

CPE334

Software Testing

# Who am I ? (again)



Tipatai Puthanukunkit (P'Ju)  
Principal Engineer at **muvmi**



Previous Exp. Pomelo.  fastwork 

CPE24 @ KMUTT

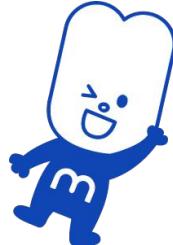


New user?

Get 50% off your first trip  
with discount code

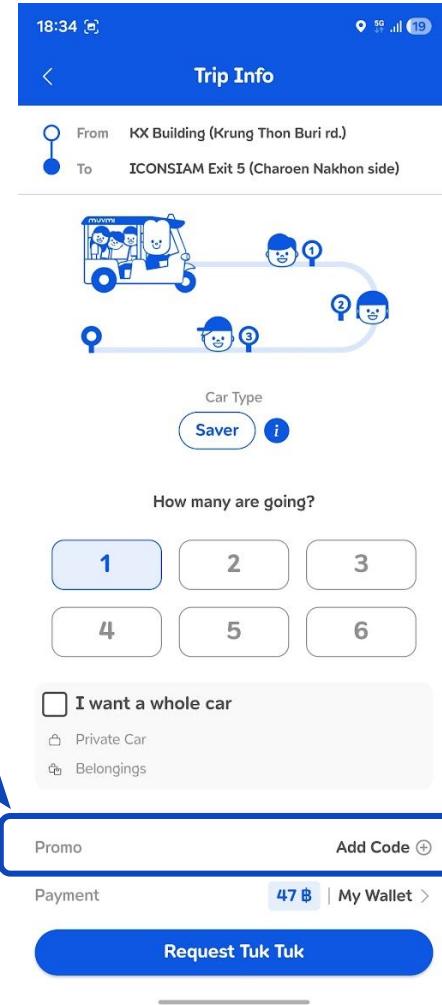
CPE50

MuvMi App



Download on the  
App Store

GET IT ON  
Google Play



This lecture is my own opinion, so you don't have to agree with everything I say.

If you have any questions or arguments to discuss, feel free to raise your hand or ask me after class.

Recap from Our  
Last Session

```
export function register(email: string, password: string): Result {
  if (/^[\^s@]+@[^\s@]+\.[^\s@]+$/ .test(email)) {
    if (email.endsWith("@cpe.kmutt.ac.th")) {
      if (password.length >= 8) {
        if (password.length <= 32) {
          if (/^[a-z]/.test(password)) {
            if (/^[A-Z]/.test(password)) {
              if (/^\d/.test(password)) {
                return createUserInDatabase(email, password); // All checks passed! ✓
              } else {
                return { success: false, error: "Password must contain at least one number." };
              }
            } else {
              return { success: false, error: "Password must contain at least one uppercase letter." };
            }
          } else {
            return { success: false, error: "Password must contain at least one lowercase letter." };
          }
        } else {
          return { success: false, error: "Password must be 32 characters or less." };
        }
      } else {
        return { success: false, error: "Email must end with @cpe.kmutt.ac.th." };
      }
    } else {
      return { success: false, error: "Email must be valid." };
    }
  } else {
    return { success: false, error: "Email must not be empty." };
  }
}
```

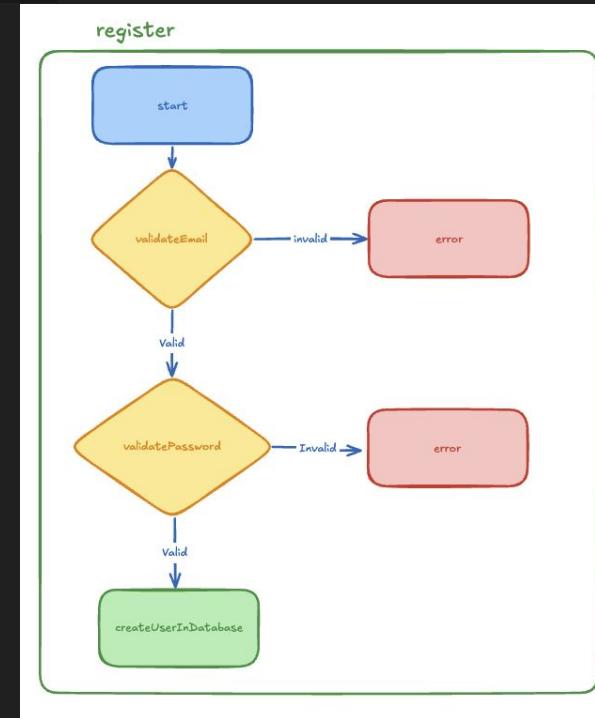
```

export function register(email: string, password: string): Result {
    const emailValidationResult = validateEmail(email);
    if (!emailValidationResult.success) {
        return emailValidationResult;
    }

    const passwordValidationResult = validatePassword(password);
    if (!passwordValidationResult.success) {
        return passwordValidationResult;
    }

    return createUserInDatabase(email, password);
}

```



<https://github.com/Judrummer/cpe-se-code-quality-bun>

# Good Code Quality

📖 Readability - Making Code Understandable

🔧 Maintainability - Easy to fix bugs and add features

🧪 Testability - Easy to test

🎯 Reliability - Consistent results, no surprises

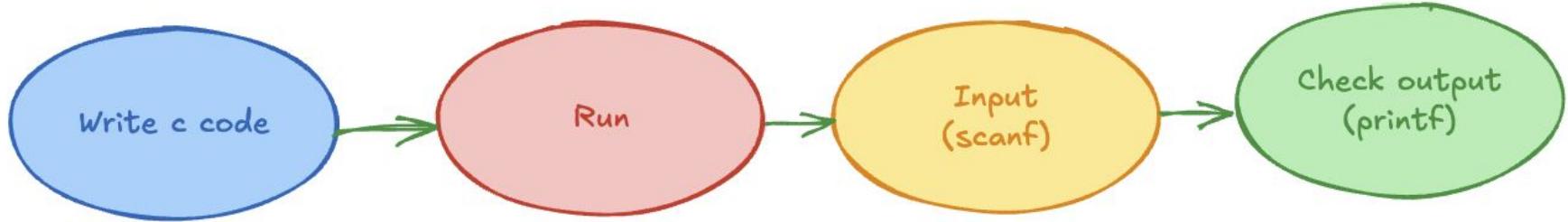
# Tips for making good code quality

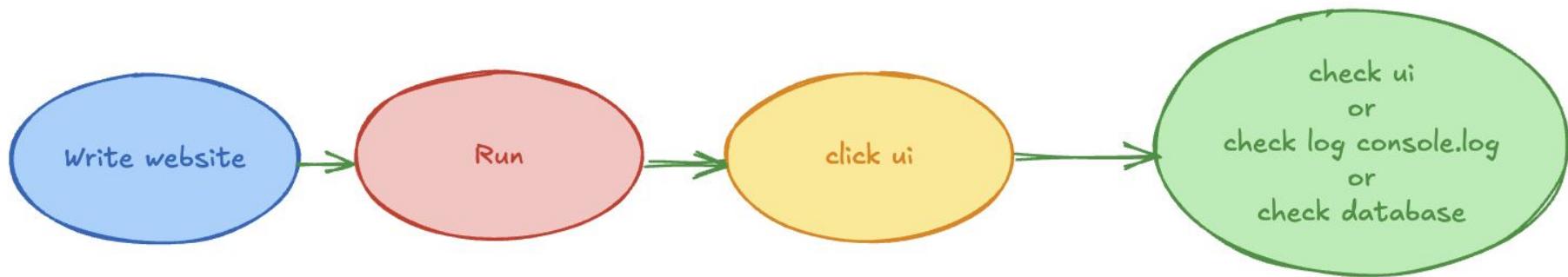
- Defining Types is Preferred
- Effective Naming Conventions
- Break problem into sub problems
- Use Guard Clauses
- Prioritize Immutable Variables
- Avoid Magic Numbers
- Pure Function
- Dependency Injection



# Testing as Documentation

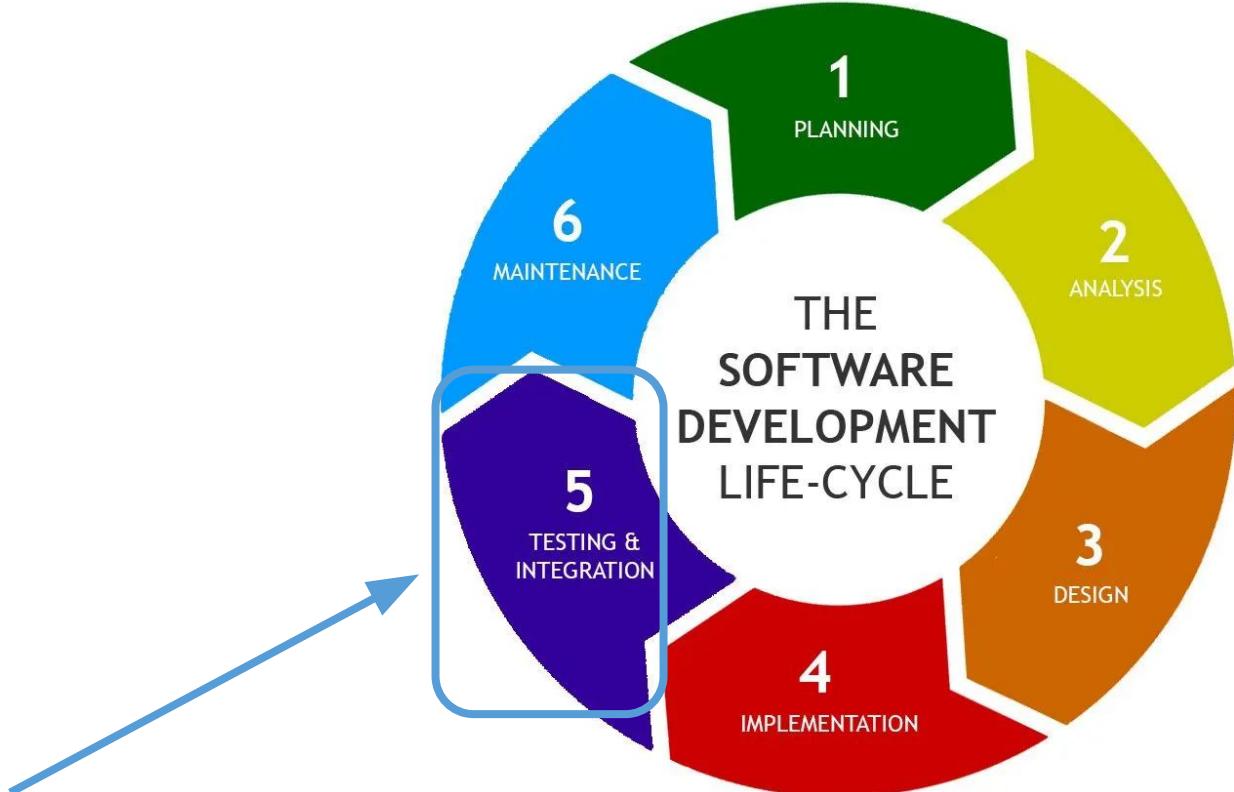
Did you test  
your software before?





Every time you fix code, you  
have to test the entire  
system again

What happens when you have  
to do this manual  
continuously in large  
projects?



# Software Testing

# Software Testing

- a testing process to ensure that the software we build (whether web or application) works as expected
- Testing occurs before the application reaches real users
- Main goal is to act as a "barrier" to help reduce problems that may occur when users start using it

Quick recap about 3 tiers  
and  
What is black box, white  
box?

| Perspective | QA Testing                                | Dev Testing                              |
|-------------|---|--|
| Focus       | Black Box Testing (The System as a Whole) | White Box Testing (Each Individual Part) |
| Detail      | Doesn't care about implementation details | Must know the module's specification     |
| Goal        | Does the system work according to spec?   | Does my module work as expected?         |
| Problem     | "The software is broken."                 | "Which specific part needs to be fixed?" |

A QA is like a driver who only cares if the car starts and drives.

A Dev is the mechanic who needs to know if the engine's timing is correct.





# Real life example

**Situation:** The website crashes when a user tries to register with an email and password.

**QA's:** "The website is broken, it's unusable!"

**Dev's:** "Is it the website's fault, or is the bug inside the register api?"

This is why we need to test the register api in isolation, without needing to run the entire website.

QA Testing cannot replace Dev Testing. They have different goals.

# Types of Testing

# 1. Non-Functional Testing

# Performance Testing

- **Load Testing:** Normal expected load (e.g., 100 concurrent users)
- **Stress Testing:** Beyond normal capacity (e.g., 1000+ users)
- **Endurance Testing:** Long-term stability (e.g., 24-hour continuous load)
- **Volume Testing:** Large amounts of data processing
- **Spike Testing:** Sudden increases in load



<https://k6.io/>

Grafana Labs

# Security Testing

- **Vulnerability Testing:** Scanning for known security flaw
- **Penetration Testing:** Simulated attacks by ethical hackers
- **Authentication Testing:** Login, session management, access control
- **Data Protection:** Encryption, privacy compliance (GDPR, HIPAA, PDPA)

# Usability Testing

- **User Experience (UX)**: Navigation, layout, user flows
- **Accessibility Testing**: Screen readers, keyboard navigation
- **User Interface(UI) (Screenshot testing)**: Visual design, consistency, responsiveness

# Reliability Testing

- **Failover Testing:** System recovery from failures
- **Disaster Recovery:** Backup and restore procedures
- **Uptime Testing:** 99.9% availability requirements

# 2. Functional Testing

# Functional Testing

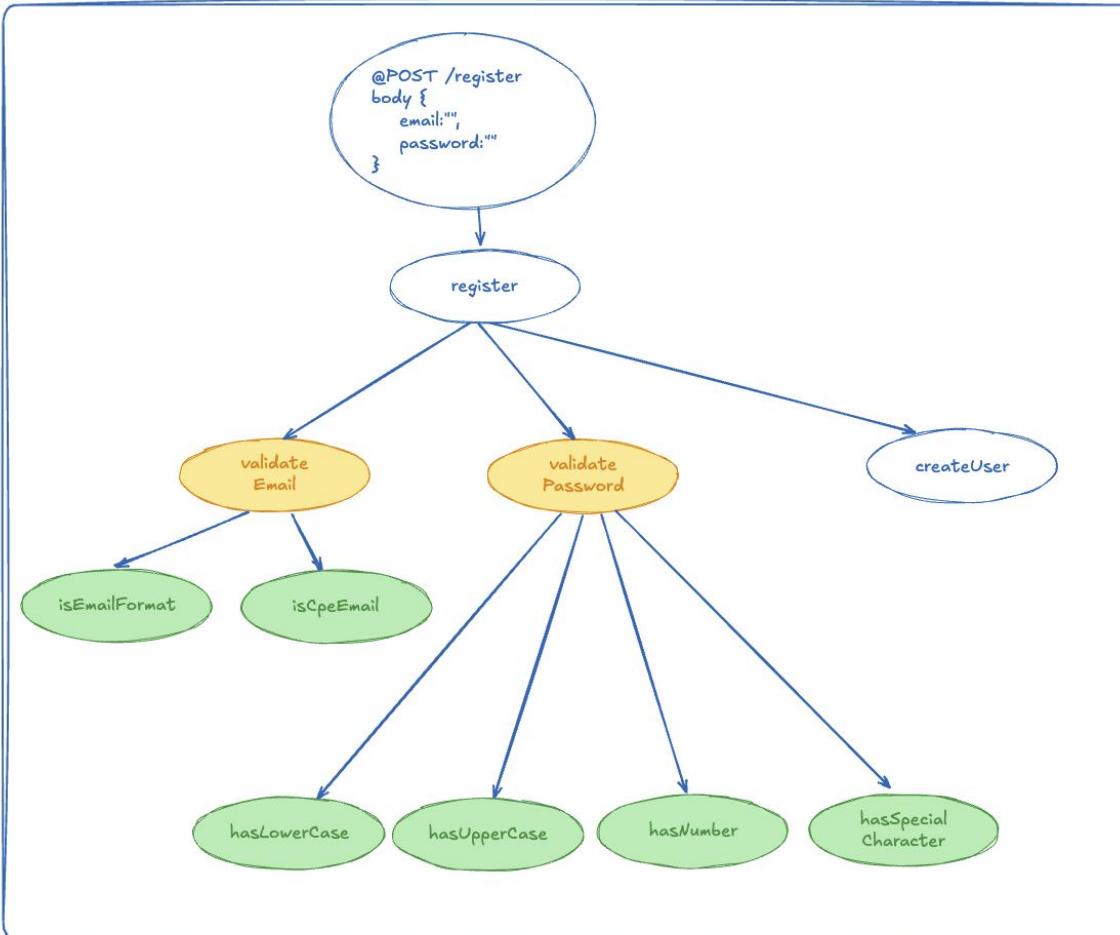
- focuses on testing the functions within the software to ensure they meet our expectations or requirements.

# Unit Test

# Unit Test

- Test on **unit** part (The team must agree on what constitutes a "unit" for your project).
- **Normally unit is**
  - Function (need to be Pure Function)
  - Class, method (need to use Dependency Injection if has Dependencies)
- **Primarily White Box:** The tester knows exactly how the function works internally, allowing easy identification of problem locations
- **the easiest type of test to write**, but mapping the result back to the overall system picture is more difficult
- focuses on the **expectation** of the code itself

Backend



# Example Unit Testing tools

- Java (JUnit)
- Javascript or Typescript (jest, vitest, bun test)
- Python (pytest)
- Dart/Flutter (Dart testing)

# Function (need to be Pure Function)

```
export const CPE_EMAIL_DOMAIN = '@cpe.kmutt.ac.th'

export function isCpeEmail(email: string): boolean {
  return email.endsWith(CPE_EMAIL_DOMAIN);
}
```



# Function (need to be Pure Function)

```
import { describe, test, expect } from "bun:test";

describe("isCpeEmail", () => {
    test("should return true for CPE email addresses", () => {
        expect(isCpeEmail("tipatai.p@cpe.kmutt.ac.th")).toBe(true);
    });

    test("should return false for non-CPE email addresses", () => {
        expect(isCpeEmail("john.doe@example.com")).toBe(false);
    });

    test("should return false for empty email addresses", () => {
        expect(isCpeEmail("")).toBe(false);
    });
});
```

# The AAA Pattern (Arrange, Act, Assert)

- **Arrange:** Prepare data and dependencies.
- **Act:** Call the function you want to test.
- **Assert:** Check if the result is correct.

\*\*Someone prefers “given when then”

# The AAA Pattern (Arrange, Act, Assert)

```
import { describe, test, expect } from "bun:test";

describe("isCpeEmail", () => {
    test("should return true for CPE email addresses", () => {
        // Arrange
        const email = "tipatai.p@cpe.kmutt.ac.th";

        // Act
        const result = isCpeEmail(email);

        // Assert
        expect(result).toBe(true);
    });
});
```

# Impure Function

```
export interface TimeRangeInHHmm {
  start: string; // e.g., "09:00"
  end: string;   // e.g., "17:30"
}

export interface Place {
  id: string;
  name: string;
  serviceHourRange: TimeRangeInHHmm;
}

function isPlaceOpen(place: Place): boolean {
  const now = new Date();
  return isDateInTimeRange(now, place.serviceHourRange);
}
```

# Impure Function

Your function is not "pure" if it:

- Connects to an API
- Have mutable global state
- Have side effect
- Accesses a Database
- Read/Write files
- Depends on an 3rd party/external service
- Depends on DateTime, Random Number

Solution: Make it Pure Function

```
function isPlaceOpen(place: Place, now: Date): boolean {  
    return isDateInTimeRange(now, place.serviceHourRange);  
}
```

```
describe("isPlaceOpen function", () => {
  test("should return true when current time is within same-day service hours", () => {
    // Arrange
    const mockDate = new Date("2024-01-15T10:30:00+07:00"); // 10:30 AM Bangkok time
    const place: Place = {
      id: "1",
      name: "Test Place",
      serviceHourRange: { start: "09:00", end: "17:00" }
    };

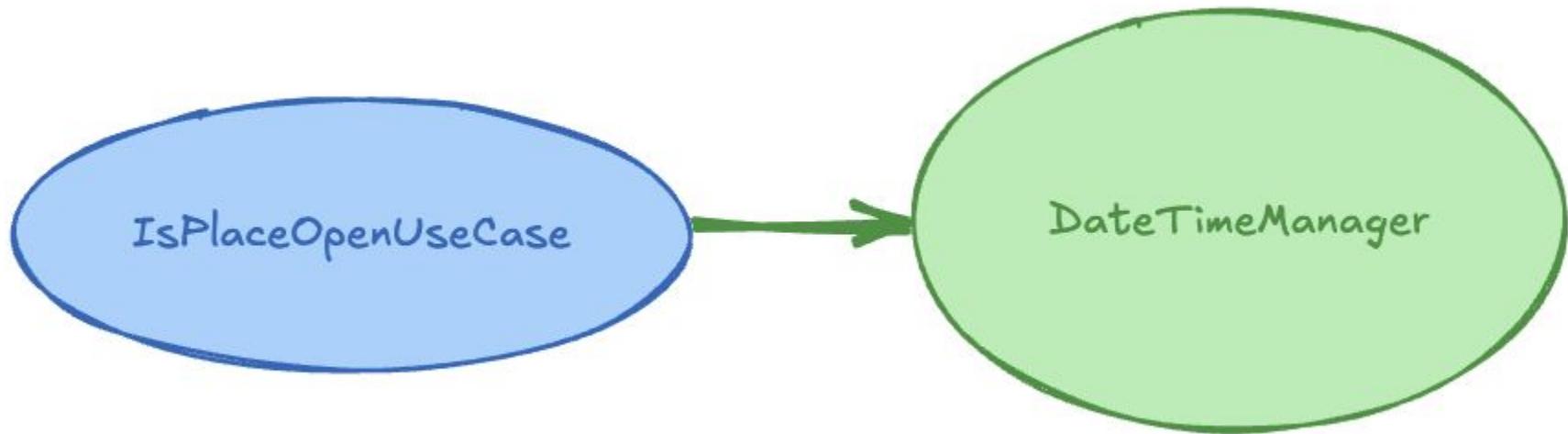
    // Act
    const result = isPlaceOpen(place, mockDate);

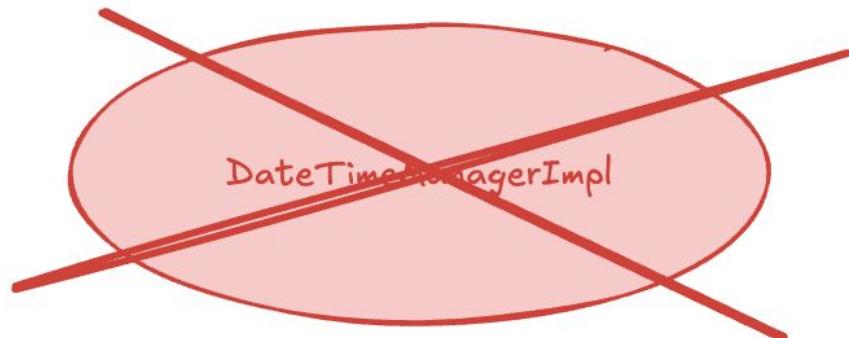
    // Assert
    expect(result).toBe(true);
  });
}
```

# Solution: Dependency Injection

# Dependency Injection (DI)

- Provide dependencies from the outside. (normally by constructor)
- Wrap uncontrolled factor (API, Database, DateTime) by interface/class
- Test Doubles (Mocks/Stubs): Create fake versions of dependencies for your tests.
- This allows you to turn an "impure" function into a "pure" one during testing, making it fast and predictable.





# Wrapped by interface / class

```
export interface DateTimeManager {  
    now(): Date;  
}  
  
export class DateTimeManagerImpl implements DateTimeManager {  
    public now(): Date {  
        return new Date();  
    }  
}
```

# Convert function to class method

```
export class IsPlaceOpenUseCase {  
    constructor(  
        private readonly dateTimeManager: DateTimeManager,  
    ) {  
    }  
  
    async execute(place: Place): Promise<boolean> {  
        const now = this.dateTimeManager.now();  
        return isDateInTimeRange(now, place.serviceHourRange)  
    }  
}  
  
const isPlaceOpenUseCase = new IsPlaceOpenUseCase(new DateTimeManagerImpl());
```

```
describe("IsPlaceOpenUseCase", () => {
  let useCase: IsPlaceOpenUseCase;
  const mockDateTimeManager = { now: mock() };
  beforeEach(() => { useCase = new IsPlaceOpenUseCase(mockDateTimeManager); });
  afterEach(() => { mock.clearAllMocks(); });

  test("should return true when current time is within same-day service hours", async () => {
    // Arrange
    const place: Place = {
      id: "place-1",
      name: "Test Place",
      serviceHourRange: { start: "09:00", end: "17:00" }
    };
    mockDateTimeManager.now.mockReturnValue(new Date("2024-01-15T10:30:00+07:00"));

    // Act
    const result = await useCase.execute(place);

    // Assert
    expect(result).toBe(true);
  });
}
```

# Tools for DI

- **NestJS** module
- **ElysiaJS** derived,decorate
- **TypeScript** Inversify
- **Java** SpringBoot
- **Flutter** GetIt, Riverpod

# Integration Test

# Integration Test

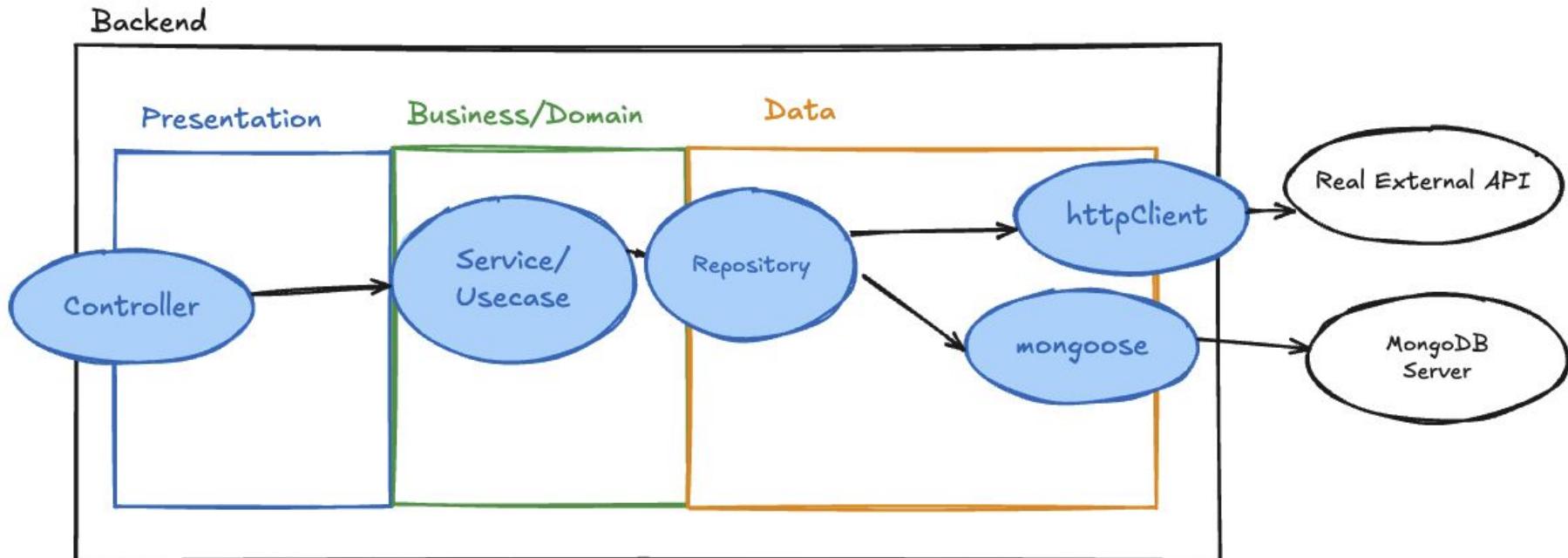
- Test how multiple units/components collaborate.
- Testing from the top-most layer (Presentation), through various logic (Domain), down to the data management layer (Data)
- The dependencies between your units should be real.
- For external layers (like APIs and databases) that the data layer connects to, I recommend configuring them not to connect to the live versions. You have two options for this.

# Approach 1 - Simulated External Services

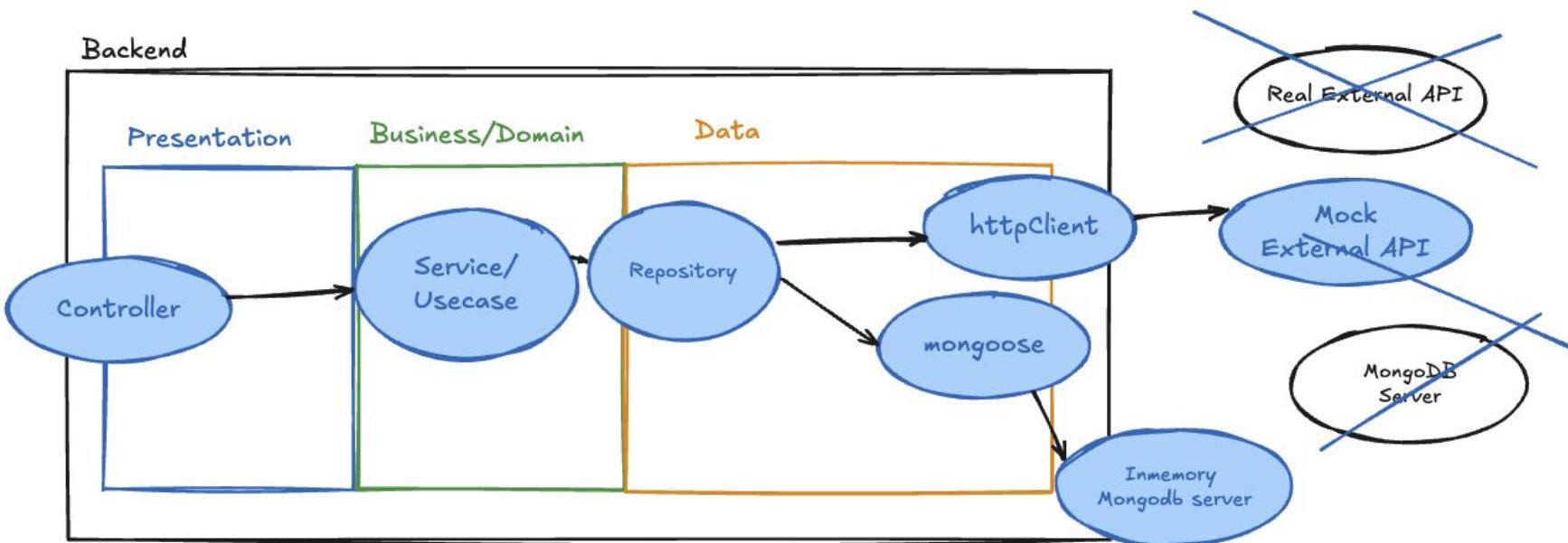
- Simulate the environment as closely as possible to the real one by using real classes in every layer and creating simulated external systems.
- This approach uses real classes in every layer, only simulating the external systems.

**Fake Server, In-Memory DB, Test containers**

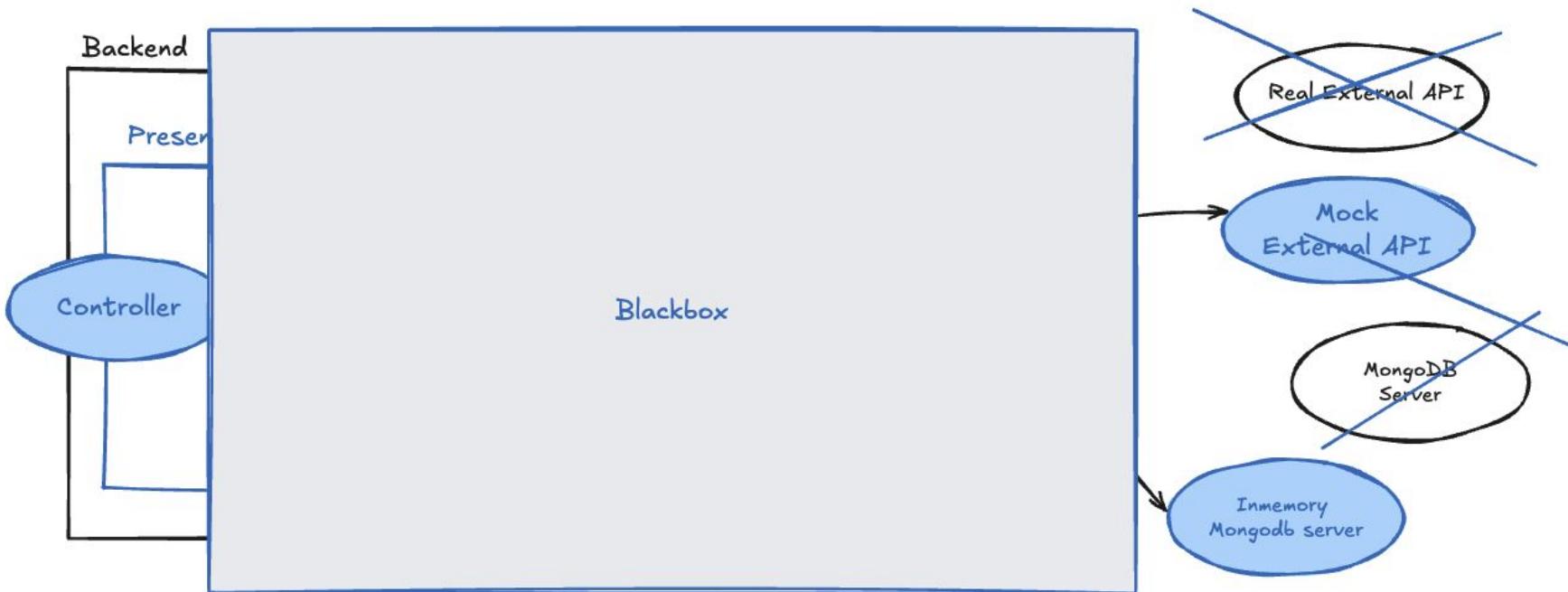
# Approach 1 - Simulated External Services



# Approach 1 - Simulated External Services



# Approach 1 - Simulated External Services



# Approach 1 - Pros and cons

## Pros:

- **✓ Most realistic:** Provides end-to-end testing coverage that is very similar to production.
- **✓ Tests all components:** Allows testing of data transformation logic (Serialization/Deserialization) and Header/Error handling within the Data Source.

## Cons:

- **✗ Complex and slow:** The setup is complex and test execution is slower.
- **✗ High maintenance:** If the external system changes (e.g., API spec update), the simulation code must also be updated.

# Simulated External Services tools

Fake Server

<https://wiremock.org/>

In-Memory DB

<https://github.com/typegoose/mongodb-memory-server>

<https://github.com/mhassan1/redis-memory-server>

<https://sqlite.org/>

Test containers

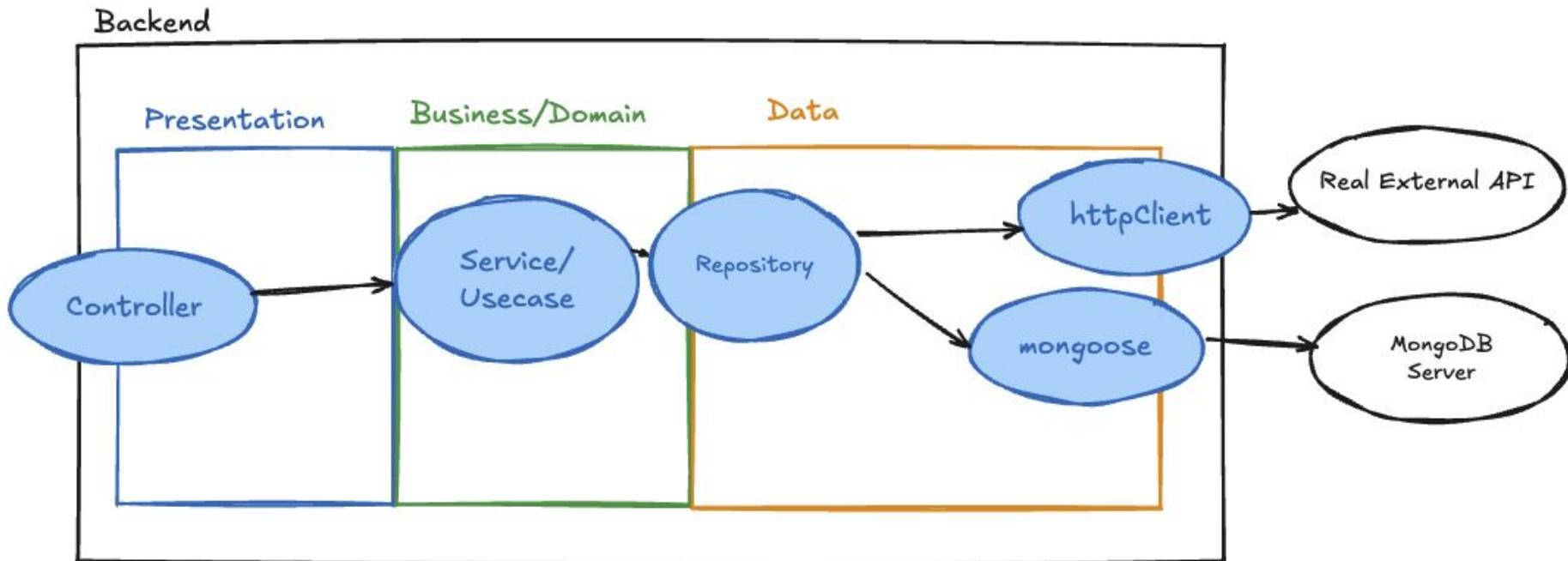
<https://testcontainers.com/>

# Approach 2 - Mocking the Intermediary Class

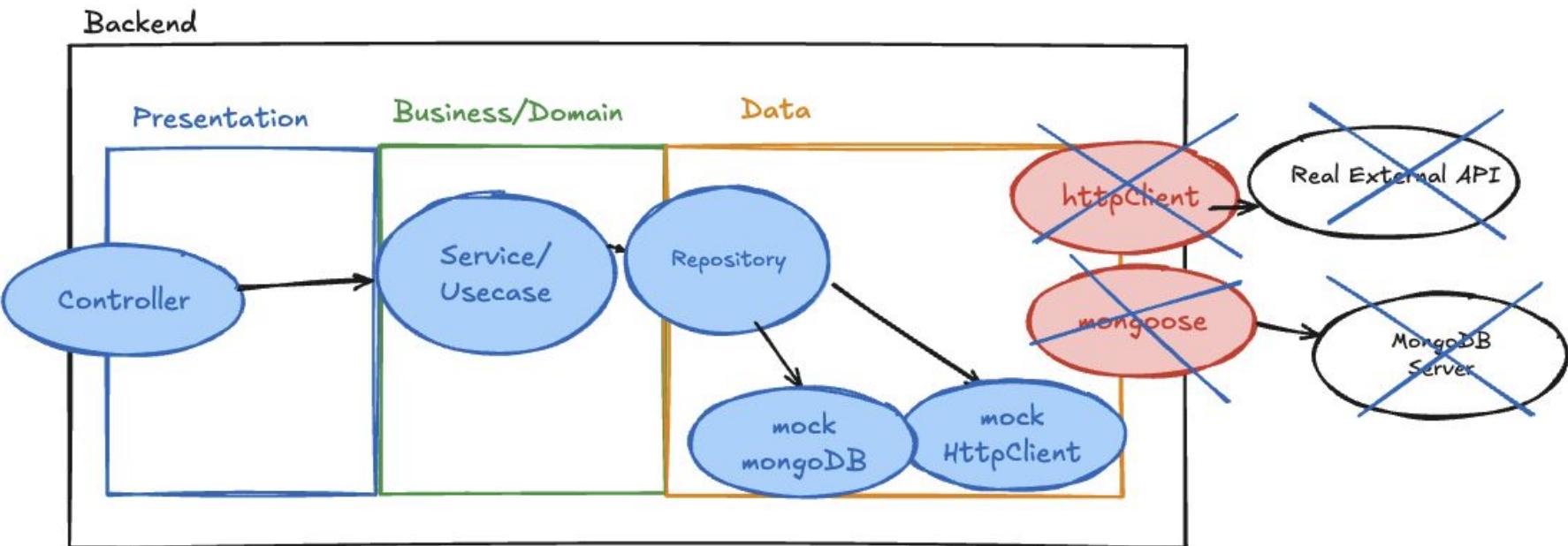
## Fake Server, In-Memory DB, Test containers

- Required Dependency Injection
- Focus on speed and simplicity. Use real classes for almost all layers but mock the specific class that directly interfaces with the external system.
- Use a mocking library (e.g., Mockito, MockK) to create a fake DataSource class (or you can manually construct it without libs). Define the behavior of the mocked class to specify what value to return or what exception to throw.
- Only the DataSource layer is mocked - all other layers use real implementations.

# Approach 2 - Mocking the Intermediary Class



# Approach 2 - Mocking the Intermediary Class



# Approach 2 - Pros and cons

## Pros:

-  **Fast and simple:** Quick to set up and tests run very fast.
-  **Easy to control:** Precisely define the desired outcomes for testing specific scenarios.

## Cons:

-  **Less realistic:** Does not test the actual connection to the external system.
-  **Incomplete coverage:** Skips testing the logic within the real DataSource (e.g., JSON conversion, header manipulation).

# Example

Backend: Testing a POST /users/register endpoint. The test would:

1. Start an in-memory database (e.g., MongoDB In memory server).
2. Send a real HTTP request to the registration endpoint running on a test server.
3. The UserService would process the request, hash the password, and save the new user.
4. The test would then connect directly to the in-memory database to assert that the user was created correctly with the hashed password.

# Example

**Mobile App:** Testing that a UserProfileScreen displays a user's name. The test would:

1. Use a mocking library (e.g., Mockito in Flutter) to create a MockUserRemoteDataSource.

2. Configure the mock to return a predefined user object:

```
when(mockDataSource.getUser()).thenAnswer((_) async => UserModel(name: 'Alice'));
```

3. Provide this mock to the real UserRepositoryImpl, which is then used by the real GetUserProfileUseCase and the screen's ProfileViewModel.

4. Render the UserProfileScreen in a test environment.

5. The ViewModel calls the full chain of real classes, but the call chain stops at the MockUserRemoteDataSource, which returns "Alice" without a network call.

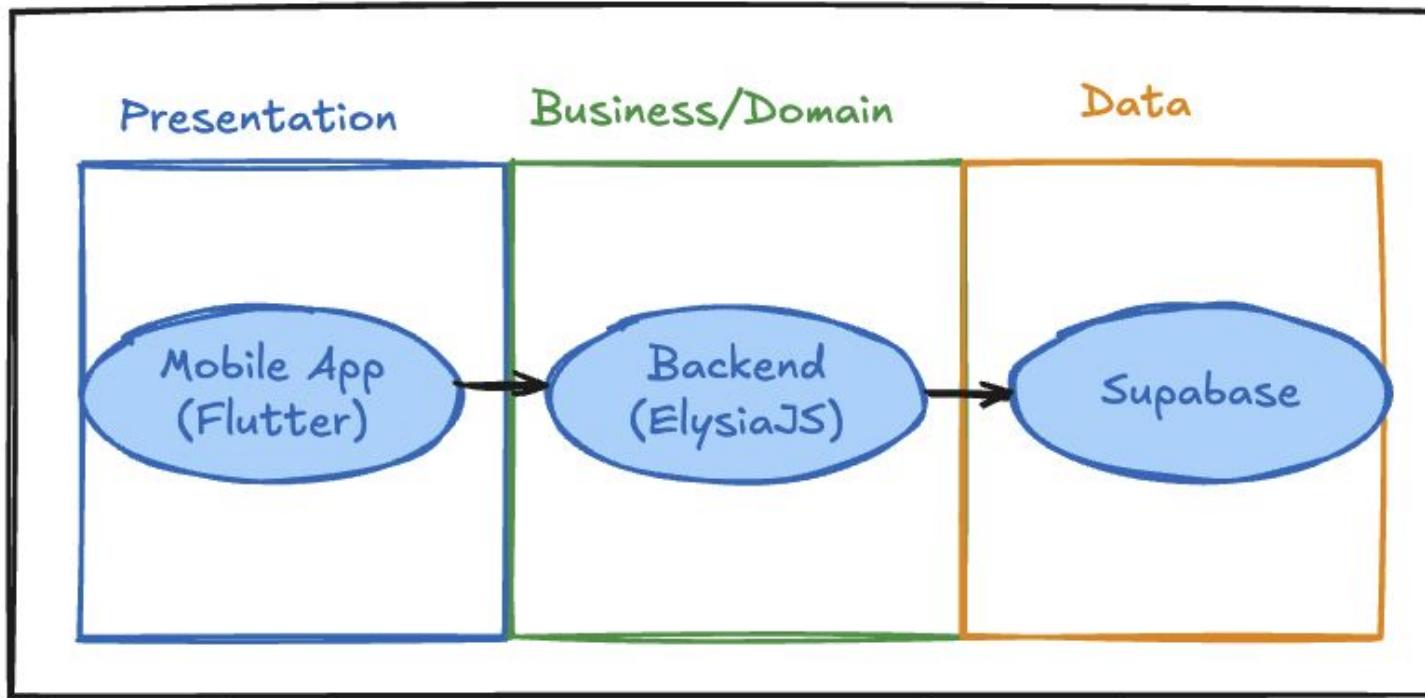
6. The test asserts that a widget with the text "Alice" is visible on the screen.

# E2E Test

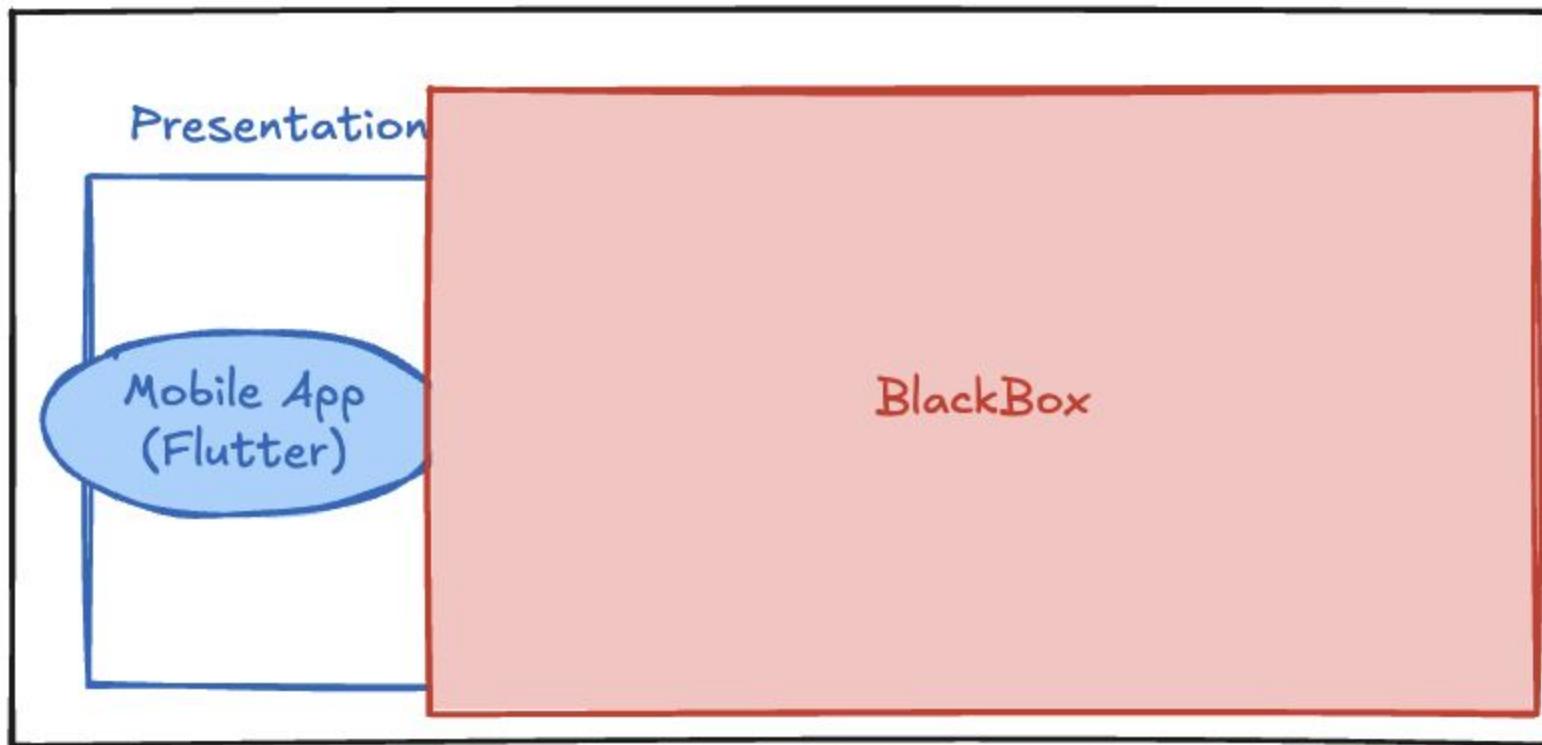
# E2E Test

- Test the application from end-to-end, just as a user would.  
(mostly UI Test)
- Don't worry about the internal implementation.
- The system should be as close to the production environment as possible.
- often cover the "happy path" to ensure critical user flows are working
- **Primarily Black Box:** The tester doesn't know how the function works internally, hard identification of problem locations
- Can align with QA Test case scenarios

## Project Architecture Diagram

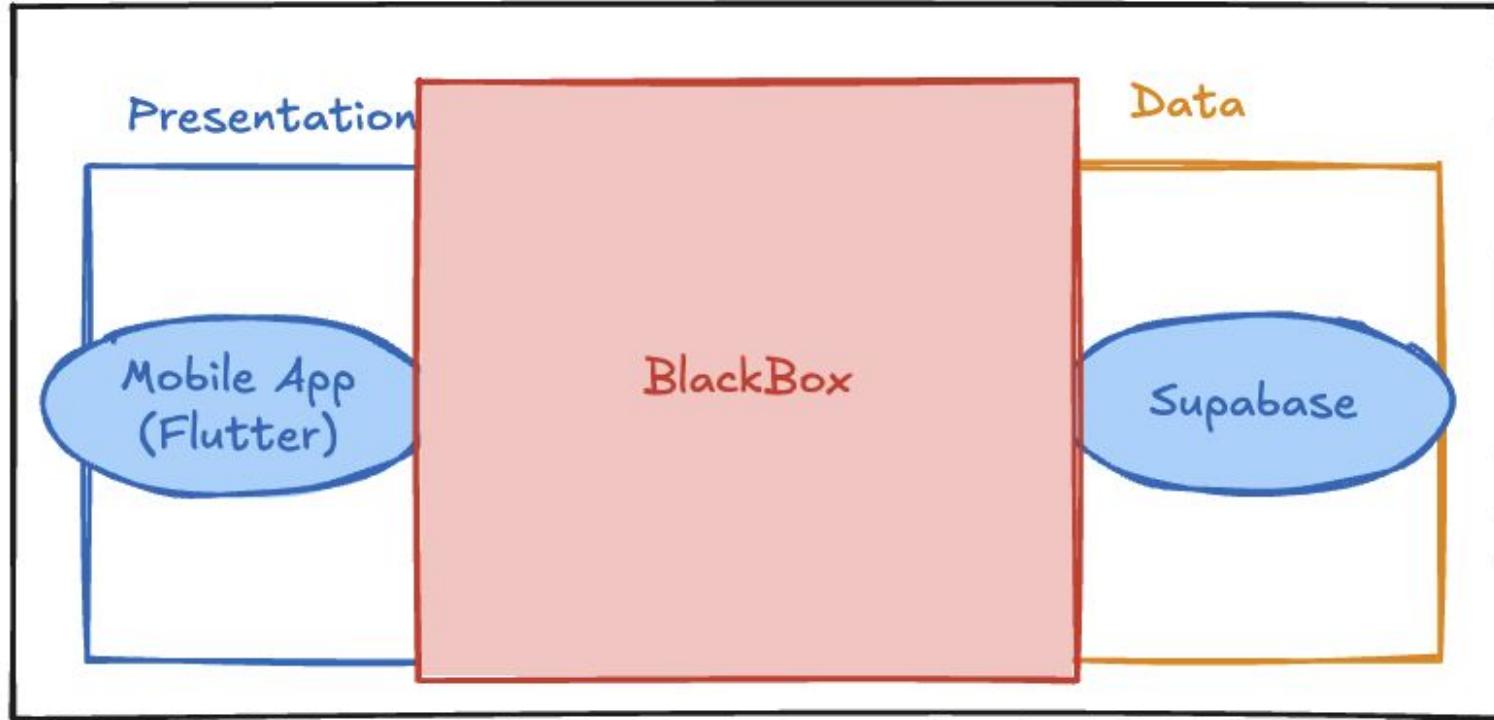


# Project Architecture Diagram



# Some situation

Project Architecture Diagram



# Example

**Web** A script opens the browser, navigates to the login page, enters credentials, adds a product to the cart, and completes the checkout.

# Tools for E2E (or Integration)

**Web** - Playwright, Cypress, Selenium

**Mobile** - Flutter Test, Appium, Robot Framework

**Backend** - Postman, Playwright

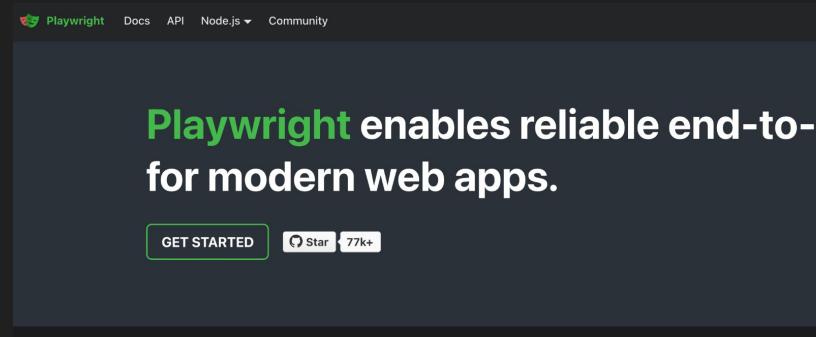
# Playwright example

```
test('get started link', async ({ page }) => {
  await page.goto('https://playwright.dev/');

  // Expect a title "to contain" a substring.
  await expect(page).toHaveTitle(/Playwright/);

  // Click the get started link.
  await page.getByRole('link', { name: 'Get started' }).click();

  // Expects page to have a heading with the name of Installation.
  await expect(page.getByRole('heading', { name: 'Installation' })).toBeVisible();
});
```



The problem of over-testing  
(integration, e2e)

# The problem of over-testing (integration, e2e)

**Takes too long:** The test suite runs so slowly that no one wants to run it.

**Hard to debug:** "100 tests failed" → Where is the actual bug?

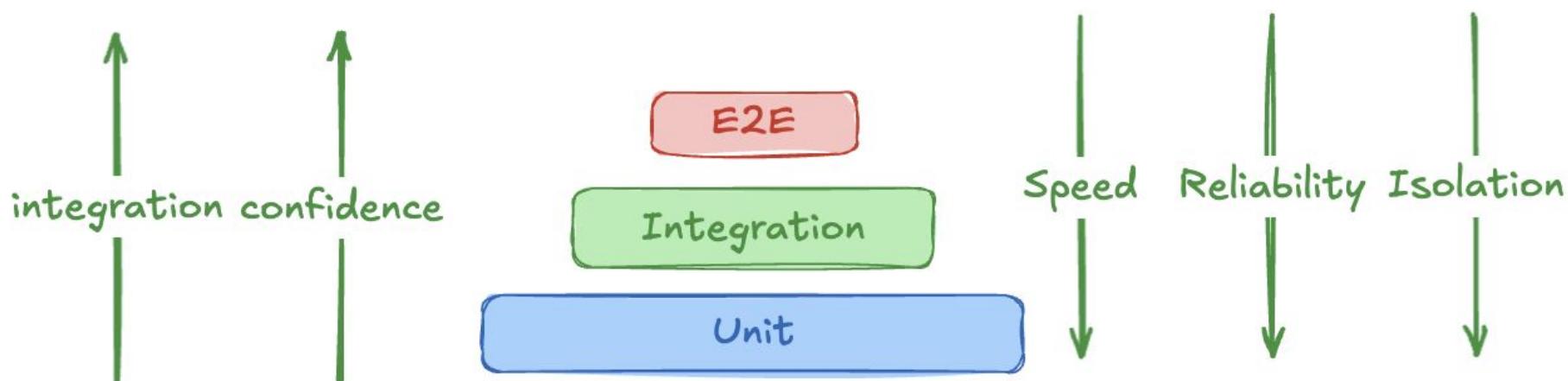
**Brittle and high maintenance:** A small requirement change breaks hundreds of test cases.

**False confidence:** Teams start ignoring or skipping failing tests to meet deadlines.

If your tests are painful, you're doing it wrong!

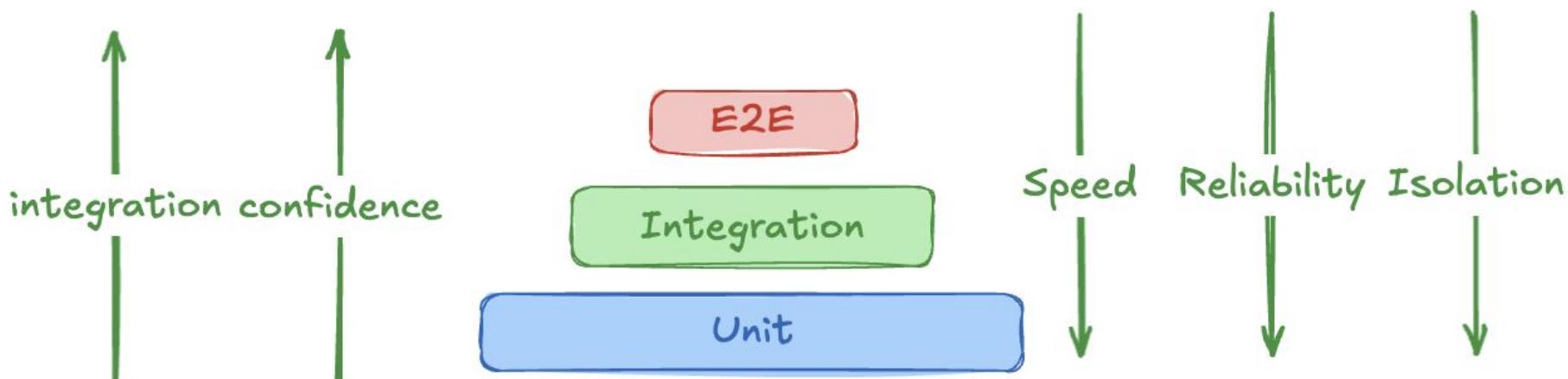
# The Test Pyramid Philosophy

- Write significantly more Unit Tests than Integration Tests, and more Integration Tests than E2E Tests.



# The Sampling Technique

- Your Integration/E2E Tests should be a sample of the most important interactions.
- Don't re-test every edge case that your Unit Tests already cover.

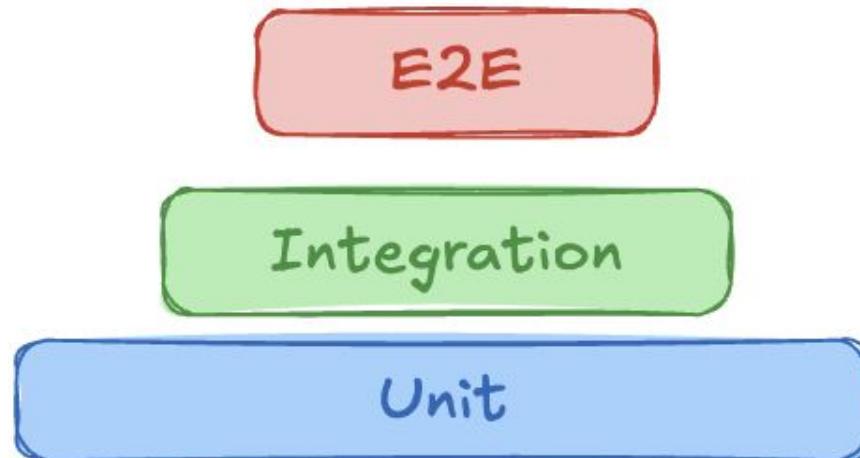


The goal is the best possible quality assurance for the lowest cost and effort.

When should you follow  
the Test Pyramid or not?

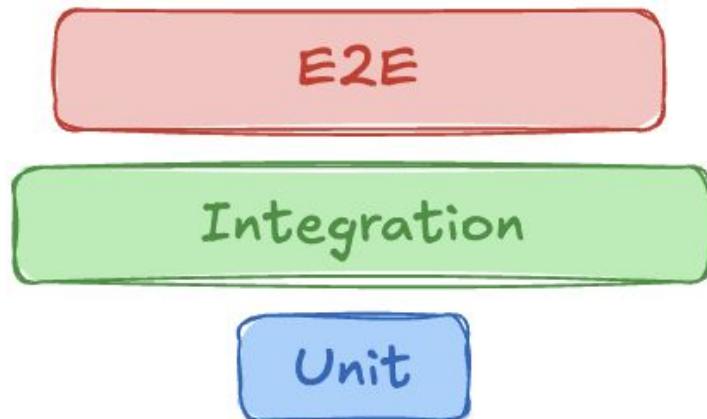
# Have time to focus on quality, design great unit?

Follow the Test Pyramid.



# Rushing a feature or building a prototype?

It's might be faster to focus on Integration/E2E tests first, then go back and add Unit Tests later during refactoring.



The perfect testing strategy is the one  
that fits your project's timeline and resources.

What makes  
a *GOOD* test?

# F.I.R.S.T principles

**Fast:** Tests should run quickly. Slow tests are a drag on development.

**Independent:** Tests should not depend on each other or a shared state.

**Repeatable:** Tests must produce the same result every time, in any environment.

**Self-Validating:** The test itself determines pass or fail. No manual check is needed.

**Timely:** Write tests just before or alongside the production code they test.

Following these principles prevents flaky tests and makes your test suite a reliable safety net.

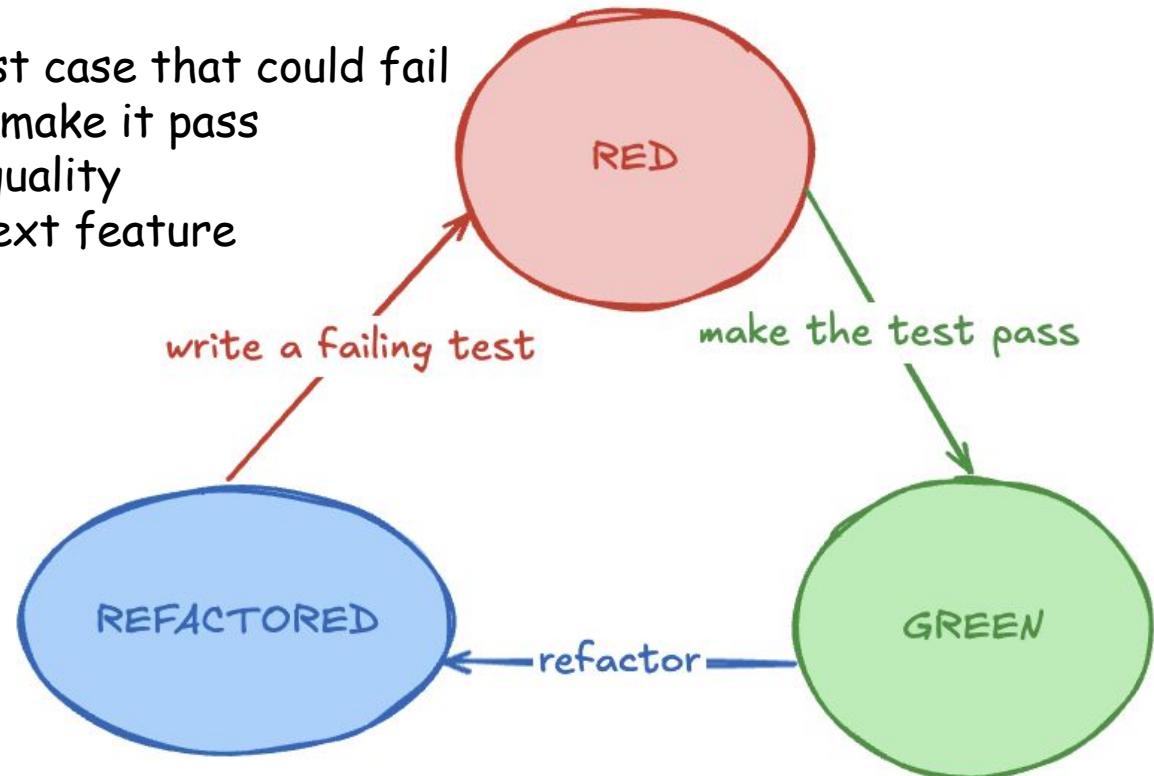
# Test Driven Development

# Test Driven Development (TDD)

- A development approach where you write tests before writing the actual code
- Follows the Red-Green-Refactor cycle to build software incrementally
- Tests serve as both specifications and safety nets

# TDD Cycle

- Start with the simplest test case that could fail
- Write just enough code to make it pass
- Refactor to improve code quality
- Repeat the cycle for the next feature



# Benefits of TDD

- **Better Design:** Writing tests first forces you to think about the interface
- **Fewer Bugs:** Catch issues early in the development cycle
- **Living Documentation:** Tests serve as executable specifications
- **Confident Refactoring:** Change code knowing tests will catch regressions

# When TDD Works Best

- **New features** with clear requirements
- **Bug fixes** (write test that reproduces the bug first)
- **Refactoring** existing code
- **Function/SDK/API design** (tests help define the interface)

# TDD Example

# Initialize function signature

```
export function sumNumbers(numbers: number[]): number {  
    // TODO: impl  
    return 0  
}
```

# Writing the failing tests

```
describe("sumNumbers", () => {
    test("should return 0 for an empty array", () => {
        expect(sumNumbers([])).toBe(0);
    });
    test("should return first number for an array with one number", () => {
        expect(sumNumbers([1])).toBe(1);
    });
    test("should return the sum of the array with two numbers", () => {
        expect(sumNumbers([1, 2])).toBe(3);
    });
    test("should return the sum of the array with three numbers", () => {
        expect(sumNumbers([1, 2, 3])).toBe(6);
    });
});
```

```
bun run test
```

#### Bun for Visual Studio Code

- ✖ should return first number for an array with one number expect(received).toBe(expected)
- ✖ should return the sum of the array with two numbers expect(received).toBe(expected)
- ✖ should return the sum of the array with three numbers expect(received).toBe(expected)
- ✓ should return 0 for an empty array

# Make the test pass

```
export function sumNumbers(numbers: number[]): number {
  let sum = 0;
  for (const number of numbers) {
    sum += number;
  }
  return sum;
}
```

```
bun run test
```

## Bun for Visual Studio Code

- ✓ should return first number for an array with one number
- ✓ should return the sum of the array with two numbers
- ✓ should return the sum of the array with three numbers
- ✓ should return 0 for an empty array

# Refactoring

```
export function sumNumbers(numbers: number[]): number {  
    return numbers.reduce((sum, number) => sum + number, 0);  
}
```

```
bun run test
```

## Bun for Visual Studio Code

- ✓ should return first number for an array with one number
- ✓ should return the sum of the array with two numbers
- ✓ should return the sum of the array with three numbers
- ✓ should return 0 for an empty array

# Code Coverage

# Code Coverage

- A metric that measures how much of your source code is executed when your tests run.
- It tells you which lines, branches, or functions are covered by your test suite.
- Mostly generated by unit testing, sometimes by integration testing

# Types of Code Coverage

**Line Coverage:** Percentage of code lines executed

**Branch Coverage:** Percentage of conditional branches tested

**Function Coverage:** Percentage of functions called

**Statement Coverage:** Percentage of statements executed

<https://bun.com/docs/test/coverage>

| File        | % Funcs | % Lines | Uncovered Line #s |
|-------------|---------|---------|-------------------|
| All files   | 38.89   | 42.11   |                   |
| index-0.ts  | 33.33   | 36.84   | 10-15,19-24       |
| index-1.ts  | 33.33   | 36.84   | 10-15,19-24       |
| index-10.ts | 33.33   | 36.84   | 10-15,19-24       |
| index-2.ts  | 33.33   | 36.84   | 10-15,19-24       |
| index-3.ts  | 33.33   | 36.84   | 10-15,19-24       |
| index-4.ts  | 33.33   | 36.84   | 10-15,19-24       |
| index-5.ts  | 33.33   | 36.84   | 10-15,19-24       |
| index-6.ts  | 33.33   | 36.84   | 10-15,19-24       |
| index-7.ts  | 33.33   | 36.84   | 10-15,19-24       |
| index-8.ts  | 33.33   | 36.84   | 10-15,19-24       |
| index-9.ts  | 33.33   | 36.84   | 10-15,19-24       |
| index.ts    | 100.00  | 100.00  |                   |

# The Coverage Reality Check

"80% coverage\*\* with good tests"  
better than  
"95% coverage with poor tests"

Quality over Quantity: Focus on testing critical business logic thoroughly

Coverage is a tool, not a goal: It helps identify untested code, but doesn't guarantee bug-free software

High coverage without good test quality gives you false confidence. Aim for meaningful coverage of your most important code paths.

# Break

New user?

Get 50% off your first trip  
with discount code

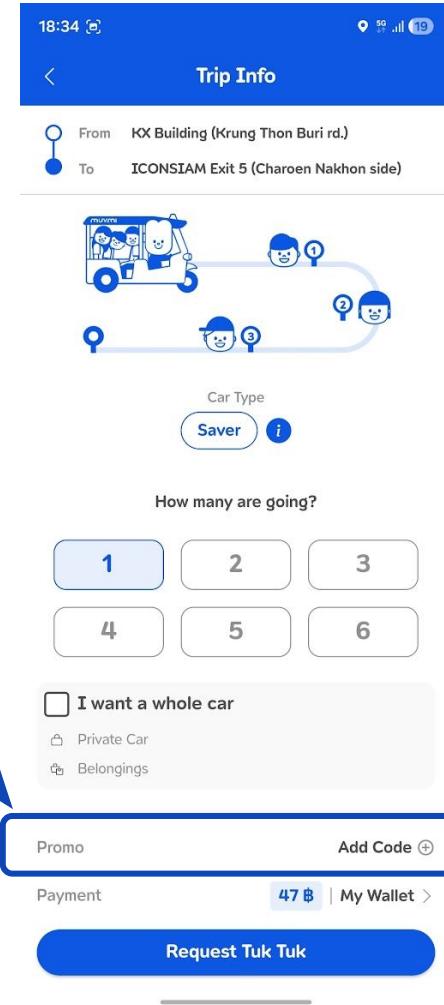
CPE50

MuvMi App



Download on the  
App Store

GET IT ON  
Google Play



# Live coding

# Requirements

- The system shall provide a function to register a new user using their email and password. On successful validation, the function shall create a new user record in the database.
- Input `email:string, password:string` Output {`success:boolean, error?:string`}
- **Email Validation:**
  - The `email` must be a valid email format.
  - The `email` domain **must** be `@cpe.kmutt.ac.th`.
- **Password Validation:**
  - **Length:** The `password` must be between **8 and 32 characters inclusive**.
  - **Complexity:** The `password` must contain at least:
    - One lowercase letter (a-z).
    - One uppercase letter (A-Z).
    - One number (0-9).
- Each validation error need to show actual reason message
- No need to implement function `createUserToDatabase(email,password)`

# More Register Requirements

- api "@POST register"
- can register only in 8:00-22:00
- send email to customer when register success

# Summary

- The crucial differences between Dev and QA testing.
- How to use the Testing Pyramid to guide your strategy.
- Functional vs Non Functional Testing
- Pure Function, Dependency Injection, Mock
- How to apply Unit, Integration, E2E, and Manual testing appropriately.
- How to avoid the trap of over-testing by seeking balance.

Testing isn't a burden. It's the tool that gives you the freedom and confidence to build great software

Thank you  
Happy coding + testing :)

Facebook : Tipatai Puthanukunkit

Instagram: judrummer

Github: judrummer

Email: judrummercpe@gmail.com

Lab