producción, donde habitarán las versiones estables de la aplicación. Desplegar una actualización a cualquiera de los entornos conlleva tiempo, ya que puede que la aplicación esté siendo usada por terceras personas y que se vean afectadas por cualquier cambio.

Para poder ejecutar los tests end-to-end la aplicación debe estar desplegada en un servidor, por lo que si no se tiene la posibilidad de tenerlo todo montado en una red local, se tendrán que cumplir una serie de requisitos previos antes de poder ejecutar los tests end-to-end.

Es por ello que en casos reales, lo normal es encontrar tests de integración en los que se hacen pruebas con un servidor simulado como apoyo, que se repiten en los tests end-to-end.

Los tests de integración originales del módulo Register, supusieron un esfuerzo en tiempo superior que al desarrollo del resto de tests de integración de la aplicación en conjunto.

Esos tests innecesarios, que fueron reemplazados por tests unitarios tras la optimización, se pudieron haber evitado con una fase en la que se haga un planteamiento sobre que se va a testear y como. Esto se trata de una parte vital de la ingeniería de pruebas y es la parte que concierne al planteamiento de los tests a realizar. Es decir, la fase previa a escribir el código que dará forma a los tests.

El motivo de omitir esa fase en este proyecto es que, de haber empezado por el planteamiento, la parte de optimizar las pruebas sería más escueta, pues esas mejoras se habrían detectado previamente a la realización de los tests.

Entonces, se ha partido de una estructura de tests ya creada basada en prácticas habituales; testear toda la aplicación sólo y exclusivamente con tests end-to-end. Y basada también en errores habituales; testear en integración, lo que se debería de hacer en tests unitarios.

Es decir, se ha simulado el aplicar una estrategia de pruebas a un proyecto ya hecho al cual se han buscado mejoras con tests ya escritos.

En un proyecto que empiece de cero, y en el que se tengan estos conocimientos, habría una fase en la que se trazaría que tipo de tests se quiere realizar y sobre este planteamiento pensar en mejoras para después escribir los tests siguiendo una metodología ya optimizada.

Respecto a la aplicación, es posible deducir su funcionamiento sin ni siquiera abrir la aplicación. Solamente con seguir el flujo de los tests es posible entender su uso. Además a nivel de código, los tests de integración y unitarios son la mejor guía para que otros desarrolladores entiendan el ciclo de vida de la aplicación.

Si se quisiera seguir mejorando la aplicación habría que empezar por desarrollar un servidor más sólido que también disponga de sus propios tests y después cubrir todo lo mencionado en el capítulo sobre las pruebas manuales, es decir, validar que lo que se recibe del servidor es correcto y un sistema de notificaciones que informe al usuario de cambios en los datos del servidor.

También se podría separar la aplicación (cliente) del servidor y los tests end-to-end, teniendo tres repositorios individuales. En un caso real en el que la aplicación siguiera creciendo, llegaría un momento en el que una persona no podría mantenerlo todo.

Por otro lado, en este proyecto todo está lanzado en servidores locales, lo cual para entornos de desarrollo es lo ideal debido a la velocidad de respuesta que se obtiene, pero para producción, dado que su finalidad es servir a otros usuarios, conviene lanzarlo a un servidor dedicado. Lo mismo se aplicaría al servidor. Llegados a este caso, los pasos desde que se escribe código hasta que se lanza la aplicación son más complejos ya que depende de agentes externos. No se tendría control para sincronizar el cliente con el servidor y el entorno que ejecute los tests end-to-end, pues no se controla desde el mismo repositorio.

Con este fin se podría utilizar una herramienta como Jenkins[11], que permite automatizar y sincronizar varios repositorios. Entonces, como ejemplo, al desarrollar una nueva actualización en el front-end, todo seguiría un proceso en el que se ejecuten los tests de integración y unitarios, si los tests devuelven un resultado exitoso, lanza la aplicación en un entorno seguro, se aplican las pruebas end-to-end y si todo ha salido bien, se lanzaría la aplicación en el entorno de producción.

Todos estos cambios no cambiarían el hecho de que realizar pruebas manuales es necesario. Los experimentos realizados en esa fase quieren demostrar que da igual la cantidad y alcance de tus tests, siempre habrá algo que se escape. Este hecho no indica una mala gestión en los test, pues como se ha observado, son casos límite que incluso los tests a veces detectan aunque el propósito inicial del test sea diferente al error obtenido.

Como analogía a este experimento se menciona la cita con la que se abre el proyecto:

- *"Sólo porque hayas contado todos los árboles no significa que hayas visto el bosque."*

ANEXO A: MONTAJE DE LA APLICACIÓN Y EJECUCIÓN DE LOS TESTS

En este anexo se va a explicar como acceder al código de la aplicación, base de datos y tests junto con su posterior ejecución.

Ha sido probado en una máquina con el sistema Ubuntu 18.04.5 LTS pero todo debería funcionar en cualquier sistema UNIX (Linux o MAC). En Windows podría ser posible, pero no se ha probado, es por ello que se desconoce el tipo de dependencias necesario para poder realizar todos los pasos que se describirán en este apartado.

Antes de empezar es indispensable tener instalado *yarn* (https://classic.yarnpkg.com/en/docs/install/#debian-stable) en la máquina. Este gestor de dependencias de JavaScript requiere de *NodeJS* (https://nodejs.org/en/download/package-manager/), es por ello que primero se instalará este último.

Todo está alojado en un repositorio de GitHub por lo que el primer paso sería clonar el repositorio al equipo local. Antes de esto, habría que asegurarse de instalar Git (https://git-scm.com/book/en/v2/Getting-Started-Installing-Git) en el equipo y, tras esto, ejecutar el siguiente comando:

git clone https://github.com/mario1105/tfg-app

Una vez clonado habría que instalar las dependencias lo cual se realiza con el siguiente comando:

yarn install

En este momento ya estaría todo listo para proceder con la ejecución de los tests unitarios y de integración la cual se realizaría con:

yarn test-all

Para lanzar el servidor, la aplicación y Cypress conviene utilizar una terminal independiente para cada uno pues así podría verse todo lo que ocurre en estos.

Para el servidor existen dos comandos, uno con una base de datos para el uso normal de la aplicación y otro con una base de datos preparada para los tests end-to-end. Estos comandos son los siguientes:

yarn start-server

yarn start-testing-server

Para montar la aplicación en el entorno de desarrollo, el cual es usado para los tests end-to-end se introduciría el comando:

yarn start

Para montar la aplicación en el entorno de producción habría que ejecutar los siguientes comandos:

yarn build

yarn global add serve

serve -s build

Para abrir la interfaz gráfica de Cypress se utiliza:

yarn run cypress open

Antes de seguir con los tests de Cypress habría que asegurarse de que Chrome está instalado.

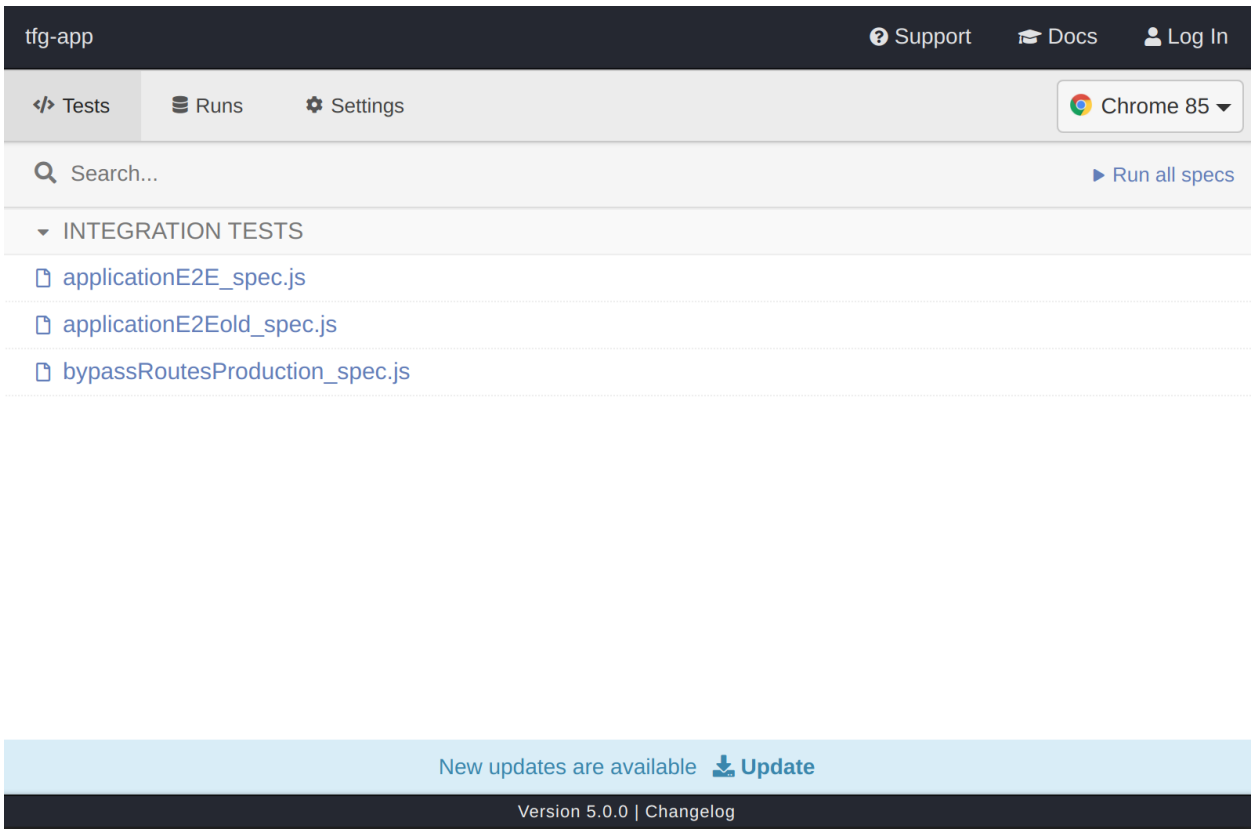

Figura 37 – Vista inicial de Cypress

En la interfaz de Cypress, mostrada en la figura 37, aparecen tres ficheros de tests:

- 'applicationE2E_spec.js': tests end-to-end después de la optimización (finales).
- 'applicationE2Eold_spec.js': tests end-to-end antes de la optimización.
- 'bypassRoutesProduction_spec.js': tests end-to-end para comprobar el no acceso a las rutas bypass en producción.

Para ejecutarlos solo habría que hacer click en cualquiera de estos y se lanzaría Chrome manejado por Cypress, donde se vería a tiempo real como se van ejecutando los tests.

REFERENCIAS

[1] https://es.wikipedia.org/wiki/Boeing_737_MAX

[2] https://es.wikipedia.org/wiki/Vuelo_610_de_Lion_Air

[3] https://es.wikipedia.org/wiki/Vuelo_302_de_Ethiopian_Airlines

[4] https://www.economist.com/gulliver/2019/07/10/the-end-is-not-yet-in-sight-for-boeings-737-max-crisis

[5] https://github.com/typicode/json-server

[6] https://jestjs.io/docs/en/expect

[7] https://jestjs.io/docs/en/jest-object

[8] https://enzymejs.github.io/enzyme/

[9] https://docs.cypress.io/guides/getting-started/writing-your-first-test.html#Write-your-first-test

[10] https://docs.cypress.io/api/api/table-of-contents.html

[11] https://www.jenkins.io/

[12] https://material-ui.com/