Estudio comparativo de herramientas end-to-end testing.

Cypress y Testcafe

2º Desarrollo de Aplicaciones Web — Javier Heredia Molero

I.E.S. Francisco Ayala — Jesús López Pinzolas

Índice

1. Introducción al end-to-end (E2E) testing

Debido al gran incremento en el desarrollo y uso de aplicaciones web, así como el aumento de la complejidad de éstas, se ha vuelto cada vez más necesario realizar tests a estas aplicaciones para comprobar que funcionan correctamente. Estas aplicaciones suelen hacerse siguiendo procesos de desarrollo ágil, por ejemplo siguiendo la metodología de Desarrollo Rápido de Aplicaciones (RAD), por lo tanto tienen un ciclo de desarrollo más corto. Ésto, sumado a que los requisitos del usuario/cliente suelen ampliarse y cambiarse, aumenta el trabajo que tiene que dedicarse al testeo y mantenimiento de las aplicaciones web [1, 2].

No dedicar el trabajo suficiente al testeo de las aplicaciones puede ser problemático, ya que cuanto más se tarde en encontrar un bug, más costoso es arreglarlo, especialmente si ya ha llegado a un entorno de producción y está siendo utilizado por los clientes, cosa que puede ocurrir bastante a menudo con las metodologías de desarrollo ágil. La suma de las razones expuestas anteriormente hacen que sea necesario automatizar este tipo de test [1].

Estos test se pueden dividir en tres grandes grupos: Tests unitarios (unit testing), test de integración (integration testing) y test funcionales (functional testing o end-to-end testing).

- Test unitarios: Este tipo de test se encargan de probar pequeños fragmentos de código, por lo general funciones individuales. Si un test necesita acceso a un recurso externo más allá de esa función, por ejemplo acceso a una base de datos o internet, no es un test unitario. Su ventaja es que suelen ser rápidos, fiables y fácilmente repetibles [3, 4].

- Test de integración: Comprueban cómo funcionan en conjunto distintas partes del sistema. Son similares a los test unitarios pero si utilizan recursos externos como bases de datos [3].

- Test funcionales o E2E: Estos tests prueban la funcionalidad completa de una aplicación simulando el comportamiento de un usuario final. En el caso de las aplicaciones web implican utilizar algún tipo de herramienta para automatizar un navegador que luego se dedica a interactuar con las páginas. Si bien este tipo de test hacen un buen trabajo probando aplicaciones desde el punto de vista del usuario, su ejecución requiere más tiempo que la de los test unitarios y se pueden producir resultados falsos debido a problemas de conexión [3, 4].

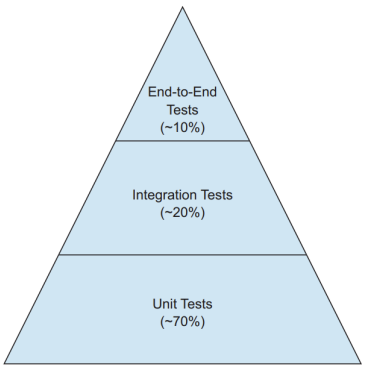

Figura 1: Pirámide de la distribución recomendada para los diferentes tipos de test en un proyecto [4]

Como se puede observar en la figura 1 en la página anterior, según [4] la distribución óptima de los diferentes tipos de tests automatizados en una aplicación debe se aproximadamente del 70 % de tests unitarios, el 20 % de integración y el 10 % E2E. Por lo tanto no es buena idea abusar de este último tipo de test y querer implementarlo en absolutamente todo el proyecto.

1.1. Buenas prácticas

Es muy importante mantener el punto de vista del usuario final cuando se están haciendo test E2E, por lo que hay que centrarse más en las funcionalidades de la aplicación que en su implementación [5]. Debido a que este tipo de test requieren bastante tiempo para ejecutarse hay que incluir solo los escenarios con más impacto para los objetivos que tiene que cumplir la aplicación. Por ejemplo en una aplicación para ver películas uno de los escenarios que habría que comprobar sería que el usuario pueda elegir una película concreta y que ésta se reproduzca sin problemas.

Otra forma de mejorar la implementación de los test E2E es hacer lo que se conoce como Risk-Based Testing (RBT). Ésto consiste en determinar cuales son las características que más fácilmente pueden fallar y el impacto potencial que éstas tendrían en los usuarios finales. Una vez hecho este análisis se dividen los tests en distintos grupos de prioridad para que si es necesario por falta de tiempo se sepa cuál es la parte más crítica que hay que tener testeada antes de lanzar la aplicación [5].

Es necesario que los tests no dependan unos de otros, cada ejecución de un test tiene que ser completamente independiente y su resultado no debe depender del resultado o acciones de un test anterior. Por lo tanto, si estos tests se están ejecutando en un navegador es necesario asegurarse de que corran en nueva instancia de este con todas las cookies limpiadas [5].

1.1.1. Erradicando los tests no deterministas

Un test no determinista es aquel que unas veces se completa correctamente y otras falla, sin que haya ningún cambio apreciable en el código, el test o el entorno.
Por lo general, cuando este tipo de tests empiezan a aparecer en el entorno de testeo son como una «plaga» y si no se arreglan o eliminan pronto pueden arruinar todo el entorno de testeo. Esto se debe a que cuando varios de los tests que se ejecutan tienen resultados no deterministas ya no es posible saber si estos fallos se deben a un bug o son parte del resultado «normal» de esos tests, por lo tanto los desarrolladores dejarán de prestar atención tanto a si los tests están fallando o si funcionan correctamente [6].

Cuando se tienen tests no deterministas lo primero que hay que hacer es ponerlos en cuarentena, de esta forma no afectan a la utilidad del resto de tests «sanos».
Una vez estos tests han sido puestos en cuarentena deben corregirse cuando sea posible, si acaso cuando haya varios, para así volver a tener un sistema de testeo robusto. Si los tests que están en cuarentena se quedan ahí durante mucho tiempo eso contribuye a que el sistema de detección de bugs se vaya degradando, ya que estos test no están cumpliendo su función [6].

Además de no seguir las buenas prácticas para hacer la suite de tests expuestas anteriormente, otra razón por la que los tests pueden dar resultados no deterministas es que utilicen servicios remotos. Es decir, si un tests depende de la respuesta de un servidor es posible que falle a veces, ya sea por que el servidor está caído, ha tardado mucho en responder o la respuesta esa vez sea incorrecta. Por lo tanto

la solución es utilizar un servidor de testeo, independiente del de producción, que vaya a responder rápido y con la respuesta correcta [6].

2. Herramientas a analizar

En este estudio sobre herramientas dedicadas al testeo end 2 end nos vamos a centrar en TestCafé y Cypress. Hemos valorado otros frameworks sugeridos en [3] como Selenium, Webdriver, Puppeteer y Jest, pero tras realizar varias pruebas se han descartado debido a que varios se basan en Selenium, no se pueden ejecutar en remoto o los tests no se escriben en javascript y los desarrolladores Frontend tendrían que aprender otro lenguaje lo que implicaría más tiempo de implementación.

2.1. Características de las herramientas

2.1.1. TestCafé

Figura 2: Logo de TestCafé

TestCafé es un framework publicado en 2016 por DevExpress para hacer tests automatizados. TestCafé permite realizar tests tanto en navegadores viejos como en los más modernos, así como en dispositivos móviles.
Se puede encargar tanto de lanzar las aplicaciones antes de comenzar los tests, de lanzar diferentes navegadores, de ejecutar este tomando capturas de pantalla durante estos así como de mostrar resultados. Además no requiere de ningún otro paquete ni webdrivers adicionales para ejecutarse. [12]

2.1.1.1. Ventajas

Tescafe presenta las siguientes ventajas: [12]

- Los tests se escriben en Javascript, por lo que el desarrollador frontend no necesita aprender ningún nuevo lenguaje para desarrollarlos.
- Permite ejecutar tests en navegadores headless, es decir, sin interfaz gráfica. Esto es muy útil cuando se corren los tests en un sistema de Integración Continua como puede ser Jenkins.
- Permite ejecutar tests en remoto en casi cualquier navegador.
- Espera automáticamente a las aserciones y selectores.
- Tiene una comunidad muy activa y una gran cantidad de plugins.

2.1.1.2. Inconvenientes

Según [13], las principales desventajas de TestCafé son:

- Tescafe no controla el navegador. Eso implica que no se puede automatiza todas las acciones del usuario. Por ejemplo, no se pueden abrir nuevas pestaás o ventanas.

- Eventos simulados. Las acciones como click o doubleClick son simuladas por javascript de forma interna. Eso puede hacer que se den casos en los que algún botón no pueda ser pulsado por el usuario pero sí pueda ser clicado por TestCafé, lo que daría un falso positivo.

2.1.2. Cypress

Figura 3: Logo de Cypress

Cypress es un framework creado por Brian Mann para programar tests End to End hecho en Javascript. Cypress permite realizar pruebas sobre sitios webs de una manera fácil y rápida. La particularidad de Cypress radica en que no usa Selenium para hacer tests, sino que corre por si mismo en un navegador conjuntamente con Node.js. Además, utiliza Mocha para ejecutar runners, Chai para aserciones y Sinon para "mocks". [7]

2.1.2.1. Ventajas Según [8, 9, 10, 11], Cypress tiene las siguientes ventajas frente a otros frameworks de testeo:

- Está escrito en Javascript, lo que permite al desarrollador frontend hacer test sin necesidad de aprender otro lenguaje, de hecho es un proceso de Node.js el que se comunica, sincroniza y lleva a cabo tareas en común.
- No utiliza selenium como "navegador controlado", sino que se ejecuta en el mismo bucle que la aplicación.
- Tiene una comunidad activa, que podemos observar si entramos en stackoverflow.
- Es fácil de instalar y de ejecutar los tests.
- Es *Cross Browser*, es decir, se puede utilizar en varios navegadores como Chrome, Chromium, Firefox, Electron, Edge, etc.
- Se pueden crear test unitarios, test de integración y test end to end.
- Hace instantáneas de los momentos de ejecución de los tests. Lo que te permite debuggear fácilmente pasando el cursor por encima de cada paso.
- No es necesario añadir comandos 'wait', porque Cypress espera automáticamente a comandos y aserciones.
- Es posible un test y al mismo tiempo ver cómo se está ejecutando.
- Verifica y controla el comportamiento de las funciones, las responses o los timers.

2.1.2.2. Inconvenientes

Según [9, 11], Cypress tiene las siguientes desventajas:

- No puede ejecutar tests en otros dispositivos más allá del ordenador donde está corriendo.
- Solo soporta una pestaña abierta.
- Solo soporta Javascript para crear tests, lo que limita su flexibilidad.
- No soporta Safari ni Internet Explorer, por el momento.
- Tiene soporte limitado para los *iframes*.

2.2. Diferencias

Cypress se puede utilizar exclusivamente para ejecutar test en local, lo que limita demasiado su uso. Además, utiliza selectores de jQuery para hacer aserciones [7], pero TestCafé utiliza selectores standards de CSS junto a una API propia para los selectores que permite, entre otra cosas subir en el DOM. Cypress ejecuta runners con Mocha, TestCafé sin embargo utiliza su propio runner.

Tescafe funciona sirviendo la web donde se va a hacer el test mediante un servidor proxy. Este servidor inyecta los scripts necesarios en la página y permite a TestCafé interactuar con ella de forma nativa. Esto implica que puede correr en cualquier navegador, incluyendo móviles y televisiones.
Por otro lado, Cypress controla el navegador mediante su propia API de automatización, lo que obliga a que haya un nuevo driver por cada navegador soportado. [7]

3. Uso de los test en la empresa

Figura 4: Logo de Foxxum

Foxxum es una empresa que se dedica a hacer aplicaciones multimedia con un framework de Javascript para Smart TV's, Android TV's, Fire TV, PlayStation y Xbox como RlaxxTV, Pantaflix y 5Flix entre otros.

3.1. Requisitos específicos de Foxxum

La principal tesitura que tiene Foxxum para desarrollar aplicaciones para televisiones es que no se cambian con tanta frecuencia como los dispositivos móviles, es decir, suelen tener una vida útil más larga y a veces los usuarios tienen televisiones de hace 5 a 6 años de media, por tanto para poder tener la mayor compatibilidad posible con todas las televisiones, se usan métodos e implementaciones antiguas u obsoletos para desarrollar aplicaciones.

Estas aplicaciones deben estar pensadas para ser controladas con un mando a distancia, Foxxum tiene una librería dedicada a interpretar los códigos de las teclas que reciben las televisiones y transformarlos en la tecla correcta.

Éstos son algunos ejemplos:

- Teclas importantes para la navegación:
  - `VK_ENTER`: Para pulsar enter u ok
  - `VK_LEFT`: Desplazamiento a la izquierda
  - `VK_UP`: Desplazamiento hacia arriba
  - `VK_RIGHT`: Desplazamiento a la derecha
  - `VK_DOWN`: Desplazamiento hacia abajo
  - `VK_BACK_SPACE`: Botón hacia atrás
- Teclas para controlar el multimedia:
  - `VK_PLAY`: Botón play
  - `VK_PAUSE`: Botón pause
  - `VK_PLAY_PAUSE`: Botón para reanudar y pausar.
  - `VK_STOP`: Botón para parar.
  - `VK_PREV`: Botón para ir al anterior.

- `VK_NEXT`: Botón para ir al siguiente.
- `VK_FAST_FWD`: Botón para adelantar.
- `VK_REWIND`: Botón para rebobinar.

- Teclas menos importantes, pero que se usan con funcionalidades especiales, dependiendo de la aplicación:
  - `VK_RED`: Botón rojo.
  - `VK_GREEN`: Botón verde.
  - `VK_YELLOW`: Botón amarillo.
  - `VK_BLUE`: Botón azul

3.2. Limitaciones de cada herramienta en el contexto de la empresa

3.2.1. TestCafé

Si bien TestCafé puede ejecutarse en las televisiones no funciona en algunos modelos, especialmente los más antiguos, debido a que los navegadores no tienen las funciones necesarias para que testcafe-hammerhead, el proxy que utiliza TestCafé, se ejecute correctamente.

Otra limitación es que no puede mandar los códigos de teclas nativos de los mandos, pero este problema lo hemos solucionado creando un helper que permite hacerlo. Estas funciones se explicarán en el siguiente apartado.

```
import { tvPressKey, initializeKeymap } from './helpers/keyHelpers';
import Keymap from './dependencies/keymap';

fixture`keyHelpers`.page`https://example.com`;

test( name: 'Press Key test', fn: async (t : TestController ) => {
  // Inicializar el keymap
  await initializeKeymap();

  // Pulsar abajo una vez
  await tvPressKey(Keymap.VK_DOWN, times: 1);
  // Pulsar derecha una vez y esperar 500 ms después
  await tvPressKey(Keymap.VK_ENTER, times: 1, wait: 500);
  // Pulsar abajo dos veces
  await tvPressKey(Keymap.VK_DOWN, times: 2);
  // Pulsar derecha tres veces
  await tvPressKey(Keymap.VK_RIGHT, times: 3);
  // Pulsar izquierda una vez y esperar 500 ms después
  await tvPressKey(Keymap.VK_LEFT, times: 1, wait: 500);
  // Pulsar arriba dos veces y esperar 500 ms después de cada pulsación
  await tvPressKey(Keymap.VK_UP, times: 2, wait: 500);
  // Esperar 40 segundos
  await t.wait( timeout: 40000);
});
```

Figura 5: Test realizado en TestCafé utilizando el helper `tvPressKey`

3.2.2. Cypress

El mayor inconveniente de Cypress es que no se puede utilizar en televisiones, pero se pueda utilizar en local simulando las teclas de la televisión. Para esta tarea se desarrolló una función específica para simular las teclas del mando a distancia.

Este ejemplo es un pequeño adelanto del helper `tvPressKey`:

```
import KeyMap from './helpers/keymap.js';

describe('SoundCloud test', () => {
  it('Visiting SoundCloud', () => {
    // Inicializa el KeyMap
    KeyMap.init();
    // Visita la aplicación de Rlaxx-TV
    cy.visit('https://localhost:3000');
    // Espera hasta que el botón con el id continue se haga visible
    cy.get('#continue', { timeout: 100000 }).should('be.visible');
    // pulsa tres veces enter, para saltar los mensajes de cookies, políticas de privacidad...
    cy.tvPressKey(KeyMap.VK_ENTER, 3);
    // pulsa tres veces el botón a la derecha para cambiar de canal
    cy.tvPressKey(KeyMap.VK_RIGHT, 3);
    // pulsa enter para seleccionar el canal y espera 5 segundos
    cy.tvPressKey(KeyMap.VK_ENTER, 1, 5000);
    // pulsa dos veces la flecha hacia arriba para ir al menu superior
    cy.tvPressKey(KeyMap.VK_UP, 2);
    // pulsa enter para entrar en el menu superior
    cy.tvPressKey(KeyMap.VK_ENTER);
    // pulsa dos veces la flecha hacia abajo y espera dos segundos para navegar por la guía de programas
    cy.tvPressKey(KeyMap.VK_DOWN, 2, 2000);
    // selecciona el programa escogido y entra en el programa
    cy.tvPressKey(KeyMap.VK_ENTER, 2, 3000);
  });
});
```

Figura 6: Test realizado utilizando el helper `tvPressKey`

4. Desarrollo de la librería para Foxxum

Tanto para TestCafé como para Cypress realizamos unas funciones específicas para la empresa con la que podían automatizar los test que se iban a realizar, porque hasta ahora, casi todos los tests eran a mano. Ésto ahorraría a la empresa horas y a los empleados, la monotonía de hacer el mismo test para diferentes televisiones.

Los principales objetivos a la hora de hacer tests automatizados en Foxxum son:

- Emular las teclas de los mandos a distancias.
- Hacer aserciones para comprobar si un elemento tiene una propiedad.
- Usar un plugin de React para comprobar *states*, *props* y poder utilizar componentes de React en los tests.
- Poder comprobar tanto las requests como sus responses.
- Añadir estos tests a un servicio de integración continua como Jenkins.
- Utilizar estos tests como pruebas de regresión para las nuevas actualizaciones.

4.1. TestCafé

Para Tescafé se desarrolló un helper enfocado a enviar los códigos correctos de las teclas de cada televisión utilizando el keymap proporcionado por Foxxum. De este modo el test es más cercano a la experiencia del usuario final.
Esta función se utiliza pasándole el código de la tecla que se quiere mandar, el número de veces que se quiere pulsar esa tecla y el tiempo de espera después de cada pulsación.

Para controlar el tiempo de espera después de cada pulsación se utiliza la función de la figura 7. Por defecto es de 150 ms, pero se puede cambiar al valor deseado utilizando `changeDefaultTimeBetweenActions` que modifica el valor de todas las esperas desde que se llama en adelante. Esta función recibe el tiempo de espera deseado en milisegundos.

```
/**
 * Default time to wait after custom actions
 * @type {{timeBetweenActions: number}}
 */
const config = {
  timeBetweenActions: 150,
};

/**
 * Change the time to wait after custom actions
 * @param {Number} timeBetweenActions
 */
export const changeDefaultTimeBetweenActions = (timeBetweenActions : Number ) => {
  if (isNaN(timeBetweenActions)) return;
  config.timeBetweenActions = parseInt(timeBetweenActions,  radix: 10);
};
```

Figura 7: Configuración del tiempo de espera después de cada pulsación

Para lanzar el evento de la pulsación de la tecla se utiliza la función de la figura 8. Esta función recibe el código de la tecla de la cual se quiere simular la pulsación, el tipo de evento que se quiere lanzar, por defecto `keydown`, y modificadores que se le pueden añadir al evento.

```
/**
 * Dispatch an event with the indicated keyCode
 * @function
 * @param {Number} keycode
 * @param {String} type
 * @param {Object} modifiers
 */
const simulateKey = ClientFunction( fn: (keyCode, type = 'keydown', modifiers = {}) => {
  const evtName = typeof type === 'string' && type.indexOf('key') !== -1 ? `${type}` : 'keydown';
  const modifier = typeof modifiers === 'object' ? modifiers : {};
  const event = document.createEvent( eventInterface: 'HTMLEvents');
  event.initEvent(evtName,  bubbles: true,  cancelable: false);
  event.keyCode = keyCode;
  for (const i in modifier) {
    if (Object.prototype.hasOwnProperty.call(modifier, i)) {
      event[i] = modifier[i];
    }
  }
  event.force = true;
  document.dispatchEvent(event);
});
```

Figura 8: Función que lanza el evento con la pulsación de la tecla

La función que se utiliza desde el test que se está escribiendo es `tvPressKey`, esta función (figura 9) recibe el código de la tecla que se quiere pulsar, el número de veces que se quiere pulsar y el tiempo que hay que esperar después de cada pulsación. Si no recibe número de veces ni tiempo de espera los valores por defecto son 1 y 150 ms respectivamente.

```
/**
 * Simulate a key from a TV Remote. It uses the Keymap VK keys.
 * @param {Number} key VK key from Keymap
 * @param {Number} times How many key presses to do
 * @param {Number} wait
 * @returns {Promise<void>}
 */
export async function tvPressKey(key : Number , times : Number  = 1, wait : Number  = config.timeBetweenActions) {
  if (isNaN(times) || isNaN(wait)) {
    throw new Error('times or wait isNaN');
  }

  for (let i = 0; i < parseInt(times,  radix: 10); i++) {
    await simulateKey(key);
    await t.wait(parseInt(wait,  radix: 10));
  }
}
```

Figura 9: Función para simular las pulsaciones de las teclas del mando a distancia

Un último problema a solucionar es cargar el teclado correcto para la televisión en la que se está ejecutando. Para esto Foxxum guarda en localStorage la plataforma en la que se está ejecutando la aplicación, por lo que utilizando las funciones de la figura 10 cuando ya se haya inicializado la aplicación podemos saber en qué televisión estamos.

```
/**
 * Initialize the values of the VK from Keymap to the platform the test is running on.
 * @returns {Promise<void>}
 */
export async function initializeKeymap() {
  Keymap.init(await getDevicePlatform());
}

/**
 * Get the device platform info from localStorage
 * @function
 */
const getDevicePlatform = ClientFunction( fn: () => {
  return JSON.parse(window.localStorage.getItem( key: 'FXM_DEVICE_INFO')).platform;
});
```

Figura 10: Función para inicializar el keymap al dispositivo correcto

4.2. Cypress

Para Cypress se desarrolló un *helper* (programa o función encargado de resolver problemas de incompatibilidades) específico para poder usar el *keymap* de foxxum, emulando la navegación de las teclas que se 'envían' a las televisiones.

La función *simulateKey*, figura 11, simula una tecla, recibe como parametros el objeto window, ID de la tecla, el tipo de evento y un objeto vacío que se usa para introducir propiedades del *event.initEvent* e introducir detalles personalizados como por ejemplo que una tecla se utilice para otro evento, por ejemplo *abrir menú*.

Respecto el objeto window, fue importante introducirlo en cypress porque no lo detectaba cuando queríamos pulsar una tecla.

```
const simulateKey = (win, keyCode, type = 'keydown', modifiers = {}) => {
    const evtName = typeof type === 'string' && type.indexOf('key') !== -1 ? `${type}` : 'keydown';
    const modifier = typeof modifiers === 'object' ? modifiers : {};
    const event = win.document.createEvent('HTMLEvents');
    event.initEvent(evtName, true, false);
    event.keyCode = keyCode;
    for (const i in modifier) {
        if (Object.prototype.hasOwnProperty.call(modifier, i)) {
            event[i] = modifier[i];
        }
    }
    win.document.dispatchEvent(event);
};
```

Figura 11: Función *simulateKey*

La constante config, indica el tiempo de espera entre acciones y la función permite aumentar el tiempo de espera, figura 12. El valor mínimo es 150ms.

```
const config = {
  timeBetweenActions: 150,
};
export const changeDefaultTimeBetweenActions = (timeBetweenActions) => {
  if (isNaN(timeBetweenActions)) return;
  config.timeBetweenActions = parseInt(timeBetweenActions, 10);
};
```

Figura 12: Constante config y función para cambiar el tiempo entre acciones

A raíz de estas funciones, desarrollamos un comando específico para Cypress llamado *tvPressKey*, figura 13. Recibe como parametros la tecla, las veces que se repite dicha tecla, y el tiempo de espera entre pulsar una tecla y luego otra.

```
Cypress.Commands.add('tvPressKey',(key,times = 1, wait = config.timeBetweenActions)=>{
    if (isNaN(times) || isNaN(wait)) {
        throw new Error('times or wait isNaN');
    }
    for (let i = 0; i < parseInt(times, 10); i++) {
        cy.window().then((win)=> {
            simulateKey(win, key);
        });
        cy.wait(parseInt(wait, 10));
    }
})
```

Figura 13: Función *tvPressKey*

5. Documentación

La documentación completa se adjunta en el archivo zip subido en moodle. Esta documentación ha sido realizada con docz.

6. Ejemplos de tests sobre aplicaciones de televisión

Se han ejecutado unos tests sobre una aplicación de televisión para SoundCloud y así mostrar lo estudiado anteriormente.

6.1. TestCafé

En el test de la figura 14 navegamos por la home, empezamos a reproducir una canción, avanzamos la canción, volvemos hacia el home y por último comprobamos que hay tres elementos h2. Como esto es cierto obtenemos el resultado de la figura 15 en la que nos indica que todo es correcto.

```
import { Selector, ClientFunction } from 'testcafe';
import keymap from './keymap';
import { initializeKeymap, tvPressKey } from './keyHelpers';

fixture`testPassed`.page('http://javier.fxmtest.com/1/SoundCloud720/#/home');
test('generalTest', async (t) => {
  // esperamos 2 segundos para que termine de hacer todas las peticiones
  await t.wait(2000);
  // inicializamos el keymap
  await initializeKeymap();
  // ahora navegamos por la home hasta la sección de todos los tracks.
  await tvPressKey(keymap.VK_DOWN, 2, 500);
  await tvPressKey(keymap.VK_RIGHT, 2, 500);
  // entramos en el elemento deseado y pausamos la canción
  await tvPressKey(keymap.VK_ENTER, 2, 1000);
  // adelantamos 90 segundos la canción
  await tvPressKey(keymap.VK_RIGHT);
  await tvPressKey(keymap.VK_ENTER, 3, 300);
  // vamos hacia al home
  await tvPressKey(keymap.VK_LEFT, 4, 300);
  await tvPressKey(keymap.VK_ENTER, 1, 300);
  // comprobamos si el elemento existe
  const titleTopViewExists = Selector('#get > p').exists;
  await t.expect(titleTopViewExists).ok();
  // comprobamos que hay 3 elementos h2
  const threeH2 = Selector('h2');
  await t.expect(threeH2.count).eql(3);
});
```

Figura 14: Test con tescafé utilizando selectores nativos y la función tvPressKey

```
> react-project@0.1.0 testPass C:\Users\JavierHeredia\Documents\projects\template-soundcloud
> testcafe \"chrome --autoplay-policy=no-user-gesture-required\" tests/testcafe/testPassed.js

 Running tests in:
 - Chrome 91.0.4472.101 / Windows 10

 testPassed
 √ generalTest


 1 passed (9s)

Process finished with exit code 0
```

Figura 15: Resultado del test de la figura 14

En el test de la figura 16 comprobamos si existe un h3. Como esto es falso obtenemos el resultado de la figura 17 en la que nos indica en qué línea en concreto se ha producido el fallo.

```
import { Selector, ClientFunction } from 'testcafe';
import keymap from './keymap';
import { initializeKeymap, tvPressKey } from './keyHelpers';
/*eslint-disable*/
fixture`testPassed`.page('http://javier.fxmtest.com/1/SoundCloud720/#/home');
test('generalTest', async (t) => {
  // esperamos 2 segundos para que termine de hacer todas las peticiones
  await t.wait(2000);
  // Vamos a hacer una aserción errónea para ver el resultado
  // En el home vamos a comprobar si existe algun h3
  const existH3 = Selector('h3').exists;
  await t.expected(existH3).ok();
});
```

Figura 16: Test con tescafé comprobando algo que va a ser falso

```
> react-project@0.1.0 testError C:\Users\JavierHeredia\Documents\projects\template-soundcloud
> testcafe \"chrome --autoplay-policy=no-user-gesture-required\" tests/testcafe/testError.js

 Running tests in:
 - Chrome 91.0.4472.101 / Windows 10

 testPassed
 × generalTest

   1) TypeError: t.expected is not a function

      Browser: Chrome 91.0.4472.101 / Windows 10

          7 |  // esperamos 2 segundos para que termine de hacer todas las peticiones
          8 |  await t.wait(2000);
          9 |  // Vamos a hacer una aserción errónea para ver el resultado
         10 |  // En el home vamos a comprobar si existe algun h3
         11 |  const existH3 = Selector('h3').exists;
       > 12 |  await t.expected(existH3).ok();
         13 |});
         14 |

         at <anonymous> (C:\Users\JavierHeredia\Documents\projects\template-soundcloud\tests\testcafe\testError.js:12:11)
         at asyncGeneratorStep (C:\Users\JavierHeredia\Documents\projects\template-soundcloud\tests\testcafe\testError.js:3:236)
         at _next (C:\Users\JavierHeredia\Documents\projects\template-soundcloud\tests\testcafe\testError.js:3:574)
```

Figura 17: Resultado del test de la figura 16

Ahora vamos a realizar otros tests utilizando los selectores de React incluidos en el plugin `testcafe-react-selectors`. En el test de la figura 18 comprobamos que la posición guardada en el estado de la home sea 0, luego nos movemos al siguiente slider y comprobamos que sea 1. Como esto es cierto obtenemos el resultado de la figura 19 en la que nos indica que todo es correcto.

```
import { Selector } from 'testcafe';
import { ReactSelector, waitForReact } from 'testcafe-react-selectors';
import Keymap from './dependencies/keymap';
import { tvPressKey, initializeKeymap } from './helpers/keyHelpers';

// Crear una fixture que espere a que React esté cargado antes de comenzar los tests
fixture`SoundCloud`.page`http://javier.fxmtest.com/1/SoundCloud720/#/home`.beforeEach( fn: async () => {
  await waitForReact();
});

test( name: 'Comprobar los slides se muevan correctamente', fn: async (t : TestController ) => {
  // Esperar a que las rutas se hayan montado. Si esto no ocurre en 30 segundos el test fallará.
  const routes = ReactSelector( selector: 'Routes');
  await t.expect(routes.exists).ok( options: { timeout: 30000 });

  // Inicializar el keymap
  await initializeKeymap();

  // Seleccionar el Home
  const homeView = ReactSelector( selector: 'Home');
  // Guardar el estado actual del home
  let homeViewState = await homeView.getReact( filter: ({ state }) => state);

  // Comprobar que la posición guardada en el estado es 0
  await t.expect(homeViewState.position).eql( expected: 0);

  // Moverse al slide de abajo y comprobar que ahora la posición valga 1
  await tvPressKey(Keymap.VK_DOWN);
  // Volver a guardar el estado de nuevo
  homeViewState = await homeView.getReact( filter: ({ state }) => state);
  // Comprobar que la posición guardada en el estado es 1
  await t.expect(homeViewState.position).eql( expected: 1);
});
```

Figura 18: Test con tescafé utilizando selectores para react y la función tvPressKey

```
Running tests in:
- Chrome 91.0.4472.101 / Windows 10

SoundCloud
√ Comprobar los slides se muevan correctamente


1 passed (3s)
```

Figura 19: Resultado del test de la figura 18

En el test de la figura 20 hacemos lo mismo pero luego comprobamos si la posición es 2. Como esto es falso obtenemos el resultado de la figura 21 en la página siguiente en la que nos indica en qué línea en concreto se ha producido el fallo.

```
import { Selector } from 'testcafe';
import { ReactSelector, waitForReact } from 'testcafe-react-selectors';
import Keymap from './dependencies/keymap';
import { tvPressKey, initializeKeymap } from './helpers/keyHelpers';

// Crear una fixture que espere a que React esté cargado antes de comenzar los tests
fixture`SoundCloud`.page`http://javier.fxmtest.com/1/SoundCloud720/#/home`.beforeEach( fn: async () => {
  await waitForReact();
});

test( name: 'Comprobar los slides se muevan correctamente', fn: async (t : TestController ) => {
  // Esperar a que las rutas se hayan montado. Si esto no ocurre en 30 segundos el test fallará.
  const routes = ReactSelector( selector: 'Routes');
  await t.expect(routes.exists).ok( options: { timeout: 30000 });

  // Inicializar el keymap
  await initializeKeymap();

  // Seleccionar el Home
  const homeView = ReactSelector( selector: 'Home');
  // Guardar el estado actual del home
  let homeViewState = await homeView.getReact( filter: ({ state }) => state);

  // Comprobar que la posición guardada en el estado es 0
  await t.expect(homeViewState.position).eql( expected: 0);

  // Moverse al slide de abajo y comprobar que ahora la posición valga 1
  await tvPressKey(Keymap.VK_DOWN);
  // Volver a guardar el estado de nuevo
  homeViewState = await homeView.getReact( filter: ({ state }) => state);
  // Comprobar que la posición guardada en el estado es 2
  // Esta línea va a fallar
  await t.expect(homeViewState.position).eql( expected: 2);

  // El test no llegará a esta línea
  console.info( data: 'Test terminado correctamente');
});
```

Figura 20: Test con tescafé comprobando algo que va a ser falso

```
Running tests in:
- Chrome 91.0.4472.101 / Windows 10

SoundCloud
× Comprobar los slides se muevan correctamente

  1) AssertionError: expected 1 to deeply equal 2

     + expected - actual

     -1
     +2


     Browser: Chrome 91.0.4472.101 / Windows 10

        28 |  await tvPressKey(Keymap.VK_DOWN);
        29 |  // Volver a guardar el estado de nuevo
        30 |  homeViewState = await homeView.getReact(({ state }) => state);
        31 |  // Comprobar que la posición guardada en el estado es 2
        32 |  // Esta línea va a fallar
      > 33 |  await t.expect(homeViewState.position).eql(2);
        34 |
        35 |  // El test no llegará a esta línea
        36 |  console.info('Test terminado correctamente');
        37 |});
        38 |

        at <anonymous> (C:\Users\                                    test1.js:33:42)
        at asyncGeneratorStep (C:\Users\                                \test1.js:4:244)
        at _next (C:\Users\                                        \test1.js:4:582)
```

Figura 21: Resultado del test de la figura 20

6.2. Cypress

En el test de la figura 22 navegamos por la home, empezamos a reproducir una canción, avanzamos la canción, volvemos hacia el home y por último comprobamos que hay tres elementos h2. Como esto es cierto obtenemos el resultado de la figura 23 en la página siguiente en la que nos indica que todo es correcto.

```
import KeyMap from '@foxxum-modules/virtual-keys';

describe('SoundCloud test', () => {
  it('Visiting SoundCloud', () => {
    // Inicializa el KeyMap
    KeyMap.init();
    // Visita la aplicación de SoundCloud y espera un segundo hasta que termine de hacer las peticiones
    cy.visit('http://javier.fxmtest.com/1/SoundCloud720/#/home').wait(1000);
    // esperamos a obrener el root de React
    cy.waitForReact(2000, '#root');
    // Presiona una vez la tecla de direccion abajo
    cy.tvPressKey(KeyMap.VK_DOWN);
    // Presiona la tecla hacia la derecha
    cy.tvPressKey(KeyMap.VK_RIGHT);
    // Pulsa 'ok' para entrar en la canción escogida y espera 5 segundos
    cy.tvPressKey(KeyMap.VK_ENTER, 1, 2000);
    // Adelantamos 30 segundos la cancion
    cy.tvPressKey(KeyMap.VK_RIGHT);
    cy.tvPressKey(KeyMap.VK_ENTER);
    // Después pasamos a la siguiente canción
    cy.tvPressKey(KeyMap.VK_RIGHT);
    cy.tvPressKey(KeyMap.VK_ENTER);
    // Nos salimos de la vista de playlist pulsando la tecla hacia atrás
    cy.tvPressKey(KeyMap.VK_LEFT, 5, 200);
    cy.tvPressKey(KeyMap.VK_ENTER, 1, 1000);
    // Ahora que estamos de vuelta en la pantalla principal, vamos a realizar algunas aserciones.
    cy.get('p').should('have.text', 'Microcosmos ChillOut');
    // Comprobamos que hay tres h2
    cy.get('h2').should('have.length',3);
    // Comprobamos que la sección de arriba tiene un width de 1280px
    cy.get('#topView').should('have.css','width', '1280px')
    // Comprobamos la props location de nuestra aplicación:
    cy.getReact('Home').getProps('location');
  });
});
```

Figura 22: Test con cypress utilizando la función tvPressKey

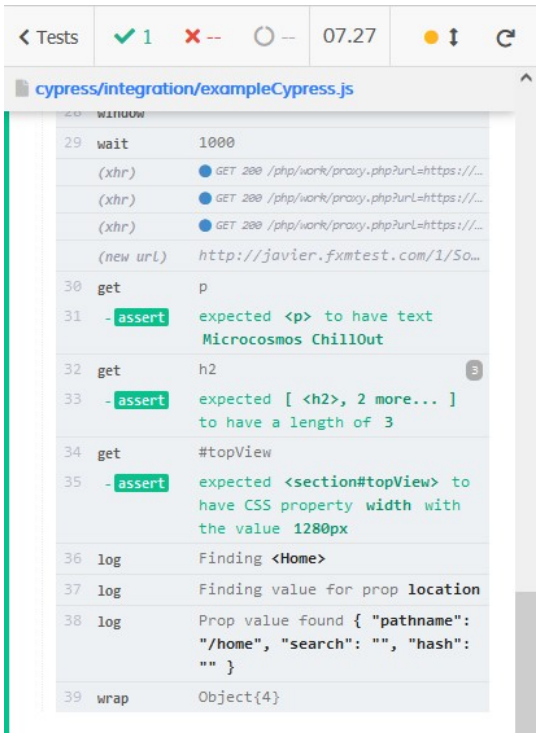

Figura 23: Resultado del test de la figura 22

En el test de la figura 24 entramos a una canción y comprobamos que el título de esa canción sea «I.E.S. Francisco Ayala». Como esto es falso obtenemos el resultado de la figura 25 en la que nos indica dónde se ha producido el fallo.

```
import KeyMap from '@foxxum-modules/virtual-keys';

describe('SoundCloud test', () => {
  it('Visiting SoundCloud', () => {
    // Inicializa el KeyMap
    KeyMap.init();
    // Visita la aplicación de SoundCloud y espera un segundo hasta que termine de hacer las peticiones
    cy.visit('http://javier.fxmtest.com/1/SoundCloud720/#/home').wait(1000);
    // esperamos a obrener el root de React
    cy.waitForReact(2000, '#root');
    // Presiona una vez la tecla de direccion abajo
    cy.tvPressKey(KeyMap.VK_DOWN);
    // Presiona la tecla hacia la derecha
    cy.tvPressKey(KeyMap.VK_RIGHT);
    // Pulsa 'ok' para entrar en la canción escogida y espera 5 segundos
    cy.tvPressKey(KeyMap.VK_ENTER, 1, 2000);
    // vamos a hacer una aserción que falle aposta para poder ver cómo se visualizaría
    // comprobamos que el encabezado tenga el texto: 'I.E.S. Francisco Ayala'
    cy.get('p').should('have.text', 'I.E.S. Francisco Ayala');
  });
});
```

Figura 24: Test con cypress comprobando algo que va a ser falso

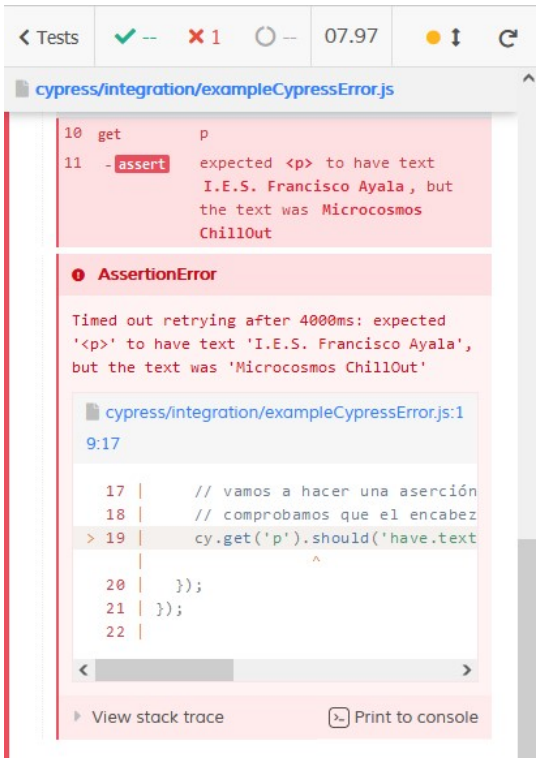

Figura 25: Resultado del test de la figura 24

7. Conclusiones

Después de este estudio de herramientas de testeo end 2 end hemos terminado que la que más se adapta a las necesidades de la empresa es TestCafé.

Si bien Cypress, al igual que TestCafé, puede simular la pulsación de los botones de un mando de televisión, no permite ejecutar tests en navegadores para los que no tenga el driver necesario. Por lo tanto aunque las dos herramientas son muy buenas, están activas y tienen una comunidad grande a su alrededor, la capacidad que tiene TestCafé de ejecutarse en la mayoría de las televisiones a las que Foxxum soporte, así como en VEWD, un emulador que simula una televisión, hace que la balanza se decante a favor de este.

Referencias

[1] Lakshmi, D.R. and Mallika, S.S. (2017). A Review on Web Application Testing and its Current Research Directions. International Journal of Electrical and Computer Engineering (IJECE), [online] 7(4), p.2132. Available at: https://core.ac.uk/download/pdf/189775324.pdf [Accessed 15 Jun. 2021].

[2] Xu, L., Xu, B. and Jiang, J. (2005). Testing web applications focusing on their specialties. ACM SIGSOFT Software Engineering Notes, 30(1), p.10.

[3] Zaidman, V. (2020). An Overview of JavaScript Testing in 2021. [online] Medium. Available at: https://medium.com/welldone-software/an-overview-of-javascript-testing-7ce7298b9870 [Accessed 15 Jun. 2021].

[4] Palmer, J., Cohn, C., Giambalvo, M., Nishina, C. and Green, B. (2018). Testing Angular applications. Shelter Island, Ny: Manning Publications.

[5] Fabbretti, M. (2020). End to end testing: best practices and pitfalls - Quality Assurance - Qualibrate Blog. [online] www.qualibrate.com. Available at: https://www.qualibrate.com/blog/end-to-end-testing-best-practices-and-pitfalls [Accessed 15 Jun. 2021].

[6] Fowler, M. (2011). Eradicating Non-Determinism in Tests. [online] martinfowler.com. Available at: https://martinfowler.com/articles/nonDeterminism.html [Accessed 15 Jun. 2021].

[7] Rhodes, T. (2018). Evaluating Cypress and TestCafé for end to end testing | YLD Blog. [online] YLD. Available at: https://www.yld.io/blog/evaluating-cypress-and-testcafe-end-to-end-testing/ [Accessed 15 Jun. 2021].

[8] Chan, E. (2018). Cypress: The future of end-to-end testing for web applications. [online] Medium. Available at: https://medium.com/tech-quizlet/cypress-the-future-of-end-to-end-testing-for-web-applications-8ee108c5b255 [Accessed 15 Jun. 2021].

[9] Unadkat, J. (2020). Cypress vs Selenium: Key Differences. [online] BrowserStack. Available at: https://www.browserstack.com/guide/cypress-vs-selenium [Accessed 15 Jun. 2021].

[10] Cowan, P. (2021). Cypress vs. Selenium: Why Cypress is the better option. [online] LogRocket Blog. Available at: https://blog.logrocket.com/cypress-io-the-selenium-killer/ [Accessed 15 Jun. 2021].

[11] Cypress.io (2021). Trade-offs. [online] Cypress Documentation. Available at: https://docs.cypress.io/guides/references/trade-offs#Permanent-trade-offs-1 [Accessed 15 Jun. 2021].

[12] Dmytro Shpakovskyi (2020). MODERN WEB TESTING WITH TestCafé : get to grips with end-to-end web testing with TestCafé and... javascript. S.L.: Packt Publishing Limited.

[13] CodeceptJS (2020). Testing with TestCafé | CodeceptJS. [online] codecept.io. Available at: https://codecept.io/testcafe/ [Accessed 15 Jun. 2021].