Informe de Control de Calidad y Pruebas del Software para SolidarianID

Control de Calidad y Pruebas del Software

Máster Universitario en Ingeniería del Software

Autores:

Hernán Salambay Roldán
Alejandro Montoya Toro
Pedro Nicolás Gomariz
Aurora Hervás López
Dongyue Yu

15 de Enero de 2025

Facultad de Informática
UMU

Contenidos

1 Introducción

El control de calidad y las pruebas de software son pilares fundamentales en el desarrollo de sistemas confiables y eficientes. Mientras que el control de calidad se enfoca en establecer estándares, procesos y buenas prácticas para garantizar que un producto cumpla con los requisitos y expectativas del cliente, las pruebas de software son una herramienta clave dentro de este proceso, permitiendo verificar y validar que el sistema funcione correctamente bajo diferentes condiciones.

El plan de pruebas es un componente esencial dentro del control de calidad, ya que proporciona una estrategia detallada para llevar a cabo el proceso de pruebas de manera estructurada y eficiente.

Este documento define los objetivos, el alcance, los casos de prueba, los recursos y los criterios de aceptación necesarios para garantizar la calidad del software. Un buen plan de pruebas no solo identifica posibles defectos antes de la implementación, sino que también reduce riesgos, optimiza el tiempo de desarrollo y mejora la experiencia del usuario final.

A continuación, se elaborará un plan de pruebas para el proyecto SolidarianID, aplicando los principios del desarrollo ágil, donde la calidad y la iteración constante son fundamentales. En el contexto ágil, las pruebas de software se integran desde las primeras fases del desarrollo y evolucionan junto con el proyecto, permitiendo detectar defectos de manera temprana y adaptarse a cambios en los requisitos. Este plan de pruebas está diseñado para garantizar que las funcionalidades de SolidarianID cumplan con los estándares de calidad esperados, alineándose con los objetivos del proyecto y maximizando la satisfacción del usuario final.

2 Plan de Pruebas

Proyecto SolidarianID

Plan de Pruebas

Preparado por: Hernán Salambay Roldán, Alejandro Montoya Toro, Pedro Nicolás Gomariz, Aurora Hervás López, Dongyue Yu

2.1 Introducción

El plan de pruebas propuesto tiene como objetivo definir y desarrollar un conjunto de pruebas destinadas a garantizar la calidad de la aplicación, abordando tanto su front-end como back-end, dentro de un entorno cliente-servidor. Como equipo de desarrollo, este enfoque busca identificar y prevenir posibles errores en el funcionamiento de la aplicación, asegurando una experiencia sólida y confiable para los usuarios.

En esta etapa inicial, el plan se enfocará en un conjunto limitado de funcionalidades o componentes clave, sirviendo como modelo para implementar una metodología de trabajo que pueda ser replicada y ampliada a otras áreas de la aplicación. Además, este documento establece una base para definir el alcance de las pruebas, así como consideraciones de infraestructura, riesgos asociados y supuestos que influyen en su implementación.

2.2 Recursos Humanos

En un plan de pruebas, los Recursos Humanos representan los roles y responsabilidades necesarias para llevar a cabo las pruebas del proyecto. Estos roles pueden incluir:

- **Ingenieros de Software**: responsables de diseñar y ejecutar pruebas unitarias e integrarlas en el código
- **Testers**: enfocados en ejecutar pruebas funcionales, de rendimiento y reportar resultados.
- **Scrum Masters**, quienes facilitan la coordinación y priorización de pruebas dentro del equipo ágil.
- **Clientes o Representantes de usuarios**: participan en las pruebas de aceptación para validar que el producto cumple con sus expectativas.

Cada rol tiene un nivel de implicación definido, asegurando que las pruebas cubran todas las necesidades del proyecto de manera eficiente. La siguiente tabla muestra el porcentaje de cada rol dedicado al proceso de prueba dentro del proyecto:

Rol	**Tarea**	**Participación en pruebas**
Cliente	Involucrado en las pruebas de aceptación y proporcionando feedback sobre la experiencia del usuario y funcionalidad.	10%
Scrum master	Facilitar la adquisición de requisitos y comunicación con el cliente para que se cumpla con las expectativas del usuario, buscando la optimalidad en el consumo de recursos en la implementación de los requisitos.	25%
Tester/QA	Enfocado en la planificación, ejecución y reporte de pruebas funcionales, de rendimiento y de carga.	50%
Ingeniero de software	Responsable de desarrollar las funcionalidades y corregir los errores, además de las pruebas dedicadas en las etapas de desarrollo (unitarias e integración)	100%

2.3 Alcance

En esta versión, el plan de prueba se centra en los siguientes componentes:

- **Funcionalidades del MVP**: implementación de todas las características esenciales necesarias para el producto mínimo viable.
- **Interfaz de usuario**: desarrollo y validación de una interfaz responsive diseñada específicamente para el administrador de la plataforma.
- **Pruebas presentadas en clase**: ejecución de las pruebas requeridas según los lineamientos establecidos en la materia:
    - **Pruebas unitarias**: incluye pruebas de caja blanca y pruebas de caja negra.
    - **Pruebas de mutación**
    - **Pruebas de frontend**
    - **Pruebas de backend**: incluye las pruebas unitarias, pruebas de endpoint y pruebas de integración.
    - **Pruebas end-to-end**

- **Pruebas de rendimiento**
- **Pruebas metamórficas**
- **Pruebas de regresión**

2.4 Fuera del Alcance

Las pruebas de base de datos, las pruebas no funcionales adicionales como las pruebas relacionadas con la seguridad o accesibilidad quedarán fuera del alcance en esta primera versión debido a la falta de tiempo. Esta decisión permite priorizar las funcionalidades clave del MVP y centrarse en una entrega más rápida y eficiente en el inicio del desarrollo.

2.5 Pruebas Unitarias

Las pruebas unitarias son una técnica para verificar el funcionamiento correcto de unidades específicas de código, como funciones o métodos. Se enfocan en asegurar que cada parte del software se comporta correctamente de forma independiente. Estas pruebas suelen ser automáticas y ayudan a detectar errores en etapas tempranas del desarrollo. Se pueden llevar a cabo mediante enfoques de caja blanca, que examinan la lógica interna del código, o de caja negra, que evalúan los resultados sin necesidad de conocer su implementación interna.

2.5.1 Técnicas de caja blanca

Las técnicas de caja blanca son métodos de prueba de software que requieren acceso al código fuente del programa. Estas pruebas se enfocan en verificar la estructura interna del sistema, evaluando el flujo de control, las decisiones condicionales y el comportamiento de las funciones.

Dentro de las técnicas de caja blanca, utilizaremos tres enfoques claves: camino básico, condición y bucle, que explicaremos a continuación.

2.5.1.1 Pruebas de camino básico

Estas pruebas se enfocan en identificar y probar los caminos fundamentales del código, asegurando que todas las rutas posibles sean cubiertas. Su objetivo es garantizar que cada ruta independiente en el flujo de ejecución del programa se ejecute al menos una vez.

Hemos elegido la función de actualizar el perfil de un usuario, *updateUser* (Figura 1), para aplicar este tipo de prueba.

```
async updateUser(id: string, email: string, bio: string): Promise<void> {
    // Find the existing user
S1    const existingUser = await this.userRepository.findById(id);
    // Check if the email is different
C1    if (existingUser.email === email) {
S2      throw new EmailUpdateConflictError();
    }
    // Check if the email is already in use
C2    try {
S3      await this.userRepository.findByEmail(email);
      // If the email is found, then it is already in use
S4     throw new EmailAlreadyInUseError(email);
C3    } catch (error) {
      // If the error is not an EntityNotFoundError, then throw it
C4      if (!(error instanceof EntityNotFoundError)) {
```

```
S5          throw error;
        }
    }
    // Update the user
S6  existingUser.updateProfile({ email, bio });
S7  await this.userRepository.save(existingUser);
}
```

Figura 1. Código *updateUser*

A continuación, presentamos el diagrama de flujo para el código anterior (Figura 2). Antes de empezar a definir las pruebas, calculamos la complejidad ciclomática del código. Para ello, hemos usado la siguiente fórmula de V(G) = arcos - nodos + 2 y obtenemos que la complejidad ciclomática es 4:

$$V(G) = 14 - 12 + 2 = 4$$

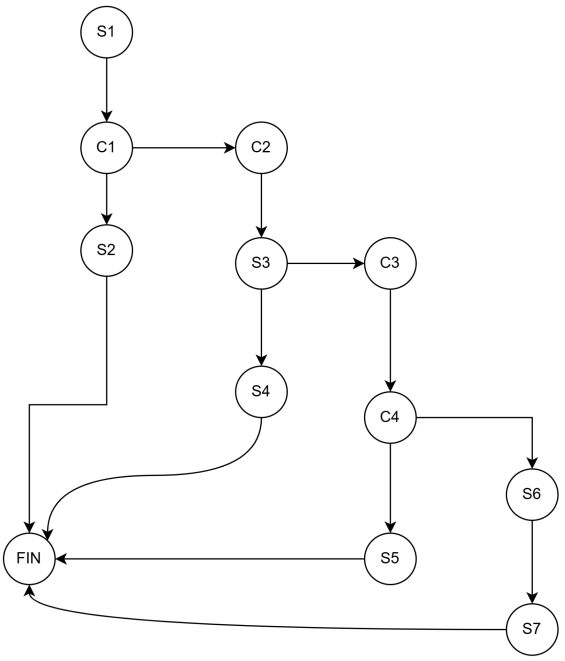


Figura 2. Diagrama de flujo de *updateUser*

Con el diagrama de la Figura 2 podemos ver que existen 4 caminos:

- **Camino 1: S1 → C1 → S2 → FIN**

  Test#1
  Entrada:*exsistingUser* (email: prueba@example.com, …)
  (“d290f1ee-6c54-4b01-90e6-d701748f0”, prueba@example.com, “Hello”)
  Salida: *EmailUpdateConflictError*

- **Camino 2: S1 → C1 → C2 → S3 → S4 → FIN**

  Test#2
  Entrada: *userA*(email: prueba@example.com, …)
  (“d290f1ee-6c54-4b01-90e6-d701748f085”, prueba@example.com, “Hello”)
  Salida: *EmailAlreadyInUseError*

- **Camino 3: S1 → C1 → C2 → S3 → C3 → C4 → S5 → FIN**

Test#3 Este caso no debería de darse, en caso de darse debería de ser un error fuera del contexto del código.
Entrada: (“d290f1ee-6c54-4b01-90e6-d701748f085”, prueba@example.com, “Hello”)
Salida: error

- **Camino 4: S1 → C1 → C2 → S3 → C3 → C4 → S6 → S7 → FIN**

  Test#4
  Entrada: *existingUser* (email: old.email@example.com)
  (“d290f1ee-6c54-4b01-90e6-d701748f085”, new.email@example.com, “Update Bio”)
  Salida: 204 No Content

Podemos observar que, con los cuatro caminos definidos, se requieren al menos tres pruebas para garantizar la cobertura de la función. Sin embargo, es importante mencionar un punto adicional: existe un camino especial, S1→ FIN, que no se refleja en los caminos previamente definidos. Esto se debe a que, en caso de fallo, la excepción redirige el flujo de ejecución a otro nivel.

La implementación de las pruebas mencionadas se encuentra en el proyecto SolidarianID, en la siguiente ruta: ‘backend/apps/users-ms/test/application/user.service.spec.ts’. A continuación, en la Figura 3, se presenta la implementación correspondiente al **Test #4**.

```
it('updates a user profile successfully', async () => {
  // Arrange
  const existingUser = await createMockUser({
    id: new UniqueEntityID('1234'),
    email: 'old.email@example.com',
  });
  userRepositoryMock.findById.mockResolvedValueOnce(existingUser);
  userRepositoryMock.findByEmail.mockRejectedValueOnce(
    new EntityNotFoundError('User not found'),
  );

  // Act
  await userService.updateUser(
    '1234',
    'new.email@example.com',
    'Updated bio',
  );

  // Assert
  expect(userRepositoryMock.findById).toHaveBeenCalledWith('1234');
  expect(userRepositoryMock.findByEmail).toHaveBeenCalledWith(
    'new.email@example.com',
  );
  expect(existingUser.updateProfile).toHaveBeenCalledWith({
    email: 'new.email@example.com',
    bio: 'Updated bio',
  });
  expect(userRepositoryMock.save).toHaveBeenCalledWith(existingUser);
});
```

Figura 3. Implementación Test#4

2.5.1.2 Pruebas de condición

Las pruebas de condición son técnicas utilizadas para evaluar si se cumple una determinada condición o conjunto de condiciones dentro de un sistema, permitiendo tomar decisiones en función de resultados verdaderos o falsos. En programación, se implementan con estructuras

como *if* o *switch* para controlar el flujo del código, y en pruebas de software, aseguran que el sistema responda correctamente ante distintos escenarios.

Partiendo del código de ejemplo de la Figura 1, correspondiente al método *updateUser*, tenemos las siguientes condiciones:

- **C1**: *if (existingUser.email === email)*

  Esta condición tiene dos posibles resultados: true o false

  - El caso true ya está cubierto por el **Test#1**.
  - El caso false está cubierto por los **Test#2**, **Test#3** y **Test#4**.

- **C2**: *try-catch*

  Eel bloque *try*, donde no ocurre ningún error, está cubierto por el **Test#3**. Para el bloque *catch*, que se ejecuta cuando ocurre un error, está cubierto por el **Test#4**.

- **C3**: *try-catch* (excepciones en el *catch*)

  Para esta condición, se espera que entre en el *catch*. Sin embargo, como se mencionó previamente, esto no sucederá en el contexto actual.

- **C4**: *if (!(error instanceof EntityNotFoundError)*

  Esta condición también tiene dos posibles valores: true o false

  - El caso true está cubierto por el **Test#3**.
  - El caso false está cubierto por el **Test#4**.

2.5.1.3 Pruebas de bucle

Las pruebas de bucle son un tipo de pruebas de software que se centran en verificar el comportamiento de los bucles dentro del código, asegurándose de que iteran correctamente sobre los elementos y realizan las acciones esperadas durante cada repetición. El propósito de estas pruebas es garantizar que los bucles funcionen de acuerdo con lo esperado en diferentes escenarios y que no se presenten errores como bucles infinitos, saltos incorrectos o iteraciones faltantes.

En el código de nuestro proyecto, hay pocos bucles. Se ha elegido el fragmento de código de la Figura 4 para mostrar el uso de esta prueba.

```
public addOds(ods: ODSEnum | Set<ODSEnum>): void {
  if (ods instanceof Set) {
    ods.forEach((item) => this.ods.add(item));
  } else {
    this.ods.add(ods);
  }
}
```

Figura 4. Código con bucle.

Para el código anterior, no hemos implementado el test dentro del proyecto, en caso de implementarlo sería algo parecido a esto:

- Cuando la entrada es un set vacío.

```
describe('when handling an empty Set', () => {
      it('should not add any values', () => {
        const odsSet = new Set<ODSEnum>(); // empty set
        odsService.addOds(odsSet); // Attempt to add an empty set

        expect(odsService.getOds().size).toBe(0); // Verify that nothing was
added
      });
    });
```

- Cuando hay un elemento en el set.

```
describe('addOds()', () => {
    describe('when adding a single ODS value', () => {
      it('should add the value to the Set', () => {
        odsService.addOds(ODSEnum.Item1);
        expect(odsService.getOds().has(ODSEnum.Item1)).toBe(true);
      });
    });
```

- Cuando hay varios elementos en el set.

```
describe('when adding multiple ODS values from a Set', () => {
      it('should add all values to the Set', () => {
        const odsSet = new Set<ODSEnum>([
          ODSEnum.Item1,
          ODSEnum.Item2,
          ODSEnum.Item3,
        ]);

        odsService.addOds(odsSet);
        expect(odsService.getOds().has(ODSEnum.Item1)).toBe(true);
        expect(odsService.getOds().has(ODSEnum.Item2)).toBe(true);
        expect(odsService.getOds().has(ODSEnum.Item3)).toBe(true);
      });
    });
```

2.5.2 Técnicas de caja negra

Las pruebas de caja negra son un tipo de prueba de software que evalúa la funcionalidad de un sistema o componente sin considerar cómo está implementado internamente. Es decir, estas pruebas se enfocan exclusivamente en las entradas y las salidas del sistema, sin tener acceso al código fuente o la lógica interna.

A continuación, se profundiza en dos técnicas fundamentales utilizadas dentro de las pruebas de caja negra: **particiones de equivalencia** y **análisis de valores límite**, las cuales permiten diseñar casos de prueba efectivos al identificar subconjuntos representativos de entradas y verificar el comportamiento del sistema en los límites de estas.

Para realizar las pruebas hemos elegido la función que crea la contraseña de un usuario cuando se registra. En la Figura 6, podemos ver la implementación de esta función, el *PASSWORD_PATTERN* corresponde a la expresión regular para validar si la contraseña es

válida o inválida. Una contraseña es válida si al menos tiene 8 caracteres, incluyendo al menos una mayúscula, una minúscula, un dígito y un carácter especial.

```
public static async create(value: string): Promise<UserPassword> {
  if (!PASSWORD_PATTERN.test(value)) {
    throw new InvalidPasswordError();
  }

  const hashedPassword = await bcrypt.hash(value, PASSWORD_HASH_SALT_ROUNDS);
  return new UserPassword({ password: hashedPassword });
}
```

Figura 6. Función de creación de contraseñas.

2.5.2.1 Particiones de equivalencia

Las pruebas de partición de equivalencia son una técnica de pruebas de caja negra que se basa en dividir el conjunto de entradas posibles en particiones o grupos que se comportan de manera similar. La idea es que si una entrada pertenece a una partición válida, cualquier valor de esa partición debería ser tratado de la misma manera por el sistema. De igual forma, para particiones inválidas, cualquier valor en la partición debe desencadenar un comportamiento de error similar.

Para la función de *create* de la contraseña podemos definir las siguientes particiones:

- **Particiones válidas**
    - Contraseña que tiene una longitud mínima de 8 caracteres, incluyendo una mayúscula, una minúscula, un número y un carácter especial: *123456Test**.
- **Particiones inválidas** (solo se citarán algunas, ya que combinando los patrones pueden salir 32 combinaciones)
    - Contraseña de menos de 8 caracteres: *Pass1**
    - Contraseña con solo números: *12345678*
    - Contraseña con solo letras: *password*
    - Contraseña con solo caracteres especiales: *****@@!!*

En la Figura 7 podemos ver un ejemplo de implementación de la partición válida. Las demás implementaciones de las particiones se encuentran en la siguiente ruta del proyecto: ‘backend/apps/users-ms/test/domain/password.spec.ts’.

```
it('Should accept a valid password that meets the pattern', async () => {
  const validPassword = '123456Test*';

  const userPassword = await UserPassword.create(validPassword);

  expect(userPassword).toBeInstanceOf(UserPassword);
});
```

Figura 7. Test de partición válida

2.5.2.2 Análisis de valores límite

El análisis de valores límite es una técnica de pruebas que se enfoca en verificar el comportamiento del sistema en los límites de las particiones de equivalencia. En lugar de probar solo un valor dentro de una partición válida o inválida, se prueban los valores en los extremos de estas particiones (por ejemplo, el valor más bajo y más alto permitido, o el valor justo fuera de los límites).

Para la función de *create* de la contraseña podemos definir los siguientes AVL de dos puntos:

- **Límite inferior**: contraseña exactamente 8 carácteres que cumple con el patrón (*123Test**)
- **Límite inferior fuera de rango**: contraseña con 7 carácteres (*123Test*).

Si queremos que sea AVL de tres puntos, podríamos añadir este caso:

- Contraseña con 9 caracteres: *1234Test**

En la Figuras 8 se muestra la implementación del límite inferior y el límite inferior fuera de rango. De acuerdo con el patrón utilizado, no existen límites superiores para la longitud de la contraseña. Los tests correspondientes están implementados en el proyecto, en la siguiente ruta: 'backend/apps/users-ms/test/domain/password.spec.ts'.

```
describe('Password length boundary tests', () => {
  it('Should accept a valid password with an exact length of 8 characters', async () => {
    const validPassword = '123Test*';

    const userPassword = await UserPassword.create(validPassword);

    expect(userPassword).toBeInstanceOf(UserPassword);
  });

  it('Should throw InvalidPasswordError for passwords with less than 8 characters', async () => {
    const shortPassword = '123Tes*';

    await expect(UserPassword.create(shortPassword)).rejects.toThrow(
      InvalidPasswordError,
    );
  });
});
```

Figura 8. Test de valores límite

2.6 Pruebas de mutación

Una vez definidos los tests unitarios, y como complemento a las pruebas realizadas en el módulo de usuarios, se llevaron a cabo pruebas de mutación para evaluar la efectividad de los casos de prueba y garantizar la calidad del sistema. Estas pruebas consisten en modificar deliberadamente partes del código ("mutantes") para verificar si los tests existentes son capaces de detectarlas. Un mutante eliminado indica que el caso de prueba asociado es eficaz, mientras que uno superviviente señala posibles mejoras en la cobertura de las pruebas.

2.6.1 Stryker

Para ejemplificar este método de pruebas, utilizamos la herramienta Stryker, que ofrece soporte para la generación automática de mutantes a partir de los métodos y validaciones implementados. Stryker nos permite evaluar y detectar áreas que requieren un mayor nivel de cobertura de una manera simple desde su interfaz web. Su capacidad de configuración es ideal para proyectos complejos como SolidarianID.

stryker.config.json > ...

You, 25 seconds ago | 1 author (You)

```
{
  "$schema": "./node_modules/@stryker-mutator/core/schema/stryker-schema.json",
  "_comment": "This config was generated using 'stryker init'. Please take a look at: https://stryker-mutator.io/do
  "packageManager": "npm",
  "reporters": ["html", "clear-text", "progress", "dashboard"],
  "testRunner": "jest",
  "testRunner_comment": "Take a look at https://stryker-mutator.io/docs/stryker-js/jest-runner for information abou
  "coverageAnalysis": "perTest",
  "mutate": [
    "apps/*/src/**/*.ts",
    "libs/*/src/**/*.ts",
    "!**/*.{spec,module,dto,enum,service,repository,mapper,guard,decorator}.ts",
    "!**/{envs,main,index,constant}.ts",
    "!**/*config*.ts"
  ],
  "ignoreStatic": true
}
```

Figura 9: Configuración de Stryker

En la Figura 9 podemos ver la configuración de Stryker. Esta configuración optimiza el análisis al enfocarse en los archivos clave dentro de las carpetas *apps* y *libs*, excluyendo elementos que no son relevantes para las pruebas, como archivos de configuración o definición de constantes. La opción *ignoreStatic: true* permite omitir los elementos estáticos, reduciendo así falsos positivos y concentrando el análisis en el código más relevante. Asimismo, se utiliza el análisis de cobertura por test (*coverageAnalysis: "perTest"*), lo que facilita identificar con precisión qué mutantes son detectados por cada caso de prueba. Con estos ajustes, Stryker nos genera un informe detallado que incluye un puntaje de mutación y aspectos clave para la mejora de nuestras pruebas de una manera intuitiva.

2.6.2 Mutación

Nuestro proceso de "mutación" y refinamiento de las pruebas definidas se centra principalmente en el módulo de usuarios, abarcando las funcionalidades del servicio UserService y las entidades del dominio correspondientes. En las siguientes secciones, detallamos los resultados iniciales obtenidos, las estrategias adoptadas para mejorar las pruebas y los beneficios alcanzados tras la eliminación de los mutantes supervivientes.

Podemos iniciar este análisis observando, mediante el dashboard de Stryker, las estadísticas de nuestras pruebas unitarias definidas previamente (Figura 10), enfocándonos en el archivo 'user.service.impl.ts'. Obtuvimos un Mutation Score del 100%, con 19 mutantes eliminados y 5 marcadas como "Timeout", sin supervivientes ni líneas sin cobertura. Estos resultados confirman la efectividad de nuestras estrategias de pruebas unitarias para el servicio.

File / Directory	Mutation Score		Killed	Survived	Timeout	No coverage	Ignored	Runtime errors	Compile errors	Detected	Undetected	Total
	Of total	Of covered										
TS user.service.impl.ts	100.00	100.00	19	0	5	0	0	0	0	24	0	24

Killed (19) Timeout (5)

Figura 10: Estadísticas de las pruebas del servicio de usuarios.

En contraste, en el resto del módulo se identificaron otros apartados que requieren nuestra atención, como se puede observar en la Figura 11:

File / Directory	Mutation Score		Killed	Survived	Timeout	No coverage	Ignored	Runtime errors	Compile errors	Detected	Undetected	Total
	Of total	Of covered										
domain	86.54	96.77	90	3	0	11	8	0	0	90	14	112
TS Password.ts	72.73	100.00	8	0	0	3	0	0	0	8	3	11
TS User.ts	84.38	94.74	54	3	0	7	0	0	0	54	10	64
TS UserBirthDate.ts	95.00	100.00	19	0	0	1	0	0	0	19	1	20
TS UserEmail.ts	100.00	100.00	9	0	0	0	8	0	0	9	0	17

Figura 11: Estadísticas del dominio del módulo usuarios.

Donde podemos observar que para el dominio de nuestro módulo usuarios, se obtuvo un Mutation Score total del 86.54%, con resultados positivos como el archivo *UserEmail.ts*, que alcanzó un **100% de mutantes eliminados**, mientras que la entidad de valor *Password.ts* muestra posibles áreas de mejora con un score de **72.73%,** que es el porcentaje resultante de haber eliminado 8 de los 11 mutantes encontrados**.** Estos resultados nos indican dónde debemos centrar nuestros esfuerzos para optimizar la cobertura y fortalecer la calidad global de las pruebas en estos casos.

2.6.3 Análisis y refinamiento de pruebas.

2.6.3.1 UserBirthDate

Podemos iniciar este análisis con un caso sencillo, el cual es el que encontramos en la entidad de valor *UserBirthDate*, para el cual habíamos observado un 95% de mutantes eliminados. Inspeccionando su archivo de mutaciones, podemos observar aquellos que hayan quedado sin eliminar. En este caso concreto, en la Figura 12 podemos observar que solamente ha quedado un mutante sin eliminar:

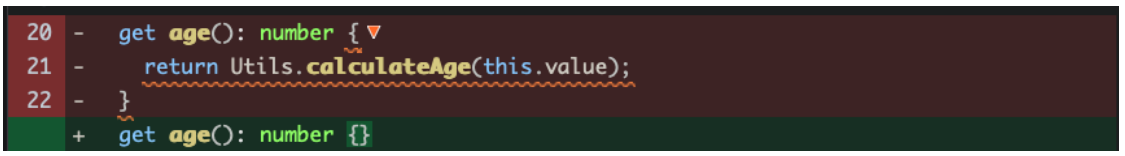


Figura 12: Mutante sin eliminar ni cubrir.

En la Figura 13 podemos ver que este mutante se produce cuando la obtención de la edad no retorna ningún valor. Este caso lo podemos eliminar sencillamente, incluyendo algún *assert* que verifique que la edad retornada por la entidad de valor corresponde con la fecha de nacimiento con la que fue inicializada.

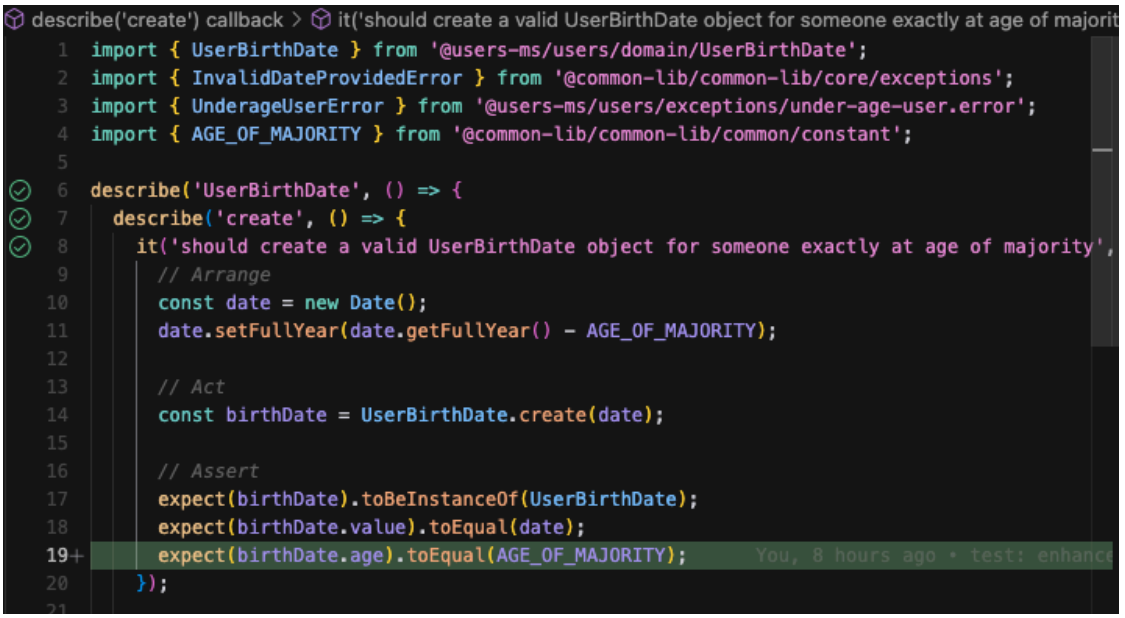


Figura 13: Test de *userBirthDate*.

Asserts como los incluidos al test anterior eliminarían al mutante, como se puede observar en la Figura 14:

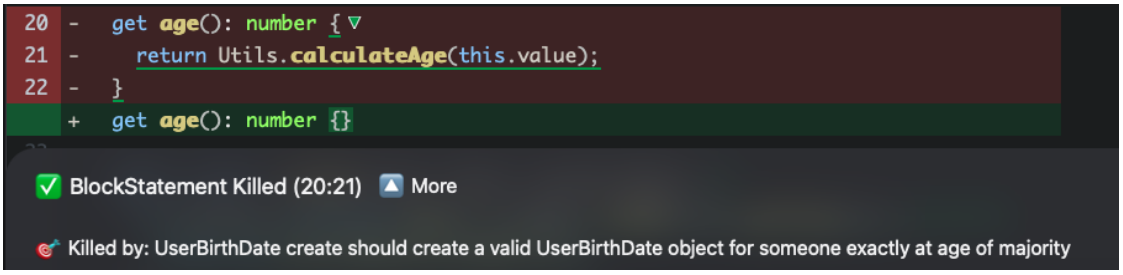


Figura 14: Eliminación del mutante.

Puliendo y ampliando el alcance de las pruebas para esta entidad, en la Figura 15 observamos que se nos había pasado por alto comprobar que la edad representada fuera correcta en cada test. Eliminando dicho mutante, obtenemos un 100% de mutantes eliminados para este archivo

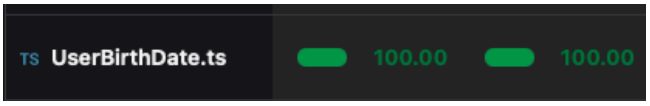


Figura 15: Porcentaje de mutantes eliminados (Mutation score).

2.6.3.2 User

En cuanto a la clase *User*, tenemos unos cuantos mutantes más por tratar. El resumen inicial es el siguiente: tenemos 54 mutantes eliminados, 3 supervivientes y 7 que no han sido tratados, como podemos observar en la Figura 16

Figura 16: Resumen inicial.

En este ejemplo, nos centraremos en los supervivientes, ya que los casos de *NoCoverage* son ejemplos sencillos como los anteriores (*getter* y *setter*), que se han eliminado agregando *asserts* a los tests existentes. En la figura 17 podemos ver los mutantes supervivientes que se encuentran en los métodos *create* y *updateProfile.*

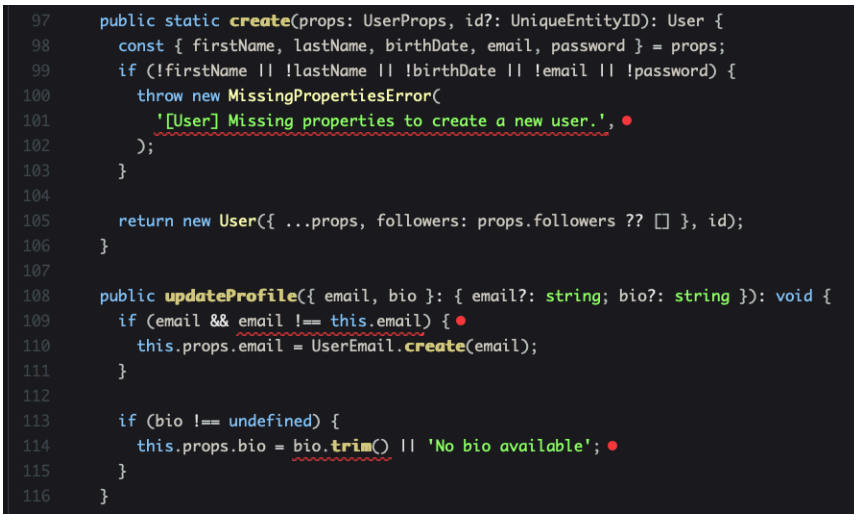


Figura 17: Mutantes supervivientes

Stryker nos indica tres casos de mutantes que no se han tratado correctamente:

- El caso en el que se retorna un mensaje de error incorrecto.
- El caso en el que el email siempre se considera igual al anterior.
- El tercero es una mutación que provoca que no se realice el *trim* de la biografía facilitada.

El primer caso se resuelve definiendo una prueba que verifique que el mensaje retornado sea el esperado (Figura 18).

```
118   it('should throw MissingPropertiesError with correct message', async () => {
119     // Arrange
120     const props = await createValidProps();
121     delete props.firstName;
122
123     // Act & Assert
124     expect(() => User.create(props)).toThrow(
125       '[User] Missing properties to create a new user.',
126     );
127   });
```

Figura 18: Solución del primer caso.

El segundo caso se soluciona implementando una prueba que verifique que, si el *email* es igual al anterior, no se ejecute el contenido del *if* (Figura 19).

```
185   it('should not update email when same email is provided', async () => {
186     // Arrange
187     const user = User.create(await createValidProps());
188     const originalEmail = user.email;
189
190     // Act
191     user.updateProfile({ email: originalEmail });
192
193     // Assert
194     expect(user.email).toBe(originalEmail);      You, 21 hours ago • feat: enhance user validation and testing …
195   });
196
197   it('should not call UserEmail.create if the new email is the same as the current email', async () => {
198     // Arrange
199     const user = User.create(await createValidProps());
200     const originalEmail = user.email;
201
202     const createSpy = jest.spyOn(UserEmail, 'create');
203
204     // Act
205     user.updateProfile({ email: originalEmail });
206
207     // Assert
208     expect(createSpy).not.toHaveBeenCalled();
209     expect(user.email).toBe(originalEmail);
210   });
```

Figura 19: Solución del segundo caso.

El caso de la Figura 19 es más rebuscado, ya que, aunque se había implementado un test que verifica que, al introducir un *email* idéntico al actual, se mantuviera el anterior, el test solo comprobaba si el *email* del usuario actualizado coincidía con el anterior. Este test era de baja calidad, ya que, si el nuevo *email* es idéntico al anterior, el test siempre pasaría.

El tercer y último caso se resuelve verificando que efectivamente se realice el *trim* de la biografía, una validación que se omitió en las pruebas iniciales (Figura 20)

```
208    it('should trim bio when updating', async () => {
209      // Arrange
210      const user = User.create(await createValidProps());
211      const bioWithSpaces = '  bio with spaces  ';      You, 2 minutes ago •
212
213      // Act
214      user.updateProfile({ bio: bioWithSpaces });
215
216      // Assert
217      expect(user.bio).toBe('bio with spaces');
218    });
219  });
```

Figura 20: Solución del tercer caso.

Una vez definidos estos nuevos casos obtenemos un 100% de mutantes eliminados (Figura 21).

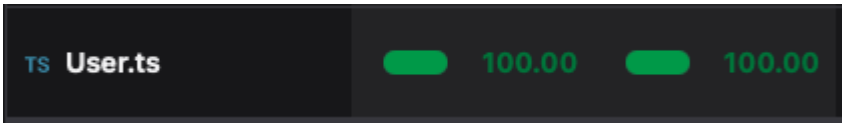


Figura 21: Porcentaje de mutantes eliminados.

2.6.3.3 UserPassword

En este ejemplo se detecta un porcentaje mayor de mutantes sin tratar. Sin embargo, al analizarlo en detalle, observamos que únicamente hay 3 mutantes sin cubrir (Figura 22).

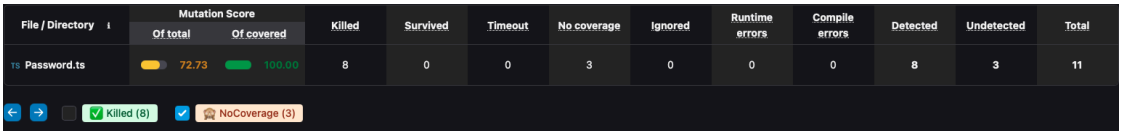


Figura 22: Estado inicial del mutante.

En la Figura 23 podemos ver que los mutantes que están sin cubrir se encuentran en las funciones de *fromHashedPassword* y *compare*.

```
31    public static fromHashedPassword(value: string): UserPassword {
32      return new UserPassword({ password: value });
33    }
34
35    public async compare(password: string): Promise<boolean> {
36      return bcrypt.compare(password, this.value);
37    }
```

Figura 23: Funciones con mutantes sin cubrir.

Los mutantes representan los casos en los que la implementación es vacía para ambas funciones y el caso en el que se crea un *UserPassword* con un valor vacío. Siguiendo la misma estrategia que en los ejemplos anteriores, estos mutantes se eliminarían añadiendo nuevos casos de prueba, ya que no fueron contemplados en los tests iniciales.

```
describe('UserPassword fromHashedPassword method', () => {
  it('Should create password from hashed value', () => {
    // Arrange
    const hashedPassword = '$2b$10$someHashedPasswordValue';

    // Act
    const userPassword = UserPassword.fromHashedPassword(hashedPassword);

    // Assert
    expect(userPassword).toBeInstanceOf(UserPassword);
    expect(userPassword.value).toBe(hashedPassword);
  });
});

describe('UserPassword compare method', () => {
  it('Should correctly compare matching passwords', async () => {
    // Arrange
    const plainPassword = '123456Test*';
    const userPassword = await UserPassword.create(plainPassword);

    // Act
    const isMatch = await userPassword.compare(plainPassword);

    // Assert
    expect(isMatch).toBe(true);
  });

  it('Should correctly compare non-matching passwords', async () => {
    // Arrange
    const originalPassword = '123456Test*';
    const wrongPassword = '123456Test#';
    const userPassword = await UserPassword.create(originalPassword);

    // Act
    const isMatch = await userPassword.compare(wrongPassword);

    // Assert
    expect(isMatch).toBe(false);
  });
```

Figura 24: Test iniciales de *UserPassword*.

Los tests anteriores nos aseguran que la contraseña creada a partir de un valor *hasheado* realmente contenga el valor *hasheado*, además de comprobar los casos en los que dos contraseñas deben coincidir, devolviendo *true*, y los casos en los que no deben coincidir, devolviendo *false*. Una vez analizados todos estos casos, hemos eliminado todos los mutantes del módulo usuarios, logrando un dominio limpio y cubriendo los casos límite previamente no tratados (Figura 25).

File / Directory	Mutation Score		Killed	Survived	Timeout	No coverage	Ignored	Runtime errors	Compile errors	Detected	Undetected	Total
	Of total	Of covered										
domain	100.00	100.00	104	0	0	0	8	0	0	104	0	112
Password.ts	100.00	100.00	11	0	0	0	0	0	0	11	0	11
User.ts	100.00	100.00	64	0	0	0	0	0	0	64	0	64
UserBirthDate.ts	100.00	100.00	20	0	0	0	0	0	0	20	0	20
UserEmail.ts	100.00	100.00	9	0	0	0	8	0	0	9	0	17

Figura 25: Informe final.

Finalmente, gracias a Stryker, hemos detectado diversos mutantes supervivientes que evidenciaron la falta de *asserts* y casos de prueba clave. Al incorporar estas mejoras, logramos eliminar todos los mutantes y, al mismo tiempo, incrementamos la calidad y el alcance de nuestras pruebas, asegurando un correcto funcionamiento en escenarios límite y reforzando la fiabilidad de las funcionalidades evaluadas.

2.7 Pruebas Frontend

Las pruebas de frontend son un conjunto de técnicas y procesos diseñados para verificar que la interfaz de usuario de una aplicación web o móvil funcione correctamente y cumpla con las expectativas del usuario. Estas pruebas garantizan que los componentes visuales, la lógica de interacción y la experiencia del usuario final sean consistentes, funcionales y libres de errores.

Para realizarlos utilizamos **Jest-DOM**, una extensión de Jest que proporciona una serie de *matchers* (métodos de aserción) específicos para probar interfaces de usuario en aplicaciones web.

Debido a la limitada implementación del frontend en esta etapa del desarrollo, se han realizado pocas pruebas enfocadas en esta parte. La mayoría de las pruebas realizadas hasta ahora se han centrado principalmente en verificar aspectos básicos, como comprobar si un elemento u objeto existe en el DOM, validar si contiene un valor esperado, o confirmar atributos y contenido textual. Para ello, se han utilizado métodos de aserción como *toHaveValue*, *toHaveAttribute*, *toBeInTheDocument* y *toHaveTextContent*. En la Figura 26 se muestra un ejemplo de test usando el método de aserción *toHaveTextContent*.

```
46  it('should display the Welcome message when the user is authenticated', () => {
47    // Set the user data
48    const user = {
49      firstName: 'John',
50      lastName: 'Doe',
51      roles: ['user'],
52    };
53
54    // Render the template with the user data
55    const htmlContent = renderTemplate({ user });
56    document.body.innerHTML = htmlContent;
57
58    // Verify that the Welcome message appears
59    expect(document.body).toHaveTextContent('Welcome, John Doe [ user ]');
60  });
61 });
```

Figura 26. Test frontend

2.8 Pruebas Backend

Las pruebas de backend son pruebas diseñadas para garantizar que el código que maneja la lógica del servidor (la capa del backend) funcione de la manera esperada. Estas pruebas se utilizan para verificar que los controladores, servicios, repositorios, y otros componentes funcionen correctamente. A continuación se explican con más detalle las pruebas implementadas para esta parte.

2.8.1 Pruebas unitarias

Las pruebas unitarias se centran en probar funciones o métodos individuales de forma aislada. Aseguran que una función o un servicio específico haga lo que se espera de él, sin depender de otros componentes. Estas pruebas se utilizan para comprobar la lógica interna de clases o funciones.

En el caso del módulo de *users*, se ha implementado este tipo de pruebas para garantizar que la lógica relacionada con la gestión de usuarios funcione correctamente de forma aislada. Las pruebas unitarias para el módulo de *users* aseguran que los métodos del servicio *UserService*, así como los controladores que gestionan las solicitudes relacionadas con los usuarios, se comporten de manera esperada.

En el apartado de **Pruebas unitarias**, se explica con detalle cómo se ha implementado esta serie de pruebas. Además, se proporciona un análisis de cómo se gestionan las entradas y salidas en el método probado, asegurando que la lógica de negocio detrás de la gestión de usuarios esté libre de errores y funcione independientemente de otras partes del sistema.

2.8.2 Pruebas de endpoint

Las pruebas de endpoint verifican el correcto funcionamiento de los puntos finales (endpoints) de una API. Se realizan simulando solicitudes HTTP (como GET, POST, PUT, DELETE, etc.) y comprobando que las respuestas sean correctas, tanto en el código de estado como en el

contenido. El objetivo es asegurar que los endpoints gestionen adecuadamente las entradas del usuario, interactúen correctamente con el servidor y la base de datos, y devuelvan los resultados esperados. Estas pruebas pueden ser manuales o automatizadas y son esenciales para garantizar que una API funcione correctamente en condiciones reales.

En este caso, hemos implementado este tipo de pruebas utilizando archivos *.rest* ubicados en la carpeta *request/* dentro del proyecto. Las pruebas de endpoints con archivos .rest permiten realizar pruebas manuales de la API de manera rápida y eficiente directamente desde el editor de código, sin necesidad de escribir código adicional. La Figura 27 muestra un ejemplo.

```
@baseUrl = http://localhost:3000/api/v1/users

### Login success
Send Request
POST  {{baseUrl}}/auth/login
Content-Type: application/json

{
  "email": "admin@example.com",
  "password": "123456Test*"
}
```

Figura 27. Prueba del endpoint /users/auth/login con rest

2.8.3 Pruebas de integración en NestJS

Las pruebas de integración, que en NestJS se denominan end-to-end, cubren la interacción de las clases y módulos de la misma forma que lo haría el usuario final en el sistema de producción. Para simular las peticiones HTTP y poder definir casos de prueba automáticos, usamos la librería **supertest**.

Durante las pruebas de integración, se debe probar que el acceso a la base de datos se hace correctamente. Para no utilizar la base de datos de producción pero trabajar en el mismo contexto en el que estaría trabajando la aplicación real, configuramos una versión de la base de datos para pruebas. Lo conseguimos definiendo una cadena de conexión a la base de datos distinta para el entorno de pruebas, como se observa en la Figura 28.

```
NODE_ENV=test

# Backend
COMMUNITIES_MS_HOST=localhost
COMMUNITIES_MS_PORT=3002

# MongoDB
MONGO_HOST=localhost
MONGO_PORT=27017
MONGO_DB=solidarianid-test
MONGO_URI=mongodb://${MONGO_HOST}:${MONGO_PORT}/${MONGO_DB}
```

Figura 28. Conexión a la base de datos de pruebas en *.env.test*

Definimos el fichero *actions.e2e-spec.ts* para los tests de integración del módulo actions. En este fichero, en primer lugar incluimos el hook *beforeAll* con la creación del módulo y la base de datos (Figura 29), además del vaciado de la base de datos y la inserción de algunos datos de prueba.

```
// Setup the app and the action model
beforeAll(async () => {
  const moduleFixture: TestingModule = await Test.createTestingModule({
    imports: [
      ActionModule,
      ConfigModule.forRoot(),
      MongooseModule.forRoot(process.env.MONGO_URI),

      MongooseModule.forFeature([
        { name: 'Action', schema: Persistence.ActionSchema },
      ]),
    ],
  }).compile();
```

Figura 29. Hook *beforeAll* para las pruebas del módulo *actions*

Por otra parte, en el hook *afterAll*, cerramos la conexión con la base de datos y la aplicación, como se indica en la Figura 30.

```
// Close the app and the database connection
afterAll(async () => {
  await app.close();
  await mongoose.connection.close();
});
```

Figura 30. Hook *afterAll* para las pruebas del módulo *actions*

Una vez configurado el módulo de pruebas, definimos los casos de prueba para los endpoints. En primer lugar, la solicitud GET del endpoint '*/actions/:id*', que devuelve una acción dado su identificador. Para probar este endpoint, utilizamos uno de los *ids* de los datos de prueba insertados en la base de datos, que se han almacenado en la variable *predefinedActionsIds*. Hacemos referencia a la aplicación que hemos creado para simular la petición HTTP, con la función *request* de supertest y definimos expectativas sobre el resultado de la petición que se ha ejecutado, este test se incluye en la Figura 31. También se realiza una prueba con un identificador inválido, comprobando que la petición devuelve '*Not Found*'. De manera similar, tenemos el test para el endpoint GET '*/actions*' que devuelve todas las acciones.

```
describe('GET /actions/:id', () => {
  it('should return action details for a valid id', async () => {
    const id = predefinedActionsIds[0];
    const response = await request(app.getHttpServer())
      .get(`/actions/${id}`)
      .set('Authorization', `Bearer ${token}`);
    expect(response.status).toBe(HttpStatus.OK);
    expect(response.body).toHaveProperty('id', predefinedActionsIds[0]);
    expect(response.body).toHaveProperty('title', 'Action 1');
  });
  it('should return 404 for an invalid id', async () => {
    const id = new UniqueEntityID().toString();
    const response = await request(app.getHttpServer())
      .get(`/actions/${id}`)
      .set('Authorization', `Bearer ${token}`);
    expect(response.status).toBe(HttpStatus.NOT_FOUND);
  });
});
```

Figura 31. Prueba del endpoint GET /actions/:id

Por otra parte, para probar el endpoint POST '*actions/:id/contributions*', definimos el payload de la contribución que queremos añadir y simulamos la petición con supertest. Entre los asertos definidos, se encuentra la comprobación de que el estado (status) de la acción ha pasado a *IN_PROGRESS.*

Por último, al probar el endpoint GET '*/actions/:id/contributions*', que devuelve todas las contribuciones de una acción, comprobamos que la acción cuyo id se ha usado anteriormente, tiene una única contribución.

Para ejecutar todos los test e2e, se define un fichero de configuración '*jest-e2e.json*' en el api-gateway, y se indica esta configuración en el script correspondiente del *'package.json*', junto con la variable de entorno *NODE_ENV* correspondiente (Figura 32).

```
"test:e2e": "cross-env NODE_ENV=test jest --config ./apps/api-gateway/test/jest-e2e.json",
```

Figura 32. Script definido para la ejecución de los test e2e

2.9 Pruebas End-to-End (e2e)

En las pruebas End-to-End, se debe probar todo el flujo de ejecución, desde que se insertan los datos en la interfaz gráfica, hasta que se envía la solicitud al backend y se recupera la información de la base de datos, integrando el backend y el frontend de la aplicación para comprobar que funcionan como se espera de extremo a extremo. Debido a la sencillez del frontend de la aplicación para el MVP, los tests e2e han permitido probar la funcionalidad del login y de validación de la creación de una comunidad.

Para realizar las pruebas End-to-End, se ha utilizado la aplicación **Cypress**. En primer lugar, configuramos en el fichero '*cypress.config.js*' la url base de la aplicación (Figura 33), para poder utilizarla durante los test poniendo '/'.

```
e2e: {
  baseUrl: 'http://localhost:3005',
},
```

Figura 33. Configuración de la url base de la aplicación.

Definimos los tests en el fichero '*solidarianId_test.cy.ts*' en la carpeta 'frontend/cypress/e2e'. Utilizamos el hook *beforeEach* para ejecutar el comando que visita la página principal de la aplicación y el que limpia las cookies (Figura 34).

```
beforeEach(() => {
  cy.visit('/');
  // Reset application state before each test
  cy.clearLocalStorage();
  cy.clearCookies();
});
```

Figura 34. Hook *beforeEach* para los test e2e

Una vez configurado todo, definimos el test mínimo que prueba que se abre la página principal de la aplicación y contiene el título '*Welcome to SolidarianID*'. También configuramos un test que, al acceder a la página principal, navega a la página de inicio de sesión e introduce sus credenciales, se comprueba que ha iniciado sesión correctamente si ahora aparece la opción de *'logout'* (Figura 35).

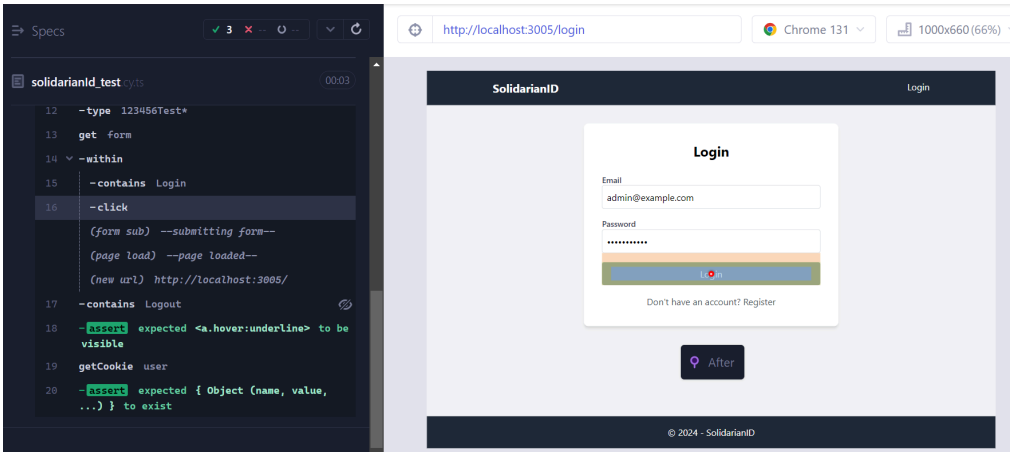


Figura 35. Test de login con Cypress.

Para los demás test, definimos un *command* en el fichero 'cypress/support/commands', que realice al backend la petición de inicio de sesión con las credenciales (Figura 36).

```
Cypress.Commands.add('login', (email: string, password: string) => {
  cy.request({
    url: '/authenticate',
    method: 'POST',
    body: { email: email, password: password },
  });
  cy.visit('/');
});
```

Figura 36. Command definido para realizar el login a la app.

Definimos es test e2e que permite rechazar la solicitud de creación de una comunidad. Para ello utilizamos el comando de *'login'* definido anteriormente para iniciar sesión directamente con una petición al backend. Además, se realiza una petición al backend para crear una nueva solicitud de creación de comunidad.

Una vez se ha iniciado sesión, se navega a la vista de solicitudes, (*Validation*), se selecciona la solicitud cuyo nombre coincide con la que se acaba de crear, se selecciona el checkbox y se pulsa el botón *Reject* para rechazar su creación. Se escribe el motivo del rechazo en el área de texto que aparece, se pulsa *Submit* y posteriormente se comprueba que no existe en la lista de solicitudes pendientes ninguna con ese nombre.

2.10 Pruebas de Rendimiento.

Las pruebas de rendimiento son un tipo de pruebas no funcionales, cuyo objetivo es evaluar la capacidad de un sistema para manejar solicitudes bajo diferentes condiciones de carga. Para llevar a cabo estas pruebas, se ha utilizado la herramienta **Apache JMeter**, que permite simular cargas de trabajo en aplicaciones web y APIs, configurando diferentes escenarios de carga, estableciendo métricas de rendimiento y analizando el comportamiento del sistema.

En este apartado, se ha definido un plan de pruebas de rendimiento (Figura 37) orientado a evaluar la funcionalidad específica de recuperar todas las causas solidarias desde la API del proyecto SolidarianID, combinado con el acceso a la página principal del frontend de la aplicación.

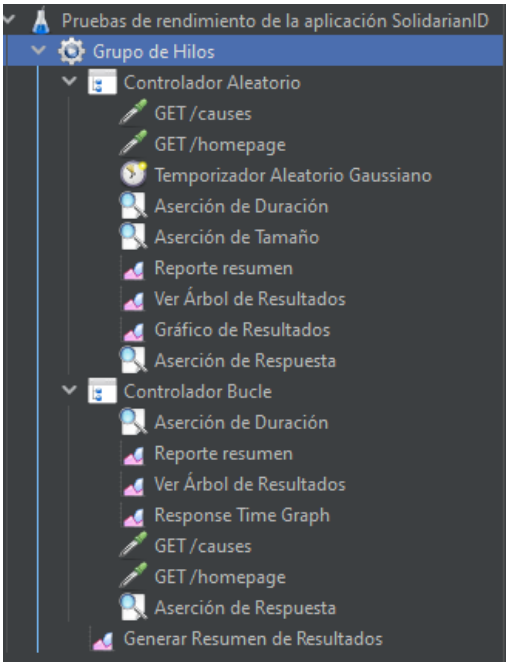


Figura 37. Plan de pruebas de rendimiento.

Para simular escenarios realistas y evaluar el rendimiento de la aplicación, se definieron varios elementos clave en JMeter. A continuación se describen los componentes utilizados en el plan de pruebas.

- **Grupo de Hilos**: se configura un grupo de 100 hilos con un periodo de subida de 5 segundos, para simular una carga progresiva y ver cómo el sistema maneja la carga creciente.
- **Petición HTTP**: se incluyen dos muestreadores de petición HTTP, una para la obtener el listado de causas solidarias, y otra para simular el acceso a la página principal del frontend de la aplicación.
- **Aserciones**:
    - **Aserción de Respuesta**: se utiliza para comprobar que el código de respuesta obtenido por las peticiones HTTP sea el *200 OK*, indicando que la solicitud se procesó correctamente.
    - **Aserción de Duración**: se configura para verificar que el tiempo de respuesta de cada petición no supere el límite de 1 segundo.
    - **Aserción de Tamaño**: Se asegura de que el tamaño de la respuesta no supere los 7kB, dado que la respuesta de la API incluirá un objeto JSON con un array de máximo 10 causas, además de los metadatos de paginación.
- **Temporizador Aleatorio Gaussiano** (desviación de 1 segundo y desplazamiento de medio segundo): simula el comportamiento de usuarios reales al introducir tiempos de espera aleatorios entre las solicitudes. La distribución gaussiana asegura que los retrasos no sean uniformes, sino que imiten más naturalmente los patrones de comportamiento de los usuarios.
- **Receptores**: recopilan y presentan datos de las peticiones que realizan los muestreadores.

- **Reporte resumen**: muestra un resumen detallado de las métricas clave como el número de peticiones exitosas, tiempos de respuesta, errores, y throughput.
- **Ver Árbol de Resultados**: proporciona un informe detallado de cada solicitud realizada, mostrando datos como el código de error, las cabeceras, y el cuerpo de la respuesta (Figura 38).
- **Gráfico de resultados**: presenta una representación gráfica de los tiempos de respuesta, la media, la mediana y la desviación, lo que ayuda a visualizar el comportamiento bajo diferentes niveles de carga.
- **Response Time Graph**: muestra cómo evolucionan los tiempos de respuesta a medida que se aumenta la carga.

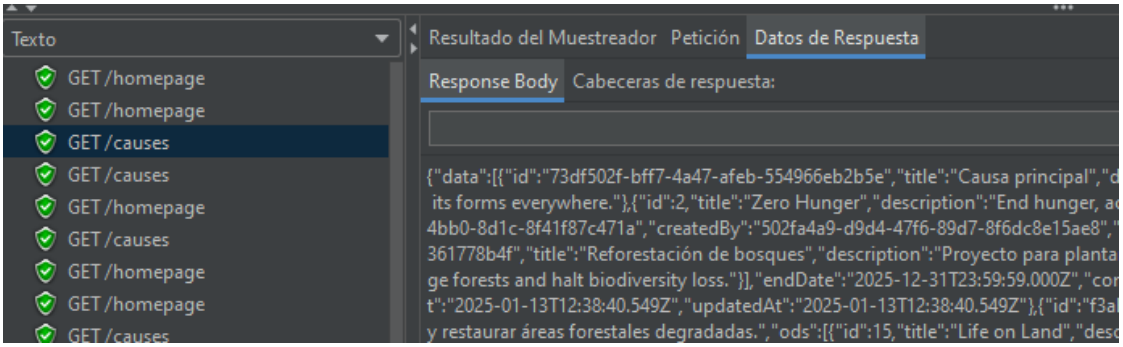
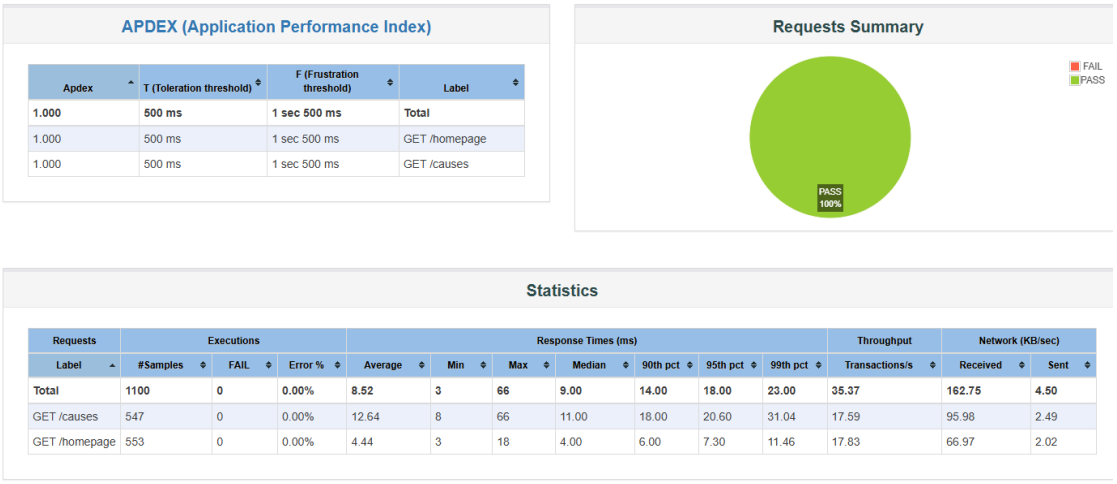


Figura 38. Árbol de resultados del plan de pruebas.

El plan de pruebas incluye dos tipos de controladores que simulan diferentes escenarios de uso:

- **Controlador Aleatorio**: simula el comportamiento de un usuario normal que interactúa aleatoriamente con las funcionalidades de la aplicación. Ejecuta de manera aleatoria las dos peticiones HTTP (recuperar causas y acceder a la página principal) y añade retrasos entre las solicitudes mediante el Temporizador Aleatorio Gaussiano. Este controlador permite emular la variabilidad y los patrones de comportamiento de los usuarios reales.
- **Controlador Bucle**: este controlador ejecuta las mismas peticiones, pero en un bucle de 10 repeticiones sin introducir tiempos de espera entre las solicitudes. Se utiliza para evaluar la capacidad de la aplicación bajo una carga más intensiva y sostenida, simulando un mayor volumen de solicitudes en un corto periodo de tiempo.

Una vez realizadas pruebas con la aplicación y viendo que pasan correctamente, seguimos las recomendaciones de ejecutar JMeter desde la línea de comandos por motivos de eficiencia. Aquí indicamos que se genere un informe csv así como un html con los resultados de la ejecución. Estos informes se encuentran en el repositorio del grupo, dentro del directorio ‘doc/test-reports/jmeter’.



APDEX (Application Performance Index)

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
1.000	500 ms	1 sec 500 ms	Total
1.000	500 ms	1 sec 500 ms	GET /homepage
1.000	500 ms	1 sec 500 ms	GET /causes

Requests Summary

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	1100	0	0.00%	8.52	3	66	9.00	14.00	18.00	23.00	35.37	162.75	4.50
GET /causes	547	0	0.00%	12.64	8	66	11.00	18.00	20.60	31.04	17.59	95.98	2.49
GET /homepage	553	0	0.00%	4.44	3	18	4.00	6.00	7.30	11.46	17.83	66.97	2.02

Figura 39. Informe html generado por las pruebas de rendimiento.

Observando la Figura 39, en la que se incluyen algunos resultados del html generado, vemos que todas las respuestas son exitosas y el plan de pruebas ha demostrado que la aplicación cumple con los límites establecidos en cuanto a tiempos de respuesta, tamaño de respuesta y experiencia de usuario (APDEX), para la funcionalidad probada.

Con un índice APDEX perfecto (1.000) y tiempos de respuesta muy bajos, el sistema es robusto y proporciona una experiencia óptima para los usuarios, ya que el 100% de las respuestas se encuentran dentro del umbral de tolerancia.

2.11 Pruebas Metamórficas

Las pruebas metamórficas son una técnica de pruebas de software utilizada para abordar problemas en casos en los que las pruebas tradicionales pueden ser difíciles de aplicar, como en sistemas complejos o aplicaciones que no tienen un "oráculo" claro para determinar si el resultado de una ejecución es correcto.

En este caso, las pruebas metamórficas se aplican al componente *CauseService*, específicamente en el método *getAllCauses*, que recupera y ordena una lista de causas almacenada en la BBDD. Las relaciones metamórficas empleadas están centradas en verificar propiedades clave del sistema relacionadas con el orden y la consistencia de los datos al aplicar filtros y ordenaciones. A continuación se detallan las relaciones metamórficas empleadas:

- **Consistencia de Ordenación:** Se asegura que la lista obtenida en orden ascendente sea la inversa exacta de la lista en orden descendente. Esto se verifica solicitando las causas ordenadas en ambas direcciones y comparando si invertir una lista produce la otra, validando así la consistencia en el comportamiento del sistema.

- **Idempotencia:** Comprueba que ejecutar la misma consulta con idénticos parámetros de ordenación y filtros siempre produce el mismo resultado. Esto se valida ejecutando el método *getAllCauses* repetidamente con los mismos argumentos y verificando que las respuestas coinciden, garantizando estabilidad ante consultas repetidas.

- **Ordenación Correcta:** Valida que el sistema ordene correctamente los resultados según lo solicitado, ya sea en orden ascendente o descendente. Esto se realiza comparando las listas devueltas por el método con listas previamente ordenadas localmente, asegurando que el sistema cumple con las expectativas de ordenamiento.

La implementación de esas pruebas se encuentra en la siguiente ruta: 'backend/apps/communities-ms/test/causes/application/cause.service.spec.ts'.

Hemos aplicado las pruebas metamórficas al *getAllCauses* de *CauseService* (Figura 40) debido a su responsabilidad. Dado que el filtrado y ordenamiento incorrectos pueden influir negativamente en la experiencia del usuario, validar este servicio es crucial para garantizar la calidad del sistema.

```
it('should return all causes in descending order', async () => {
  const sortedData = [...data.data].sort((a, b) =>
    b.title.localeCompare(a.title),
  );
  mockCauseRepository.findAll.mockResolvedValueOnce(sortedData);

  const result = await service.getAllCauses(
    undefined,
    undefined,
    CauseSortBy.TITLE,
    SortDirection.DESC,
  );

  expect(result.data).toEqual(sortedData);
});
```

Figura 40. Prueba metamórfica para *getAllCauses*.

2.12 Pruebas de Regresión

Las pruebas de regresión verifican que los cambios recientes en el código no hayan afectado negativamente las funcionalidades existentes, asegurando que el software siga funcionando correctamente después de modificaciones. De las tres estrategias, selección, minimización y priorización hemos escogido la estrategia de selección.

La estrategia de selección de pruebas de regresión consiste en elegir un conjunto representativo de pruebas que cubren las funcionalidades más críticas, en lugar de ejecutar todas las pruebas. Se seleccionan aquellas que validan las características esenciales que podrían verse afectadas por los cambios, como el método *getAllCauses* en cuanto a ordenación y paginación.

Hemos elegido esta estrategia porque es eficiente para desarrollo ágil, ya que reduce el tiempo de ejecución de las pruebas sin comprometer la calidad. Al centrarse en las pruebas más relevantes, como las de ordenación y filtrado de *getAllCauses*, se garantiza que los cambios no afecten a funcionalidades críticas sin perder cobertura.

En el proceso ágil, se integran pruebas de regresión mediante **CI/CD**. Cada vez que un desarrollador hace un cambio, las pruebas seleccionadas se ejecutan automáticamente en el pipeline de integración continua. Si las pruebas pasan, el código se despliega, y si fallan, se notifica al equipo de desarrollo para su corrección.

Este enfoque permite detectar errores rápidamente, asegurando que las funcionalidades clave, como la ordenación y paginación de *getAllCauses*, no se vean afectadas tras los cambios.

2.12 Pruebas de Aceptación de usuario

Las pruebas de aceptación de usuario se utilizan para validar si una aplicación cumple con los requisitos y expectativas del usuario final antes de ser lanzada al público.

En nuestro caso, estas pruebas tienen como objetivo asegurar que las funcionalidades principales de la aplicación sean intuitivas y funcionen según lo esperado. Debido a la

situación, no tenemos usuario finales como tal, pero en caso de simulación, para aplicar las pruebas de aceptación en SolidarianID, el proceso debe implicar lo siguiente:

- **Identificación de usuarios finales**: los usuarios que participen en las pruebas de aceptación deberían ser personas que representan los perfiles clave de la aplicación, como voluntarios, organizaciones sin fines de lucro o administradores de la plataforma.
- **Definir los criterios de aceptación**: los criterios de aceptación deben alinearse con las funcionalidades principales de la aplicación, tales como registro y autenticación de un usuario, visualización de comunidades, causas y acciones, etc.
- **Validación de la interfaz de usuario (UI)**: durante las pruebas, los usuarios finales también revisarán la interfaz para asegurar que sea intuitiva y accesible.
- **Feedback de los usuarios**: tras realizar las pruebas, los usuarios proporcionarán comentarios sobre la experiencia general, las posibles mejoras y las áreas de dificultad en la navegación o en el uso de funcionalidades.
- **Revisión de errores y ajustes**: cualquier error o inconsistencia que se detecte durante las pruebas se reportará al equipo de desarrollo para su corrección antes del lanzamiento final de la plataforma.

2.13 Consideraciones de Infraestructura

Para la ejecución de las pruebas unitarias no es necesario realizar configuraciones específicas del entorno, ya que estas pruebas se ejecutan de forma aislada y sin dependencias externas. Sin embargo, para las demás pruebas (como pruebas de integración, funcionales o de aceptación), es fundamental contar con un entorno bien configurado y preparado.

En este proyecto, se requiere tener instaladas herramientas clave como Docker, Docker Compose y Make. Estas herramientas permiten gestionar de forma eficiente los entornos necesarios para las pruebas. Hemos desarrollado y preparado archivos Makefile dentro del proyecto para automatizar el levantamiento de los contenedores necesarios. Estos contenedores incluyen todos los servicios críticos para la ejecución de las pruebas, tales como:

- **Base de datos**: para almacenar y gestionar los datos de la aplicación durante las pruebas.
- **Frontend**: la interfaz de usuario de la aplicación.
- **Backend**: el núcleo de la lógica del negocio y las API de la aplicación.
- **Kafka**: para pruebas relacionadas con mensajería y comunicación asincrónica.

Además, el proyecto incluye archivos Makefile específicos para levantar de forma independiente el frontend y el backend, facilitando así el trabajo en caso de que se desee probar únicamente una parte de la aplicación.

El uso de contenedores permite replicar un entorno de producción de manera fiable, garantizando consistencia y aislando las pruebas de posibles conflictos con el entorno local del desarrollador. Esto no solo optimiza el flujo de trabajo del equipo, sino que también asegura que las pruebas se ejecuten bajo condiciones controladas y reproducibles.

2.14 Suposiciones

Dentro de este plan de pruebas, se asume que el enfoque principal estará en los módulos críticos de la aplicación, aquellos que son esenciales para garantizar una experiencia de usuario fluida y funcional. Entre estos módulos prioritarios se encuentran el módulo de usuarios, que incluye funcionalidades de autenticación y autorización para gestionar el acceso seguro a la plataforma; el módulo de comunidades, que organiza las interacciones y colaboraciones entre los usuarios; el módulo de causas, encargado de la visualización y gestión de iniciativas solidarias; y el módulo de acciones, que permite a los usuarios participar activamente en dichas iniciativas. Estas áreas se consideran fundamentales debido a su impacto directo en los objetivos del proyecto y la experiencia de usuario.

2.15 Riesgos

Los riesgos son factores o situaciones potenciales que podrían afectar negativamente la calidad del software, la ejecución de las pruebas o el cumplimiento de los objetivos del proyecto. Estos riesgos pueden surgir debido a problemas técnicos, humanos, de recursos o de planificación, y su identificación temprana es clave para mitigarlos o gestionarlos adecuadamente. En nuestro caso hemos identificado los siguientes riesgos:

Riesgos	**Mitigación**
Problemas al conectar los módulos principales	Diseñar y ejecutar pruebas de integración automatizadas para validar la interacción entre los módulos principales. Realizar revisiones técnicas frecuentes entre equipos responsables de los módulos.
La aplicación podría no manejar adecuadamente un alto volumen de usuarios	Ejecutar pruebas de carga y rendimiento con herramientas como JMeter para simular escenarios de alto tráfico. Monitorizar el comportamiento de los módulos clave y optimizar el rendimiento.
Falta de datos reales para pruebas	Generar conjuntos de datos simulados basados en escenarios reales. Utilizar herramientas para poblar bases de datos con datos relevantes y representativos para pruebas de carga.
Plazos insuficientes para ejecutar pruebas completas en todos los módulos	Priorizar pruebas en funcionalidades críticas y planificar pruebas paralelas para maximizar el tiempo disponible. Mantener reuniones regulares de seguimiento para ajustar prioridades.
Priorización inadecuada	Involucrar al equipo de producto y a los testers en la planificación inicial de las pruebas. Usar una matriz de riesgos para identificar áreas críticas y garantizar que se prioricen.
Ajustes inesperados en las funcionalidades de los módulo	Mantener flexibilidad en los casos de prueba mediante el uso de metodologías ágiles. Establecer un proceso claro para incorporar cambios en los requisitos sin interrumpir las pruebas planificadas.

2.16 Criterio de finalización

El criterio de finalización define las condiciones bajo las cuales se considera que las actividades de prueba han concluido con éxito. Estos criterios permiten evaluar si las pruebas han sido suficientes y si el sistema cumple con los requisitos establecidos. En nuestro caso tenemos los siguientes criterios:

- Se haya alcanzado una cobertura de código del 80% en pruebas unitarias.
- Se ha ejecutado el 100% de los casos de pruebas definidos.
- Las pruebas de aceptación realizadas por el cliente o representante de usuarios han sido aprobadas y no hay solicitudes de cambios mayores.
- El sistema ha pasado pruebas de carga y rendimiento sin problemas significativos, mostrando estabilidad en escenarios simulados de alto tráfico.
- Al menos un 80%-90% de los mutantes generados durante las pruebas de mutación deben ser detectados y eliminados por los casos de prueba.
- Las funcionalidades probadas cumplen con los criterios de aceptación definidos en los requisitos funcionales y no funcionales.

3 Revisiones de Código

Las revisiones de código son esenciales para garantizar la calidad del software y mantener un estándar consistente en el proyecto. En SolidarianID, se han implementado las guías de estilo de *Airbnb* mediante la configuración de *ESLint* y *Prettier*, junto con extensiones de *Visual Studio Code* como *Error Lens*, *ESLint* y *Prettier* para optimizar la experiencia de desarrollo.

ESLint se utiliza para realizar análisis estático del código, lo que permite identificar patrones problemáticos y garantizar que se sigan las mejores prácticas. *Prettier*, por su parte, asegura un formato uniforme en todo el proyecto, mejorando la legibilidad y la consistencia del código. Las extensiones *Error Lens, ESLint* y *Prettier* se han empleado para resaltar errores y advertencias directamente en el editor, lo que facilita su detección y corrección durante el desarrollo.

La configuración de *ESLint* se ha adaptado a partir de las guías de estilo de *Airbnb*, con las siguientes reglas personalizadas:

- **'prettier/prettier': 'error', {endOfLine: ‘auto’}**

  Se incluye para integrar *Prettier* como una regla de *ESLint*, garantizando que el formato de código cumpla con las normas de *Prettier*. Los errores de formato se marcan como errores de *ESLint*, obligando a los desarrolladores a corregirlos y al mismo tiempo defines el formato de salto de línea a auto debido a que trabajamos en sistemas operativos distinto.

- **'no-useless-constructor': 'off'**

  Se desactiva para permitir constructores que no tengan lógica explícita, lo cual es útil en proyectos con *NestJS* donde empleamos la inyección de dependencias.

- **'no-empty-function': ['error', { allow: ['constructors'] }]**

  Permite el uso de constructores vacíos, lo cual es necesario en *NestJS* donde para la inyección de dependencias.

- **'import/extensions': 'off'**

  Se desactiva para evitar forzar la inclusión de extensiones de archivo en las importaciones, haciendo que el código sea más limpio y compatible con los módulos TypeScript y Node.js.

- **'import/prefer-default-export': 'off'**

Se desactiva para permitir el uso de exportaciones nombradas en lugar de exportaciones por defecto, lo cual mejora la claridad y flexibilidad del código, especialmente en proyectos grandes.

- **'no-underscore-dangle': ['error', { allow: ['_id', '_props'], allowAfterThis: true }]**

  Esta regla prohíbe el uso de guiones bajos al inicio de las variables, pero permite excepciones como _id y _props, por el uso de la clase base *Entity*, que tiene un *_id* y también permite su uso después de *this*.

- **'dot-notation': 'off'**

  Desactiva la recomendación de usar notación de punto en lugar de corchetes para acceder a propiedades de objetos, esto nos permite ser más flexible y adaptarse mejor a las necesidades específicas del código.

- **'class-methods-use-this': 'off'**

  Desactiva la regla que exige que los métodos de clase usen *this* ayuda a evitar advertencias innecesarias cuando se implementan métodos estáticos o funciones de utilidad dentro de clases, que no requieren acceso a las propiedades de la instancia.

- **'import/no-extraneous-dependencies': ['error', { devDependencies: true }]**

  Desactiva la advertencia sobre importar dependencias que no están listadas en package.json. La ventaja de eso es que previene dependencias innecesarias y permite importar las dependencias de desarrollo

- **'import/no-relative-packages': 'error'**

  Esta regla marca como error las importaciones de paquetes usando rutas relativas, lo que ayuda a evitar la dependencia de módulos locales específicos.

La configuración de *Prettier* se ha adaptado para mantener un formato uniforme en el proyecto, con las siguientes reglas personalizadas:

- **singleQuote: true**

  Fuerza el uso de comillas simples en lugar de comillas dobles, manteniendo la consistencia con las guías de estilo de JavaScript.

- **trailingComma: 'all'**

  Agrega comas al final de elementos en objetos, arrays y parámetros de funciones siempre que sea posible. Esta práctica facilita la adición de nuevas líneas y reduce posibles errores de sintaxis.

- **semi: true**

  Agrega punto y coma al final de cada declaración, siguiendo una convención que mejora la claridad y previene errores Automatic Semicolon Insertion (ASI), evitando la dependencia de este mecanismo.

- **tabWidth: 2**

  Establece la anchura de tabulación a 2 espacios, mejorando la legibilidad y manteniendo un código compacto y fácil de leer.

- **endOfLine: auto**

Se utiliza para definir cómo manejar los saltos de línea (end-of-line) en los archivos. Permite que Prettier respete automáticamente el estilo de saltos de línea que ya existe en los archivos, adaptándose a la configuración del sistema operativo.

4 Análisis de Calidad del Código

Un análisis de calidad del código es fundamental para garantizar que el software sea robusto, seguro y fácil de mantener. Para realizarlo hemos utilizado **SonarQube**, una herramienta de análisis estático que permite detectar posibles errores, vulnerabilidades y malas prácticas en el código fuente.

Para asegurar que el análisis de calidad se ejecute de manera consistente en todo el repositorio se definió un archivo de configuración con las propiedades de SonarQube. Este archivo, *sonar-project.properties*, contiene los parámetros necesarios para personalizar y ajustar las reglas de análisis, como los directorios a analizar, las exclusiones y el informe de cobertura del código generado por *Jest*.

Cabe destacar que al ejecutar el análisis del código por primera vez, no se detectaron muchos problemas debido al uso de *ESLint*. Este linter nos ha permitido asegurarnos de trabajar con código limpio desde el principio, ya que nos ayuda a identificar y resolver una gran cantidad de problemas, como errores de sintaxis, estilo de codificación y posibles inconsistencias antes de que lleguen a SonarQube.

El análisis nos ha ayudado a identificar y resolver varios problemas pequeños en el código, como el uso de *readonly* para propiedades que no se redefinen, lo que contribuye a mejorar la inmutabilidad del código. Sin embargo, algunos problemas han quedado sin resolver, principalmente aquellos relacionados con el número máximo de parámetros en una función, el uso de *map* en lugar de *forEach* y el uso de *switch* en lugar de *if* en ciertas condiciones. En estos casos, hemos decidido mantener estas estructuras tal como están, ya que preferimos su uso por razones de legibilidad o por la naturaleza específica de la implementación.

En cuanto a los *Hotspots*, se detectó como un punto crítico el uso de contraseñas *hard-coded* en el código. Tras revisar el contexto, confirmamos que estas contraseñas se encuentran únicamente en el archivo de pruebas de la clase *Password*, donde se declaran algunas contraseñas para permitir la ejecución de los tests, por lo que consideramos que el uso de estas contraseñas en este caso es seguro.

Por otro lado, se han mantenido los umbrales por defecto para las distintas métricas de SonarQube, ya que se considera que están dentro de los estándares adecuados para el proyecto y no se identificaron problemas significativos en áreas como la complejidad o mantenimiento. En cuanto a la cobertura de pruebas, también se ha mantenido el umbral del 80%, teniendo en cuenta que no se logra alcanzarlo porque el objetivo de la asignatura no era realizar pruebas exhaustivas en todo el código. Como resultado, la cobertura de pruebas queda por debajo del umbral pero se considera que en un proyecto real debería alcanzarlo para asegurar una calidad adecuada. En la figura 41 se incluyen los resultados del análisis de calidad del código según las métricas establecidas por SonarQube.

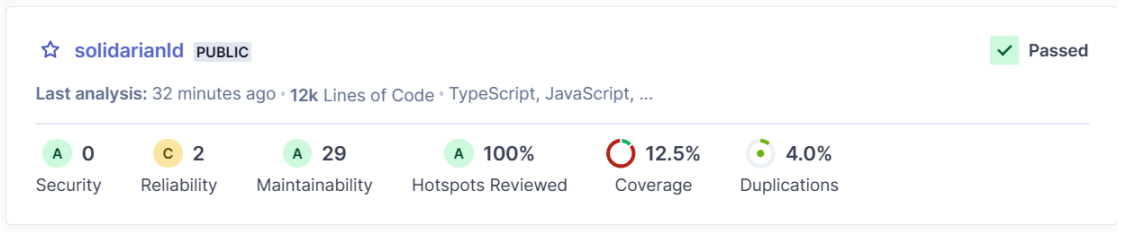


Figura 41. Resultado del análisis de calidad del código con SonarQube.

5 Conclusiones

En conclusión, la fase de testing sigue siendo crucial para garantizar la calidad y fiabilidad de cualquier aplicación, independientemente de su finalidad. Ya sea una aplicación de compras, un juego o una herramienta de apoyo para otras aplicaciones, todas requieren pruebas para asegurar su correcto funcionamiento.

En este proyecto, se ha realizado un plan de pruebas que incluye pruebas unitarias, pruebas de mutación con Striker, pruebas end-to-end con Cypress y pruebas de rendimiento con Apache JMeter. Aunque no se ha podido realizar una cobertura exhaustiva debido a limitaciones de tiempo, las pruebas realizadas han sido fundamentales para verificar el comportamiento de los módulos clave de la aplicación, como el de usuarios, autenticación y autorización, así como para asegurarse de que la aplicación sea capaz de manejar un volumen adecuado de usuarios sin problemas de rendimiento.

Asimismo, se han aplicado pruebas metamórficas para evaluar la coherencia y robustez de la lógica de la aplicación ante transformaciones y cambios, aunque estas pruebas fueron limitadas en número. Durante el proceso, también se ha asegurado la calidad del código mediante ESLint y Prettier, garantizando que esté limpio y sin errores de estilo, y se ha realizado un análisis con SonarQube, identificando vulnerabilidades y áreas de mejora.

Si bien no se ha logrado una cobertura más amplia en todos los tipos de pruebas debido a las restricciones de tiempo, este enfoque ha permitido identificar algunos errores importantes y asegurar una calidad mínima en los componentes analizados, así como familiarizarnos con herramientas para llevar a cabo numerosos tipos de pruebas.

El testing, aunque limitado, contribuye significativamente a evitar costes adicionales y retrasos en fases posteriores del desarrollo, lo que garantiza que la aplicación esté lo más preparada posible para ofrecer una experiencia de usuario óptima y un rendimiento adecuado.

6 Bibliografía

1. Prettier. (s.f.). *Options*. https://prettier.io/docs/en/options.html
2. ESLint. (s.f.). *Configure ESLint*. https://eslint.org/docs/latest/use/configure/
3. NestJS. (s.f.). *Testing*. https://docs.nestjs.com/fundamentals/testing
4. Torres, A. M. (s.f.). *Testing de APIs NestJS*. https://ualmtorres.github.io/SeminarioTesting/
5. QAlified. (2023, octubre 2). *¿Qué son las Pruebas de Regresión? Cómo hacerlas, herramientas y más*. https://qalified.com/es/blog/pruebas-regresion/
6. LoadView. (2023, diciembre 5). *Pruebas de carga de JMeter: La guía definitiva para 2023*. https://www.loadview-testing.com/es/la-guia-definitiva-de-jmeter-tutorial-de-pruebas-de-carga-y-rendimiento/
7. Cazzcode. (s.f.). *¿Qué son las Pruebas Metamórficas?*. https://www.cazzcode.com/articulos/que-son-las-pruebas-metamorficas-metamorphic-testing.html
8. Wikipedia. (s.f.). *Prueba unitaria*. https://es.wikipedia.org/wiki/Prueba_unitaria

9. Universidad de Murcia. (2025). Prácticas y clases grabadas de la asignatura Control de Calidad y Pruebas. Máster en Ingeniería del Software, curso 2024-2025.