Testing

Introducción a las pruebas de software

CertiDevs

Índice de contenidos

1. Introducción

El **testing de software** es un proceso esencial en el desarrollo de aplicaciones y sistemas informáticos, que tiene como objetivo verificar y validar la **calidad**, **funcionalidad** y **rendimiento** de un producto de software.

Este proceso busca identificar posibles **errores**, **inconsistencias** o **vulnerabilidades** que podrían afectar negativamente la experiencia del usuario y la seguridad de la información.

El testing es fundamental para **garantizar** que un producto de software cumple con los **requisitos** y expectativas del cliente, así como para prevenir posibles problemas que puedan surgir durante su uso.

La **calidad del software** es un aspecto crítico en la industria de la tecnología, ya que de ella depende la satisfacción del usuario, la eficiencia operativa y la reputación de una empresa o desarrollador. Por esta razón, el testing de software es considerado una práctica indispensable en cualquier proceso de desarrollo, y se lleva a cabo en diferentes etapas, desde la fase de diseño hasta la implementación y mantenimiento.

2. Tipos de testing

Existen varios **tipos de testing de software**, cada uno con sus propias técnicas y objetivos específicos. Algunos de los más comunes incluyen:

- **Testing funcional**: Se enfoca en verificar que el software cumple con las funcionalidades y requisitos establecidos en la documentación. Esto incluye pruebas de interfaces, entradas y salidas, bases de datos y otros componentes del sistema.
- **Testing no funcional**: Evalúa aspectos como el rendimiento, la seguridad, la escalabilidad, la usabilidad y la compatibilidad del software. Estas pruebas buscan garantizar que el producto funcione de manera eficiente y segura en diferentes entornos y situaciones.
- **Testing de regresión**: Consiste en volver a probar un software después de realizar cambios o actualizaciones, con el objetivo de asegurar que las modificaciones no hayan generado nuevos errores o afectado funcionalidades previamente validadas.
- **Testing de integración**: Verifica la correcta interacción entre diferentes componentes y módulos del software, asegurando que trabajen de manera conjunta y eficiente.
- **Testing unitario**: Se realiza a nivel de código, evaluando cada función o componente individualmente para garantizar su correcto funcionamiento.
- **Testing de aceptación**: También conocido como pruebas de usuario, este tipo de testing tiene como objetivo validar que el software cumple con las expectativas y necesidades del cliente o usuario final.

El testing de software es una disciplina en constante evolución, con metodologías y herramientas que se adaptan a las nuevas tendencias y desafíos de la industria tecnológica.

Sin importar el enfoque o la tecnología empleada, el objetivo principal del testing de software siempre será garantizar la calidad y confiabilidad del producto, asegurando que cumpla con las

necesidades y expectativas de los usuarios y clientes.

3. Testing en Angular

Una de las características más importantes del framework **Angular** es su enfoque en el **testing** y la **calidad** del código, lo que facilita la creación de aplicaciones robustas y fiables.

Para lograr esto, Angular se integra con diversas herramientas de testing, siendo **Jasmine** uno de los frameworks más utilizados para realizar pruebas en aplicaciones Angular.

Sitio web de Jasmine

Jasmine es un **framework de testing** de JavaScript de código abierto que permite a los desarrolladores escribir y ejecutar **pruebas unitarias** y de **integración** en un entorno fácil de usar y con una sintaxis clara y descriptiva.

Jasmine es especialmente adecuado para el testing en Angular debido a su enfoque en el comportamiento y las funcionalidades de la aplicación, lo que lo convierte en una herramienta ideal para probar **componentes**, **servicios**, **directivas** y otras partes de una aplicación Angular.

Algunas de las ventajas de utilizar Jasmine para el testing en Angular incluyen:

- **Sintaxis clara y fácil de leer**: Jasmine utiliza una sintaxis similar a BDD (Behavior-Driven Development), lo que permite escribir pruebas de manera legible y descriptiva, facilitando la comprensión y mantenimiento del código de prueba.
- **Dobles de prueba**: Jasmine proporciona funcionalidades para crear **espías** (spies) y **dobles de prueba** (mocks) que permiten aislar y simular comportamientos y dependencias en las pruebas, lo que facilita la realización de pruebas unitarias y de integración en aplicaciones Angular.
- **Integración con Angular**: Jasmine se integra de manera **nativa con Angular**, lo que permite a los desarrolladores utilizar herramientas y utilidades específicas de Angular, como `TestBed`, para facilitar la configuración y ejecución de pruebas.
- **Amplia comunidad y soporte**: Jasmine es una herramienta ampliamente utilizada y respaldada por una gran comunidad de desarrolladores, lo que garantiza su continuidad y evolución, así como la disponibilidad de recursos y soporte para resolver dudas y problemas.

Al trabajar con Angular y Jasmine, los desarrolladores pueden aprovechar las funcionalidades de ambas herramientas para crear y ejecutar pruebas eficientes y fiables en sus aplicaciones.

4. Soporte nativo para testing en Angular

Cuando se crea un proyecto Angular utilizando **Angular CLI**, este incorpora por defecto varias herramientas y frameworks de testing para facilitar la creación y ejecución de pruebas en la aplicación.

Los **frameworks de testing** incluidos **por defecto** en un proyecto **Angular** generado con Angular CLI son:

- **Jasmine**: Jasmine es el framework de testing principal utilizado en proyectos Angular. Está diseñado para realizar pruebas unitarias y de integración en aplicaciones JavaScript y se integra perfectamente con Angular. Jasmine proporciona una sintaxis clara y descriptiva basada en **BDD (Behavior-Driven Development)** que facilita la escritura y comprensión de pruebas.
- **Karma**: Karma es un **test runner** creado por el equipo de Angular que permite ejecutar pruebas en diferentes navegadores y entornos. Karma funciona como un **intermediario** entre el código de prueba y los navegadores, facilitando la ejecución de pruebas en múltiples navegadores y plataformas de forma simultánea. Karma se integra con Jasmine para ejecutar pruebas unitarias y de integración en aplicaciones Angular.

Con estas herramientas, se pueden realizar diferentes **tipos de testing** dentro de una aplicación Angular:

- **Pruebas unitarias**: Con Jasmine y Karma, los desarrolladores pueden crear y ejecutar pruebas unitarias para componentes, servicios, directivas y otros elementos de la aplicación. Las pruebas unitarias se enfocan en probar cada parte del código de forma aislada, verificando que funcione correctamente.
- **Pruebas de integración**: Jasmine y Karma también permiten realizar pruebas de integración en aplicaciones Angular, verificando que diferentes componentes y módulos del sistema interactúen correctamente entre sí.

5. Dobles de prueba

6. Test doubles

Test Double es el término general para **stubs**, **mocks**, **fakes**.

Un Test double o doble de prueba es cualquier objeto o sistema que usa en una prueba en lugar de la dependencia real. La mayoría de las pruebas de software automatizadas implican el uso de dobles de prueba de algún tipo u otro.

6.1. 1. Stub

Un **stub** es una implementación que se comporta de forma "antinatural". Está preconfigurado (generalmente por la configuración de testing) para responder a entradas específicas con salidas específicas.

Es un objeto que contiene unos datos de prueba predefinidos y los utiliza para responder a llamadas durante las pruebas.

El **propósito** de un stub es poner su sistema bajo prueba en un estado específico. Por ejemplo, si está escribiendo una prueba para algún código que interactúa con una API REST, puede cerrar la API REST con una API que siempre devuelva una respuesta ya predefinida (canned/hardcoded) o que responda a una solicitud de API con un error específico. De esta manera, podría escribir pruebas que hagan afirmaciones sobre cómo reacciona el sistema a estos estados; por ejemplo,

probar la respuesta que obtienen sus usuarios si la API devuelve un error 404.

Un stub generalmente se implementa para responder solo a las interacciones exactas a las que le ha dicho que responda. Pero la característica clave que hace que algo sea un stub es su propósito: un stub se trata de configurar su caso de prueba.

Stub vs. mock

Stub al igual que mock significa crear un objeto simulado, pero un stub solo imita el comportamiento, no el objeto completo. No sería tan sofisticado como un mock, porque el stub no registra las llamadas.

Resumen

- Un **stub** es la versión más ligera y estática de esta cadena de test doubles.
- Stub siempre devuelve la salida predefinida independientemente de la entrada.
- No podemos controlar el comportamiento del stub.
- Un stub puede ser útil para imitar los objetos de la base de datos. Por ejemplo devolver unos objetos predefinidos, sin tener una base de datos real.

6.2. 2. Fake

Un fake es un objeto con capacidades limitadas creado únicamente con propósito de testing y no debería ser usado en producción.

Un fake o falsificación es una implementación que se comporta "naturalmente", pero no es "real".

El propósito de una falsificación no es afectar el comportamiento del sistema bajo prueba, sino simplificar la implementación de la prueba (eliminando dependencias innecesarias o pesadas).

Estos son conceptos confusos y, por lo tanto, diferentes personas tienen diferentes interpretaciones de lo que hace que las cosas sean falsas. Ejemplos:

1. Una **base de datos en memoria** (por ejemplo, usar sqlite con :memory: store o H2). Nunca usaría esto para producción (ya que los datos no se conservan), pero es perfectamente adecuado como base de datos para usar en un entorno de prueba. También es mucho más ligero que una base de datos "real".
2. **Almacén de objetos** (por ejemplo, Amazon S3) en producción, pero en una prueba simplemente puede guardar objetos en archivos en el disco; entonces su implementación de "guardar en disco" sería falsa. (O incluso podría falsificar la operación "guardar en disco" utilizando un sistema de archivos en memoria).
3. **Caché**, un objeto que implementa la interfaz correcta pero que simplemente no realiza ningún almacenamiento en caché pero siempre devuelve una falta (miss) de caché sería una especie de falsificación.
4. Un **servicio web** falso, por ejemplo un cliente para paypal o coinbase.
5. Un **cliente mail** falso.

6. Conector de **mensajería asíncrona** en memoria para rabbit/kafka.

Resumen

- Un Fake es más poderoso que un Stub.
- Las clases falsas pueden cambiar el comportamiento según la entrada.
- Las funciones de clase falsas pueden devolver una salida diferente para diferentes entradas a diferencia de la de stub.
- Los fakes nos pueden ayudar a imitar todo el comportamiento posible de las interfaces.

6.3. 3. Spy

Un espía es un objeto real pero parcialmente mockeado, es decir, cierta parte del comportamiento puede ser cambiada. Significa que trabajas con un objeto no doble, excepto el comportamiento simulado. Por ejemplo, si se tiene una clase con múltiples funciones, un espía le permitiría imitar una de esas funciones y dejar que el resto de la clase se comporte normalmente.

Permiten que la entidad a imitar conserve su comportamiento original al tiempo que proporciona información sobre cómo interactuó con esa entidad. El espía puede decirle a la prueba qué parámetros se le dieron, cuántas veces se llamó y cuál fue el valor de retorno, si hubo alguno.

Resumen

- Un Spy es una versión avanzada del Fake que puede almacenar el estado anterior del objeto y hacer verificaciones.
- Un Spy nos permite ejecutar el comportamiento real de un objeto e imitar parcialmente ciertas partes del objeto.
- Un Spy nos permite verificar si un comportamiento del objeto fue invocado, por ejemplo si una función se llamó al menos una vez.
- También puede se crear un espía para que los loggers almacenen y validen todos los logs registrados mientras ejecutan el caso de prueba.

6.4. 4. Mock

Los **mocks** son un enfoque en las pruebas unitarias que permite hacer afirmaciones sobre cómo el código está interactuando con dependencias.

Por ejemplo: un método1 que llama a un método2, en este caso método2 es una dependencia, pero si estamos testeando método1 lo mejor es enfocarse en método1 y utilizar un objeto simulado o mock para método2.

De esta forma podemos comprobar cómo método1 interactúa con método2 sin llegar a probar método2 (método2 ya tiene su test unitario) y si lo está haciendo.

Los mocks son sofisticados en el sentido de que permiten verifican las interacciones, por ejemplo:

- que un método de ese mock ha sido invocado

- comprobar cuántas veces ha sido invocado, si 0, una o muchas
- simular respuesta
- comprobar argumentos recibidos
- … etc …

mock vs. stub

Un **mock** es similar a un **stub**, pero con **verificación** añadida. El propósito de un mock es hacer verificaciones sobre cómo su sistema bajo prueba interactuó con la dependencia.

Por ejemplo, si está escribiendo una prueba para un sistema que carga archivos en un sitio web, podría crear un mock que acepte un archivo y que pueda usar para afirmar que el archivo cargado era correcto.

Otro ejemplo, en una escala más pequeña, es común usar un mock de un objeto para verificar que el sistema bajo prueba llame a métodos específicos del objeto simulado y comprobar cuántas veces han sido llamados.

Los mocks están vinculados a las pruebas de interacción, que es una metodología de prueba específica. Las personas que prefieren probar el estado del sistema en lugar de las interacciones del sistema usarán mocks con moderación, si es que lo hacen.

Resumen

- Un Mock es la versión más potente y flexible de todas las opciones
- El comportamiento de la interfaz simulada se puede cambiar dinámicamente en función de los escenarios.
- Podemos aplicar una variedad de afirmaciones mediante la creación de objetos simulados utilizando mock frameworks como Mockito.
- Mock da el control total sobre el comportamiento de los objetos simulados.
- En frameworks como Mockito un Mock también permite llamar al método real haciendo uso de funciones `thenCallRealMethod`, `willCallRealMethod`, `doCallRealMethod`.

7. Dummy

Objetos dummy o dummy data son objetos con datos ficticios como Book, Employee, etc, para ser usados en las pruebas. Normalmente se usan para completar listas de parámetros y evitar errores de compilación.

También se puede llamar dummy data a archivos csv que se cargan en base de datos para tener un entorno demo para las pruebas.

8. Estructura de archivos de testing en Angular

- **Archivos de prueba unitaria e integración**: Por lo general, cada **componente**, **servicio**, **directiva** u otra entidad en un proyecto Angular tiene su propio archivo de prueba asociado. Estos archivos de prueba se ubican en el mismo directorio que el archivo correspondiente al elemento que se está probando y siguen la convención de nombres `nombre-del-elemento.spec.ts`. Por ejemplo, si se tiene un componente llamado `my-component`, el archivo de prueba asociado se llamaría `my-component.component.spec.ts`.
- **Configuración de Karma**: La configuración de Karma se encuentra en un archivo llamado `karma.conf.js` en la raíz del proyecto. Este archivo contiene configuraciones importantes para la ejecución de pruebas, como los navegadores donde se ejecutarán las pruebas, los informes generados, los preprocesadores utilizados y las rutas a los archivos de prueba.

9. Ejecutar tests en Angular

```
# ejecutar Karma test runner
ng test
```

```
# ejecutar karma test runner con cobertura:
ng test --code-coverage
```

```
# ejecutar un test concreto
ng test --include src/app/models/contact.spec.ts
```