# Instalación y uso de Cypress en proyectos Angular.

Cuando hablamos de buenas prácticas, nos vienen a la cabeza múltiples conceptos como el código limpio, la estandarización, las librerías compartidas, los controles… Todos ellos muy útiles e importantes. Sin embargo, hay una práctica que generalmente pasamos por alto (ya sea sin querer o intencionadamente, dada su complejidad): las pruebas **E2E** .

Pero… ¿Qué son las pruebas **E2E** ?

Las pruebas **E2E** (end-to-end) son pruebas que validan los ciclos de vida completos de una aplicación. Nos permiten comprobar el comportamiento de nuestra web en casos de uso complejos, simulando lo que haría un usuario real, pero de forma automatizada. Esto representa un cambio radical en cómo una empresa afronta tanto el desarrollo como el mantenimiento de sus aplicaciones. Si se logra una buena cobertura con pruebas **E2E** y se integran satisfactoriamente en los flujos de CI, se invertirá algo de tiempo al principio (formando a los desarrolladores e implementándolas), pero se logrará un ahorro enorme en mantenimiento y un incremento considerable en la confianza del cliente, ya que los errores serán cada vez menos frecuentes.

La principal virtud de las pruebas **E2E** es también su mayor defecto: están vivas. Cada vez que se detecta un error, este debe corregirse junto con una nueva prueba que asegure que no volverá a ocurrir. A cambio, esto implica que no se pueden abandonar: cada vez que se modifique un componente, deberemos adaptar también sus pruebas para garantizar que siguen siendo válidas y útiles.

Adoptar las pruebas **E2E** como parte fundamental del desarrollo no es sencillo. Se necesita un equipo con conocimientos técnicos (JavaScript o TypeScript ,aunque no son complejas, hay que aprender a escribirlas) y funcionales (entender qué son pruebas positivas, negativas, cómo estructurarlas…). Pero, sin duda, son una de las mejores herramientas para garantizar la seguridad de tu código.

Otra cuestión clave es que las pruebas **E2E** no son exclusivamente para frontales (aunque, si usas datos simulados (mock, podrían limitarse a ello). Estas pruebas pueden validar llamadas a la **E2E** , comprobar el contenido de la respuesta, las cabeceras, los tiempos de respuesta, etc. En general, una buena batería de pruebas antes de desplegar a producción proporciona una garantía muy alta de que los cambios introducidos no generarán errores.

En nuestro caso, vamos a utilizar **Cypress**. Esta herramienta nos permite ejecutar pruebas **E2E** escritas en **JavaScript** o **TypeScript**, lo que facilita enormemente su desarrollo y permite llevar un control detallado del histórico de ejecuciones. Además, puede integrarse como paso de control en los flujos de **CI/CD** para impedir despliegues si hay errores.

En este documento veremos cómo instalar, configurar y gestionar diferentes proyectos con pruebas **E2E** que garanticen la calidad de nuestro código.

# Instalación. Comencemos por los cimientos.

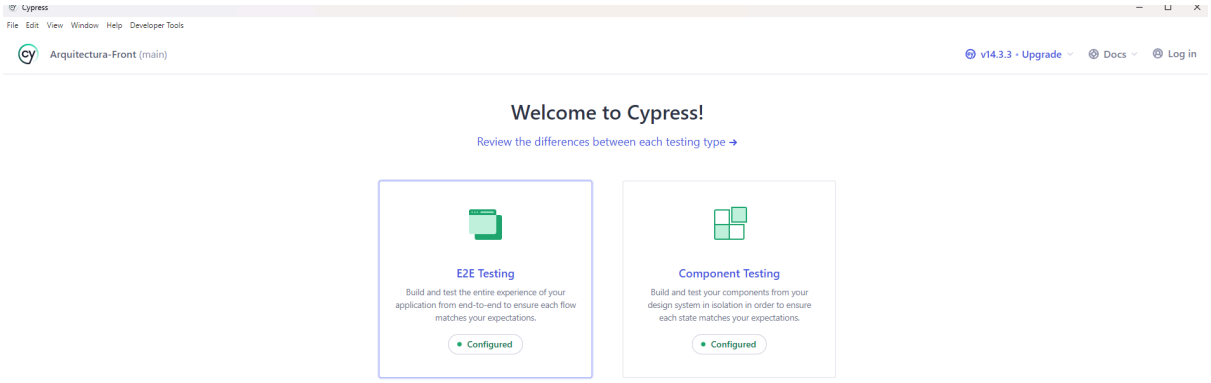
Instalar **Cypress** es una tarea realmente sencilla. Basta con instalar la librería de Cypress desde su página oficial (con npm) o con el siguiente comando: npm install cypress --save-dev.

Una vez tenemos la librería instalada, es una buena idea añadir los scripts a tu **packaje.json** para ejecutarlo (aunque no es obligatorio):

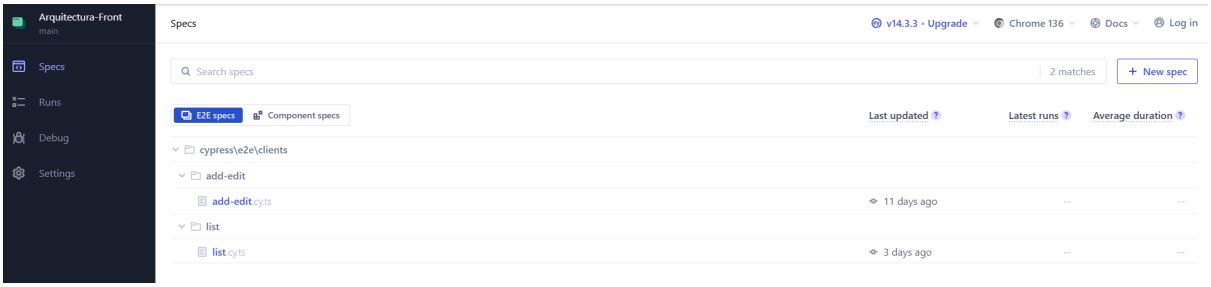
```
"scripts": {
  "cypress:open": "cypress open",
  "cypress:run": "cypress run"
}
```

Por último, para que cypress genere toda la estructura de carpetas que necesita, es necesario lanzarlo por primera vez con el siguiente comando: npx cypress open. para ejecutar este comando es importante que el proyecto este ejecutándose.

Ahora nos aparecerá la ventana principal, en la que generalmente utilizaremos la opción de **E2E testing**. Estas pruebas están diseñadas para hacer un flujo completo de un usuario, mientras que **Component testing** se centra más en las funcionalidades de componentes aislados. A futuro haremos ambas, pero para comenzar, es más sencillo y rentable hacerlo con una de flujo completo. A continuación nos pedirá el navegador que queremos utilizar(en este caso no hay ninguna opción buena o mala, sino que dependerá del proyecto).




Trás seleccionar el navegador, llegarás a la home de Cypress, indicando que cypress ha sido correctamente configurado y está disponible para comenzar a utilizarse (en este caso, el proyecto ya tiene alguna prueba creada).




# Vista general. Guía de supervivencia a la carpeta e2e.

Antes de comenzar a tratar temas más enrevesados, vamos a explicar para qué valen todas las carpetas que ha creado Cypress de manera automática y otras que hemos creado nosotros. Esta es la estructura general de carpetas que debe tener cypress:

```
cypress/
├── downloads/                    # Archivos descargados durante pruebas
├── e2e/                          # Tests end-to-end principales
│   └── clients/                  # Pruebas relacionadas con clientes
│       ├── add-edit/             # Casos de prueba para añadir o editar
│       │   └── add-edit.cy.ts    # Prueba: flujo de alta/modificación
│       └── list/                 # Casos de prueba para listados
│           └── list.cy.ts        # Prueba: vista y paginado de lista
├── fixtures/                     # Datos falsos para simular respuestas
├── screenshots/                  # Capturas automáticas al fallar un test
├── support/                      # Funciones compartidas y comandos globales
└── component-relation-map.json   # Mapa de relaciones entre componentes
```

Vamos a ver cada una de estas carpetas un poco más a fondo:

- **downloads**: Son archivos con los que trabajaremos durante las pruebas para descargar o subirlos. Son ficheros auxiliares que nos ayudarán a trabajar en algunos elementos como los inputs para subir ficheros.

- **e2e**: estas son las pruebas que han escrito los programadores y el equipo de pruebas. Como se puede ver en el diagrama, la organización interna de esta carpeta es por funcionalidades. Generalmente el esquema interno será el mismo que el que encontramos en esos mismos componentes en nuestro proyecto.

- **fixtures**: aquí almacenaremos todos los datos mockeados que necesitemos para trabajar en las pruebas. Es importante destacar que las pruebas e2e no están exentas de las buenas prácticas habituales del código, por lo que si tenemos objetos, servicios u otros elementos comunes, no deben implementarse repitiendo código en cada componente.

- **screenshots**: almacena las imágenes que genera Cypress cuando una prueba falla para el reporte final. Esta carpeta generalmente no es necesario tocarla, ya que cypress se encarga de guardar y borrar sus registros de manera automática.

- **support**: se encarga de almacenar clases comunes, importaciones de scripts o configuraciones de librerías externas que apoyan a nuestras pruebas.

- **component-relation-map.json**: este componente es un diccionario de datos el cual almacena las referencias entre los componentes y las pruebas que tienen asociadas. Este fichero se explicará cómo funciona más adelante, ya que será el encargado de decirnos qué pruebas deben ejecutarse de manera automática.

En general, estas son las carpetas que nos vamos a encontrar cuando trabajemos con cypress, aunque la gran mayoría del tiempo lo haremos en **e2e** haciendo nuestras pruebas.

## De cero a test: anatomía de una prueba Cypress.

Ahora que ya tenemos una vista general de la carpeta de Cypress, vamos a comenzar a profundizar en cómo es una prueba E2E internamente.
Lo primero es entender que en un fichero pueden convivir múltiples pruebas. Estas pruebas se agrupan utilizando la función **describe**. Esta sería una prueba E2E muy sencilla:

```
describe('Comprueba que el listado de clientes se carga correctamente.', () => {
   beforeEach(() => {
     cy.intercept('GET', '**/api/v1/clients').as('getClients');
   });

   it("Obtener el listado completo de clientes", () => {
     cy.viewport(1900, 1200);
     cy.visit('/clients/list');

     cy.wait('@getClients').then((interception) => {
       expect(interception.response?.statusCode).to.eq(200);
       return expect(interception.response?.body).to.be.an('array').and.not.be.empty;
     });
   });
});
```

- **describe**: Lo primero que encontramos es el describe, que sirve para empaquetar pruebas relacionadas. Agrupa tests que pertenecen a una misma funcionalidad, un componente concreto o cualquier otra lógica común. Básicamente: todo lo que tenga sentido revisar en bloque.
  Los **describe** siempre deben llevar una descripción clara. Esto es fundamental, ya que cuando tengamos muchos bloques de pruebas, este texto será lo único que nos ayude a identificarlas y agruparlas.

- **beforeEach**: Esta sentencia nos permite ejecutar comandos antes de cada una de las pruebas. Es muy útil no solo para interceptores, sino también para reiniciar variables, inicializar componentes o generar datos dinámicos.

- **it**: Indica a Cypress que se trata de una prueba. Dentro de cada **it** escribiremos los pasos y validaciones, indicando en el texto qué se está probando exactamente.

Como introducción, estos tres bloques son más que suficientes. La gran mayoría de pruebas los tendrán, aunque con más líneas de código y muchas más validaciones.

# ¿Cómo se construye una prueba?.

Para responder a esta pregunta, hay que tener muy claro cómo funciona una prueba **E2E**. Estas pruebas pretenden simular las interacciones reales que tendría un usuario en nuestra web, por lo que cuando hagamos nuestras pruebas, debemos intentar actuar como lo haría un usuario.
Pero… ¿cómo simulo ser un usuario? Bueno, la navegación y la forma de interactuar con la web será mediante clics, escritura con el teclado, arrastrar elementos… Todo esto mediante la sentencia **cy.get** que nos da **Cypress**.

## cy.get. Encontrando nuestros elementos.

cy.get es la sentencia más importante de **cypress**. Esta nos permite indicar qué elemento de toda el DOM queremos utilizar. Existen diferentes maneras de seleccionar un elemento. Cada una tiene sus ventajas e inconvenientes, pero conocerlas todas nos dará una buena flexibilidad a la hora de trabajar.

- **cy.get('[data-cy="add-request"]')**: esta es la mejor manera de trabajar. Gracias a esta sentencia, podemos identificar de manera única a un elemento de nuestro html mediante el atributo **attr.data-cy=xxxx**. Es la forma más correcta, ya que el identificador es único y no afecta en nada a la programación o maquetación del componente, separando así el uso de pruebas con el de programación.

- **cy.get('#botonEnviar')**: esta manera es bastante eficiente, aunque siempre deberíamos utilizar la primera opción, obtener un elemento por Id no es incorrecto. Para añadir este id al html, se hará mediante **id=botonEnviar**.

- **cy.get('.botonEnviar')**: esta forma es la menos recomendada de lejos. Nos permite acceder a un elemento mediante su clase css. Si bien en algunos casos es la única solución, se debería evitar en la medida de lo posible, ya que es tremendamente volátil y puede verse afectado sin ni siquiera tocar el html. En el html se crea mediante el atributo **class="botonEnviar"**.

Gracias a estos elementos podemos seleccionar casi cualquier elemento de la pantalla (más adelante veremos cómo acceder a elementos html anidados uno dentro de otro). Con esto ya tenemos la primera parte de las pruebas, que consiste en generar el flujo de **clicks.**

### Cuando el get no es suficiente.

En una web sencilla, **get** será suficiente para encontrar siempre los elementos que queremos, sin embargo, cuando la página sea más compleja, con elementos visuales más intrincados, usar el get solo podría no ser suficiente. Para esto, **cypress** nos otorga varias herramientas muy útiles que nos ayudan:

- **find('input')**: permite buscar elementos dentro del elemento marcado, es decir, si ya hemos hecho un get, podemos encadenar un contains para obtener un elemento concreto. Ej: **cy.get('[data-cy="add-request"]').find('input[name="email"]');** Esto nos permitirá agrupar diferentes pruebas agrupadas dentro del mismo elemento.

- **contains('xxx')**: permite buscar un elemento en base a su texto. Si bien para obtener elementos "normales" no es lo mejor (un botón). Puede ayudarte a buscar algunos textos informativos de la web, como por ejemplo mensajes emergentes o textos de advertencia al no haber rellenado un campo. También ayuda a validar que las traducciones funcionan correctamente.

- **cy.get('@alias')**: este formato no nos permite acceder a sitios diferentes, si no que nos permite cambiar el cómo lo hacemos, generalmente con la finalidad de hacer las pruebas más pequeñas y evitar la duplicación del código. El alias es un elemento que guardamos bajo un nombre, gracias al cual, desde ese momento, podremos llamarlo de una manera sencilla. Ej:

```
cy.get('.menu').as('menu')
cy.get('@menu')
```

## .click, type… Dando funcionalidad a las selecciones.

Gracias al **click** podemos seleccionar los diferentes elementos de nuestra web, sin embargo, ahora debemos hacer algo con esta selección, es decir, interactuar con la web como lo haría un usuario. A continuación se describen las principales:

- click: como su nombre indica, permite hacer click sobre el elemento seleccionado. Esto aplica todos los efectos normales de esta acción (eventos en Angular, cambios en el css…). Ej: **cy.get('[data-cy="button-create-request"]').click();**

- type: permite escribir si el componente lo permite. Por ejemplo en un input. Ej: **cy.get('[data-cy="input-name-person"]').type('Gonzalo');** En caso de que la web lo soporte, existe la opción **dblclick()**, el cual se usa igual y simula un doble click.

- **select("xxx")**:Permite seleccionar un elemento de un select. Su funcionamiento es igual que en los anteriores casos.

Cypress posee muchas más opciones a la hora de interactuar con los diferentes elementos de la web, sin embargo, los mencionados anteriormente son los más comunes. Si quieres o necesitas ampliar información sobre los diferentes eventos que puede generar cypress, te dejo aquí el enlace a la documentación oficial de **cypress** para qué peludas consultarlos.

## Validaciones, comprobaciones y condiciones.

Hasta ahora hemos interactuado con la interfaz, haciendo **click**, escribiendo y navegando por nuestras ventanas, sin embargo, te habrás dado cuenta que no hemos probado nada. Ahora debemos empezar a comprobar que nuestra aplicación responde como nosotros queremos.

Para esto cypress nos ofrece varias funciones de validación.

- **should()**: este comando nos permite validar el funcionamiento de un elemento Html. Existen muchos parámetros con los que funciona **should()**. Cada uno de ellos te permite comprobar un aspecto distinto del elemento (**exist**. **not.exist**, **be.visible**…), sin embargo, para no alargar este documento, te dejo aquí el enlace a la documentación oficial, en donde podrás ver todos con sus ejemplos.
- **expect()**:Se emplea con valores obtenidos directamente, por ejemplo dentro de un .then(). Esto nos va a permitir obtener valores asíncronos, como por ejemplo cuando accedemos a datos de una llamada a la API. Ej:
  expect(interception.request.body.name).to.equal('Gonzalo'');
- **and()**: Nos permite concatenar diferentes validaciones en una misma sentencia, permitiéndonos tener un código más limpio al no tener que repetir las sentencias para obtener el selector. Ej:
  cy.get('button').should('be.visible').and('have.class','btn-primary');

Existen más elementos para comparar y poder validar si nuestro código está funcionando, sin embargo, estos que hemos visto son los principales.

Hay que entender que las pruebas **E2E** deben probar ciclos de vida de un usuario, por lo que siempre se deberán hacer pruebas positivas (todo funciona) y negativas (forzar un fallo y asegurarse que está controlado). También es muy conveniente comprobar las llamadas a las **API**, asegurándonos no solo de establecer tiempos máximos de carga, comprobar que el objeto devuelto es correcto o que los POST, PUT, PATCH… funcionan con el modelo que tienen los frontales.

Por último, como extra, hay que indicar que también se deben probar los css de nuestras vistas, asegurándonos de que cuando un elemento debe estar deshabilitado, tiene una clase asociada, que los elementos invisibles sean invisibles… Aunque lo principal de las pruebas **E2E** es verificar el flujo de la aplicación, estas otras pruebas nos ayudarán a alcanzar un nivel mucho más alto, dejando nuestra web blindada no solo en funcionalidad de Fron, si no de Back e incluso maquetación.

# Sincronización y tiempos de espera.Jugando con el tiempo.

Cuando nosotros trabajamos en nuestra página, somos conscientes de las animaciones y los tiempos de carga, sin embargo, cuando nosotros trabajamos con **cypress** este no sabe de manera automática cuando debe seguir con la ejecución de su código. Para solucionar este problema, cypress no da un par de herramientas muy útiles y sencillas:

- **cy.wait()**:  este comando nos permite indicar a **cypress** en ms el tiempo que queremos esperar hasta ejecutar la siguiente operación. Esto es fundamental con muchas animaciones de css, campos que aparecen al ejecutar una opción y cuando se esperan recibir datos de la BBDD.
- **cy.wait('@interceptor')**: el comando es el mismo que el anterior, pero he querido separarlo ya que este se usa de manera distinta. Dentro del parámetro de la función, si en vez de pasarle un tiempo en ms se le manda el identificador de un interceptor, **cypress** obtendrá la respuesta de la llamada a la API, permitiéndonos validar diferentes elementos de la propia llamada.

Con esto tenemos resuelta buena parte de la asincronía con **cypress**, pero surgen nuevas dudas, ¿qué es un interceptor?, ¿cómo se utiliza?, pues bien, en el siguiente punto, entraremos más en detalle sobre los interceptores.

## Interceptores. Hazle el trabajo al BACK.

Los interceptores son elementos que podemos crear durante nuestras pruebas para controlar las llamadas que hacemos a nuestras APIS. Estas funciones interceptan la llamada y obtienen todos los datos, tanto los que enviamos como los que recibimos.

Estas herramientas suponen un avance enorme en nuestras pruebas, ya que ahora podemos no solo garantizar la interacción entre nuestros componentes, si no que podemos validar si la API funciona y si Front y Back se entienden, garantizando el funcionamiento completo.

### ¿Cómo son?, ¿En dónde están?. Entendiendo más a los interceptores.

Vamos a comenzar por lo más básico: ¿dónde se deben declarar los interceptores?
Lo ideal es declararlos dentro de los bloques **before()** o **beforeEach()** cuando se comparten entre varias pruebas (por ejemplo, interceptar una petición de login o configurar una credencial).
Esto no solo ayuda a evitar la duplicación de código, sino que además garantiza que el interceptor esté activo antes de que se dispare la petición real.
Aunque técnicamente puedes declarar un interceptor dentro de un **it()**, no es recomendable salvo que sea estrictamente necesario y exclusivo de ese test. Hacerlo dentro del **it()** puede provocar errores difíciles de detectar si la petición ocurre antes de que el interceptor esté listo.
Por buenas prácticas y coherencia en el código, deberías evitar declarar interceptores dentro de los **it()** y centralizarlos siempre que sea posible en los **before()** o **beforeEach()**.

## Anatomía de un interceptor.

A continuación, vamos a mostrar un ejemplo muy sencillo de un interceptor para comprender sus componentes más básicos:

```
cy.intercept('GET', '**/clients/**').as(
    'get-client-by-id',
);
```

Como hemos visto antes, generalmente se declaran en el before() o beforeEach(). Estas son sus partes más relevantes:

- **cy.intercept**: indica a Cypress que estamos creando un interceptor.

- **Tipo de llamada**: debemos indicar si se trata de un GET, POST, PUT, etc.

- **URL**: se debe especificar la **URL** de la llamada. Es muy importante tener en cuenta que estas pruebas deben funcionar en diferentes entornos. Por ello, es recomendable reemplazar la dirección base del entorno con **, lo que indica a **Cypress** que esta parte puede aceptar cualquier valor.
  Además, si se trata,como en este caso, de un **GET** por **ID** o si se utilizan **QueryString** para construir filtros (como marca el estándar **HTTP REST**), la parte final de la **URL** también suele reemplazarse por **.

- **.as()**: para poder acceder a este interceptor durante nuestras pruebas, se le asigna un alias. Esto nos permitirá hacer referencia a él más adelante, por ejemplo, usando **cy.wait()**.

Con esto ya tenemos un ejemplo básico de la declaración de un interceptor, sin embargo, debo hacer referencia a la generación de errores. Anteriormente en este documento hemos visto que deben existir pruebas negativas, es decir, pruebas que comprueban cómo se comporta la aplicación ante un fallo. para esto podemos utilizar el **mockeo** de datos en un interceptor. Dicho de otra manera, podemos sustituir la respuesta que nos da el servidor por la que nosotros queramos. Esto es útil para ver cómo se comporta la página ante un error inesperado:

```
cy.intercept('GET', '**/clients/**', {
  statusCode: 500,
  body: { message: 'Error del servidor' },
}).as('get-client-error');
```

Como se puede ver, después de la url podemos añadir un json con datos mockeados (recuerda crear los objetos en modelos independientes que se importen en las pruebas), de tal manera que la prueba a sustituir la respuesta real del servidor por la nuestra, activando todos nuestros mecanismos para verificar que no se descontrola.

### Implementando las validaciones en nuestras pruebas.

Crear un interceptor es solo la primera parte, ya que ahora debemos decir a cypress cuando queremos utilizarlo. para esto, en nuestra prueba (it), cuando se haga la llamada(por ejemplo al hacer **click** en un botón **Guardar**) podremos llamar al interceptor de la siguiente manera:

```
cy.wait('@get-client-by-id').then((clientData) => {
        expect(clientData.response?.statusCode).to.eq(200);
});
```

Vamos a ver más en detalle cada una de las partes de esta implementación:

- **cy.wait**: como ya hemos visto antes, nos permite generar una sincronia en el código, obligando a cypress a obtener una respuesta antes de seguir con lo que haya dentro del then.
- **Nombre del alias**: para indicar a cypress cual de todos nuestros interceptores debe utilizar, utilizaremos la sentencia **@get-client-by-id**. Gracias a esto cypress sabrá cómo debe comportarse.
- **Estructura síncrona**: **then** y **clientData**(se puede llamar como quieras, pero asegurate de que sea un nombre claro y específico) nos permitirán obtener el valor de la respuesta de la llamada en nuestro código.
- **Bloque de validaciones**: una vez estamos dentro del **then()** debemos hacer nuestras validaciones con cualquiera de las sentencias que hemos visto anteriormente en este documento.

Es importante destacar la necesidad de hacer múltiples validaciones de una llamada, no solo comprobar que nos devuelve un 200, si no asegurarnos de que lo que le enviamos es correcto, como por ejemplo, si un campo es obligatorio, debemos asegurarnos de que no esté vacío…
Uno de los problemas que podemos encontrar cuando trabajamos con interceptores, es hacer un mal uso de ellos. Siempre debemos evitar obtener datos de una llamada usando el comando **cy.wait(1000)**, ya que aunque pueda parecer pesado trabajar con interceptores, nos aseguramos de que si el Back está saturado en ese momento y va más lento, nuestras pruebas no fallen de forma aleatoria.

## Funciones avanzadas. Obtener la excelencia no es fácil.

Con lo visto hasta ahora, podemos gestionar nuestras pruebas E2E de una manera sencilla. Tenemos todo lo necesario para comenzar a implementarlas y hacer las subidas a producción mucho más tranquilos, sin embargo, antes de pasar a otros temas como la integración continua, es necesario destacar algunos fallos que se cometen al usar **cypress** y que aunque no son errores graves, conviene evitarlos desde el principio.

## ¿before o beforeEach?.

Aunque ya lo hemos visto anteriormente por encima, quiero mencionar algo que ocurre muy a menudo en nuestras pruebas, y es el login. Casi cualquier aplicación que usemos, deberá tener un token para poder validar nuestro usuario cuando hagamos una llamada a la API. Este token se puede utilizar de varias maneras, sin embargo, se haga como se haga, solo deberá hacerse una vez por bloque de pruebas.

Nosotros tendremos un método común que será el que se encargue de hacer el login, y este será llamado siempre desde el **before()** para que solo se ejecute una sola vez.

## Bloques comunes. No te repitas.

Es posible que varias pruebas tengan una base común. En estos casos jamás se debe repetir el código para validar algo que ya ha sido validado una vez, por lo que podemos hacer uso de las funciones comunes.

Estás funciones se declaran a la misma altura que **before()** y pueden ser llamadas desde cada una de nuestras pruebas para evitar repetirnos:

```
 function crearCliente(nombre: string) {
    cy.get('button#crear-cliente').click();
    cy.get('input[name="nombre"]').type(nombre);
  }
```

Para llamar a esta función desde nuestra prueba, bastará con usar **crearCliente('Juan');**

Es posible que para algunas pruebas necesitemos hacer uso de elementos externos a nuestro componente, como por ejemplo cambiar el rol del usuario, o modificar algún dato de nuestro usuario, teniendo que hacer login. Para esto podemos hacer uso del fichero commands ubicado dentro de nuestra carpeta support. Esto nos permite crear fragmentos de pruebas comunes a toda nuestra aplicación, reduciendo el número de líneas que escribimos y asegurando un mantenimiento mucho más fácil.

## El tamaño importa. Pruebas versión extendida con comentarios del programador.

Cuando se realizan pruebas e2e generalmente se cae en hacer pruebas muy extensas, aprovechando que ya has hecho una parte de un alta de un cliente (por poner un ejemplo) para probar otras funcionalidades.

Se debe ser consciente de que una prueba se asegura de validar una funcionalidad específica. Si hay otros ciclos de vida que te gustaría comprobar, debes crear más pruebas y utilizar las funciones vistas anteriormente para no duplicar código.

Un mal uso del tamaño de las pruebas puede hacer que sean difíciles de mantener, largas de ejecutar y en el peor de los casos, errores por falta de memoria en la máquina en la que se están ejecutando.

### Mantén todo en su sitio. Hacer pruebas E2E también es programar.

Cuando desarrollamos pruebas, debemos ser tan cuidadosos como cuando programamos en nuestro lenguaje favorito. Se debe mantener el orden, una nomenclatura clara y seguir los patrones de buenas prácticas como hacemos habitualmente.

Cuando trabajemos con datos que vamos a reutilizar debemos crear esos datos en carpetas comunes ubicadas siempre dentro de la carpeta **support**. Esto nos va  permitir reutilizar métodos, modelos o datos mockeados facilitando la reutilización y la edición de nuestras pruebas.

Otra cuestión a destacar es que las pruebas E2E aceptan comentarios, por lo que no te cortes a la hora de explicar lo que estás haciendo. Si consideras que hay ciertas zonas de tus pruebas que son muy complejas, deja un comentario explicando el por que debe funcionar así. También se muy explícito a la hora de describir cada una de la pruebas, ya que cuando se ejecute cypress, toda la información que vas a poseer sobre la prueba que se esté ejecutando es la descripción que tienes.

### Pruebas independientes. Evita falsos positivos.

Cuando hablamos de pruebas E2E y su funcionalidad de probar ciclos de vida, muchas veces se tiende a pensar que cualquier dato es bueno, encadenando diferentes pruebas las cuales dependen unas de otras.

Cada prueba debe poder funcionar de manera aislada, por lo que no puedes depender de crear un cliente para poder editarlo. La prueba específica para el alta de un cliente, creará un cliente, mientras que la de la edición, o se crea una batería de datos que garantice que ese cliente siempre existirá en la cuenta de pruebas o antes de ejecutarse, se creará ese cliente de manera manual (en vez de mediante una prueba, en el before dar de alta ese cliente). Para facilitar estas gestiones, lo más recomendable es que haya un usuario especial para las pruebas E2E, cuyos datos sean siempre iguales y eso está replicado en cada uno de los entornos. También puede ayudar tener uno o varios endpoints/ scripts en la BBDD que una vez finalicen las pruebas, devuelvan ese usuario a su estado inicial, garantizando así que las pruebas nunca fallarán por problemas de datos.

## Integración continua, pruebas en commits y otras formas de garantizar las subidas.

Hasta ahora, podemos decir que ya contamos con todo lo necesario para incorporar pruebas E2E en nuestros proyectos. Sabemos qué son, cómo se implementan, cómo se

gestionan y cómo se ejecutan. Sin embargo, el siguiente paso es definir cuándo y cómo vamos a ejecutarlas.

Puede parecer sencillo lanzarlas desde nuestras pipelines de forma indiscriminada, pero esto resulta costoso tanto en términos económicos ya que Cypress cobra por bloques de resultados (test results), como en términos de tiempo, ya que en proyectos grandes, ejecutar todo el conjunto de pruebas puede ser muy lento.

Por tanto, necesitamos diseñar una estrategia eficiente que garantice que el código se prueba sólo cuando es necesario, y no en cada cambio. Para ello, distinguiremos dos líneas principales:

1. **Qué pruebas ejecutar**: Solo se deben ejecutar las pruebas afectadas por los cambios en el código. Esto implica detectar qué componentes se han modificado y lanzar únicamente los tests relacionados.

2. **Cuándo ejecutarlas**: No todas las pruebas deben ejecutarse en todos los commits. Podemos establecer diferentes momentos de ejecución según el tipo de acción, como:

    - En pull requests hacia ramas de integración.

    - En la rama principal antes de un despliegue.

## ¿Qué pruebas ejecutamos?. Buscando a los sospechosos.

Imaginemos un proyecto con 15 vistas, cada una de ellas con sus componentes y una batería de pruebas E2E sólidas, sin llegar a una cobertura del 100%, pero sí lo suficientemente grande como para sentirnos tranquilos antes de publicar. Un programador añade una funcionalidad nueva, creando un botón que permite vaciar todo el formulario.

Por supuesto que haciendo uso de la herramienta de Cypress tanto en su interfaz gráfica como por comando el puede ejecutar solo las pruebas de ese componente, sin embargo, cuando suba su código a la pipe, esta ejecutará el comando **cypress run**, haciendo que se ejecuten todas las pruebas. La parte buena de esta historia, es que el código del programador siempre será correcto, garantizando así la subida. Como parte negativa, está pipe entre **npm i**, **ng build** y la ejecución de las pruebas ha durado 1 hora. En ese tiempo, otros compañeros han podido subir cambios, los cuales se han encolado, además, si al empresa solo tiene contratada una o 2 pipelines simultáneas, han bloqueado completamente los despliegues del departamento.

Para evitar esto tenemos dentro del proyecto de ejemplo un fichero llamado run-changed-tests.js, el cual se encargará de lanzar las pruebas de los ficheros que se han modificado.

### run-changed-tests.js. Nuestro aliado de confianza.

Este script se encarga de obtener los cambios que tenemos en **git**, y mediante un diccionario de datos con clave/valor (fichero/pruebas asociadas) podemos saber qué ficheros se han modificado, por lo que también sabemos qué pruebas debemos ejecutar. Después se encarga de ejecutar todas las pruebas marcadas como pendientes.

Si sumamos esto que hemos dicho a husky en un **precommit**, podemos hacer que de manera automática se ejecuten las pruebas de los ficheros cambiados antes de subir nuestros cambios. Si todo se ha ejecutado bien, se subirá con normalidad, sin embargo, si una falla, no se terminará el **commit**.

De esta manera, podemos hacer que cada vez que un programador necesite subir algo, de manera automática se inviertan un par de minutos en ejecutarse las pruebas, garantizando que lo que se sube a develop o main trás un merge funciona correctamente.

### Plan de pruebas para CI.

Gracias al script anterior, el número de pruebas que ejecutaremos será mucho más comedido, reduciendo los tiempos de compilación de las pipes, sin embargo, esto no es suficiente, ya que si 4 programadores cambian 4 ficheros distintos y en todas las ramas lanzamos estas pruebas, volveríamos al problema anterior.

Aquí hay múltiples estrategias válidas, aunque en mi opinión, lo más eficiente es que las pruebas en **CI** solo se ejecuten en la rama destinada a producción, es decir, main, release o la que se use.

De esta manera, si agrupamos los commits en una subida, todo el código estará validado pero no se producirán colas en las pipelines. Esto tiene otra ventaja, y es que reducir el número de pruebas que guardamos en nuestro servidor para el control de código, nos facilita la lectura de estos análisis en el futuro. Más adelante veremos cómo funciona el panel de control de **cypress** de administración, que será el principal afectado en caso de realizar muchos ciclos de pruebas.