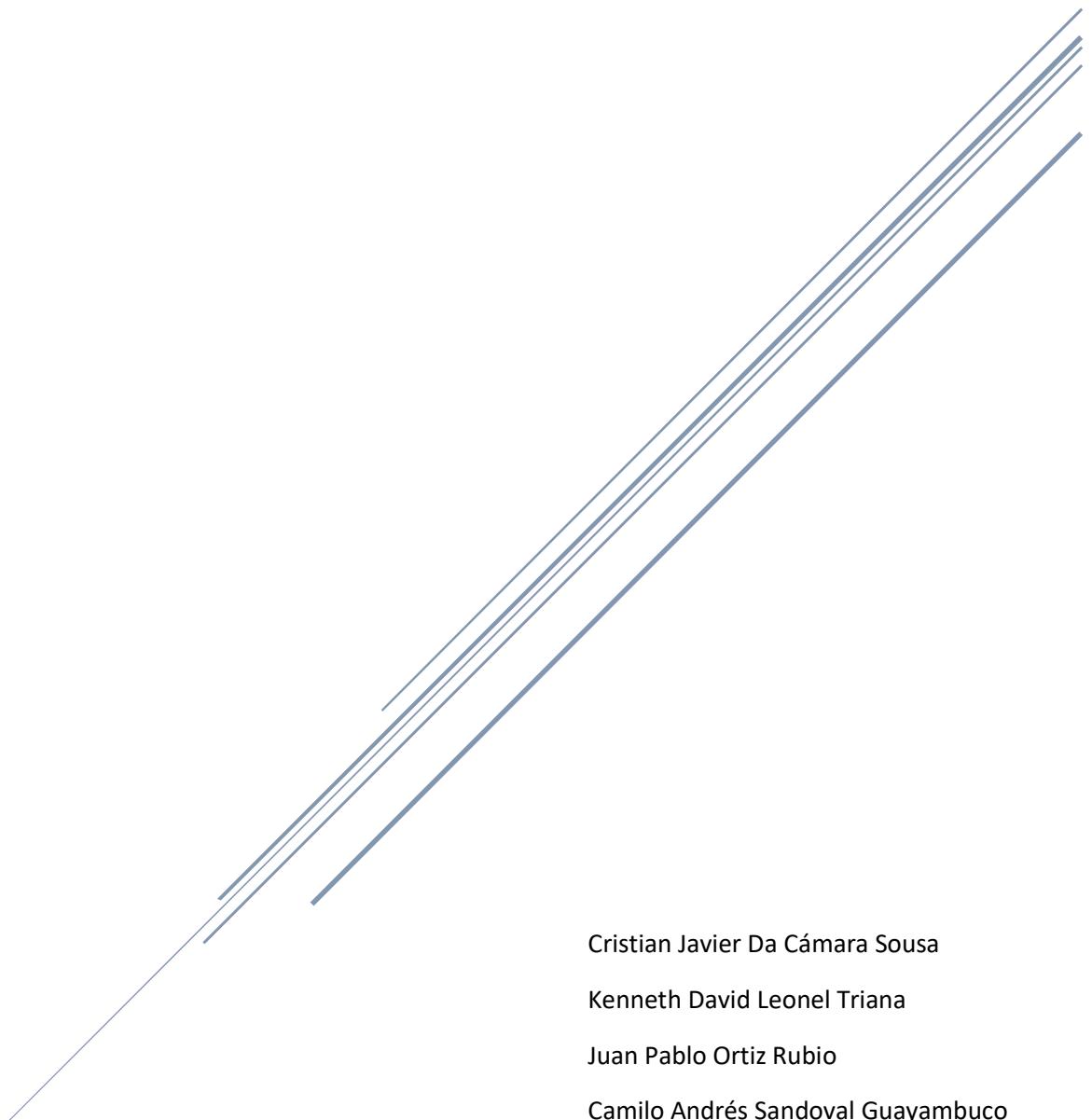


PLAN DE PRUEBAS – APRENDIZAJE AUTOMÁTICO DE LENGUA DE SEÑAS COLOMBIANA

CIS2210CP03



Cristian Javier Da Cámara Sousa

Kenneth David Leonel Triana

Juan Pablo Ortiz Rubio

Camilo Andrés Sandoval Guayambuco

Noviembre 2022

1. Introducción

El propósito principal de este documento es detallar las pruebas del Sistema, mirando su correcto funcionamiento tanto en la parte de Interfaz donde el usuario eventualmente interactuara, como en la parte de procesamiento y de lógica del Sistema. A su vez se documentará los factores externos que pueden afectar el buen funcionamiento de este, para ofrecer la satisfacción máxima de nuestros servicios a los usuarios.

1.1 Objetivo general

La finalidad del Plan de pruebas es poner el Sistema bajo a ciertas circunstancias que puedan ser controladas y evaluadas, para llegado acaso perfeccionar el Software, dejando ventajas a la hora de presentar el servicio para los usuarios, dejando un producto que satisfaga sus necesidades, que pueda utilizar nuestro Sistema de la forma más dinámica, generando como resultado una buena experiencia y confianza de la aplicación.

1.2 Estrategia de pruebas

Para la realización de las pruebas tanto para *Front-End* como para *Back-End*, fue necesario averiguar los diferentes *frameworks* que puedan encapsular nuestro Sistema y ponerlo a prueba, a continuación, se explicará las herramientas optadas:

- **Front-End:**

En el caso de Angular a la hora de crear el proyecto, brindan ya servicios para efectuar las pruebas, dejando como ventaja solo enfocarse en materializar los diferentes ambientes que se van a querer probar. Seguidamente se aclara las tecnologías usadas.

- **Karma:** Es una herramienta que genera un servidor web como test runner, que ejecuta cada una de las pruebas en el navegador especificado, dichas configuraciones se pueden ver en el archivo “karma.conf.js.”, esto facilita para ver en detalle si pasaron las pruebas correctamente, o en caso de error nos provee información de este [1].
- **Jasmine:** Es el frameWork de testing que justamente sirve para aplicarse en cualquier plataforma habilitada para JavaScript, su gran ventaja es la sintaxis que optaron, ya que hace que sea entendible, una de las ventajas de Jasmine es que nos provee lo que serían ‘Spys’ que son esenciales para poder simular objetos o procedimientos que se comporten como se espera que lo haga el Sistema, generando el resultado que esperamos [2].

- **Back-End:**

En la parte de back-end al usarse el *framework* de *FastApi* se tomó en cuenta la siguiente librería para poder hacer las pruebas unitarias:

- **Pytest:** Es la herramienta utilizada, ya que facilita la escritura de pruebas pequeñas, nos provee una documentación detallada haciendo su uso sencillo, se tiene similitudes *JUnit*, que es un marco de pruebas unitarias en Java que ya se ha tenido experiencia en ella haciendo su uso más efectivo por parte de desarrollo [3].

1.3 Alcance

El plan de pruebas será delimitado específicamente por las pruebas unitarias que fueron ejercidas tanto para *front-end* como *back-end* del Sistema. En cuanto la parte de *front-end* se tuvieron en cuenta tanto las pruebas orientadas a componentes como las de servicios con un total de 44 pruebas. Por otro lado, para el *back-end* se probó que la ejecución de las pruebas nos dé como resultado esperado que las funciones que ofrecemos a los usuarios del aplicativo sean las correctas, para ello se realizaron un total de 27 pruebas.

1.4 Propósito

Al llevar a cabo la ejecución del plan de pruebas esperamos que, con los resultados obtenidos, se pueda determinar que las funcionalidades que van a hacer los usuarios finales tengan congruencia, que satisfagan sus necesidades y que ayuden a mejorar el Sistema gracias a la detección oportuna de errores que con el futuro mejoren la experiencia del aplicativo.

2. Entregables

2.1 Documentación a entregar.

Con el fin de realizar una descripción detallada de los entregables de nuestro proyecto se tomó la decisión de repartir ítems según se requiera para cada entrega. Las entregas están asociadas con los siguientes documentos:

Entregable	Descripción
Memoria de trabajo	Es el documento final de trabajo donde se tiene la información esencial de la tesis desarrollada por el grupo de trabajo.
Plan de proyecto	Es un documento de control para administrar un proyecto de software. En él se establece las políticas, procedimientos, normas, tareas, horarios y recursos necesarios para completar el proyecto.
Especificación de requisitos	Es un conjunto de recomendaciones para la especificación de los requerimientos o requisitos de software el cual tiene como producto final la documentación de los acuerdos entre el cliente y el grupo de desarrollo para así cumplir con la totalidad de exigencias estipuladas.
Especificación del diseño	Es un documento que describe el diseño de software en cuanto a la arquitectura del proyecto y las características de cada componente del sistema.
Propuesta del diseño	Es el documento el cual contiene la descripción de la solución propuesta asociada al diseño
Manual de usuario	Es un conjunto de instrucciones y recomendaciones para el usuario final, en el que se explica el funcionamiento del aplicativo y las diversas funciones que el usuario puede hacer.
Plan de pruebas.	Documento Actual.

Tabla 1 Entregables del proyecto

3. Características de las pruebas

En esta sección se listará cada una de las pruebas que fueron ejecutadas, en forma resumida, si quiere detallar el funcionamiento uno a uno de dichas pruebas es recomendable dirigirse al quinto apartado “Sistema Bajo Pruebas” del documento actual. En la siguiente tabla se ilustrará lo anterior.

Pruebas unitarias del Sistema Front-end		
Característica	Descripción	Modulo orientado
Creación Componente	Se corrobora que se cree el componente sin inconveniente	Componentes: 7 pruebas
Creación de Servicio	Se corrobora que se cree el servicio sin inconveniente	Servicio: 5 pruebas
Corroborar aspectos Interfaz	Se mira si existe los elementos vitales para correcto funcionamiento	Componentes: 14 pruebas
Corroborar funcionalidades	Se detalla que realice bien la función necesaria	Componentes: 10 pruebas
Funcionalidades de Servicios	Se hace uso de <i>Jasmine Spy</i> para simular Peticiones	Servicio: 8 Pruebas
		Total: 44 Pruebas

Tabla 2 Pruebas Front-End

Pruebas unitarias del Sistema Back-end		
Características	Descripción	Modulo orientado
Funcionalidades de peticiones	Se simulan a través de <i>mocks</i> los objetos de las peticiones	Endpoints: 27 pruebas
		Total: 27 pruebas

Tabla 3 Pruebas Back-End

4. Características no probadas

A lo largo de la producción de las pruebas unitarias, se tomaron decisiones tales como la no evaluación del componente de Traducción, es decir no se pudo simular mediante *Jasmine* la conexión de un *socket* para evidenciar y demostrar que los datos y funciones del Sistema sean los correctos, este aspecto es en cuanto al *front-end*.

Con respecto al *back-end* no se realizaron las pruebas a las diferentes funcionalidades de los componentes como lo son los controladores, repositorios, funciones externas que ayudan a la composición y arquitectura definida del aplicativo, ya que se le dieron una prioridad alta a probar los *endpoints*.

5. SUT (SISTEMA BAJO PRUEBAS)

Pruebas Unitarias Front-End

Para estas pruebas unitarias de Angular se realizaron mediante la utilización de *Jasmine* y *Karma*, con un total de cuarenta y cuatro Pruebas tanto enfocadas a componentes como a servicios, se resume las pruebas en la ilustración 1.

```
44 specs, 0 failures, randomized with seed 74419
DEPRECATION: describe with no children (describe() or it()) is deprecated and will be removed in the next major version. Please use describe('Component Name') or it('Test Name') instead.
describe or add children to it. Note: This message will be shown only once. Set the verbose option to true to see it every time.

Pruebas Componente ChartResultsComponent
• Realizar promedio de los datos
• Creación del componente ChartResultsComponent
• Verificar que la variable hasData sea falsa
• Verificar que la variable hasData sea verdadera

Pruebas Componente HomeComponent
• Comprobar existencia información de uno de los integrantes
• Comprobar existencia apartado integrantes
• Comprobar título del componente HomeComponent
• Creación del HomeComponent

Pruebas Servicio SignalService
• Comprobar creación nueva seña
• Servicio activo correctamente SignalService

Pruebas Componente SignupComponent
• Comprobar formulario de registro incompleto
• Creación del SignupComponent
• Comprobar formulario de registro válido

Pruebas Servicio TrainService
• Servicio activo correctamente TrainService
• Comprobar retorno estructura de estado del entrenamiento
• Comprobar aviso comienzo entrenamiento del modelo
• Comprobar retorno arreglo de señas por entrenar

Pruebas Componente TrainingComponent
• Comprobar botón de entrenamiento deshabilitado
• Comprobar modelo actual este vacío
• Creación del componente TrainingComponent

Pruebas Servicio RestService
• Servicio funcionando correctamente

Pruebas Componente TranslationComponent
Signal
ModelVariables

Pruebas Componente AppComponent
• Comprobar redirección y creación componente de login
• Creación del componente app.component
• Comprobar que el título sea 'Reconocimiento LSC'
• Comprobar Usuario no logueado
• Comprobar nombre de usuario vacío
• Comprobar item de redirección a otras páginas

Pruebas Componente NewSignalComponent
• Comprobar arreglo de fotos vacía
• Comprobar la función de vaciar arreglo de fotos capturadas
• Comprobar input de la seña lleno
• Comprobar input de la seña vacío
• Comprobar la función de eliminar foto del arreglo de fotos capturadas
• Creación del NewSignalComponent

Pruebas Componente LoginComponent
• Comprobar login incorrecto
• Comprobar formulario de login incompleto
• Comprobar formulario de login válido
• Creación del LoginComponent
• Comprobar login correcto

Pruebas Servicio UserService
• Comprobar retorno de error de falta credenciales del usuario
• Comprobar datos del usuario retornados
• Comprobar error de usuario no registrado
• Comprobar registro exitoso de un usuario
• Servicio activo correctamente UserService

Pruebas Servicio WebsocketService
• Servicio funcionando correctamente
```

Ilustración 1 pruebas unitarias Front-End

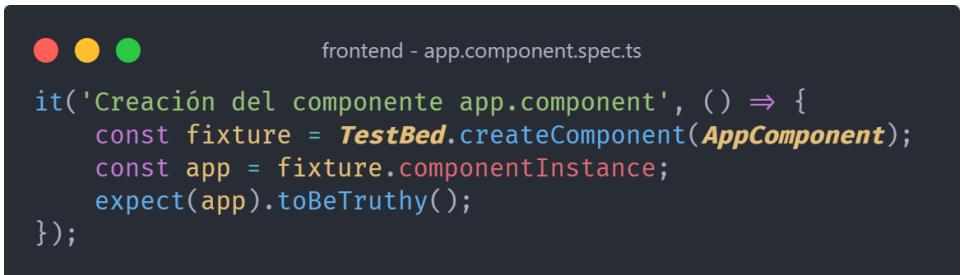
Pruebas de componentes:

Las pruebas unitarias orientadas a componentes tienen como finalidad probar las variables, funciones, corroborar que existan ciertas características visuales del HTML, que deban existir al ejecutarse. Se explicarán cada una de las pruebas por componente.

- **Componente App**

- **Creación del componente**

El fin de esta prueba es corroborar que se cree el componente, para ello la variable app se la va a instanciar el componente esperando que esto sea verdadero.

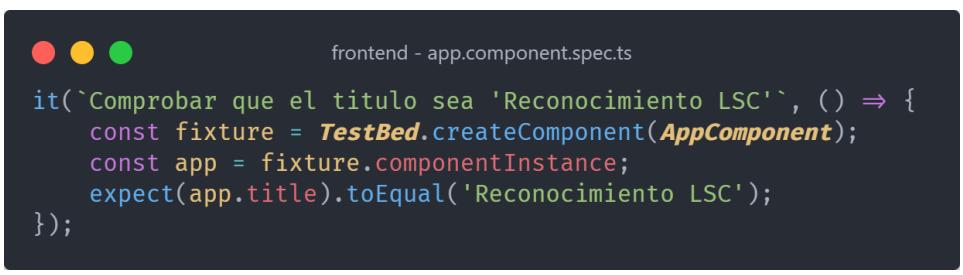


```
● ● ● frontend - app.component.spec.ts
it('Creación del componente app.component', () => {
  const fixture = TestBed.createComponent(AppComponent);
  const app = fixture.componentInstance;
  expect(app).toBeTruthy();
});
```

Ilustración 2 Creación componente app

- **Corroborar título**

La prueba espera que el título sea igual al que se tiene en el componente en este caso “Reconocimiento LSC”, como app tiene la instancia del componente y este contiene una variable *title* la manipulamos para hacer la respectiva validación.



```
● ● ● frontend - app.component.spec.ts
it(`Comprobar que el título sea 'Reconocimiento LSC'`, () => {
  const fixture = TestBed.createComponent(AppComponent);
  const app = fixture.componentInstance;
  expect(app.title).toEqual('Reconocimiento LSC');
});
```

Ilustración 3 Corroborar título en componente app

- **Usuario no autenticado**

Se verifica que la variable *isLoggedIn* cuando se crea la instancia del componente mediante la variable accedemos a ella y validamos que sea falso, es decir que el usuario no se ha autenticado en el sistema.



frontend - app.component.spec.ts

```
it(`Comprobar Usuario no logueado`, () => {
  const fixture = TestBed.createComponent(AppComponent);
  const app = fixture.componentInstance;
  expect(app.isLoggedIn).toEqual(false);
});
```

Ilustración 4 Usuario no autenticado en componente app

- **Nombre usuario vacío**

Mediante la instancia del componente accedemos a la variable *username* para corroborar que este se encuentre vacío, ya que el usuario no se ha autenticado en el sistema.



frontend - app.component.spec.ts

```
it(`Comprobar nombre de usuario vacío`, () => {
  const fixture = TestBed.createComponent(AppComponent);
  const app = fixture.componentInstance;
  expect(app.username).toEqual('');
});
```

Ilustración 5 Nombre usuario vacío en componente app

- **Encontrar función goTo**

Se corrobora que exista en el componente la funcionalidad *goTo()*, función vital para acceder a nuevos componentes del aplicativo.



frontend - app.component.spec.ts

```
it(`Comprobar item de redirección a otras páginas`, () => {
  const fixture = TestBed.createComponent(AppComponent);
  const app = fixture.componentInstance;
  expect(app.goTo).toBeTruthy();
});
```

Ilustración 6 Función *goTo()* en componente app

- **Componente Home**

- **Creación del componente**

esta prueba es la misma que la de la creación del componente app, se verifica que se realiza correctamente la instancia del componente.



```
frontend - home.component.spec.ts

it('Creación del HomeComponent', () => {
  const fixture = TestBed.createComponent(HomeComponent);
  const app = fixture.componentInstance;
  expect(app).toBeTruthy();
});
```

Ilustración 7 Creación Componente home

- **Título del componente**

En la realización de esta prueba fue necesaria la manipulación del DOM, debido a que de esta forma podemos tomar los elementos del HTML del componente a testear, en este caso se agarra la etiqueta h1 y esperamos que el contenido de esta debe ser “Manual de uso”.



```
frontend - home.component.spec.ts

it('Comprobar título del componente HomeComponent', () => {
  const fixture = TestBed.createComponent(HomeComponent);
  fixture.detectChanges();
  const compiled = fixture.nativeElement;
  expect(compiled.querySelector('h1').textContent).toContain('Manual de uso');
});
```

Ilustración 8 Corroborar título en componente home

- **Existencia de apartado integrantes**

Esta prueba se tomó debido a que se utilizó la librería GSAP, para darle pequeños detalles de animación al manual de usuario del Sistema. Para ello se realizó por medio de la manipulación del DOM, se tomó la etiqueta h2 para determinar si esta contiene el texto de “Grupo de Investigación”.



```
frontend - home.component.spec.ts

it('Comprobar existencia apartado integrantes', () => {
  const fixture = TestBed.createComponent(HomeComponent);
  fixture.detectChanges();
  const compiled = fixture.nativeElement;
  expect(compiled.querySelector('h2').textContent).toContain('Grupo de Investigación');
});
```

Ilustración 9 Apartados integrantes en componente home

- **Verificar información Integrante**

Esta prueba solo corrobora que mediante la manipulación del DOM tomemos la información que contiene la etiqueta h3 que está dentro de un contenedor con la clase *card*, esta debe contener el nombre de uno de los integrantes del desarrollo del proyecto, en este caso esperamos que contenga el nombre de “Juan Pablo Ortiz Rubio”.



```
frontend - home.component.spec.ts

it('Comprobar existencia información de uno de los integrantes', () => {
  const fixture = TestBed.createComponent(HomeComponent);
  fixture.detectChanges();
  let usuario = fixture.debugElement.query(By.css('.card h3'));
  expect(usuario.nativeElement.textContent).toContain('Juan Pablo Ortiz Rubio');
});
```

Ilustración 10 Información Integrante en componente home

- **Componente NewSignal**

- **Creación del componente**

Esta prueba es la misma que la de la creación del componente app, se verifica que se realiza correctamente la instancia del componente.



```
frontend - new-signal.component.spec.ts

it('Creación del NewSignalComponent', () => {
  const fixture = TestBed.createComponent(NewSignalComponent);
  const app = fixture.componentInstance;
  expect(app).toBeTruthy();
});
```

Ilustración 11 Creación del componente NewSignal

- **Verificar arreglo fotos vacío**

Esta prueba rectifica que el arreglo *processedPhoto* tenga un tamaño de cero, ya que, si esta prueba corriera mal, se tendría inconsistencia en la variable cosa que puede afectar negativamente el buen funcionamiento del Sistema.



frontend - new-signal.component.spec.ts

```
it('Comprobar arreglo de fotos vacía', () => {
  const fixture = TestBed.createComponent(NewSignalComponent);
  const app = fixture.componentInstance;
  fixture.detectChanges();
  expect(app.processedPhotos.length).toBe(0);
});
```

Ilustración 12 Arreglo fotos vacío en componente NewSignal.

- **Probar función deleteAll()**

En esta prueba se simula que la variable captures que es el arreglo donde se guardan las imágenes de una nueva Seña contiene tres imágenes, con Jasmine esperamos que el arreglo tenga un tamaño de tres, después utilizamos la función del componente llamado *deleteAll()*. Para que finalice con éxito esta prueba después de ejecutarse la función se espera que el tamaño del arreglo sea cero.



frontend - new-signal.component.spec.ts

```
it('Comprobar la función de vaciar arreglo de fotos capturadas', () => {
  const fixture = TestBed.createComponent(NewSignalComponent);
  const app = fixture.componentInstance;
  fixture.detectChanges();
  app.captures.push('foto1');
  app.captures.push('foto2');
  app.captures.push('foto3');
  expect(app.captures.length).toBe(3);
  app.deleteAll();
  expect(app.captures.length).toBe(0);
});
```

Ilustración 13 Función deleteAll() en componente NewSignal.

- **Probar función eliminar foto.**

Esta prueba tiene como finalidad probar que se puede eliminar una foto del arreglo *captures*, para ello se simula que se toma una foto es decir llenamos dicho arreglo con una foto, probamos la función *deletePhoto()*, donde se le pasa por parámetro el índice de la foto que se desea eliminar, si sale correcto la prueba significa que se espera que el tamaño del arreglo sea cero.



frontend - new-signal.component.spec.ts

```
it('Comprobar la función de eliminar foto del arreglo de fotos capturadas', () => {
  const fixture = TestBed.createComponent(NewSignalComponent);
  const app = fixture.componentInstance;
  fixture.detectChanges();
  app.captures.push('foto1');
  app.deletePhoto(0);
  expect(app.captures.length).toBe(0);
});
```

Ilustración 14 Eliminar foto en componente NewSignal

- **Nombre de la seña vacío**

Se espera que cuando se crea la instancia del componente, la variable *inputText* debe estar vacío, esta variable representa el nombre de la seña que se va a crear en el componente.



```
● ● ● frontend - new-signal.component.spec.ts
it('Comprobar input de la seña vacío', () => {
  const fixture = TestBed.createComponent(NewSignalComponent);
  const app = fixture.componentInstance;
  fixture.detectChanges();
  expect(app.inputText).toBe('');
});
```

Ilustración 15 Nombre de la seña vacío en componente NewSignal

- **Comprobar nombre seña lleno**

Esta prueba consiste en que cuando el usuario va a guardar la seña se valide que el nombre de la seña deba contener su información correcta. Para ello se valida que la variable *inputText* contenga “Hola”.



```
● ● ● frontend - new-signal.component.spec.ts
it('Comprobar input de la seña lleno', () => {
  const fixture = TestBed.createComponent(NewSignalComponent);
  const app = fixture.componentInstance;
  fixture.detectChanges();
  app.process();
  app.inputText = 'Hola';
  expect(app.inputText).toBe('Hola');
});
```

Ilustración 16 Simular nombre seña lleno en componente NewSignal

- **Componente ChartResults**
 - **Creación del componente**

Esta prueba es la misma que la de la creación del componente app, se verifica que se realiza correctamente la instancia del componente.

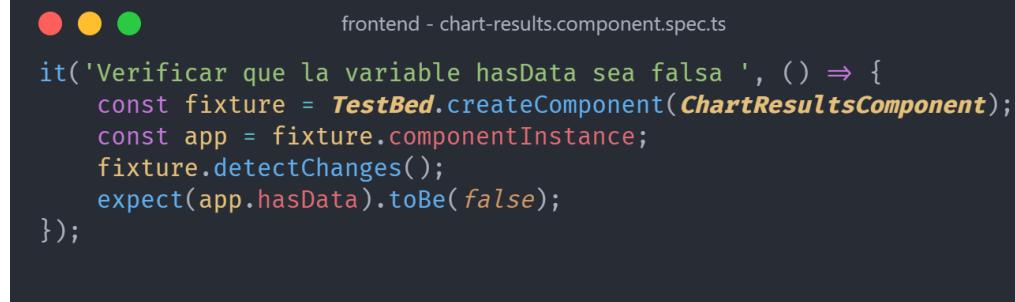


```
● ● ● frontend - chart-results.component.spec.ts
it('Creación del componente ChartResultsComponent', () => {
  const fixture = TestBed.createComponent(ChartResultsComponent);
  const app = fixture.componentInstance;
  expect(app).toBeTruthy();
});
```

Ilustración 17 Creación del componente ChartResults

- **Variable hasData sea falsa**

La finalidad de esta prueba es mirar que la variable *hasData* al crearse el componente tenga como valor booleano falso.



```
● ● ● frontend - chart-results.component.spec.ts
it('Verificar que la variable hasData sea falsa ', () => {
  const fixture = TestBed.createComponent(ChartResultsComponent);
  const app = fixture.componentInstance;
  fixture.detectChanges();
  expect(app.hasData).toBe(false);
});
```

Ilustración 18 Variable hasData sea falsa en componente ChartResults

- **Verificar variable hasData sea verdadero**

Se realiza la simulación de cuándo debe cambiar el valor de la variable *hasData*, para ello tomamos el arreglo “dato” lo inicializamos como un arreglo vacío, y lo llenamos con un objeto que contiene la estructura necesaria (seña, precisión), si el tamaño del arreglo es uno le asignamos a la variable *hasData* el valor true y esperamos que esto sea verdad.

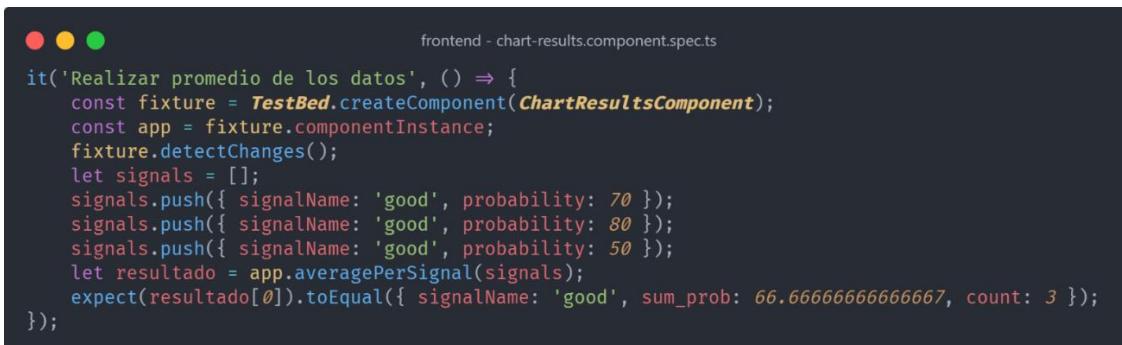


```
● ● ● frontend - chart-results.component.spec.ts
it('Verificar que la variable hasData sea verdadera ', () => {
  const fixture = TestBed.createComponent(ChartResultsComponent);
  const app = fixture.componentInstance;
  fixture.detectChanges();
  let signals = [];
  signals.push({ signalName: 'good', probability: 70 });
  expect(signals.length).toBe(1);
  app.hasData = true;
  expect(app.hasData).toBe(true);
});
```

Ilustración 19 Variable *hasData* sea verdadero en componente *ChartResults*

- **Promedio de los datos correcto**

Esta prueba verifica que la función que se tiene para calcular el promedio de las señas que sea el esperado, para ello se hace la simulación que el arreglo dato contiene tres señas con el mismo nombre, es decir que se calcula el promedio del peso en este caso de la señal con nombre “*good*”, esperamos que al ejecutar la función *averagePerSignal()*, nos dé como resultado un objeto que contenga como atributos la señal, el peso donde este debe contener el promedio y el contador que es la cantidad de datos del arreglo dato de la misma señal.



```
● ● ● frontend - chart-results.component.spec.ts
it('Realizar promedio de los datos', () => {
  const fixture = TestBed.createComponent(ChartResultsComponent);
  const app = fixture.componentInstance;
  fixture.detectChanges();
  let signals = [];
  signals.push({ signalName: 'good', probability: 70 });
  signals.push({ signalName: 'good', probability: 80 });
  signals.push({ signalName: 'good', probability: 50 });
  let resultado = app.averagePerSignal(signals);
  expect(resultado[0]).toEqual({ signalName: 'good', sum_prob: 66.66666666666667, count: 3 });
});
```

Ilustración 20 Promedio de los datos correcto en componente *ChartResults*

- **Componente Training**
 - **Creación componente**

Esta prueba es la misma que la de la creación del componente app, se verifica que se realiza correctamente la instancia del componente.



frontend - training.component.spec.ts

```
it('Creación del componente TrainingComponent', () => {
  const fixture = TestBed.createComponent(TrainingComponent);
  const app = fixture.componentInstance;
  expect(app).toBeTruthy();
});
```

Ilustración 21 Creación del componente Training

- **Verificar botón deshabilitado**

Se pretende verificar que el botón no se encuentre habilitado al crearse el componente, para ello se hace uso del DOM, se toma la etiqueta que tenga como id “button_train”, y esperamos que el botón en verdad no se encuentra disponible.



frontend - training.component.spec.ts

```
it('Comprobar botón de entrenamiento deshabilitado', () => {
  const fixture = TestBed.createComponent(TrainingComponent);
  const app = fixture.componentInstance;
  fixture.detectChanges();

  let button = fixture.debugElement.query(By.css('#button_train'));
  expect(button.nativeElement.disabled).toBeTruthy();
});
```

Ilustración 22 Verificar botón deshabilitado en componente Training.

- **Verificar modelo actual vacío**

Se verifica que la variable “actualModel” tenga sus parámetros vacíos.

```

● ● ● frontend - training.component.spec.ts
it('Comprobar modelo actual este vacío', () => {
  const fixture = TestBed.createComponent(TrainingComponent);
  const app = fixture.componentInstance;
  fixture.detectChanges();

  expect(app.actualModel.id).toBe(0);
  expect(app.actualModel.name).toBe('');
  expect(app.actualModel.loss).toBe(0);
  expect(app.actualModel.val_loss).toBe(0);
  expect(app.actualModel.accuracy).toBe(0);
  expect(app.actualModel.val_accuracy).toBe(0);
  expect(app.actualModel.mean_time_execution).toBe(0);
  expect(app.actualModel.epoch).toBe(0);
  expect(app.actualModel.cant_epochs).toBe(0);
  expect(app.actualModel.training_state).toBe(0);
  expect(app.actualModel.begin_time).toBe('');
  expect(app.actualModel.end_time).toBe('');
  expect(app.actualModel.trained_signals.size).toBe(0);
});

```

Ilustración 23 Verificar modelo actual vacío en componente Training

- **Componente Login**

- **Creación del componente**

Esta prueba es la misma que la de la creación del componente app, se verifica que se realiza correctamente la instancia del componente.

```

● ● ● frontend - login.component.spec.ts
it('Creación del LoginComponent', () => {
  const fixture = TestBed.createComponent(LoginComponent);
  const app = fixture.componentInstance;
  expect(app).toBeTruthy();
});

```

Ilustración 24 Creación del componente login

- **Comprobar formulario incompleto**

Para realizar la prueba se toma en cuenta nuestro formulario que tiene como variable lo que sería el correo electrónico y la contraseña, en este caso queremos validar que el formulario no sea correcto ya que la variable contraseña se simulo que se encuentra vacío.

```
● ● ● frontend - login.component.spec.ts
it('Comprobar formulario de login incompleto', () => {
  const fixture = TestBed.createComponent(LoginComponent);
  const app = fixture.componentInstance;
  fixture.detectChanges();

  // form
  app.formFields.setValue({
    email: 'kennethleo1@hotmail.com',
    password: '',
  });
  expect(app.formFields.valid).toBeFalsy();
});
```

Ilustración 25 Formulario incompleto en componente login

- **Comprobar de formulario de login válido**

En esta prueba se tomó la variable *formFields* que representa el formulario de ingreso, para ello se les asigna a los parámetros de contraseña y correo electrónico. Esperando que la variable *formFields.valid* sea verdadero ya que se encuentra los datos completos de este.

```
● ● ● frontend - login.component.spec.ts
it('Comprobar formulario de login válido', () => {
  const fixture = TestBed.createComponent(LoginComponent);
  const app = fixture.componentInstance;
  fixture.detectChanges();

  // form
  app.formFields.setValue({
    email: 'kennethleo1@hotmail.com',
    password: '123456',
  });
  expect(app.formFields.valid).toBeTruthy();
});
```

Ilustración 26 formulario de login válido en componente login

- **Comprobar login correcto**

La prueba consiste en simular que el usuario ingresa correctamente en el sistema es decir que llenamos los datos del formulario (contraseña y correo electrónico) y mediante el uso del DOM, se manipula el HTML para poder tomar la etiqueta con

la “buttonFix”, después se simula que damos click al botón, se le asigna a la variable token un valor y se espera que sea el mismo.

```
frontend - login.component.spec.ts

it('Comprobar login correcto', () => {
  const fixture = TestBed.createComponent(LoginComponent);
  const app = fixture.componentInstance;
  fixture.detectChanges();

  // form
  app.formFields.setValue({
    email: 'kennethleo1@hotmail.com',
    password: '123456',
  });

  let button = fixture.debugElement.nativeElement.querySelector('.buttonFix');
  button.click();
  app.token = 'token';
  expect(app.token).toBe('token');
});
```

Ilustración 27 Comprobación login correcto en componente login.

○ Comprobar Login incorrecta

Esta prueba tiene como fin simular que se realiza la autenticación de forma incorrecta, es decir que al dar click al botón el token del usuario autenticado ocurrió un error y no pudo ser asignado como es el caso se espera que dicha variable sea *undefined*.

```
frontend - login.component.spec.ts

it('Comprobar login incorrecto', () => {
  const fixture = TestBed.createComponent(LoginComponent);
  const app = fixture.componentInstance;
  fixture.detectChanges();

  let debugElement = fixture.debugElement;

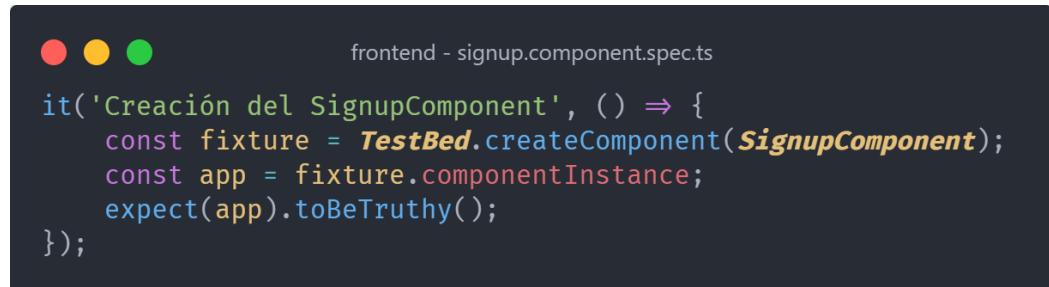
  // form
  app.formFields.setValue({
    email: 'kennethleo1@hotmail.com',
    password: '123456',
  });

  debugElement.nativeElement.querySelector('.buttonFix');
  expect(app.token).toEqual(undefined);
});
```

Ilustración 28 Comprobar login incorrecto del usuario en componente login

- **Componente Signup**
 - **Creación del componente**

Esta prueba es la misma que la de la creación del componente app, se verifica que se realiza correctamente la instancia del componente.



```
● ● ● frontend - signup.component.spec.ts
it('Creación del SignupComponent', () => {
  const fixture = TestBed.createComponent(SignupComponent);
  const app = fixture.componentInstance;
  expect(app).toBeTruthy();
});
```

Ilustración 29 Creación del componente Signup

- **Formulario de registro incompleto**

Para la realización de esta prueba se tomó la variable del componente llamado *formFields*, a este formulario que contiene los siguientes atributos (correo, contraseña, nombre, rol) se le asignan los valores correspondientes exceptuando al del atributo contraseña, la ventaja que nos ofrece la variable es que se puede acceder a una propiedad llamada *valid*, en el cual si *formFields.valid* es false significa que nuestro formulario no se ha completado correctamente.



```
● ● ● frontend - signup.component.spec.ts
it('Comprobar formulario de registro incompleto', () => {
  const fixture = TestBed.createComponent(SignupComponent);
  const app = fixture.componentInstance;
  fixture.detectChanges();

  // form
  app.formFields.setValue({
    email: 'kennethleo1@hotmail.com',
    name: 'kenneth',
    role: 'admin',
    password: '',
  });
  expect(app.formFields.valid).toBeFalsy();
});
```

Ilustración 30 registro incompleto en componente Signup

- **Formulario de registro válido.**

Para la realización de esta prueba se tomó la variable del componente llamado *formFields*, a este formulario que contiene los siguientes atributos (correo, contraseña, nombre, rol) se le asignan los valores correspondientes, la ventaja que nos ofrece la variable es que se puede acceder a una propiedad llamada *valid*, en el cual si *formFields.valid* es true significa que nuestro formulario se ha completado correctamente.



frontend - signup.component.spec.ts

```
it('Comprobar formulario de registro válido', () => {
  const fixture = TestBed.createComponent(SignupComponent);
  const app = fixture.componentInstance;
  fixture.detectChanges();

  // form
  app.formFields.setValue({
    email: 'kennethleo1@hotmail.com',
    name: 'kenneth',
    role: 'admin',
    password: '123456',
  });
  expect(app.formFields.valid).toBeTruthy();
});
```

Ilustración 31 Registro válido en componente Signup.

Prueba de Servicios:

Las pruebas enfocadas en servicios son importantes para simular las interacciones que tiene el Front, para ello es necesario utilizar lo que sería un objeto SPY que puede simular las peticiones HTTP (GET, POST).

- **Servicio Signal**

- **Activar servicio**

Para que se puedan utilizar los servicio es necesario comprender lo que sería el *beforeEach* ya que de esta forma se configura lo que serían la simulación de una petición por HTTP, y la utilización de un cliente espía para realizar dichas pruebas orientada a servicio, donde se menciona que métodos se deben simular en este caso POST. Para ello esperamos que el servicio sirva correctamente.



frontend - signal.service.spec.ts

```
let service: SignalService;
let httpClientSpy: { post: jasmine.Spy };
```

```
beforeEach(() => {
  TestBed.configureTestingModule({
    imports: [HttpClientTestingModule],
  });

  httpClientSpy = jasmine.createSpyObj('HttpClient', ['post']);
  service = new SignalService(new RestService(httpClientSpy as any));
});

it('Servicio activo correctamente SignalService', () => {
  expect(service).toBeTruthy();
});
```

Ilustración 32 Servicio Signal creado correctamente

- **Comprobar creación nueva seña**

Para este caso vamos a Simular la creación de una nueva seña es por ello que se debe crear el objeto que se debe enviar en el servicio y lo que esperamos que nos devuelva el servicio. Se llama el servicio *createSignal()* que necesita como parámetros el arreglo de imágenes junto con el nombre de la seña, para que finalice con éxito la prueba se espera que el resultado que nos devuelve sea el que esperamos.



frontend - signal.service.spec.ts

```
it('Comprobar creación nueva seña', (done: DoneFn) => {
  const signal = {
    name: 'Casa',
    images: ['foto1', 'foto2', 'foto3'],
  };
  const resultado = ['Seña creada correctamente'];
  httpClientSpy.post.and.returnValue(of(resultado));

  const { name, images } = signal;
  service.createSignal(images, name).subscribe((res) => {
    expect(res).toEqual(resultado);
    done();
  });
});
```

Ilustración 33 Comprobar creación nueva seña

- **Servicio Train**

- **Activar servicio**

Para que se puedan utilizar los servicio es necesario comprender lo que sería el `beforeEach` ya que de esta forma se configura lo que serían la simulación de una petición por HTTP, y la utilización de un cliente espía para realizar dichas pruebas orientada a servicio, donde se menciona que métodos se deben simular en este caso GET. Para ello esperamos que el servicio sirva correctamente.

```
● ● ● frontend - train.service.spec.ts
let service: TrainService;
let httpClientSpy: { get: jasmine.Spy };

beforeEach(() => {
  TestBed.configureTestingModule({
    imports: [HttpClientTestingModule],
  });

  httpClientSpy = jasmine.createSpyObj('HttpClient', ['get']);
  service = new TrainService(new RestService(httpClientSpy as any));
});

it('Servicio activo correctamente TrainService', () => {
  expect(service).toBeTruthy();
});
```

Ilustración 34 Servicio Train creado correctamente

- **Retornar el arreglo de las señas por entrenar**

Para esta prueba creamos un arreglo que simulo lo que esperamos que nos retorne el servicio, en este caso denotamos que es una petición y le decimos al espía que simule lo que se encuentra en la variable esperada. Para finalizar ejecutamos lo que sería el servicio y decimos que lo que nos devuelve el servicio sea igual a lo que esperamos.

```

● ● ● frontend - train.service.spec.ts
it('Comprobar retorno arreglo de señas por entrenar', (done: DoneFn) => {
  const resultado = [
    new Signal('Signal 1', [], false),
    new Signal('Signal 2', [], false),
    new Signal('Signal 3', [], false),
  ];
  httpClientSpy.get.and.returnValue(of(resultado));

  service.getStatusTrainModel().subscribe((res) => {
    expect(res).toEqual(resultado);
    done();
  });
});

```

Ilustración 35 Retornar el arreglo de las señas por entrenar.

- **Entrenamiento del modelo empezó**

Esta prueba consiste en simular que lo que nos devuelve la petición GET, para ello se creó el arreglo resultado con la asignación de lo que esperamos nos devuelva el servicio, se hace el llamado a la función del servicio *trainModel()* donde este recibe como parámetro la época en el que se encuentra el proceso de entrenamiento. Esperamos que lo que nos devuelve el servicio sea igual al arreglo resultado que tenemos previsto.

```

● ● ● frontend - train.service.spec.ts
it('Comprobar aviso comienzo entrenamiento del modelo', (done: DoneFn) => {
  const epochs = 10;
  const resultado = ['El modelo tiene entrenamiento en proceso'];
  httpClientSpy.get.and.returnValue(of(resultado));

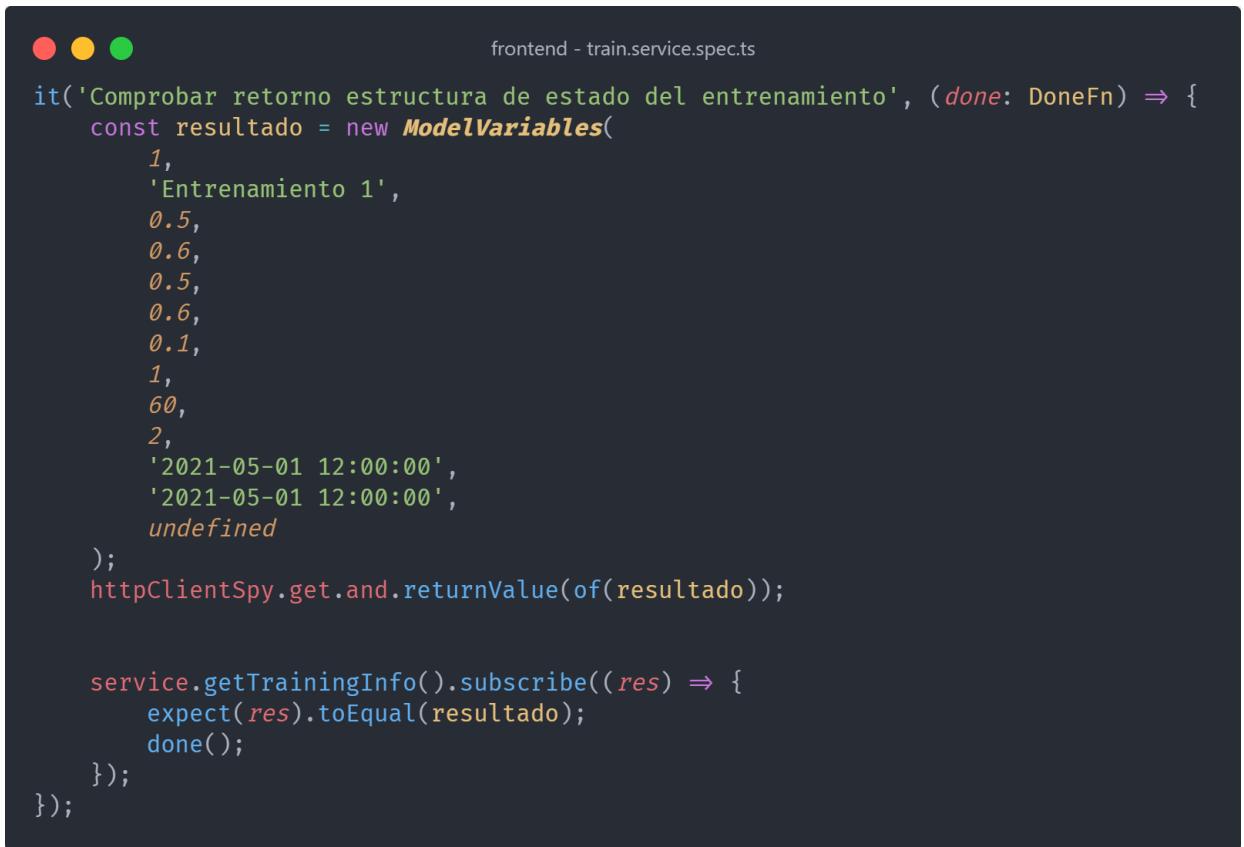
  service.trainModel(epochs).subscribe((res) => {
    expect(res).toEqual(resultado);
    done();
  });
});

```

Ilustración 36 Entrenamiento del modelo en proceso

- **Estructura de cómo va el entrenamiento**

El fin de esta prueba es calidad que nos llegue la estructura necesaria de cómo va el entrenamiento, es importante ya que dicho objeto será relevante para el componente de entrenamiento, para ello creamos el objeto que esperamos que nos devuelva el servicio y miramos si al realizar la función del servicio llamada `getTrainingInfo()` sea igual a la que tenemos prevista.



```
frontend - train.service.spec.ts

it('Comprobar retorno estructura de estado del entrenamiento', (done: DoneFn) => {
  const resultado = new ModelVariables(
    1,
    'Entrenamiento 1',
    0.5,
    0.6,
    0.5,
    0.6,
    0.1,
    1,
    60,
    2,
    '2021-05-01 12:00:00',
    '2021-05-01 12:00:00',
    undefined
  );
  httpClientSpy.get.and.returnValue(of(resultado));

  service.getTrainingInfo().subscribe((res) => {
    expect(res).toEqual(resultado);
    done();
  });
});
```

Ilustración 37 Estructura del entrenamiento

- **Servicio User**

- **Activar servicio**

Para que se puedan utilizar los servicio es necesario comprender lo que sería el `beforeEach` ya que de esta forma se configura lo que serían la simulación de una petición por HTTP, y la utilización de un cliente espía para realizar dichas pruebas orientada a servicio, donde se menciona que métodos se deben simular en este caso POST. Para ello esperamos que el servicio sirva correctamente.

```

● ● ● frontend - user.service.spec.ts
let service: UserService;
let httpClientSpy: { post: jasmine.Spy };

beforeEach(() => {
  TestBed.configureTestingModule({
    imports: [HttpClientTestingModule],
  });
  httpClientSpy = jasmine.createSpyObj('HttpClient', ['post']);
  service = new UserService(httpClientSpy as any);
});

it('Servicio activo correctamente UserService', () => {
  expect(service).toBeTruthy();
});

```

Ilustración 38 Servicio User creado correctamente

- **Registrar usuario correctamente**

Para esta prueba se simula el objeto que se le va a enviar al servicio, que es la información del formulario de registro de un usuario (nombre, correo, contraseña y rol), y también se crea el arreglo que esperamos que nos devuelva el servicio, hacemos el llamado a la función del servicio llamada *signup()* donde este recibe como parámetro cada uno de los campos del formulario, y esperamos que lo que nos devuelva el servicio sea igual a lo que tenemos previsto.

```

● ● ● frontend - user.service.spec.ts
it('Comprobar registro exitoso de un usuario', (done: DoneFn) => {
  const usuarioRegistro = {
    name: 'Kenneth',
    correo: 'kennethleo1@hotmail.com',
    password: '1234',
    role: 'admin',
  };
  const resultado = ['Usuario registrado correctamente'];
  const { name, correo, password, role } = usuarioRegistro;
  httpClientSpy.post.and.returnValue(of(resultado));

  service.signup(name, correo, password, role).subscribe((res) => {
    expect(res).toEqual(resultado);
    done();
  });
});

```

Ilustración 34 Registrar usuario correctamente

- **Registro usuario incorrecto**

El fin de esta prueba es simular que el servicio nos dé un error, para ello creamos el objeto que se le va a enviar al servicio, que es la información del formulario de registro de un usuario (nombre, correo, contraseña y rol), en este caso se crea el objeto que esperamos nos devuelva siendo este un objeto de tipo `HttpErrorResponse`, este contiene el mensaje de error y cuál fue el estado de error en este caso esperamos que fuera 400.

Se realiza el llamado de la función del servicio llamada `signup()` donde se envía como parámetros los campos del formulario, pero esperamos que nos devuelva el servicio el objeto de error previsto.



```
frontend - user.service.spec.ts

it('Comprobar error de usuario no registrado', (done: DoneFn) => {
  const usuarioRegistro = {
    name: 'Kenneth',
    correo: 'kennethleo1@hotmail.com',
    password: '1234',
    role: 'admin',
  };
  const error = new HttpErrorResponse{
    error: { message: 'Usuario no registrado' },
    status: 400,
  });
  const { name, correo, password, role } = usuarioRegistro;
  httpClientSpy.post.and.returnValue throwError(error);

  service.signup(name, correo, password, role).subscribe(
    (res) => {},
    (err) => {
      expect(err).toEqual(error);
      done();
    }
  );
});
```

Ilustración 35 Registro usuario incorrecto

- **Retornar datos del usuario**

Lo que se espera de esta prueba es que el servicio retorne el objeto como respuesta de que el usuario se autentico correctamente, para ello creamos el objeto que le enviamos al servicio que en este caso contiene los datos del formulario de ingreso

(correo y contraseña), se hace el llamado a la función del servicio llamada *login()* que recibe como parámetro los campos del formulario y esperamos que el resultado del servicio sea el previsto.

```
frontend - user.service.spec.ts

it('Comprobar datos del usuario retornados', (done: DoneFn) => {
    const usuarioLogin = {
        email: 'kennethleo1@hotmail.com',
        contra: '1234',
    };
    const resultado = {
        role: 'admin',
        usuario: 'Kenneth',
        token: 'DevuelveUnToken',
    };
    const { email, contra } = usuarioLogin;
    httpClientSpy.post.and.returnValue(of(resultado));

    service.login(email, contra).subscribe((res) => {
        expect(res).toEqual(resultado);
        done();
    });
});
```

Ilustración 36 Comprobar retorno datos del usuario

○ Retornar faltar de credenciales del usuario

La finalidad de esta prueba es que hubo un error del servicio entonces esperamos que el servicio nos devuelva un objeto de tipo *HttpErrorResponse*, que contiene el mensaje de error y el estatus de error que en este caso es el 400. Para ello se creó el objeto que se le va a enviar al servicio que son los datos del formulario (usuario y contraseña), se hace el llamado a la función del servicio llamada *login()* que recibe como parámetros dichos campos del formulario, para finalizar esperamos que lo que nos devuelve el servicio sea igual a lo que se tiene previsto.

```
● ● ● frontend - user.service.spec.ts
it('Comprobar retorno de error de falta credenciales del usuario', (done: DoneFn) => {
  const usuarioLogin = {
    email: 'kennethleo1@hotmail.com',
    contra: '',
  };
  const error = new HttpErrorResponse{
    error: { message: 'Usuario invalido' },
    status: 400,
  });
  const { email, contra } = usuarioLogin;
  httpClientSpy.post.and.returnValue throwError(error);

  service.login(email, contra).subscribe(
    (res) => {},
    (err) => {
      expect(err).toEqual(error);
      done();
    }
  );
});
```

Ilustración 38 Comprobar retorno error servicio User

Para los demás servicios como *RestService* o *WebSocketService* no se realizaron pruebas específicas ya que no vimos relevancias o no se pudo realizar por motivos de investigar en cómo realizar pruebas unitarias al socket.

Pruebas Unitarias Back-End:

Para empezar con las pruebas unitarias cabe aclarar que se tuvo que crear lo que serían los modelos que contienen las diferentes clases como: *RequestSignal*, *TrainingState*, *ModelVariables*, *UserModel*, *UserLogin* que ayudan a la realización de las pruebas.

- Endpoint crear nueva seña:

Para entender de forma sintetizada en que consiste esta prueba, se empieza con declarar lo que sería un falso token “*fake_secret_token*”, el *mock* tiene como intención simular el objeto que recibe la petición donde esta contiene lo que sería el nombre de la seña y el arreglo de imágenes, se realiza toda la configuración de la ruta donde se condiciona que, si no se tiene el token, se devuelve un error que significa que el usuario no tiene autorización, a su vez si la seña no tiene nombre, se devuelve como respuesta que la seña está vacía, por ultimo si el objeto *mock* es igual al de la petición , se muestra con éxito que la seña se creó correctamente.

```
● ● ●
5 fake_secret_token = "secret"
6
7 mock_db = {"1": RequestSignal(name="name1", images=["images1"])}
8
9 router = APIRouter()
10
11
12 @router.post("/new-signal")
13 async def save_signal(signal: RequestSignal, req: Request):
14     if req.headers["Authorization"] != fake_secret_token:
15         raise HTTPException(status_code=401, detail="Unauthorized")
16     if signal.name == "":
17         raise HTTPException(status_code=422, detail="El nombre de la señal está vacío")
18     mock_db["2"] = signal
19     return {"message": "Señal creada correctamente", "result": signal}
```

Ilustración 39 Endpoint crear seña

- Simular solicitud cliente en crear seña:

Para realizar la simulación de que el cliente realiza la petición de crear una nueva seña, se declara el cliente que es un *TestClient*, en ella se declara la variable *response*, donde se especifica lo que sería la url donde se hace el llamado al servidor, el JSON donde contiene la información en este caso el nombre de las señas y el arreglo de imágenes y en el objeto headers se tiene el token necesario ya que de esta forma se determina si el usuario tiene permiso para realizar la solicitud. Si el estatus del response es 200 (exitoso) y a su vez se procesa y se obtiene como mensaje del *response* “Seña creada correctamente” se da por finalizada la prueba.

```

9   client = TestClient(app)
10
11 def test_save_signal():
12     response = client.post(
13         url="/new-signal",
14         json=RequestSignal(name="test", images=["test"]).dict(),
15         headers={"Authorization": "secret"},
16     )
17     assert response.status_code == 200
18     response = response.json()
19     assert response["result"] == RequestSignal(name="test", images=["test"]).dict()
20     assert response["message"] == "Señal creada correctamente"

```

Ilustración 40 Comprobar que la seña se creó con autorización de autenticación

- **Simular solicitud cliente en crear seña sin permiso**

Para realizar la simulación de que el cliente realiza la petición de crear una nueva seña, se declara el cliente que es un *TestClient*, en ella se declara la variable *response*, donde se especifica lo que sería la url donde se hace el llamado al servidor, el JSON donde contiene la información en este caso el nombre de las señas y el arreglo de imágenes y en el objeto headers se tiene el token necesario ya que de esta forma se determina si el usuario tiene permiso para realizar la solicitud en este caso el token no es permitido, lo que se espera es que el estatus del *response* sea el 401 (restringido) y a su vez se tiene como *response* el objeto que contiene lo que sería el detalle o razón de error de la solicitud.

```

23 def test_save_signal_unauthorized():
24     response = client.post(
25         url="/new-signal",
26         json=RequestSignal(name="test", images=["test"]).dict(),
27         headers={"Authorization": "wrong"},
28     )
29     assert response.status_code == 401
30     assert response.json() == {"detail": "Unauthorized"}

```

Ilustración 41 comprobar solicitud crear seña sin permiso

- **Simular solicitud cliente en crear seña con datos faltantes**

Para realizar la simulación de que el cliente realiza la petición de crear una nueva seña, se declara el cliente que es un *TestClient*, en ella se declara la variable *response*, donde se especifica lo que sería la url donde se hace el llamado al servidor, el JSON donde contiene la información en este caso el nombre de las señas que se encuentra vacío y el arreglo de imágenes y en el objeto headers se tiene el token necesario ya que de esta forma se determina si el usuario tiene permiso para realizar la solicitud. En este caso se comprueba que el estatus del *response* sea el 422 (sintaxis incorrecta) y este a su vez debe tener el objeto que contiene lo que sería el detalle o razón de error de la solicitud.



```
33 def test_save_signal_incorrect_name():
34     response = client.post(
35         url="/new-signal",
36         json=RequestSignal(name="", images=["test"]).dict(),
37         headers={"Authorization": "secret"},
38     )
39     assert response.status_code == 422
40     assert response.json() == {"detail": "El nombre de la señal está vacío"}
```

Ilustración 42 Comprobar error al crear señal con datos faltantes

- Endpoint entrenamiento

Para entender de forma sintetizada en que consiste esta prueba, se empieza con declarar lo que sería un falso token “fake_secret_token”, el mock tiene como intención simular el objeto de la base de datos donde esta contiene lo que sería la señal y dos registros de modelos donde el primer registro tiene como estado de entrenamiento finalizado y el segundo registro el estado se encuentra en proceso, se tiene también, lo que sería la simulación del objeto de imágenes donde este tiene como fin decir que señas han sido procesadas y cuáles no.

```

7   fake_secret_token = "secret"
8   mock_db = {
9     "signals": {
10       "1": {
11         "name": "prueba1",
12         "images": None,
13         "processed_signal": False,
14       }
15     },
16     "models": {
17       "1": {
18         "id": "1",
19         "name": "prueba1",
20         "loss": 0.316,
21         "val_loss": 0.316,
22         "accuracy": 0.932,
23         "val_accuracy": 0.932,
24         "mean_time_execution": 0.6396115562799787,
25         "epoch": 1000,
26         "cant_epochs": 1000,
27         "training_state": 3,
28         "begin_time": "{TinyDate}:2022-10-21T02:18:09.589691",
29         "end_time": "{TinyDate}:2022-10-21T02:20:15.966698",
30       },
31       "2": {
32         "id": "2",
33         "name": "prueba2",
34         "loss": 0.316,
35         "val_loss": 0.316,
36         "accuracy": 0.932,
37         "val_accuracy": 0.932,
38         "mean_time_execution": 0.6396115562799787,
39         "epoch": 1000,
40         "cant_epochs": 1000,
41         "training_state": 2,
42         "begin_time": "{TinyDate}:2022-10-21T02:18:09.589691",
43         "end_time": "{TinyDate}:2022-10-21T02:20:15.966698",
44       },
45     },
46   }
47
48   images_db = {
49     "notprocessed": ["Amor", "Alegria"],
50     "processed": ["Tristeza", "Enojo", "Sorpresa", "Asco", "Miedo"],
51   }

```

Ilustración 43 Endpoint entrenamiento

Esta petición recibe lo que serían la cantidad de épocas que se van a requerir del entrenamiento, como primera instancia se revisa que llegue en el *headers* lo que sería el token de autorización si este no es igual al de “fake_secret_token”, significa que hay un problema de permisos, también se revisa que la cantidad de épocas deba ser mayor a 1 si no pues se menciona el mensaje de error, llegado el caso que todo esté en orden se crea toda la estructura de un modelo nuevo para ello se hace el llamado al *ModelVariables*, se guarda este nuevo registro en el objeto de simulación de la base de datos terminando exitosamente con el mensaje que el entrenamiento inicio correctamente.

```

56     @router.get("/train/{epochs}")
57     async def train_model(epochs: int, req: Request):
58         if req.headers["Authorization"] != fake_secret_token:
59             raise HTTPException(status_code=401, detail="Unauthorized")
60         if epochs < 1:
61             raise HTTPException(status_code=422, detail="La cantidad de épocas debe ser mayor que 1")
62
63         today = datetime.now()
64         num_model = "3"
65
66         model_variables = ModelVariables(
67             id=num_model,
68             name=f"{num_model}-Modelo_{today.strftime('%Y-%m-%d_%H-%M-%S')}",
69             Loss=0.0,
70             val_Loss=0.0,
71             accuracy=0.0,
72             val_accuracy=0.0,
73             mean_time_execution=0.0,
74             epoch=0,
75             cant_epochs=epochs,
76             training_state=TrainingState.CREATED,
77             begin_time=datetime.now(),
78             end_time=datetime.now(),
79         )
80
81         mock_db["models"][str(num_model)] = model_variables.dict()
82
83     return {"proceso de entrenamiento iniciado con éxito"}

```

Ilustración 44 Endpoint entrenamiento parte 2

- Endpoint información del entrenamiento

Esta petición recibe como parámetro el identificador del entrenamiento que está en proceso, como primera instancia se revisa que llegue en el *headers* lo que sería el token de autorización si este no es igual al de “fake_secret_token”, significa que hay un problema de permisos, se revisa si el identificador del modelo se encuentra en nuestra base de datos ficticia llegado caso que la respuesta es no, significa que no se encuentra el modelo con el identificador especificado. Si, si se encuentra entonces de devuelve todo el objeto que contiene la información del modelo con ese identificador.

```

86     @router.get("/training-info/{id_training}")
87     async def get_training_info(id_training: str, req: Request):
88         if req.headers["Authorization"] != fake_secret_token:
89             raise HTTPException(status_code=401, detail="Unauthorized")
90         if id_training not in mock_db["models"].keys():
91             raise HTTPException(status_code=404, detail="No se encontró el entrenamiento con el id especificado")
92
93     return mock_db["models"][id_training]

```

Ilustración 45 Endpoint informacion del entrenamiento

- Endpoint información del entrenamiento sin identificador

Esta petición revisa que llegue en el *headers* lo que sería el token de autorización si este no es igual al de “fake_secret_token”, significa que hay un problema de permisos, por otro lado, si el tamaño de los registros de modelos es igual a cero, se menciona que no se encontraron entrenamientos.

```
95  @router.get("/training-info")
96  async def get_training_info_actual(req: Request):
97      if req.headers["Authorization"] != fake_secret_token:
98          raise HTTPException(status_code=401, detail="Unauthorized")
99      if len(mock_db["models"]) == 0:
100          raise HTTPException(status_code=404, detail="No se encontraron entrenamientos")
101      for _, v in reversed(mock_db["models"].items()):
102          return v
```

Ilustración 46 Endpoint informacion del entrenamiento sin identificar

- EndPoint estado del entrenamiento.

Esta petición revisa que llegue en el *headers* lo que sería el token de autorización si este no es igual al de “fake_secret_token”, significa que hay un problema de permisos, si no, se devuelve el objeto de imágenes que aún no se han procesado.

```
104
105  @router.get("/train-state")
106  async def get_train_state(req: Request):
107      if req.headers["Authorization"] != fake_secret_token:
108          raise HTTPException(status_code=401, detail="Unauthorized")
109      return images_db["notprocessed"]
```

Ilustración 47 Endpoint estado del entrenamiento

- EndPoint modelo actual

Esta petición revisa como primera instancia se revisa que llegue en el *headers* lo que sería el token de autorización si este no es igual al de “fake_secret_token”, significa que hay un problema de permisos, por otro lado, si el tamaño de los registros de modelos es igual a cero se menciona que no se encontraron entrenamientos, la finalidad de esta prueba es devolver el último modelo de la base de datos que tenga el estado de entrenamiento en finalizado es decir que sea el numero 3.

```

112 @router.get("/actual-model")
113 async def get_actual_model(req: Request):
114     if req.headers["Authorization"] != fake_secret_token:
115         raise HTTPException(status_code=401, detail="Unauthorized")
116     if len(mock_db["models"]) == 0 or all(
117         callback["training_state"] != 3 for callback in mock_db["models"].values()
118     ):
119         raise HTTPException(status_code=404, detail="No se encontró ningún modelo entrenado")
120
121     # Get only models with training_state = 3 (trained)
122     trained_models = [model for model in mock_db["models"].values() if model["training_state"] == 3]
123
124     return trained_models[-1]

```

Ilustración 48 Endpoint modelo actual

- Simular petición información del entrenamiento.

Para realizar la simulación de que el cliente realiza la petición de obtener información del entrenamiento, se declara el cliente que es un *TestClient*, en ella se declara la variable *response*, donde se especifica lo que sería la url donde se hace el llamado al servidor y en el objeto *headers* se tiene el token necesario ya que de esta forma se determina si el usuario tiene permiso para realizar la solicitud, si el estatus del *response* es 200 (éxito), se verifica que el identificador sea el dos, el nombre del modelo sea “prueba2” y el estado de entrenamiento sea el 2.

```

12 def test_get_training_info():
13     response = client.get(
14         url="/training-info",
15         headers={"Authorization": "secret"},
16     )
17     assert response.status_code == 200
18     response = response.json()
19     assert response["id"] == "2"
20     assert response["name"] == "prueba2"
21     assert response["training_state"] == TrainingState.PROCESSING
22

```

Ilustración 49 Simulación de una petición de información del entrenamiento

- Simular petición información del entrenamiento por identificador.

Para realizar la simulación de que el cliente realiza la petición de obtener información del entrenamiento, se declara el cliente que es un *TestClient*, en ella se declara la variable *response*, donde se especifica lo que sería la url donde se hace el llamado al servidor y en el objeto *headers* se tiene el token necesario ya que de esta forma se determina si el usuario tiene permiso para realizar la solicitud, si el estatus del *response* es 200 (exito), se verifica que el identificador del *response* sea el uno, el nombre del modelo sea “prueba1” y que el estado del entrenamiento esté finalizado.

```
24 def test_get_training_info_by_id():
25     response = client.get(
26         url="/training-info/1",
27         headers={"Authorization": "secret"},
28     )
29     assert response.status_code == 200
30     response = response.json()
31     assert response["id"] == "1"
32     assert response["name"] == "prueba1"
33     assert response["training_state"] == TrainingState.FINISHED
34
```

Ilustración 50 simulación de una petición de información del entrenamiento por identificador

- Simular petición información del entrenamiento sin autorización.

Para realizar la simulación de que el cliente realiza la petición de obtener información del entrenamiento, se declara el cliente que es un *TestClient*, en ella se declara la variable *response*, donde se especifica lo que sería la url donde se hace el llamado al servidor y en el objeto *headers* se tiene el token necesario ya que de esta forma se determina si el usuario tiene permiso para realizar la solicitud, en este caso el token no es permitido, se verifica que el estatus del *response sea el 401*, y que se tenga el objeto con el detalle de no autorización.

```
36 def test_get_training_info_unauthorized():
37     response = client.get(
38         url="/training-info/1",
39         headers={"Authorization": "wrong"},
40     )
41     assert response.status_code == 401
42     assert response.json() == {"detail": "Unauthorized"}
43
```

Ilustración 51 Simulación de una petición de información del entrenamiento sin autorización

- Simular petición información del entrenamiento no encontrado.

Para realizar la simulación de que el cliente realiza la petición de obtener información del entrenamiento, se declara el cliente que es un *TestClient*, en ella se declara la variable *response*, donde se especifica lo que sería la url donde se hace el llamado al servidor y en el objeto *headers* se tiene el token necesario ya que de esta forma se determina si el usuario tiene permiso para realizar la solicitud, se verifica que el estatus del *response sea el 404 un error de sintaxis y que el objeto response se detalle la razón de error*.

```
45 def test_get_training_info_not_found():
46     response = client.get(
47         url="/training-info/100",
48         headers={"Authorization": "secret"},
49     )
50     assert response.status_code == 404
51     assert response.json() == {"detail": "No se encontró el entrenamiento con el id especificado"}
52
53
```

Ilustración 52 Simulación de una petición de información del entrenamiento no encontrado

- Simular petición obtener estado del entrenamiento.

Para realizar la simulación de que el cliente realiza la petición de obtener el estado del entrenamiento, se declara el cliente que es un *TestClient*, en ella se declara la variable *response*, donde se especifica lo que sería la url donde se hace el llamado al servidor y en el objeto *headers* se tiene el token necesario ya que de esta forma se determina si el usuario tiene permiso para realizar la solicitud, se verifica que el estatus del *response* sea el 200 que significa que fue exitoso y que el tamaño del repsonse sea de dos y que se contenga lo que sería las señas de Amor y Alegría en el *response*.

```
54 def test_get_train_state():
55     response = client.get(
56         url="/train-state",
57         headers={"Authorization": "secret"},
58     )
59     assert response.status_code == 200
60     response = response.json()
61     assert len(response) == 2
62     assert "Amor" in response
63     assert "Alegria" in response
```

Ilustración 53 simulación de una petición de estado del entrenamiento

- Simular petición obtener estado del entrenamiento no autorizado

Para realizar la simulación de que el cliente realiza la petición de obtener el estado del entrenamiento, se declara el cliente que es un *TestClient*, en ella se declara la variable *response*, donde se especifica lo que sería la url donde se hace el llamado al servidor y en el objeto *headers* se tiene el token necesario ya que

de esta forma se determina si el usuario tiene permiso para realizar la solicitud, se verifica que el estatus del *response* sea el 401 y que el objeto del *response* contenga como detalle la no autorización.

```
● ● ●  
66 def test_get_train_state_unauthorized():  
67     response = client.get(  
68         url="/train-state",  
69         headers={"Authorization": "wrong"},  
70     )  
71     assert response.status_code == 401  
72     assert response.json() == {"detail": "Unauthorized"}
```

Ilustración 54 Simulación de una petición de estado del entrenamiento no autorizado

- Simular obtener modelo actual.

Para realizar la simulación de que el cliente realiza la petición de obtener el modelo actual, se declara el cliente que es un *TestClient*, en ella se declara la variable *response*, donde se especifica lo que sería la url donde se hace el llamado al servidor y en el objeto *headers* se tiene el token necesario ya que de esta forma se determina si el usuario tiene permiso para realizar la solicitud, se verifica que el estatus del *response* sea el 200 que significa que fue exitoso, que en *response* se tenga el identificador igual 1, que el nombre del modelo sea “prueba1” y que el estado del entrenamiento esté finalizado.

```
● ● ●  
75 def test_get_actual_model():  
76     response = client.get(  
77         url="/actual-model",  
78         headers={"Authorization": "secret"},  
79     )  
80     assert response.status_code == 200  
81     response = response.json()  
82     assert response["id"] == "1"  
83     assert response["name"] == "prueba1"  
84     assert response["training_state"] == TrainingState.FINISHED
```

Ilustración 55 Simulación obtener modelo actual

- Simular obtener modelo actual sin autorización

Para realizar la simulación de que el cliente realiza la petición de obtener el modelo actual, se declara el cliente que es un *TestClient*, en ella se declara la variable *response*, donde se especifica lo que sería la url donde se hace el llamado al servidor y en el objeto *headers* se tiene el token necesario ya que de esta forma se determina si el usuario tiene permiso, en este caso el token no es el idóneo para realizar la

solicitud, por ende, se verifica que el estatus del *response* sea el 401 y que el objeto del *response* contenga como detalle la no autorización.

```
● ● ●  
87 def test_get_actual_model_unauthorized():  
88     response = client.get(  
89         url="/actual-model",  
90         headers={"Authorization": "wrong"},  
91     )  
92     assert response.status_code == 401  
93     assert response.json() == {"detail": "Unauthorized"}
```

Ilustración 56 Simulación obtener modelo actual sin autorización

- Simular obtener modelo actual no encontrado

Para realizar la simulación de que el cliente realiza la petición de obtener el modelo actual, se declara el cliente que es un *TestClient*, en ella se declara la variable *response*, donde se especifica lo que sería la url donde se hace el llamado al servidor y en el objeto *headers* se tiene el token necesario ya que de esta forma se determina si el usuario tiene permiso para realizar la solicitud, se verifica que el estatus del *response* sea el 404 un error de sintaxis y que el objeto *response* se detalle la razón de error.

```
● ● ●  
96 def test_get_actual_model_not_found():  
97     # Change all training states to error  
98     change_all_models_to_error()  
99  
100    response = client.get(  
101        url="/actual-model",  
102        headers={"Authorization": "secret"},  
103    )  
104    assert response.status_code == 404  
105    assert response.json() == {"detail": "No se encontró ningún modelo entrenado"}
```

Ilustración 57 Simulación obtener modelo actual no encontrado

- Simular petición de entrenamiento

Para realizar la simulación de que el cliente realiza la petición de hacer el entrenamiento, se declara el cliente que es un *TestClient*, en ella se declara la variable *response*, donde se especifica lo que sería la url donde se hace el llamado al servidor y en el objeto *headers* se tiene el token necesario ya que de esta forma se determina si el usuario tiene permiso para realizar la solicitud, se verifica que el estatus del *response* sea el 200 que significa que fue exitoso y se espera que el *response* tenga como mensaje que el entrenamiento comenzó correctamente.

```
108 def test_train_model():
109     response = client.get(
110         url="/train/100",
111         headers={"Authorization": "secret"},
112     )
113     assert response.status_code == 200
114     response = response.json()
115     assert response[0] == "proceso de entrenamiento iniciado con éxito"
116
```

Ilustración 58 Simulación petición de entrenamiento

- Simular petición de entrenamiento sin autorización.

Para realizar la simulación de que el cliente realiza la petición de hacer el entrenamiento, se declara el cliente que es un *TestClient*, en ella se declara la variable *response*, donde se especifica lo que sería la url donde se hace el llamado al servidor y en el objeto *headers* se tiene el token necesario ya que de esta forma se determina si el usuario tiene permiso para realizar la solicitud, en este caso el token no es el idóneo, por ende, se verifica que el estatus del *response* sea el 401 y que el objeto del *response* contenga como detalle la no autorización.

```
118 def test_train_model_unauthorized():
119     response = client.get(
120         url="/train/100",
121         headers={"Authorization": "wrong"},
122     )
123     assert response.status_code == 401
124     assert response.json() == {"detail": "Unauthorized"}
```

Ilustración 59 Simulación petición de entrenamiento sin autorización

- Endpoint usuario

Para entender de forma sintetizada en que consiste esta prueba, se empieza con declarar lo que sería un falso token “fake_secret_token”, el mock tiene como intención simular el objeto de la base de datos donde esta contiene lo que sería la información de los usuarios que se tienen en simulación.

```
4 fake_secret_token = "secret"
5
6 users_db = {
7     "1": {
8         "username": "admin@admin.com",
9         "password": "admin",
10        "role": "admin",
11        "name": "admin",
12    }
13 }
```

Ilustración 60 Endpoint usuario

- Endpoint registro

Se realiza toda la configuración de la ruta donde para esta no es necesario tener presente el token, pero si se tiene en cuenta la información como es el correo, nombre, rol y contraseña recibida de la petición, se toma el objeto mock específicamente lo que sería el correo del usuario del primer registro y si este es igual al de la petición, se muestra el error de que esta cuenta ya existe en el sistema, en caso contrario, se crea de manera satisfactoria el usuario.

```
18 @router.post("/signup")
19 def signup(user_details: UserModel):
20     if user_details.username == users_db["1"]["username"]:
21         raise HTTPException(status_code=409, detail="Cuenta ya existe")
22     return "Usuario creado Correctamente"
23
```

Ilustración 61 Endpoint registro

- Endpoint ingresar al sistema.

Se realiza toda la configuración de la ruta donde para esta no es necesario tener presente el token, pero si se tiene en cuenta la información como es el correo y contraseña recibida de la petición, se toma el objeto mock específicamente lo que sería el correo del usuario del primer registro si este no es igual significa que el usuario no se ha registrado en el Sistema, si las contraseñas no coinciden, se notifica que

la contraseña es incorrecta, por último, si no hay ningún inconveniente con los campos y validaciones, se retorna la estructura como es la generación del token, el rol del usuario y el nombre de él.

```
● ● ●
25 @router.post("/login")
26 def login(user_details: UserLogin):
27     if user_details.username != users_db["1"]["username"]:
28         raise HTTPException(status_code=409, detail="Usuario no existe")
29     if user_details.password != users_db["1"]["password"]:
30         raise HTTPException(status_code=409, detail="Contraseña incorrecta")
31     return {"access_token": fake_secret_token, "role": users_db["1"]["role"], "name": users_db["1"]["name"]}
32
```

Ilustración 62 Endpoint ingresar al sistema

- Simular petición registro.

Para realizar la simulación de que el cliente realiza la petición de registrarse, se declara el cliente que es un *TestClient*, en ella se declara la variable *response*, donde se especifica lo que sería la url donde se hace el llamado al servidor y el JSON con la información, se verifica que el estatus del *response* sea el 200 que significa que fue exitoso y que el usuario se creó el correo correctamente.

```
● ● ●
12 def test_signup():
13     response = client.post(
14         url="/signup",
15         json={
16             "username": "test@test.com",
17             "password": "test",
18             "role": "user",
19             "name": "test",
20         },
21     )
22     assert response.status_code == 200
23     response = response.json()
24     assert response == "Usuario creado Correctamente"
25
```

Ilustración 63 Simular petición registro

- Simular petición registro ya existe

Para realizar la simulación de que el cliente realiza la petición de registrarse, se declara el cliente que es un *TestClient*, en ella se declara la variable *response*, donde se especifica lo que sería la url donde se hace

el llamado al servidor y el JSON con la información, se verifica que el estatus del *response sea el 409 que significa un conflicto* y esto a su vez se verifica el objeto del response que detalla la razón de error.



```
27 def test_signup_user_already_exists():
28     response = client.post(
29         url="/signup",
30         json=UserModel(
31             username="admin@admin.com",
32             password="admin",
33             role="admin",
34             name="admin",
35         ).dict(),
36     )
37     assert response.status_code == 409
38     assert response.json() == {"detail": "Cuenta ya existe"}
39
```

Ilustración 64 Simular petición registro ya existe

- Simular petición ingresar al Sistema.

Para realizar la simulación de que el cliente realiza la petición de ingresar al Sistema, se declara el cliente que es un *TestClient*, en ella se declara la variable *response*, donde se especifica lo que sería la url donde se hace el llamado al servidor y el JSON con la información, se verifica que el estatus del *response sea el 200 que significa que fue exitoso y que el usuario ingreso correctamente al Sistema*.

```
41 def test_login():
42     response = client.post(
43         url="/login",
44         json={
45             "username": "admin@admin.com",
46             "password": "admin",
47         },
48     )
49     assert response.status_code == 200
50     response = response.json()
51     assert response["access_token"] == "secret"
52     assert response["role"] == "admin"
53     assert response["name"] == "admin"
```

Ilustración 65 Simular petición ingreso al sistema

- Simular petición ingresar al Sistema usuario no existe.

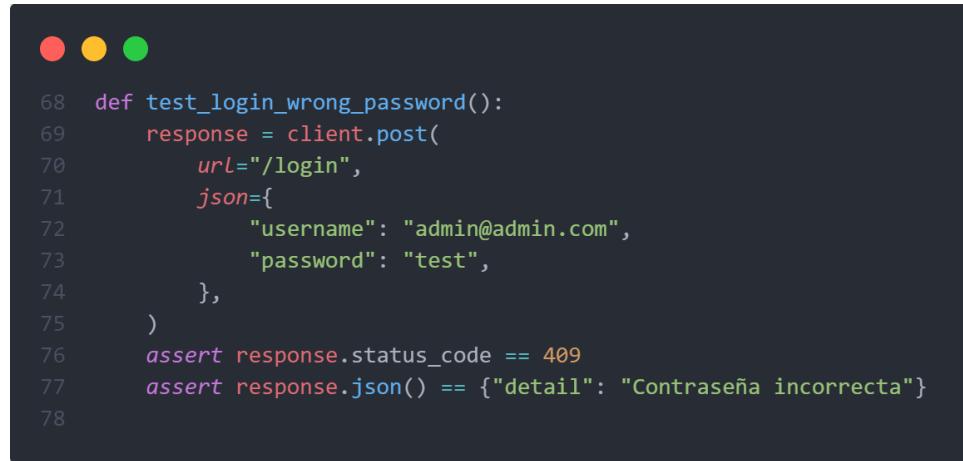
Para realizar la simulación de que el cliente realiza la petición de ingresar al Sistema, se declara el cliente que es un *TestClient*, en ella se declara la variable *response*, donde se especifica lo que sería la url donde se hace el llamado al servidor y el JSON con la información, se verifica que el estatus del *response* sea el 409 que significa un conflicto y esto a su vez se verifica el objeto del *response* que detalla la razón de error con el mensaje de usuario no existente.

```
55
56 def test_login_user_not_exists():
57     response = client.post(
58         url="/login",
59         json={
60             "username": "test",
61             "password": "test",
62         },
63     )
64     assert response.status_code == 409
65     assert response.json() == {"detail": "Usuario no existe"}
66
```

Ilustración 66 Simular petición ingresar al sistema usuario no existe

- Simular petición ingresar al Sistema con contraseña incorrecta

Para realizar la simulación de que el cliente realiza la petición de ingresar al Sistema, se declara el cliente que es un *TestClient*, en ella se declara la variable *response*, donde se especifica lo que sería la url donde se hace el llamado al servidor y el JSON con la información, se verifica que el estatus del *response* sea el 409 que significa un conflicto y esto a su vez se verifica el objeto del *response* que detalla la razón de error con el mensaje contraseña incorrecta.



```
68 def test_login_wrong_password():
69     response = client.post(
70         url="/login",
71         json={
72             "username": "admin@admin.com",
73             "password": "test",
74         },
75     )
76     assert response.status_code == 409
77     assert response.json() == {"detail": "Contraseña incorrecta"}  
78
```

Ilustración 67 Simular petición ingresar al sistema con contraseña incorrecta