**CURSO PLAYWRIGHT**

Índice por clases

[CLASE 1 – 23/07/2025 2](#_Toc206259239)

[Tests 3](#_Toc206259240)

[Posibles métodos que utilizaremos dentro de un proyecto: 3](#_Toc206259241)

[Comandos: 5](#_Toc206259242)

[Estructurar proyecto para hacerlo reutilizable y prepararlo para POM (page object model) 6](#_Toc206259243)

[CLASE 2 – 30/07/2025 8](#_Toc206259244)

[Escenarios que automatizar: 8](#_Toc206259245)

[Diferentes tipos de acciones de la clase: 9](#_Toc206259246)

[Ir a la página: 9](#_Toc206259247)

[Llamada a las clases desde el test: 9](#_Toc206259248)

[Métodos de espera: 10](#_Toc206259249)

[Encontrar elementos sin ids. 10](#_Toc206259250)

[CLASE 3 – 06/08/2025 12](#_Toc206259251)

[HOOKS 12](#_Toc206259252)

[ test.beforeEach 12](#_Toc206259253)

[PRUEBAS BACKEND O API 14](#_Toc206259254)

[Identificar endpoint en ejecución y hacer una llamada al endpoint. 14](#_Toc206259255)

[Guardar la respuesta y trabajar con ella 15](#_Toc206259256)

[Organizar nuestros datos de acceso en json 16](#_Toc206259257)

[CLASE 4 – 13/08/2025 18](#_Toc206259258)

[CLASE 5 – 20/08/2025 19](#_Toc206259259)

# CLASE 1 – 23/07/2025

Carpetas que no queremos subir al repositorio.



Package.json:

1. devDependencies:
   1. Es donde tendremos las librerías que queremos meter si alguien ha realizado algún método que queremos utilizar.



1. Scripts:
   1. Podremos crear funciones específicas, por ejemplo que ejecute solo en mobile/Android.

playRaightConfig:

1. Lo vamos a utilizar para la configuración de ejecución del proyecto.
   1. Que lance pruebas paralelas con:
      1. FullyParallel
2. Projects:
   1. Serían los navegadores

Tests:

1. Todos los que sean spec.ts son los archivos de pruebas.

Instalar:

1. playwright test for visual code
2. material icon theme

## Tests

Example.spec.ts

Dependencias que importamos con **import {test, expect} from ‘ruta’ 🡪** la ruta apunta hacia node\_modules/test/index

Todo esta encapsulado en bloques y lo que está en el import tendrán acceso para todos los casos de pruebas.

**Mejoras de playWright para entrevistas:**

1. PlayWright se ejecuta en el propio navegador no por encima del navegador como selenium.

**Documentacion PlayWrights**

**Cosas interesantes:**

1. **Actions: para todas las acciones que podemos hacer dependiendo del tipo de elemento que tengamos.**
2. **Locators: las diferentes maneras que tenemos de identificar los elementos en playWright.**

### Posibles métodos que utilizaremos dentro de un proyecto:

**En la documentación de playWright tenemos una sección que se llama actions y ahí tendremos todas las acciones que vamos a poder hacer dependiendo del tipo de elemento.**

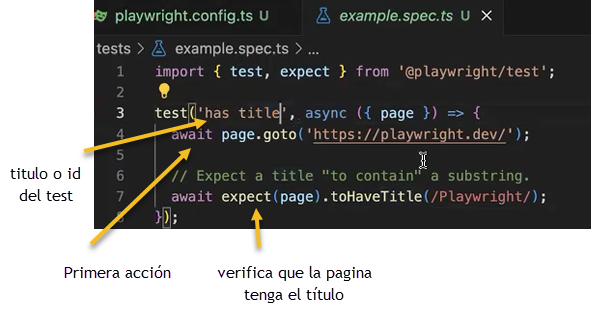
Async 🡪 asíncrona 🡪 se pueden ejecutar mas de un evento al mismo tiempo

Sync 🡪 síncrona

Await 🡪 espera

Page 🡪 sería la ventana que se lanza para realizar un caso de prueba, playWright separa cada caso de prueba por ventana o ambiente.

Page.goto (‘’) 🡪 ruta hacia la web que queramos probar



Page.locator(‘[name=”lastname”]’).fill(‘Torres’) 🡪 encontrar un elemento

.fill 🡪 rellena un campo de texto



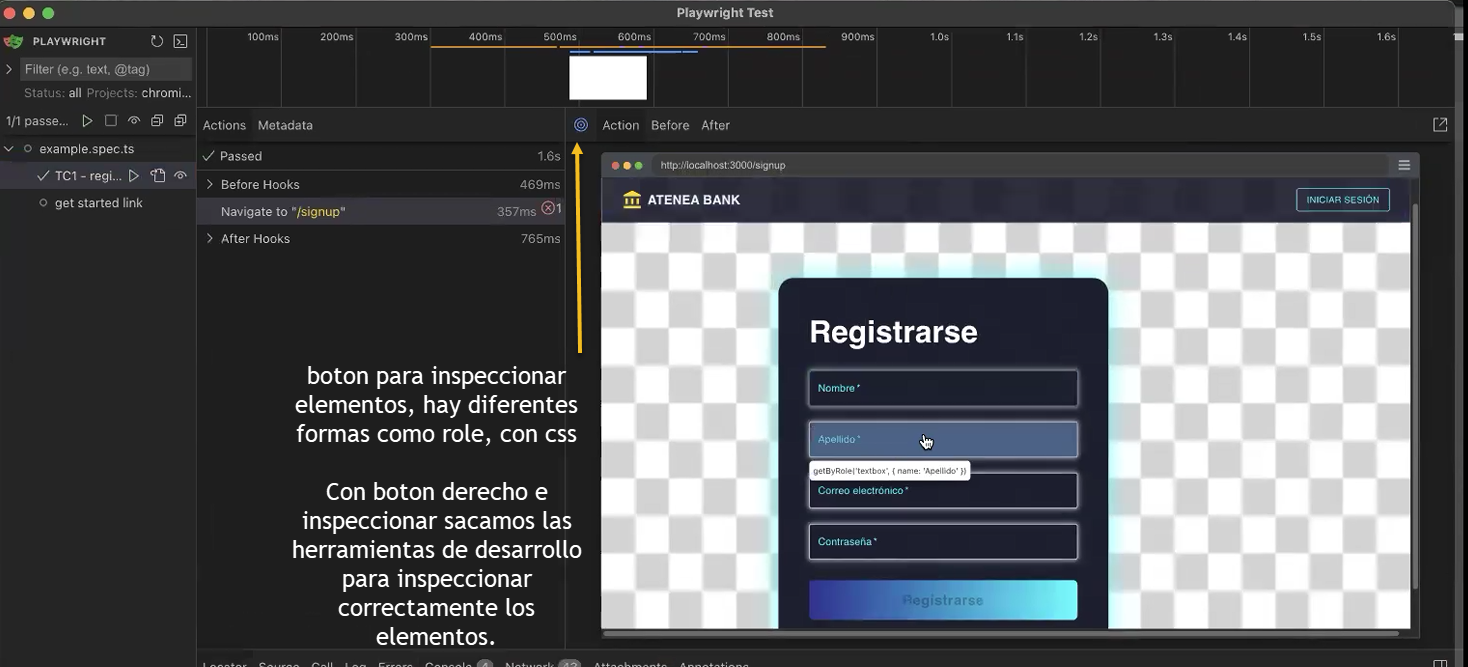
Await page.waitForTimeout(5000) 🡪 tiempo de espera en depuración **!IMPORTANTE BORRAR ESTO UNA VEZ VAYAMOS A SUBIR CAMBIOS**

.expert 🡪 para aserciones, por ejemplo mensaje de registro exitoso.



### Comandos:

**Npx playwright test --ui** 🡪 abre la ventana grafica de playWright para poder encontrar los elementos.



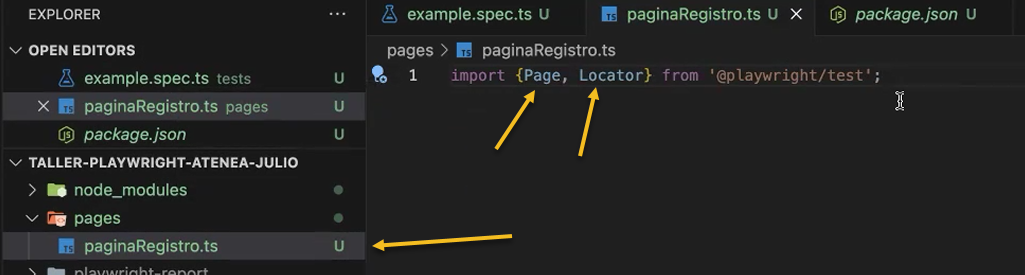
### Estructurar proyecto para hacerlo reutilizable y prepararlo para POM (page object model)

Dentro del proyecto creamos una carpeta pages, esta carpeta contendrá los POM y los archivos que crearemos dentro iremos identificándolas por paginas EJ:

* pagesRegistro.ts
* pagesLogin.ts

Dentro de estas páginas iremos estrucutandola como una guía para hacer un elemento o una guía de este tipo se genera un elemento de este tipo y se consumirá dentro de la clase.

TODAS LAS PAGINAS DE POM (PAGE OBJECT MODEL) tienen que importar Page, Locator.



Luego le pondremos export class PaginaRegistro{} para que pueda acceder a esta clase los archivos o clases que necesitemos.

EJ: export class PaginaRegistro{

Readonly page: Page 🡪 inicializamos nuestra clase

Readonly nombreInput: Locator

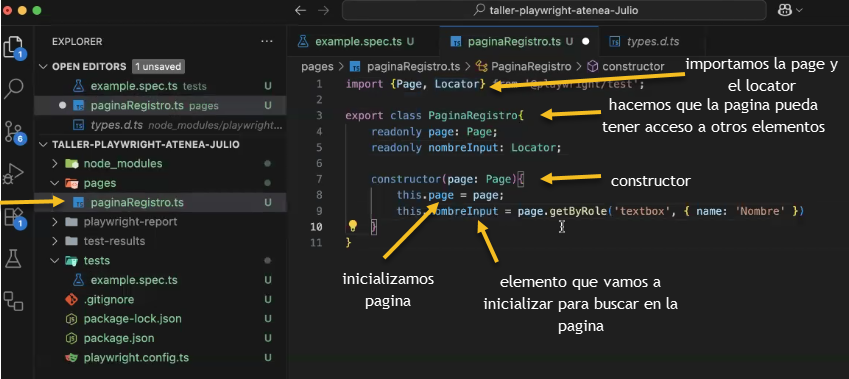
}

* Dentro del export tendremos el constructor que para lo que se utiliza para inicializar los elementos de la página.
  + Este recibe cual es el navegador que necesitara el elemento.
    - EJ: constructor (page: Page){} 🡪 variable que creamos page y la clase Page.
  + Dentro del constructor inicializaremos los elementos.
    - EJ: constructor (page: Page){

this.page = page

this.hombreInput = page.getByRole(‘textbox’, {name:’Nombre’})

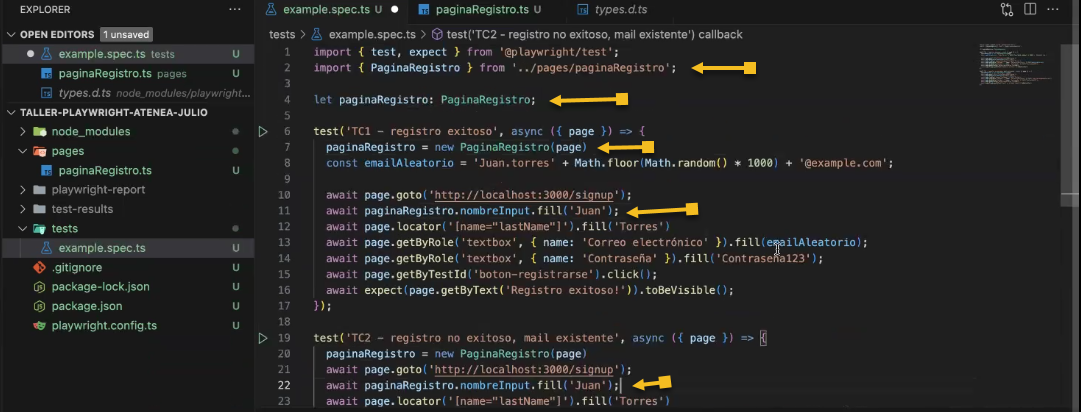
}



Ya hemos expuesto la clase para que sea importada en el caso de prueba.

Importaríamos la clase en el caso de prueba.

* Import { PaginaRegistro } from ‘../pages/paginaRegistro’; 🡪 importamos la clase PaginaRegistro
* Let paginaRegistro: PaginaRegistro; 🡪 objeto del tipo página registro.
* paginaRegistro = new PaginaRegistro(page); 🡪 Inicializar el objeto y le pasamos el objeto de la ventana que se genere del navegador que sería page.
* Await paginaRegistro.nombreInput.fill(`Juan`)



GIT:

Git status 🡪 nos muestra todos los archivos modificados.

Git checkout -b main 🡪 primera versión por eso el main

Git add . 🡪 añade todo

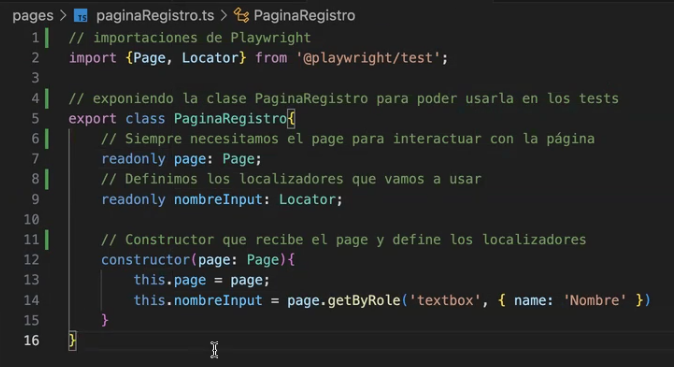
Git commit -m “Primer commit con escenarios de registro” 🡪 commit de los archivos añadidos

Git push origin main 🡪 subida al repo de git

# CLASE 2 – 30/07/2025

ReadOnly: se utiliza para usar el valor en esa pagina.

Explicación de cada parte del POM



## Escenarios que automatizar:

* TC3 – Redireccionamiento a login después de registro exitoso.
* TC4 – Login exitoso
* TC5 – Agregar cuenta de manera exitosa con dinero debito.

## Diferentes tipos de acciones de la clase:

Con esto lo que hacemos es reducir el texto de los test y la lógica agregarla en la page correspondiente.

### Ir a la página:

* Async visitarPaginaRegistro() {

Await this.page.goto(‘http://localhost:3000/signup’);

}

Rellenar el formulario:

* Async completarFormularioRegistro(nombre: string, apellido: string, email: string, contraseña: string) {

Await This.nombreImput.fill(‘Juan’);

Await This.apellidoImput.fill(‘Torres);

Await This.emailImput.fill(‘juan@gmail.com);

Await This.contraseñaImput.fill(‘Contraseña1234.);

}

* Async hacerClickBotonRegistro(){

This.botonRegistrarse.click();

}

* Async registrarUsuario(nombre: string, apellido: string, email: string, contraseña: string){

Await this.completarFormularioRegistro(nombre, apellido, email, contraseña);

Await this.hacerClickBotonRegistro();

}

### Llamada a las clases desde el test:

Await paginaRegistro.visitarPaginaRegistro();

Await paginaRegistro.completarFormularioRegistro(“Juan”, “Torres”, emailAleatorio, ”Contraseña1234.”);

Await paginaRegistro.botonRegistrarse()

//otra manera de hacerlo es con el ultimo async que hemos creado que hace el relleno de los campos y pulsa en el botón agregar solo ese método al test.

Await registrarUsuario(“Juan”, “Torres”, emailAleatorio, ”Contraseña1234.”);

### Métodos de espera:

Await this.page.waitForLoadState(‘estado’) 🡪

* domcontent loaded 🡪 Espera a que se carge todo
* load 🡪 Espera a que se carge todo
* Networkidle 🡪 no se recomienda

Await this.page.waitForUrl(‘https…’)

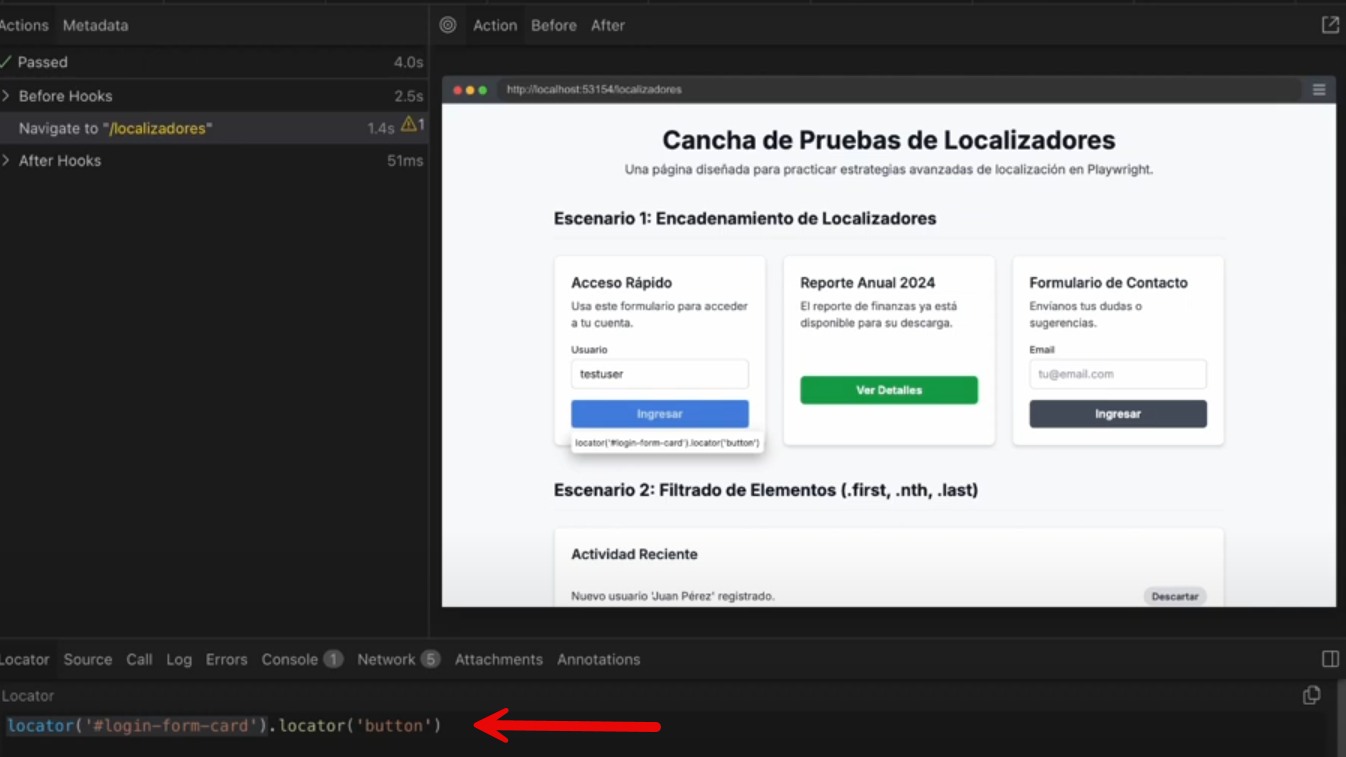
Video para coger tooltips de los formularios rellenos incorrectamente o vacíos.

<https://www.youtube.com/watch?v=kKQ_1mACv_E&t=808s>

### Encontrar elementos sin ids.

#### Encadenamiento de localizadores

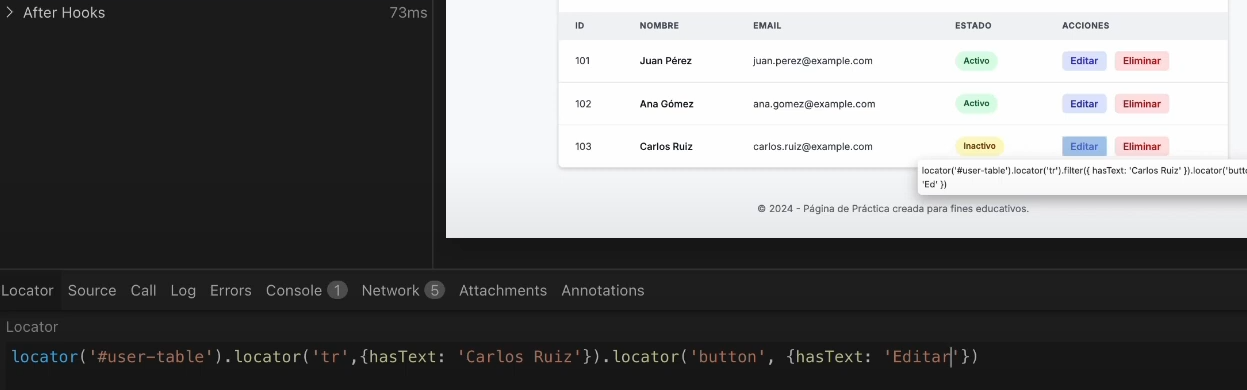
Como localizar ciertos elementos cuando se repiten, en este caso que se ve accedemos al elemento principal la caja que contiene el botón ingresar azul y luego localizamos el botón que queremos o input.

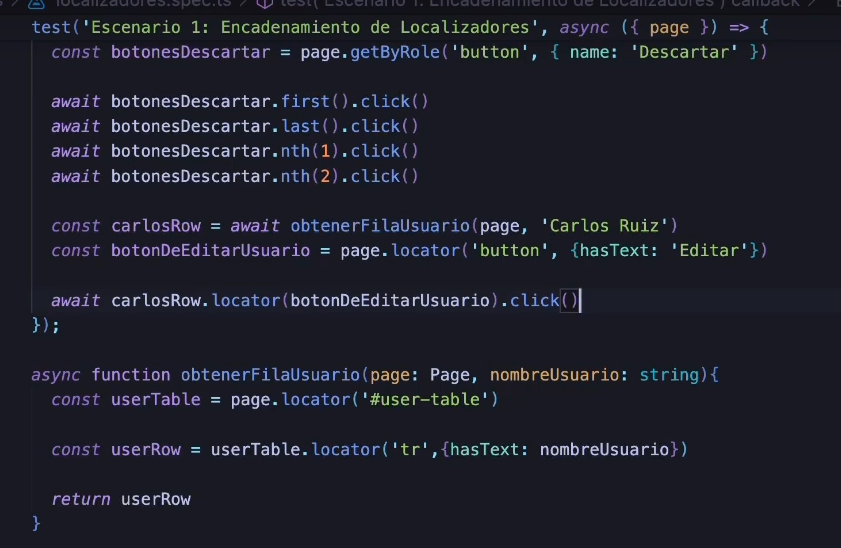


#### Filtrado de elementos con first, nth y last

#### 

#### Localizadores relativos (TABLAS)





# CLASE 3 – 06/08/2025

## HOOKS

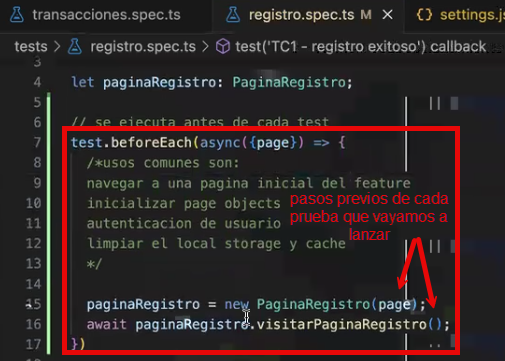
Hooks 🡪 nos ayudan a gestionar el ciclo de vida de nuestras pruebas, son funciones especiales que se ejecutan en momentos específicos del ciclo de vida de las pruebas, esto nos va a permitir configurar precondiciones limpiar recursos y mantener las pruebas organizadas y mas eficientes posibles.

* test.beforeEach 🡪 ejecuta algún bloque de código antes de alguna prueba individual, esto nos permite ahorrar código ya que estas funciones repetitivas de cada prueba solo se escriben una vez en código y se ejecutan en cada prueba.

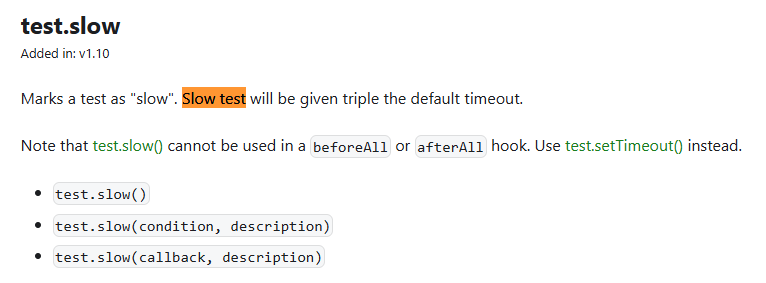
Esto se ejecuta una vez en cada prueba.

<https://playwright.dev/docs/api/class-test> 🡪 documentación.

* + Los usos comunes del beforeEach son:
    - Navegar a la página inicial del feature.
    - Inicializar page objects
    - Autenticación de usuario
    - Limpiar el localStorage y la cache.



Existe un método llamado test.slow() que ejecuta el test mas lento por si tuviéramos problemas a la hora de ejecutar alguna prueba porque playwright funciona muy rápido.



* test.afterEach 🡪 No se suele utilizar para pruebas.
* test.beforeAll
* Test.afterALL

## PRUEBAS BACKEND O API

Siempre vamos a guardar la request.

Por ejemplo, si queremos generar un usuario usaremos el endpoint signUp con respuesta POST.

Además, nos devolverá un body con los datos en el caso de atenea nos da un token de jwt y los datos del usuario.

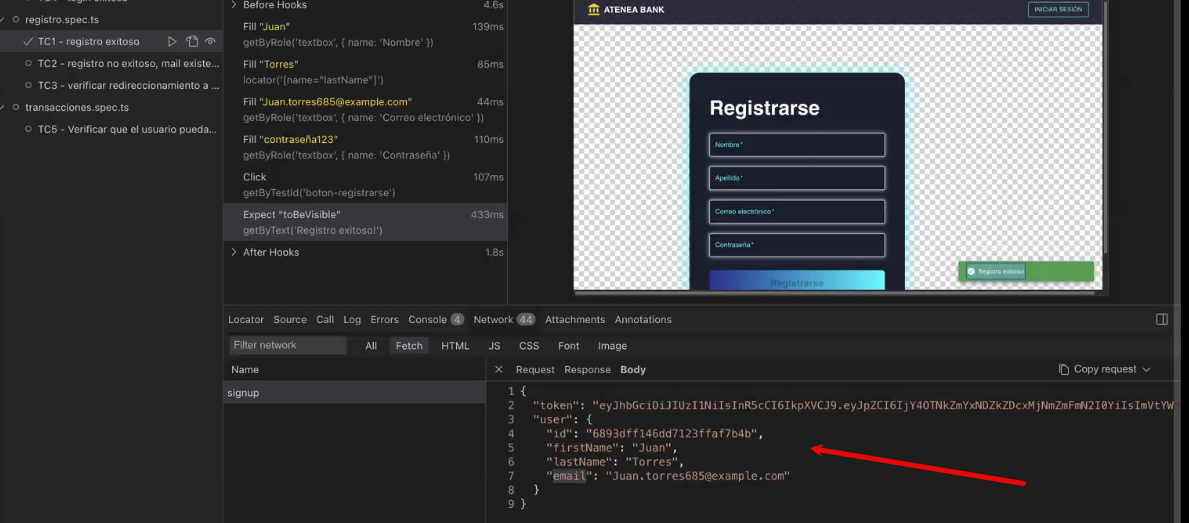
**¡¡IMPORTANTE!!**

La información de los headers a veces es importante para saber que request estamos enviando.

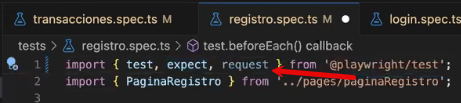
Algunos headers importantes:

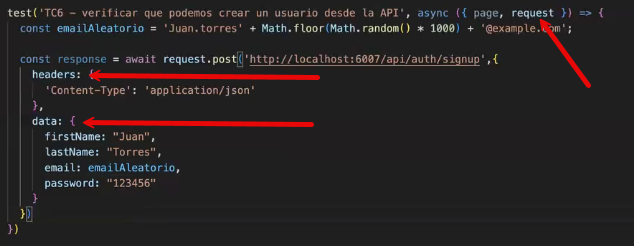
* + - * Content-type.
      * Accepts.

### Identificar endpoint en ejecución y hacer una llamada al endpoint.



En nuestro test importaríamos el request para poder usarlo en las pruebas.





### Guardar la respuesta y trabajar con ella

Los datos de la respuesta se almacenan llamando a response.json();

Propiedades que podemos verificar de la respuesta:

* Expect(response.status()).toBe(201); 🡪 que el estado de la respuesta sea un 201.
* expect(responseBody.toHaveProperty(‘token’); 🡪 que tenga la propiedad token.
* expect(typeof responseBody.token).toBe(‘string’); 🡪 que el valor del token sea un tipo de datos string.
* expect(typeof responseBody.user).toEqual(expect.objectContaining({

id: expect.any(String),

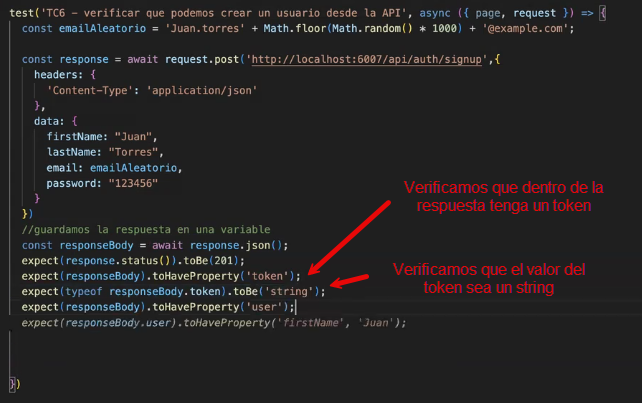
firstName: “Juan”,

lastName: “Torres”,

email: emailAleatorio,

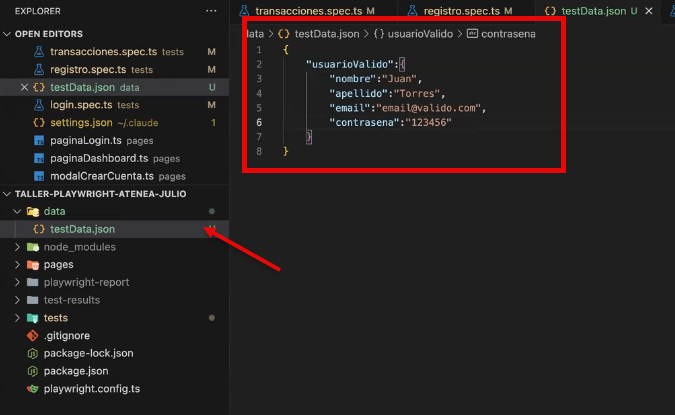
}))

**Esto verifica un objeto.**

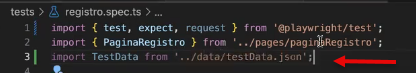


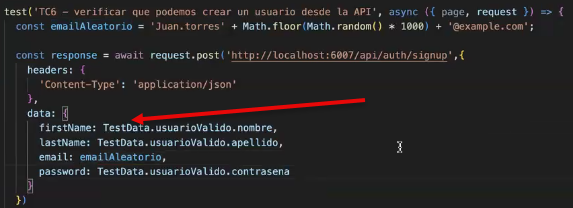


### Organizar nuestros datos de acceso en json



**Hacer referencia al archivo desde el caso de prueba**





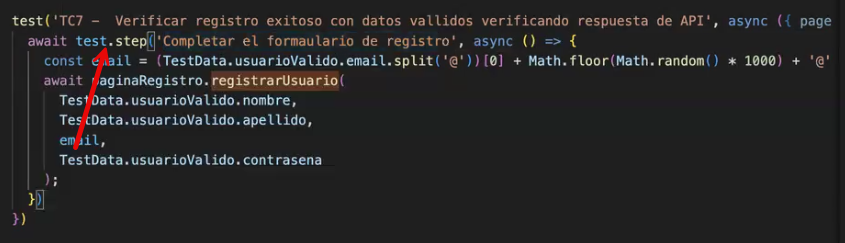
**Split 🡪 dividir nuestros strings de prueba en diferentes partes para ponerle cosas en el medio.**

**EJ: const email =** (TestData.usuarioValid.email.split(‘@’))[0] + Math.floor(Math.ramdom() \* 1000) + ‘@’ + TestData.usuarioValido.email.split(`@`)[1];

Una de las cosas que podemos utilizar en playWright para asemejar a cucumber y que el reporte se vea mas especifico cuando se lo mostremos a alguien es usar lo siguiente:

Await test.step(‘’, async () => {

})



# CLASE 4 – 13/08/2025

# CLASE 5 – 20/08/2025