UNICA
UNIVERSITÀ
DEGLI STUDI
DI CAGLIARI

sAIfer Lab
Joint lab on Safety and Security of AI

Testing

Maura Pintor
maura.pintor@unica.it

Introduzione al testing

Perché testare il codice?

- Prevenzione di bug
- Manutenibilità del codice
- Refactoring sicuro

Automatizzare i test permette di aumentare l'efficienza dello sviluppo.

Tipologie di test

- **Unit Test**: testano singole funzioni
- **Integration Test**: testano l'interazione tra più componenti
- **End-to-End Test**: testano il sistema dal punto di vista dell'utente finale
- **Regression Test**: verificano che nuove modifiche non rompano vecchie funzionalità

Test-Driven Development (TDD)

Ciclo TDD:

- Scrivi un test che fallisce
- Scrivi codice per farlo passare
- Refactor del codice

Vantaggi:

- Design più modulare
- Copertura maggiore del codice

Esercizi di test semplici in Python

```
def somma(a, b):
    return a + b

def test_somma():
    assert somma(2, 3) == 5
```

Per eseguirli: `pytest test_math.py`

Esercizi aggiuntivi:

- Scrivere test per una funzione che calcola il massimo di tre numeri
- Testare una funzione che verifica se una stringa è palindroma

Code Coverage

Il code coverage è la percentuale di codice eseguito durante i test

```
def somma_pari(a, b):
    if a % 2 == 0 and b % 2 == 0:
        return a + b # linea non testata
    return None

def test_somma():
    assert somma(2, 3) == 5
```

Attenzione: **alta coverage ≠ test efficaci**

Mocking e isolamento

In un unit test, l'obiettivo è testare una funzione **senza dipendenze esterne**
Usiamo mocking per simulare:

- Accesso al database
- API esterne
- File system

```
from unittest.mock import MagicMock

db.get_user = MagicMock(return_value={"id": 1})
```

Tutorial di mocking

Per esempio, si può utilizzare il mocking per implementare un metodo che dipende da componenti esterni

Per lo scopo dello unit test infatti, non serve testare anche i metodi esterni

In questo caso, non ci preoccuperemo di implementare la funzione che accede al database, ma vedremo che è comunque possibile fare il test di una funzione che dipende da un medodo non implementato

```
class Database:
    def get_valori(self):
        # per ora non lo implementiamo
        raise NotImplementedError


def somma(db: Database):
    a, b = db.get_valori()
    return a + b
```

Tutorial di mocking

Implementiamo un metodo mock che finge di chiamare il metodo non implementato del databse e restituisce dei valori fittizi

All'interno del test potremo chiamare il metodo senza avere un errore.

```
import unittest
from unittest.mock import MagicMock
from somma_da_db import somma


class TestConMockDatabase(unittest.TestCase):
      def test_somma_mock_db(self):
            # Crea un mock del database
            mock_db = MagicMock()
            mock_db.get_valori.return_value = (3, 5)

            risultato = somma(mock_db)

            self.assertEqual(risultato, 8)

if __name__ == "__main__":
      unittest.main()
```

Test parametrici

Si possono scrivere test che sfruttano un'interfaccia comune ma usano parametri diversi
Evita codice ripetuto, migliora la copertura

```
import pytest

@pytest.mark.parametrize("a,b,expected", [
    (2, 3, 5),
    (-1, 1, 0),
    (0, 0, 0)
])
def test_somma(a, b, expected):
    assert a + b == expected
```

Testing per il Web Programming

Perché testare una Web App è diverso

- Richiede test più strutturati:
  - Validazione input/output
  - Gestione errori
  - Stato applicazione (es. DB, autenticazione)
  - Sicurezza (es. accessi non autorizzati)
- Non è solo “il codice gira”, ma “la risposta HTTP è corretta”

Casi di fallimento da testare

- Input malformati o incompleti
- Accesso a risorse inesistenti (/user/999)
- Richieste non autorizzate
- Timeout, errori di rete, errori interni (500)
- Status code
  - 200 OK → operazione riuscita
  - 201 Created → oggetto creato con successo
  - 400 Bad Request → errore input
  - 401 Unauthorized / 403 Forbidden
  - 404 Not Found
  - 422 Unprocessable Entity → validazione fallita
  - 500 Internal Server Error → bug

Test con FastAPI

FastAPI supporta test con TestClient (simulazione richieste HTTP)

```
from fastapi import FastAPI

app = FastAPI()
@app.get("/")
def read_root():
    return {"message": "Hello World"}
```

```
from fastapi.testclient import TestClient
from server import app

client = TestClient(app)

def test_read_root():
    response = client.get("/")
    assert response.status_code == 200
```

Test di validazione degli input

FastAPI gestisce automaticamente la validazione degli input, ma si può verificare che siano effettivamente gestiti correttamente

In caso di errore: risposta 422 Unprocessable Entity

```python
class Item(BaseModel):
    name: str = Field(min_length=3)
    price: float = Field(gt=0)

@app.post("/items")
def post_item(item: Item):
    return {"item received": item}
```

```python
def test_create_item_valid():
    response = client.post("/items/", json={"name": "Book", "price": 5})
    assert response.status_code == 200

def test_create_item_invalid():
    response = client.post("/items/", json={"name": "A", "price": -5})
    assert response.status_code == 422
```

Best practice di testing API

Testare tutti i percorsi possibili:

- Successo
- Fallimento per input sbagliato
- Fallimento per stato dell'app (es. utente non esiste)

Usare test automatici con TestClient

- status code
- contenuto JSON
- headers (es. token, location)

Se c'è autenticazione, bisogna verificare anche che:

- Un utente **non autenticato** non possa accedere
- Un utente **autenticato** possa solo accedere alle le sue risorse

https://github.com/maurapintor/testing_tutorial