Boilerplate

Los ejemplos que vamos a codificar los puedes probar en la sandbox de JavaScript, si no sabes cómo funciona, échale un vistazo al módulo de setup y si te quedan dudas contacta con tu mentor.

Unit testing

Introducción

Vamos a realizar una introducción básica a pruebas unitarias, a lo largo del curso volveremos sobre este tema e iremos ampliándolo.

Todavía no hemos cubierto ni arrays ni bucles, y ya estamos hablando de pruebas unitarias ¿Por qué? porque es algo fundamental para trabajar como un profesional, en el mundo laboral te encontrarás empresas *carniceras* que no aplican estos principios, la excusa suele ser la de *no tenemos tiempo* y después lo que suele pasar es que sacan versiones de productos que son defectuosos.

Cuando empezamos a hacer limpia de código en variables, vimos que generábamos un montón de funciones independientes, es más en el módulo de *imports* separamos en varios ficheros dichas funciones, pero... *¿Y si queremos probar que esto arroja las salidas que se esperan?* El tema de levantar la aplicación e ir a probando de forma manual si va o no lo que hemos picado se hace complicado conforme la aplicación crece, ya que para ver si una función no tiene fallos hay que meter mucho ruido (te puedes haber equivocado al hacer un *getElementById* en otra función que no tiene nada que ver pero que está entre medias, y mil cosas así, esto puede acabar en un infierno de *console.logs*).

Pero pero... si el objetivo de esto es hacer las cosas más fáciles ¿Qué pasa? Pues que nos falta una pieza del puzzle, las funciones que hemos creado tienen una entrada y una salida bien definida.

```
Entrada (parámetros)
   |  |  |
   V  V  V
Código de la función
   |  |  |
   V  V  V
 Salida (return)
```

¿Y si pudiéramos tener *pequeños programas* que prueban que cada función se comporta como esperamos?: esto es lo que se conoce como *unit testing*.

- Cada *mini-programa* representa un caso de prueba (por ejemplo en el caso de la función de acertar un número, podríamos tener: caso de número menor, caso de número mayor, caso de aciertas número y caso de metes un texto).
- Esos *mini-programas* los escribes una vez pero después lo puedes ejecutar tantas veces como quieras (sí, no tienes que ir manualmente viendo que algo está bien, es un proceso automático el que lo puede

ejecutar por ti).

De esta manera puedes asegurarte de que una función se está comportando como esperas, y si introduces algún cambio en la misma (refactorizas) y rompes algo, la batería de tests que has creado, te avisa (decimos que un test *sale en verde* cuando da el resultado esperado, y en *rojo* cuando falla).

Esto es lo que llamamos dar un paso más en la industrialización de la generación de nuestro código.

¿Quiere decir esto que tengo que meterle pruebas unitarias a todos los ficheros? Aquí hay mucha discusión al respecto, lo normal es usar el sentido común, por ejemplo todo lo que son funciones de lógica de negocio (en los ejemplos que hemos hecho hasta ahora podrían ser: dame un carta, chequea si el número es correcto, etc...), sí deberían de llevar pruebas unitarias, en la parte de interfaz de usuario aquí depende, hay veces que hay componentes importantes o que se pueden usar en otras partes de la aplicación u otras aplicaciones que sí merece la pena probarlos, hay otros que se consideran más de uso de la aplicación y que pueden cambiar con facilidad, aquí te puedes plantear si implementar pruebas unitarias, o si se pueden probar utilizando otras aproximaciones (por ejemplo end to end testing).

Unit Testing básico

Dejémonos de teoría y pasemos a la práctica, en este caso la sandbox que vas a usar de JavaScript tiene preconfigurado un framework de prueba de JavaScript llamado Vitest (existen diversas alternativas pero lo que son los conceptos que vamos a manejar son similares).

Vamos a tomar como punto de partida el ejemplo de *adivina un número* del módulo de *imports* (el que lo dividimos en *modelo*, *motor*, *ui* ...), para que no tengas que volver a escribir el código, en el repositorio bootcamp-js-2 en la ruta *07-testing/00-js-advina* tienes el código de inicio.

Clonate el repo y copíate ese proyecto a una carpeta nueva (en caso de que no sepas como funciona esto, contacta con tu mentor y el te ayudará), ejecutamos un *install*

```
npm install
```

Vamos a analizar una función de motor y ver cómo podemos probarla, en concreto *comprobarNumero*

NO COPIAR Y PEGAR ESTE CODIGO

```
export const comprobarNumero = (texto) => {
  const numero = parseInt(texto);
  const esUnNumero = !isNaN(numero);

  if (!esUnNumero) {
    return NO_ES_UN_NUMERO;
  }

  if (numero === numeroParaAcertar) {
    return ES_EL_NUMERO_SECRETO;
  }

  if (hasSuperadoElNumeroMaximoDeIntentos()) {
```

```
    return GAME_OVER_MAXIMO_INTENTOS;
  }

  return numero > numeroParaAcertar ? EL_NUMERO_ES_MAYOR : EL_NUMERO_ES_MENOR;
};
```

Vamos a añadir una batería de pruebas para ese fichero, aquí lo que hacemos es crear un fichero que se llame igual que *motor.js* pero le metemos la extensión *.spec.js* (lo normal es que los tests le pongamos como extensión *.spec.js* o *.test.js*, y que queden como ficheros *hermanos* del que vamos a probar).

El siguiente paso es crear un grupo para los casos que vamos a testear.

./src/motor.spec.js

```
import { comprobarNumero } from "./motor";

describe("comprobarNumero", () => {});
```

Acabamos de crear una agrupación para probar los casos de *comprobarNumero*,

Nos ponemos manos a la obra a probar que el primer caso funciona correctamente: si meto un texto me tiene que devolver *NO_ES_UN_NUMERO*, para ello vamos a usar la función *_it* qué es la que nos permite crear un caso de prueba, aquí haremos tres pasos (la triple A se llama):

- **Arrange:** preparamos el escenario (qué valores iniciales tenemos que usar etc...)
- **Act:** ejecutamos la función que queremos probar.
- **Assert**: comprobamos que el resultado es el esperado.

```
import { comprobarNumero } from "./motor";
import { NO_ES_UN_NUMERO } from "./modelo";

describe("comprobarNumero", () => {
  it("Debería de devolver NO_ES_UN_NUMERO cuando texto no es un número", () => {
    // Arrange
    const texto = "esto no es número";

    // Act
    const resultado = comprobarNumero(texto);

    // Assert
    expect(resultado).toBe(NO_ES_UN_NUMERO);
  });
});
```

¿Qué estamos haciendo aquí?

- Definimos un caso dentro del *describe* (lo normal es que definamos varios dentro de esa agrupación).
- En el *it* le pasamos:

    - Cómo primer parámetro un texto que describe el caso que estamos probando (Es importante gastar tiempo en describir bien los casos de prueba, porque si no luego no sabes qué es lo que estás probando).
    - Cómo segundo parámetro una función que es la que va a contener el código de la prueba.
- Nos ponemos manos a la obra con la prueba:
    - En la sección de *Arrange*, preparamos el escenario, en este caso le pasamos un texto que no es un número.
    - En la sección de *Act* ejecutamos la función que queremos probar, en este caso *comprobarNumero* pasándole el texto que inicializamos en *Act*.
    - Y ahora en la sección de *Assert*, donde comprobamos si el resultado es el esperado, aquí con *expect* le decimos que esperamos que *resultado* tenga un valor concreto, ¿Ese valor concreto qué es? Lo que marcamos dentro de *toBe* en este caso estamos esperando que *resultado* sea igual a *NO_ES_UN_NUMERO*.

Si ejecutamos el test (línea de comandos):

```
npm run test
```

Verás que sale en verde.

```
 Test Files  1 passed (1)
      Tests  1 passed (1)
   Start at  08:07:29
   Duration  362ms (transform 32ms, setup 0ms, collect 22ms, tests 3ms)
```

Es decir obtenemos el resultado esperado, ojo, a veces probamos escenarios esperados que son fallos, es decir un test que sale en verde es que simplemente genera el resultado que esperamos (por ejemplo podemos probar que si no hay conexión a internet se lanza un error y eso es lo esperado).

Si quieres prueba y pon en el *toBe* otra constante, verás que sale el test en rojo, es decir no pasaría (en este caso nos habríamos equivocado al desarrollar el test).

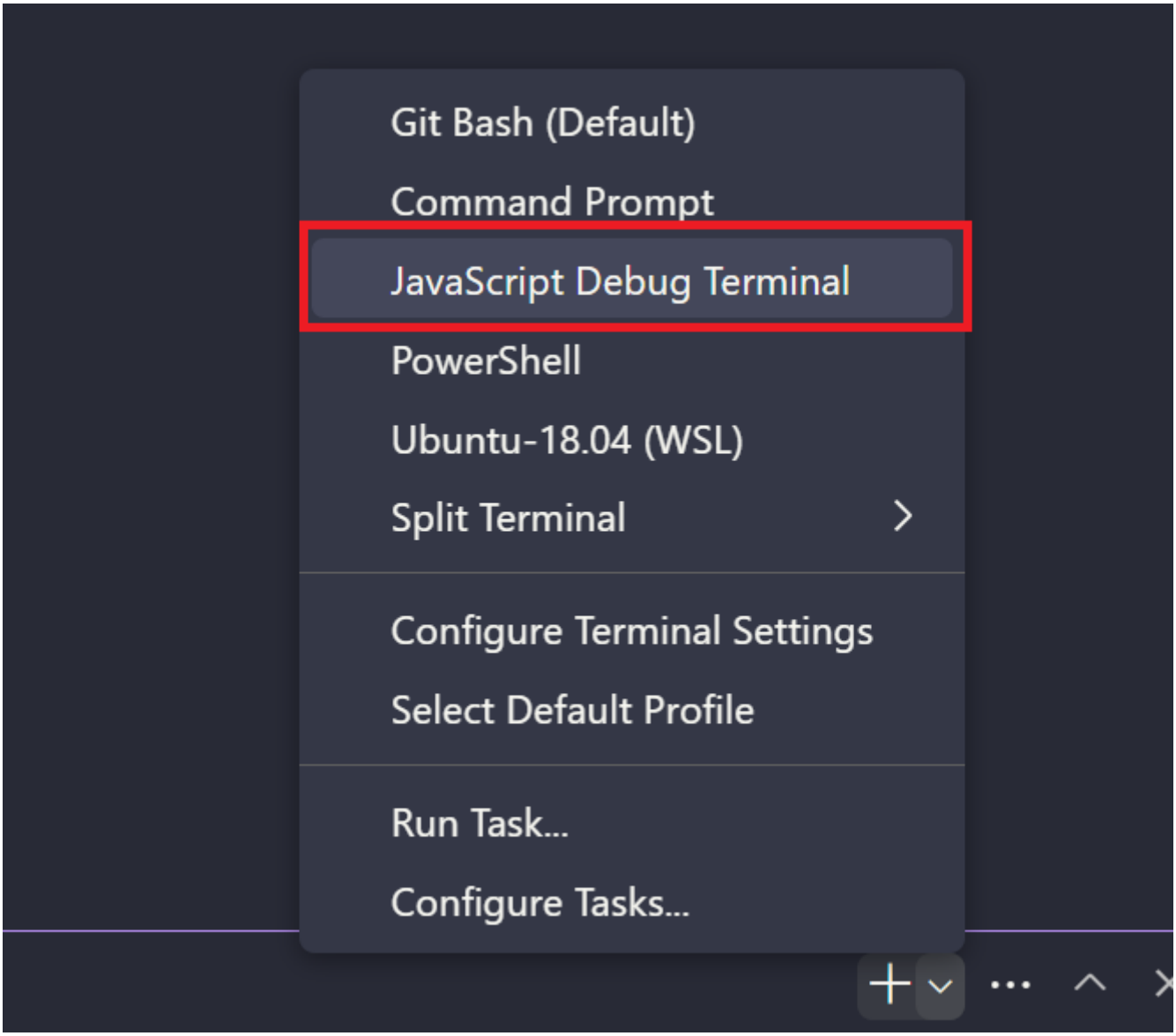
Si te fijas cuando has ejecutado *npm run test* se queda el terminal pillado esperando cambios y si introduces cambios en el código se vuelven a lanzar los test, esto es porque estamos usando *vitest* como motor de testing y tiene un modo *watch* automático, en otros motores de testing esto no funciona así por defecto y tienes que configurarlo (te puedes encontrar con un comando que se llame *npm run test:watch*).

Depurando un test

Una duda que te puede surgir ahora es *Oye y si no me funciona un test y no tengo ni idea de por qué ¿Cómo puedo depurar?* Es muy buena pregunta, aquí no tenemos las herramientas de desarrollo del navegador, peeeeroooo contamos con algo muy bueno también ¡¡El Gran Visual Studio Code!!

Si en este momento estás ejecutando los tests en el terminal tira de teclado y pulsa CTRL+C, o si eres más de tirar de ratón, pincha en el icono de la papelera que hay a la derecha del terminal.

Vamos ahora a abrir un terminal en modo debug, para ello pinchamos en el icono + del terminal y elegimos la opción *JavaScript Debug Terminal*.



Ahora podemos poner break points en el código, vamos a poner uno en el test (también podríamos ponerlos en el código principal).

```
19  export const comprobarNumero = (texto) => {
20    const numero = parseInt(texto);
21    const esUnNumero = !isNaN(numero);
22
23    if (!esUnNumero) {
24      return NO_ES_UN_NUMERO;
25    }
26
```

Y desde el terminal de debug ejecutamos el comando de *npm run test*

```
npm run test
```

Y... ¡¡Toma ya!! Se nos para en la línea de código y podemos depurar, si te fijas es parecido a la herramienta de debugging del navegador, tienes tus botones de continuar, saltar a la siguiente línea, etc... Y en la parte izquierda tienes también tu área de watch.

Bueno apenas llevamos unos módulos y ya estás viendo lo que va a ser tu día a día como desarrollador... testear, depurar, testear, depurar 😊.

Unit Testing mocking

En este momento nos podemos venir arriba y pensar... ¡ea a por el siguiente caso! pero nos vamos a pegar un morrazo 😊, si queremos probar el caso ES_EL_NUMERO_SECRETO

NO COPIAR Y PEGAR ESTE CODIGO

```
export const comprobarNumero = (texto) => {
  const numero = parseInt(texto);
  const esUnNumero = !isNaN(numero);

  if (!esUnNumero) {
    return NO_ES_UN_NUMERO;
  }

  if (numero === numeroParaAcertar) {
    return ES_EL_NUMERO_SECRETO;
  }
```

Tendríamos que compararlo contra el valor *0* que es el valor inicial que se le asigna a *numeroParaAcertar*, pero esto huele mal:

- Primero puede que el valor inicial cambie y rompamos los test.
- Después... oye igual quiero probar con otro valor, y no puedo (por narices está el cero).

¿Qué podemos hacer? Una opción sería usar *setNumeroParaAcertar* para cambiar el valor:

```
- import { NO_ES_UN_NUMERO } from "./modelo";
+ import { NO_ES_UN_NUMERO, ES_EL_NUMERO_SECRETO, setNumeroParaAcertar } from
"./modelo";
import { comprobarNumero } from "./motor";

describe("comprobarNumero", () => {
  it("Debería de devolver NO_ES_UN_NUMERO cuando texto no es un número", () => {
    // Arrange
    const texto = "texto";

    // Act
    const resultado = comprobarNumero(texto);

    // Assert
    expect(resultado).toBe(NO_ES_UN_NUMERO);
  });
```

```
+  it("Debería de devolver ES_EL_NUMERO_SECRETO cuando texto es el número a
acertar", () => {
+    // Arrange
+    const numeroParaAcertar = 23;
+    const texto = "23";
+
+    // Act
+    setNumeroParaAcertar(numeroParaAcertar);
+    const resultado = comprobarNumero(texto);
+
+    // Assert
+    expect(resultado).toBe(ES_EL_NUMERO_SECRETO);
+  });
});
```

Esto funciona, sale verde, pero no es una solución muy purista, ya que queremos probar de forma unitaria el método *comprobarNumero* y nos hace falta invocar a otra función: *setNumeroParaAcertar ¿Y si esta función tuviera un fallo?* Upa... hacemos aguas 😉.

En breve te mostramos como implementar esto de una manera más limpia, aunque también con más complejidad

De momento... TE TOCA, siguiendo la misma aproximación implementa el caso de EL_NUMERO_ES_MAYOR:

- Vamos por el primer caso, EL_NUMERO_ES_MAYOR

Para este caso vamos a añadir un nuevo test:

```
- import { NO_ES_UN_NUMERO, ES_EL_NUMERO_SECRETO, setNumeroParaAcertar } from
"./modelo";
+ import { NO_ES_UN_NUMERO, ES_EL_NUMERO_SECRETO, EL_NUMERO_ES_MAYOR,
setNumeroParaAcertar } from "./modelo";

describe("comprobarNumero", () => {

(...)

+  it("Debería de devolver EL_NUMERO_ES_MAYOR cuando texto es mayor que el número
a acertar", () => {
+      // Arrange
+      const numeroParaAcertar = 23;
+      const texto = "24";

+      // Act
+      setNumeroParaAcertar(numeroParaAcertar);
+      const resultado = comprobarNumero(texto);

+      // Assert
+      expect(resultado).toBe(EL_NUMERO_ES_MAYOR);
```

```
+    });
});
```

Ejecutamos los test y vemos que pasan en verde.

```
npm run test
```

- Ahora el siguiente caso, EL_NUMERO_ES_MENOR

Animate y TE TOCA: implementa el caso EL_NUMERO_ES_MENOR, sino has sido capaz no te desanimes, aquí tienes la solución:

```
- import { NO_ES_UN_NUMERO, ES_EL_NUMERO_SECRETO, EL_NUMERO_ES_MAYOR,
setNumeroParaAcertar } from "./modelo";
+ import { NO_ES_UN_NUMERO, ES_EL_NUMERO_SECRETO, EL_NUMERO_ES_MAYOR,
EL_NUMERO_ES_MENOR, setNumeroParaAcertar } from "./modelo";

describe("comprobarNumero", () => {

(...)

+it("Debería de devolver EL_NUMERO_ES_MENOR cuando texto es menor que el número a
acertar", () => {
+    // Arrange
+    const numeroParaAcertar = 23;
+    const texto = "22";

+    // Act
+    setNumeroParaAcertar(numeroParaAcertar);
+    const resultado = comprobarNumero(texto);

+    // Assert
+    expect(resultado).toBe(EL_NUMERO_ES_MENOR);
+  });
});
```

Nos falta un caso más, pero antes de implementarlo vamos a ver cómo podemos hacer para que nuestro código sea más limpio y no tengamos que invocar *setNumeroParaAcertar*. Ojo esto que vamos a explicar es más avanzado y lo revisitaremos en módulos siguientes, pero para que tengas una idea:

- Cuando importamos un fichero (lo podemos llamar módulo), nos traemos una serie de cosas: funciones, variables, constantes...
- Puedes ser que te interese fijar ciertos valores para poder probar tu código simulando exactamente ciertos escenarios.

En este caso nuestro objetivo es importarnos el módulo de *modelo* y podemos fijar el valor de *numeroParaAcertar* a el valor 23 para poder probar varios casos (mayor que, menor que...), así no nos

quedamos con el caso por defecto que es 0 y que podría cambiar en un futuro.

¿Cómo lo hacemos?

La forma de solucionarlo es haciendo uso de la función *spyOn* que nos proporciona *vitest*, pero podemos preguntarnos, ¿eso para qué sirve?

Con *spyOn* lo que conseguimos es trampear sobre el comportamiento de la variable *numeroDeIntentos* y que nos devuelva el valor que nosotros queramos, de manera informal podemos pensar que *spyOn* es el cliché de la _vieja del visillo_ que en el fondo sabe todo lo que pasa en la casa de al lado, incluso podría impersonarnos, es decir:

- *spyOn* se coloca entre medias de la función o variable que queramos *espiar*, y controla aspectos tales como saber cuantas veces se ha llamado una función (si, la tiene bien controlada...).
- También podemos dar un paso más allá, y es detectar cuando va a llamar la función real y ponerse en medio y devolver el valor que queramos (como si interceptara una llamada de teléfono y se hiciera pasar por nosotros ¿Habéis visto que vieja del visillo tan ciberpunk tenemos? 😉).

¿Y tanto rollo para qué? Cuando implementamos pruebas unitarias, lo normal e que queremos probar el código de la función en concreto, no el resto de funciones o variables involucradas, es decir cortocircuitamos el resto de funciones y probamos las combinaciones de casos que queramos *¿Y que pasa con esas otras funciones?* Que también les implementaremos pruebas unitarias cuando toque para ver que funcionan bien per se.

> ¿Quieres decir que es malo probar dos funciones juntas? Digamos que no es una pruebga unitaria, también existen pruebas de ensamblaje en la que probamos varias funciones / componentes juntos.

Para poder usar *spyOn* y sustituir valores en *vitest* nos va hacer falta que importemos *modelo* de una manera especial:

```
import { comprobarNumero } from "./motor";
import {
  NO_ES_UN_NUMERO,
  ES_EL_NUMERO_SECRETO,
  setNumeroParaAcertar,
  EL_NUMERO_ES_MAYOR,
  EL_NUMERO_ES_MENOR,
} from "./modelo";
+ import * as modelo from "./modelo";
(...)
```

Y pensamos, ¿pero qué es esto?, si yo solo quiero importar la variable *numeroParaAcertar* y me está haciendo un import raro.

De esta forma estamos importando todo el contenido exportado de un módulo o archivo llamado *modelo* y lo almacena en un objeto con el nombre "modelo". Esto significa que podemos acceder a todas las variables, funciones y clases exportadas desde el módulo *modelo*.

Por ejemplo, si en el módulo *modelo* se exporta una variable llamada *numeroParaAcertar*, podemos acceder a ella desde el archivo que realiza la importación de la siguiente manera:

NO COPIAR ESTE CODIGO

```
import * as modelo from "./modelo";

console.log(modelo.numeroDeIntentos);
```

Y ya en el test directamente... quitamos el *setNumeroParaAcertar* y añadimos el *spyOn*. De esta forma conseguimos que el valor de *numeroParaAcertar* sea el que nosotros queramos, fijate que :

- Con *spyOn* activamos el modo *espia* (*vieja del visillo*) sobre *numeroPAraAcertar*.
- Con *mockReturnValue* activamos el modo *vieja del visillo cyberpunk* y le decimos que cuando se invoce a *numeroParaAcertar* se meta en medio y devuelva un *23*.

```
  it("Debería de devolver ES_EL_NUMERO_SECRETO cuando texto es el número a
acertar", () => {
    // Arrange
-   const numeroParaAcertar = 23;
    const texto = "23";
+   vi.spyOn(modelo, "numeroParaAcertar", "get").mockReturnValue(23);
    // Act
-   setNumeroParaAcertar(numeroParaAcertar);
    const resultado = comprobarNumero(texto);

    // Assert
    expect(resultado).toBe(ES_EL_NUMERO_SECRETO);
  });
```

Como este tema es importante, vamos a recapitular, lo que hacemos es llamar a la función spyOn, que recibe tres parámetros:

- El objeto que queremos espiar, en nuestro caso el objeto "modelo".
- El nombre de la propiedad que queremos espiar, "numeroParaAcertar".
- El tipo de propiedad que queremos espiar, usamos "get" porque queremos espiar la propiedad que se obtiene.
- Y por último, el valor que queremos que devuelva, en este caso 23 qué es el número que tenemos que acertar.

Modificamos también del resto de tests, y añadimos el *spyOn*.

```
it("Debería de devolver EL_NUMERO_ES_MAYOR cuando texto es mayor que el número a
acertar", () => {
    // Arrange
-   const numeroParaAcertar = 23;
    const texto = "24";
```

```
+    vi.spyOn(modelo, "numeroParaAcertar", "get").mockReturnValue(23);

     // Act
-     setNumeroParaAcertar(numeroParaAcertar);
     const resultado = comprobarNumero(texto);

     // Assert
     expect(resultado).toBe(EL_NUMERO_ES_MAYOR);
   });

   it("Debería de devolver EL_NUMERO_ES_MENOR cuando texto es menor que el número a
acertar", () => {
     // Arrange
-    const numeroParaAcertar = 23;
     const texto = "22";
+    vi.spyOn(modelo, "numeroParaAcertar", "get").mockReturnValue(23);

     // Act
-     setNumeroParaAcertar(numeroParaAcertar);
     const resultado = comprobarNumero(texto);

     // Assert
     expect(resultado).toBe(EL_NUMERO_ES_MENOR);
   });
```

Ahora vamos a ver el caso que nos quedaba pendiente, vamos a implementar lo mismo para el caso en que el que hemos agotado los intentos y nos devuelve *GAME_OVER_MAXIMO_INTENTOS*.

Como pista, tiene que hacer mock de *numeroDeIntentos*

Ahora vamos crear un nuevo test, añadir un *spyOn* dentro del test y vamos a simular que se ha superado el número máximo de intentos, también podrías confiar en que el número de intentos máximo no va a cambiar y ver el inicial.

```
- import { NO_ES_UN_NUMERO, ES_EL_NUMERO_SECRETO, EL_NUMERO_ES_MAYOR,
EL_NUMERO_ES_MENOR,setNumeroParaAcertar} from "./modelo";
+ import { NO_ES_UN_NUMERO, ES_EL_NUMERO_SECRETO, EL_NUMERO_ES_MAYOR,
EL_NUMERO_ES_MENOR, GAME_OVER_MAXIMO_INTENTOS } from "./modelo";

describe("comprobarNumero", () => {
( ... )
+ it("Debería de devolver GAME_OVER_MAXIMO_INTENTOS cuando se ha superado el
número máximo de intentos", async () => {
+    // Arrange
+    const texto = "70";
+    vi.spyOn(modelo, "numeroParaAcertar", "get").mockReturnValue(23);
+    vi.spyOn(modelo, "numeroDeIntentos", "get").mockReturnValue(5);
+
+    // Act
+    const resultado = comprobarNumero(texto);
```

```
+    // Assert
+    expect(resultado).toBe(GAME_OVER_MAXIMO_INTENTOS);
+ });
});
```

Y si ahora ejecutamos los test

```
npm run test
```

¡Tatachán!, funcionan todos los test 😊.

```
 DEV  v0.29.8 C:/trabajo/javascript-sandbox

 ✓ src/dummy.spec.ts (1)
 ✓ src/motor.spec.js (5)

 Test Files  2 passed (2)
      Tests  6 passed (6)
   Start at  22:24:51
   Duration  416ms (transform 82ms, setup 0ms, collect 80ms, tests 8ms)


 PASS  Waiting for file changes...
       press h to show help, press q to quit
```

Acabas de ver que en los tests a veces hay algo de código repetitivo, vamos a optimizarlo un poco, cómo vemos estamos usando el mismo valor de *numeroParaAcertar* en todos los tests, también cabe la posiblidad de que en un futuro queramos cambiar ese valor, en ese caso tendríamos que cambiarlo en todos los tests.

Pero en este caso, en concreto, hemos usado el mismo valor en todos los tests, por lo que podríamos hacer una pequeña refactorización. Y ahora es cuando entra en escena el *beforeAll*.

Y podemos preguntarnos, ¿qué es *beforeAll*? Pues es una función que se ejecuta antes de todos los tests, y que nos permite hacer una configuración inicial. A esta función vamos a añadir el *spyOn* para *numeroParaAcertar* y así no tendremos que repetirlo en todos los tests.

Para esto nos hace falta importar *beforeAll* de *vitest*, y lo usamos de la siguiente forma:

```
+ import { beforeEach } from "vitest";
(...)

describe("comprobarNumero", () => {
+  beforeEach(() => {
+    vi.spyOn(modelo, "numeroParaAcertar", "get").mockReturnValue(
+      23
+    );
+  });
```

Ahora solo nos quedaría refactorizar nuestros test.

```
describe("comprobarNumero", () => {
  (...)

  it("Debería de devolver ES_EL_NUMERO_SECRETO cuando texto es el número a
acertar", () => {
    // Arrange
-   const numeroParaAcertar = 23;
    const texto = "23";
-   vi.spyOn(modelo, "numeroParaAcertar", "get").mockReturnValue(23);

    // Act
    const resultado = comprobarNumero(texto);

    // Assert
    expect(resultado).toBe(ES_EL_NUMERO_SECRETO);
  });

  it("Debería de devolver EL_NUMERO_ES_MAYOR cuando texto es mayor que el número a
acertar", () => {
    // Arrange
-   const numeroParaAcertar = 23;
    const texto = "24";
-   vi.spyOn(modelo, "numeroParaAcertar", "get").mockReturnValue(23);

    // Act
    const resultado = comprobarNumero(texto);

    // Assert
    expect(resultado).toBe(EL_NUMERO_ES_MAYOR);
  });

  it("Debería de devolver EL_NUMERO_ES_MENOR cuando texto es menor que el número a
acertar", () => {
    // Arrange
-   const numeroParaAcertar = 23;
    const texto = "22";
-   vi.spyOn(modelo, "numeroParaAcertar", "get").mockReturnValue(23);

    // Act
    const resultado = comprobarNumero(texto);

    // Assert
    expect(resultado).toBe(EL_NUMERO_ES_MENOR);
  });

  it("Debería de devolver GAME_OVER_MAXIMO_INTENTOS cuando se ha superado el
número máximo de intentos", async () => {
    // Arrange
-   const numeroParaAcertar = 23;
```

```
    const texto = "5";
-   vi.spyOn(modelo, "numeroParaAcertar", "get").mockReturnValue(23);
    vi.spyOn(modelo, "numeroDeIntentos", "get").mockReturnValue(5);

    // Act
    const resultado = comprobarNumero(texto);

    // Assert
    expect(resultado).toBe(GAME_OVER_MAXIMO_INTENTOS);
  });
});
```

Y si ahora ejecutamos los test

```
npm run test
```

Siguen funcionando todos 😊.

Os podéis preguntar, ¿por qué hemos mantenido el *spyOn* para *numeroDeIntentos*? Pues porque en este caso no queremos que el valor sea el mismo en todos los tests.

Volvamos ahora a un caso de mocking más controlable 😊, uno muy interesante es la función que hemos creado para generar números aleatorios entre 1 y 100 ¿Podrías afirmar con toda seguridad que lo crea entre ese rango de valores?

Se supone que *Math.random* genera valores aleatorios entre 0 y 1 (el uno no está incluido), lo podemos comprobar en la documentación oficial de MDN sobre math.random.

Y tenemos que *Math.floor* nos devuelve el entero más pequeño que es mayor o igual que el número que le pasamos, de nuevo podemos comprobarlo en la documentación oficial de MDN sobre math.floor.

Nosotros ese resultado lo multiplicamos por 101.

¿Y si pudiéramos hacer mocking de *Math.random* y meter trucados varios valores entre 0 y 1 para comprobar que nos da los datos esperados? Sobre todo están los casos frontera (también se llaman casos arista) ¿Qué pasa si le paso un 0, y un 0.99999?).

```
- import { comprobarNumero } from "./motor";
+ import { comprobarNumero, generarNumeroAleatorio } from "./motor";
( ... )
+
+ describe("generarNumeroAleatorio", () => {
+  it("MathRandom lo forzamos a que devuelva 0, debería de devolver 0", () => {
+    // Arrange
+    const numeroEsperado = 0;
+
+    vi.spyOn(global.Math, 'random').mockReturnValue(0);
+
+    // Act
```

```
+    const resultado = generarNumeroAleatorio();
+
+    // Assert
+    expect(resultado).toBe(numeroEsperado);
+  });
+ });
```

Te toca... ¿Vemos a ver si se consigue generar el número 100?

Pistas:

- Copia el caso anterior.
- Ponlo dentro del describe.
- Cambia la descripción del caso, eso es muy importante.
- Cambia el valor mockeado por 0.999
- Comprueba a ver si se genera un 100.

```
  });

+  it("MathRandom lo forzamos a que devuelva 0.9999, debería de devolver 100", ()
=> {
+    // Arrange
+    const numeroEsperado = 100;
+
+    vi.spyOn(global.Math, 'random').mockReturnValue(0.999);
+
+    // Act
+    const resultado = generarNumeroAleatorio();
+
+    // Assert
+    expect(resultado).toBe(numeroEsperado);
+  });
+ });
```

Te toca... mira a ver un par de valores en medio a ver si lo genera, ¿Cómo generarías un 50? ¿Y un 37? ... ¿Quiere esto decir que tengo que hacer un test por cada número? La respuesta es no, lo normal es probar casos comunes y ver que el comportamiento es el esperado, y luego casos extremos/arista/frontera, suele pasar que para esos valores se nos puede escapar algo.

Unit Testing DOM

Otro tema importante para testear son las actualizaciones en el DOM (tu HTML que se muestra en el navegador), ¿De verdad estamos eligiendo el ID correcto? ¿Se está mostrando la info que toca?

Para utilizar esto hay que conocer conceptos de asincronía y promesas, así que esto lo vamos a dejar aparcado de momento y volveremos a ello cuando hayamos cubierto esos tópicos.

TDD

Normalmente cuando te pones a escribir código, si dices que *vas a hacer los tests luego* casi nunca lo haces, porque (pon aquí la excusa que te de la gana): *no hay tiempo* o *para que voy a hacerlo si ya lo he probado manualmente*, hay escenarios en que esto es muy mala idea ya que:

- Si un día quieres refactorizar código te vas a encontrar con que no tienes tests y no sabes si los cambios que has introducido han podido romper algo (esto pasa a menudo).
- Si lo has probado manualmente, *¿Qué pasa si un día algo no funciona?* A probarlo todo a manubrio de nuevo... esto a poco que el proyecto tenga un poco de complejidad implica un coste y gasto de tiempo considerable.

Aquí es donde entra en juego la metodología *TDD* (*Test Driven Development*), *¿En qué consiste?* En que los tests vayan guiando tu código, *¿Y esto cómo funciona?* Supongamos una función que acepta unos parámetros y devuelve un resultado, que haríamos:

- Empiezo por crear la función vacía.
- Escribo un test para un caso concreto.
- Obviamente ese caso falla porque la función está vacía.
- Implemento lo mínimo para que pase el caso y salga verde.
- Implemento el siguiente caso de test... me volverá a fallar porque no he tenido en cuenta ese caso en la función.
- Vuelvo a la función, la refactorizo y la hago más genérica para que pase el siguiente caso de test.
- ¿Qué puede pasar aquí?
    - Qué igual el nuevo test pasa, pero el anterior rompe, entonces vuelvo y arreglo el código para que pasen todos los tests.
    - Que pasen los dos tests, y ahora toca ir a implementar el siguiente caso...
- Implemento un nuevo test... se pone en rojo y vuelta al código...
- Así en bucle hasta que tengo una batería de tests que me convencen y mientras sin haberme dado cuenta he implementado la función que tenía que hacer 😊.

¿Qué es lo bueno de esto?

- Que si la función no me convence la puedo refactorizar sin miedo a romper nada, ya que tengo una buena lista de tests automáticos que prueban mi aplicación .

- Que si mi programa no funciona por algún motivo, puedo descartar que sea esta función, ya que tengo una batería de tests que me aseguran que todo va bien (si por lo que sea se me escapó algún caso y falla mi función, añado un nuevo test para el caso adicional, así para la próxima lo tengo cubierto).

¿Quiere esto decir que a partir de ahora lo tengo que implementar todo siguiendo TDD? Bueno aquí también mucha discusión al respecto, hay personas que dicen que todo con TDD, otras que sólo donde aplique (por ejemplo si estás interactuando con el DOM y se tienen que realizar muchos cambios se puede hacer pesado trabajar con *TDD*), como consejo personal, al menos en funciones que estén más relacionadas con procesos de negocio, es muy buena idea seguir esta aproximación.

Vamos a simularlo con un ejemplo.

Imagina que queremos crear una función llamada *comprobarNumeroB*, y cómo parámetros de entrada vamos a pasarle un texto y el número que vamos a acertar. Y como resultado nos va a devolver lo que definimos antes, los valores: *NO_ES_UN_NUMERO*, *EL_NUMERO_ES_MAYOR*, *EL_NUMERO_ES_MENOR* o *ES_EL_NUMERO_SECRETO*. En este caso vamos a obviar si hemos pasado el número máximo de intentos.

- Primero creamos la función vacía.

motor.js

```
export const comprobarNumeroB = (texto, numeroAcertar) => {};
```

- Ahora vamos a crear un test para esta función

motor.spec.js

```
- import { comprobarNumero } from "./motor";
+ import { comprobarNumero, comprobarNumeroB } from "./motor";
(...)

describe("comprobarNumeroB", () => {
  it("el número es mayor que el número que acertar", () => {
    // Arrange
    const texto = "25";
    const numeroAcertar = 24;

    // Act
    const resultado = comprobarNumeroB(texto, numeroAcertar);

    // Assert
    expect(resultado).toBe(EL_NUMERO_ES_MAYOR);
  });
});
```

Ejecutamos los tests y nos da rojo, porque la función está vacía, vamos a implementar lo mínimo para que pase el test.

```
export const comprobarNumeroB = (texto, numeroAcertar) => {
  const numero = parseInt(texto);

  if (numero > numeroAcertar) {
    return EL_NUMERO_ES_MAYOR;
  }
};
```

Vamos a ejecutar los tests

```
npm run test
```

Vemos que tenemos nuestros test en verde 😊.

Vamos a crear otro test para ver qué pasa si el número introducido es menor al número que tenemos que acertar.

```
describe("comprobarNumeroB", () => {
  it("el número es mayor que el número que acertar", () => {
    // Arrange
    const texto = "25";
    const numeroAcertar = 24;

    // Act
    const resultado = comprobarNumeroB(texto, numeroAcertar);

    // Assert
    expect(resultado).toBe(EL_NUMERO_ES_MAYOR);
  });

+  it("el número introducido es menor que el número que acertar", () => {
+    // Arrange
+    const texto = "23";
+    const numeroAcertar = 24;

+    // Act
+    const resultado = comprobarNumeroB(texto, numeroAcertar);

+    // Assert
+    expect(resultado).toBe(EL_NUMERO_ES_MENOR);
+  });
});
```

Si ahora ejecutamos los tests nos da rojo, vamos a modificar la función para que pase el test.

```
export const comprobarNumeroB = (texto, numeroAcertar) => {
  const numero = parseInt(texto);

  if (numero > numeroAcertar) {
    return EL_NUMERO_ES_MAYOR;
-  }
+  } else {
+   if (numero < numeroAcertar) {
+     return EL_NUMERO_ES_MENOR;
+   }
+  }
};
```

Como vemos, hemos ido implementando la función poco a poco, y hemos ido comprobando que funciona, y si no funciona, hemos ido modificando la función para que pasen los tests.

Vamos ahora a crear un nuevo test para ver qué pasa si el número introducido es el número que tenemos que acertar.

```
describe("comprobarNumeroB", () => {
  ( ... )

+  it("el número introducido es el que teníamos que acertar", () => {
+    // Arrange
+    const texto = "24";
+    const numeroAcertar = 24;

+    // Act
+    const resultado = comprobarNumeroB(texto, numeroAcertar);

+    // Assert
+    expect(resultado).toBe(ES_EL_NUMERO_SECRETO);
+  });
});
```

Vemos que nos da rojo, vamos a modificar la función para que pase el test.

```
export const comprobarNumeroB = (texto, numeroAcertar) => {
  const numero = parseInt(texto);

  if (numero > numeroAcertar) {
    return EL_NUMERO_ES_MAYOR;
  } else {
   if (numero < numeroAcertar) {
     return EL_NUMERO_ES_MENOR;
-   }
+   } else {
+     return ES_EL_NUMERO_SECRETO;
+   }
 }
};
```

Y cómo último test vamos a comprobar que si el texto que introducimos no es un número, nos va a devolver *NO_ES_UN_NUMERO*.

```
describe("comprobarNumeroB", () => {
  ( ... )

+  it("el texto introducido no es un número", () => {
+    // Arrange
+    const texto = "Introducimos un texto";
+    const numeroAcertar = 24;

+    // Act
+    const resultado = comprobarNumeroB(texto, numeroAcertar);

+    // Assert
```

```
+    expect(resultado).toBe(NO_ES_UN_NUMERO);
+  });
});
```

Vemos que de nuevo los test se ponen en rojo, modificamos nuestra función para que pase el test.

```
export const comprobarNumeroB = (texto, numeroAcertar) => {
  const numero = parseInt(texto);
+  const esUnNumero = !isNaN(numero);

+  if (!esUnNumero) {
+    return NO_ES_UN_NUMERO;
+  } else {
    if (numero > numeroAcertar) {
      return EL_NUMERO_ES_MAYOR;
    } else {
      if (numero < numeroAcertar) {
        return EL_NUMERO_ES_MENOR;
      } else {
        return ES_EL_NUMERO_SECRETO;
      }
    }
  }

};
```

Ahora tenemos un código que funciona, pero que tiene mala pinta, ¿Lo refactorizamos? 😅, si no hubieramos seguido TDD nos daría respeto modificarlo (eso que dicen "si funciona no lo toques"), pero como tenemos una buena baterías de tests podemos mejorar el código y comprobar que los tests siguen pasando en verde, como quedaría.

```
Ahora por último podríamos retactorizar el código para que quede más limpio.

```js
export const comprobarNumeroB = (texto, numeroAcertar) => {
  const numero = parseInt(texto);
  const esUnNumero = !isNaN(numero);

+  if (!esUnNumero) {
+    return NO_ES_UN_NUMERO;
+  }
+
+  if (numero === numeroAcertar) {
+    return ES_EL_NUMERO_SECRETO;
+  }

+  return numero > numeroAcertar ? EL_NUMERO_ES_MAYOR : EL_NUMERO_ES_MENOR;
};
```
```

Vemos que los tests vuelven a pasar y salen en verde, imagínate que hubiéramos metido un fallo tonto, por ejemplo haber cambiado el > por un <, si lo pruebas fíjate que te salen tests en rojo.

Resumen

En este módulo hemos visto:

- Qué son las pruebas unitarias.
- Como implementar pruebas unitarias básicas.
- Qué son los espías y el mocking.
- Qué es el testeo de UI, aunque lo hemos dejado para módulos posteriores.
- Qué es TDD.