**UNIVERSIDAD NACIONAL DE INGENIERÍA**

# **FACULTAD DE CIENCIAS**

****

# **Tópicos de Calidad de Software CC0A5**

# **Testing de Microservicios**

***Fatama Ruiz Ademar – 20192148F***

***Yaranga Sante Julio Cesar – 20187020E***

***Cipriano Arroyo Bruno - 20190497C***

***Seminario Serna Luis - 20181328H***

**Profesor:  
 GIPSY MIGUEL ANGEL ARRUNATEGUI ANGULO**

Introducción

En el paisaje dinámico del desarrollo de software, la arquitectura de microservicios ha redefinido la agilidad y la escalabilidad. Sin embargo, esta transición ha elevado la importancia del testing de microservicios para garantizar la cohesión y el rendimiento en entornos distribuidos. Estas pruebas abordan la validación individual y la integración de servicios, siendo críticas para la robustez de la aplicación. Herramientas especializadas han surgido para automatizar pruebas unitarias, de integración y extremo a extremo, facilitando la tarea de los equipos de desarrollo.

Este informe explora el panorama del testing de microservicios, destacando pruebas esenciales y herramientas clave. La calidad del software resultante se ve directamente afectada por estas prácticas, ya que aseguran no solo la coherencia y el rendimiento, sino también la capacidad de adaptación en un entorno tecnológico dinámico.

Estado del arte del testing de microservicios

Pruebas funcionales

Las pruebas funcionales son un componente esencial en el desarrollo de software y se llevan a cabo para evaluar si un sistema cumple con los requisitos especificados y funciona según lo previsto. Estas pruebas se realizan en distintos niveles y abordan diferentes aspectos del software. Aquí hay una breve descripción de algunos tipos comunes de pruebas funcionales.

Pruebas unitarias

Son pruebas que se centran en evaluar unidades individuales de código, como funciones o métodos. El objetivo es verificar que cada unidad funcione correctamente de manera aislada, además, son generalmente realizadas de manera automática.

Pruebas de integración

Son las pruebas que se centran en evaluar la interacción e integración correcta del código, esto ayuda a evaluar de manera general e incluso y cumpla con el correcto funcionamiento del código, además de reducir los riesgos asociados con la integración de componentes.

Pruebas de sistema

Las pruebas del sistema es un tipo de prueba funcional en el cual se centra en evaluar todo el sistema, es decir el sistema completo, en lugar de unidades individuales o módulos. El objetivo principal de estas pruebas es asegurarse del funcionamiento de todo el sistema.

Pruebas de aceptación

Las pruebas de aceptación son un tipo de prueba que se realiza para asegurarse de que un sistema cumple con los criterios de aceptación definidos por el cliente o las partes interesadas. Estas pruebas están diseñadas para verificar que el software cumple con los requisitos del usuario y que satisface las necesidades y expectativas del cliente.

Pruebas de carga

Las pruebas de carga son un tipo de prueba de rendimiento que evalúa el comportamiento de un sistema bajo condiciones de carga y demanda significativas. El objetivo principal de estas pruebas es medir la capacidad de un sistema para manejar un volumen específico de usuarios, transacciones o datos durante un período prolongado, manteniendo al mismo tiempo un rendimiento aceptable.

Pruebas de estrés

Las pruebas de estrés son un tipo de prueba de rendimiento que evalúa cómo se comporta un sistema bajo condiciones extremas o situaciones límite. El propósito principal de estas pruebas es determinar el límite o la capacidad máxima de un sistema y observar cómo responde cuando se somete a una carga o demanda que excede sus capacidades normales.

Pruebas de caos

Son un enfoque sistemático para identificar y abordar debilidades en sistemas mediante la introducción deliberada de fallos o condiciones inusuales en un entorno de producción controlado. El propósito de estas pruebas es evaluar la resiliencia y la capacidad de recuperación de un sistema frente a situaciones caóticas y eventos inesperados.

Pruebas de penetración

Las pruebas de penetración, también conocidas como pruebas de pen testing o ethical hacking, son un tipo de evaluación de seguridad en la que profesionales de seguridad informática simulan ataques cibernéticos contra un sistema, red o aplicación para identificar y corregir vulnerabilidades antes de que puedan ser explotadas por actores malintencionados.

Pruebas E2E (end-to-end) -

Las pruebas end-to-end (E2E) evalúan la funcionalidad y rendimiento de una aplicación en su conjunto, simulando la experiencia del usuario desde el inicio hasta el final del flujo de trabajo. Por ejemplo, en un sistema de comercio electrónico, una prueba E2E podría involucrar la simulación de un usuario que navega por el sitio, agrega productos al carrito, realiza el pago y verifica la confirmación del pedido. Estas pruebas garantizan la integración adecuada de todos los componentes y servicios de la aplicación.

Pruebas de Escalabilidad Horizontal

Las pruebas de escalabilidad horizontal evalúan cómo una aplicación maneja el aumento de carga distribuyendo eficientemente el tráfico entre instancias adicionales del servicio. En el contexto de microservicios, esto significa verificar la capacidad del sistema para escalar horizontalmente añadiendo o eliminando instancias de microservicios según sea necesario. Por ejemplo, si una aplicación experimenta un aumento repentino en la demanda, las pruebas de escalabilidad horizontal garantizan que los microservicios puedan replicarse para mantener el rendimiento sin afectar la experiencia del usuario.

Análisis de herramientas y procesos

Docker

Empaquetamiento Consistente: Docker permite empaquetar aplicaciones y todas sus dependencias en contenedores, lo que garantiza consistencia y reproducibilidad en diferentes entornos. Esto elimina problemas de "funciona en mi máquina" al encapsular la aplicación y su entorno en un contenedor autosuficiente.

Aislamiento y Portabilidad: Los contenedores Docker ofrecen un alto grado de aislamiento, lo que significa que las aplicaciones empaquetadas en contenedores pueden ejecutarse de manera segura y sin interferencias con otras aplicaciones en la misma máquina. Además, los contenedores son portátiles y pueden ejecutarse de manera consistente en cualquier entorno que admita Docker.

Desarrollo Ágil y Entrega Continua: Docker facilita el desarrollo ágil al permitir a los desarrolladores construir, probar y distribuir rápidamente aplicaciones en contenedores. Esto acelera el ciclo de vida del desarrollo y facilita la implementación continua, lo que es crucial en un entorno de microservicios.

Eficiencia en Recursos:

Kubernetes

Orquestación y Escalabilidad: Kubernetes es una plataforma de orquestación de contenedores que facilita la implementación, escala y gestión de aplicaciones basadas en microservicios. Permite la fácil escala horizontal de microservicios en respuesta a cambios en la demanda.

Gestión de Servicios y Balanceo de Carga: Kubernetes automatiza la gestión de servicios, permitiendo una fácil exposición y acceso a los microservicios. También proporciona funcionalidades de balanceo de carga para distribuir el tráfico de manera eficiente entre las instancias de los microservicios.

Autoreparación y Actualizaciones sin Tiempo de Inactividad: Kubernetes monitoriza continuamente el estado de los contenedores y puede reiniciar automáticamente los contenedores fallidos. Además, facilita las actualizaciones sin tiempo de inactividad, permitiendo la implementación gradual de nuevas versiones de microservicios.

Declaratividad y Despliegue Consistente: Kubernetes utiliza configuraciones declarativas para definir la infraestructura y el estado deseado de las aplicaciones. Esto asegura un despliegue consistente y repetible de microservicios, lo que es esencial en entornos de producción.

Gestión de Configuración y Almacenamiento de Secretos: Kubernetes gestiona de manera centralizada la configuración y los secretos de las aplicaciones, facilitando la administración y actualización de la configuración de los microservicios de manera segura.

Servicios de AWS

**Amazon Elastic Container Service (ECS)**:

* Es un servicio de orquestación de contenedores gestionado por AWS.
* Ideal para aplicaciones más sencillas y que se integran bien con el ecosistema de AWS.
* Permite ejecutar contenedores Docker en clústeres administrados por AWS.
* Proporciona un nivel de control más simple y menos complejo que EKS.
* Puede ser hasta un 30% más económico que EKS

**Amazon Elastic Kubernetes Service (EKS)**:

* Es un servicio para implementar clústeres de Kubernetes en AWS.
* Recomendado para quienes prefieren Kubernetes y desean más control.
* Permite administrar clústeres de Kubernetes en AWS.
* Ofrece más control y flexibilidad que ECS.
* Tiene cargos adicionales, como $0.1 por hora por clúster de Kubernetes

**AWS Fargate**:

* Es un motor de contenedores sin servidor.
* Perfecto para procesos de corta duración.
* Ejecuta contenedores sin preocuparte por la infraestructura subyacente.
* Menos control que ECS y EKS, pero más sencillo.
* Paga solo por los recursos utilizados para ejecutar tus contenedores

Parasoft:

Parasoft es una plataforma de pruebas de software integral que aborda diversas fases del ciclo de vida del desarrollo de software. Ofrece herramientas para realizar pruebas unitarias, pruebas de regresión, pruebas de integración y pruebas de sistema. Además, incluye características de análisis estático de código, pruebas de seguridad y pruebas funcionales.

Los equipos de desarrollo utilizan Parasoft para automatizar y ejecutar pruebas a lo largo del proceso de desarrollo, lo que ayuda a detectar y corregir errores de manera eficiente.

JMeter:

Apache JMeter es una herramienta de código abierto desarrollada por la Apache Software Foundation. Se utiliza principalmente para realizar pruebas de carga y rendimiento en aplicaciones web. JMeter permite simular un gran número de usuarios virtuales que interactúan con una aplicación web, generando así carga para evaluar su rendimiento y detectar posibles cuellos de botella o problemas de escalabilidad.

Los equipos de pruebas y rendimiento utilizan JMeter para medir la capacidad de una aplicación bajo condiciones de carga simuladas, identificar cuellos de botella y realizar pruebas de estrés.

Gatling:

Gatling es una herramienta de pruebas de rendimiento escrita en Scala y basada en el lenguaje de programación Akka. A diferencia de JMeter, Gatling se centra en la simulación de carga utilizando una sintaxis de DSL (lenguaje específico del dominio) que permite describir escenarios de prueba de manera clara y concisa. Gatling está diseñado para ser altamente eficiente y escalable.

Equipos de rendimiento y desarrollo utilizan Gatling para realizar pruebas de carga, pruebas de estrés y pruebas de rendimiento en aplicaciones web y APIs. La sintaxis de DSL facilita la creación de escenarios complejos y realistas de pruebas de carga.

Selenium:

Como herramienta de automatización para aplicaciones web, Selenium facilita la validación y la integración continua de microservicios, asegurando la funcionalidad en entornos distribuidos.

Postman:

La plataforma de pruebas de API Postman capacita a los desarrolladores para probar, desarrollar y documentar APIs de microservicios de manera eficiente y efectiva.

WireMock:

Como simulador de servicios HTTP, WireMock simplifica las pruebas de API, permitiendo validar la interacción y la comunicación entre microservicios de manera controlada y reproducible.

Pytest**:**

Pytest, eficiente y versátil, emerge como una herramienta esencial para el testing de microservicios en entornos Python. Su diseño modular simplifica la creación y mantenimiento de pruebas unitarias e integración, contribuyendo a la calidad y fiabilidad de los microservicios.

Prueba piloto y resultados

Para entender mejor el funcionamiento de las pruebas en los microservicios vamos a explorar un proyecto fascinante alojado en el repositorio público de GitHub llamado [Microservicios\_Contenedores\_Calculadora](https://github.com/CrissUD/Microservicios_Contenedores_Calculadora), cortesía del usuario CrissUD. Este proyecto es una calculadora única, donde cada operación (suma, resta, multiplicación, división) es manejada por su propio microservicio, demostrando la belleza de la arquitectura de microservicios.

El proyecto utiliza Flask, un microframework web para Python que es conocido por su simplicidad y eficiencia. Flask facilita el desarrollo de aplicaciones web siguiendo el patrón Modelo-Vista-Controlador (MVC), lo que permite una separación clara entre la interfaz de usuario, los datos y la lógica de negocio.

Además, también utiliza Flask-CORS, una extensión vital de Flask que maneja el intercambio de recursos de origen cruzado (CORS). Esto es esencial para permitir solicitudes AJAX de origen cruzado, un mecanismo de seguridad implementado por los navegadores cuando se realiza una petición a un recurso alojado en un origen diferente.

Para lograr una mejor visión de las pruebas adoptaremos un enfoque de caja negra, donde nos centraremos en los resultados de las operaciones sin preocuparnos por los detalles internos de cómo se realizan. Para ello, utilizaremos el marco de pruebas Pytest, que nos ofrece un abanico de posibilidades para la automatización de nuestras pruebas.

Este código es un ejemplo de cómo se pueden realizar pruebas en microservicios utilizando pytest, un marco de pruebas para Python, y Flask, un microframework web para Python.

Análisis de la prueba del microservicio suma

El código comienza importando los módulos necesarios: pytest para las pruebas, math para las operaciones matemáticas, requests para hacer solicitudes HTTP y suma que es el microservicio que se va a probar.

import pytest

import math

import requests

from suma.suma import app as suma

Luego, se define una fixture de pytest llamada client. Las fixtures son funciones que se ejecutan antes de cada prueba y proporcionan un entorno de prueba aislado. En este caso, client es un cliente de prueba de Flask que se utiliza para enviar solicitudes al microservicio suma.

@pytest.fixture

def client():

    with suma.test\_client() as client:

        yield client

A continuación, se definen dos pruebas: test\_suma\_dos\_positivos y test\_suma\_live.

test\_suma\_dos\_positivos es una prueba unitaria que verifica que el microservicio suma está funcionando correctamente. Esta prueba envía una solicitud POST al endpoint /api/suma del microservicio con dos números como datos JSON. Luego, verifica que el estado de la respuesta es 200 (lo que indica que la solicitud fue exitosa) y que el resultado de la suma es correcto.

def test\_suma\_dos\_positivos(client):

    response = client.post('/api/suma', json={'n1': 2, 'n2': 3})

    assert response.status\_code == 200

    response\_data = response.data.decode('utf-8')

    print(response\_data)

    assert math.isclose(float(response\_data), 5, rel\_tol=1e-9)

test\_suma\_live es similar a test\_suma\_dos\_positivos, pero en lugar de usar el cliente de prueba de Flask, utiliza el módulo requests para enviar una solicitud HTTP real al microservicio. Esta prueba puede ser útil para verificar que el microservicio está funcionando correctamente en un entorno de producción.

def test\_suma\_live():

    response = requests.post('http://localhost:3030/api/suma', json={'n1': 2, 'n2': 3})

    assert response.status\_code == 200

    response\_data = response.json()

    assert math.isclose(float(response\_data), 5, rel\_tol=1e-9)

Ejecutando las pruebas

Para ejecutar las pruebas nos dirigiremos a la raíz a nuestro directorio de microservicios y ejecutaremos el comando: “python -m pytest”.

Primero lo haremos sin iniciar los microservicios, para fijarnos que esto es así ejecutaremos el comando: “docker ps”



Podemos observar que no hay contenedores en ejecución por lo que pasamos a ejecutar las pruebas

A screen shot of a computer

Description automatically generated

<Output omitido>

A black screen with white text

Description automatically generated

Podemos observar que 6 pruebas han fallado y tras una mayor inspección descubriremos que se debe a errores de conexión pues la aplicación no está en ejecución.

Tras una mejor inspección con el comando “python -m pytest -v” vemos que las fallas se deben a los test que terminan en \_live pues estos hacen llamadas directamente a la aplicación, mientras que el resto de test lo hace al cliente de pruebas de Flask.

A screenshot of a computer program

Description automatically generated

Ahora procederemos a levantar la aplicación:

A screenshot of a computer program

Description automatically generated

A black screen with white text

Description automatically generated

Y volveremos a ejecutar los tests:

A screenshot of a computer program

Description automatically generated

Ahora las llamadas a la aplicación si han podido concretarse, por lo que todos los test pasan.

Conclusiones

En el fascinante mundo de los microservicios, la calidad del software se ve directamente impactada por la implementación de pruebas exhaustivas. Las pruebas, que abarcan desde pruebas funcionales y unitarias hasta pruebas de estrés y caos, desempeñan un papel crucial en la construcción de aplicaciones robustas y confiables.

El uso de herramientas como Docker, junto con servicios en la nube como EKS, AKS y ECS, permite una orquestación eficiente de los microservicios. Esto no solo mejora la escalabilidad y la disponibilidad, sino que también facilita el proceso de pruebas al proporcionar entornos aislados y reproducibles.

Sobre proyecto de la calculadora, hemos experimentado de primera mano el poder de pytest en el testing de microservicios. Esta herramienta, junto con el cliente de prueba de Flask, ha demostrado ser invaluable, permitiendo pruebas aisladas y controladas de cada servicio.

En conclusión, el testing de microservicios es un campo vasto y emocionante que ofrece numerosas oportunidades para mejorar la calidad del software. A través de una variedad de pruebas y el uso de herramientas efectivas como Docker y pytest, podemos construir aplicaciones de microservicios que sean robustas, escalables y, sobre todo, confiables.

Referencias

* Flask. (2021). Welcome to Flask. Consultado el 10 de Dic. de 2023, de [Flask website](https://www.bibliotecas.unam.mx/index.php/desarrollo-de-habilidades-informativas/como-hacer-citas-y-referencias-en-formato-apa" \t "_blank).
* Docker. (2021). What is a Container? Consultado el 10 de Dic. de 2023, de [Docker website](https://www.bibliotecas.unam.mx/index.php/desarrollo-de-habilidades-informativas/como-hacer-citas-y-referencias-en-formato-apa).
* pytest. (2021). pytest: helps you write better programs. Consultado el 10 de Dic. de 2023, de [pytest website](https://www.bibliotecas.unam.mx/index.php/desarrollo-de-habilidades-informativas/como-hacer-citas-y-referencias-en-formato-apa" \t "_blank).
* Microsoft Azure. (2021). ¿Qué es Kubernetes? Consultado el 10 de Dic. de 2023, de [Microsoft Azure website](https://www.bibliotecas.unam.mx/index.php/desarrollo-de-habilidades-informativas/como-hacer-citas-y-referencias-en-formato-apa).
* CrissUD. (2023). Microservicios\_Contenedores\_Calculadora. GitHub. Recuperado el 9 de Dic. de 2023, de [CrissUD/Microservicios\_Contenedores\_Calculadora: Introducción a microservicios y contenedores utilizando el framework Flask de python para representar los microservicios de una calculadora de enteros. Ademas de utilizar Docker para componetizar los servicios (github.com)](https://github.com/CrissUD/Microservicios_Contenedores_Calculadora)