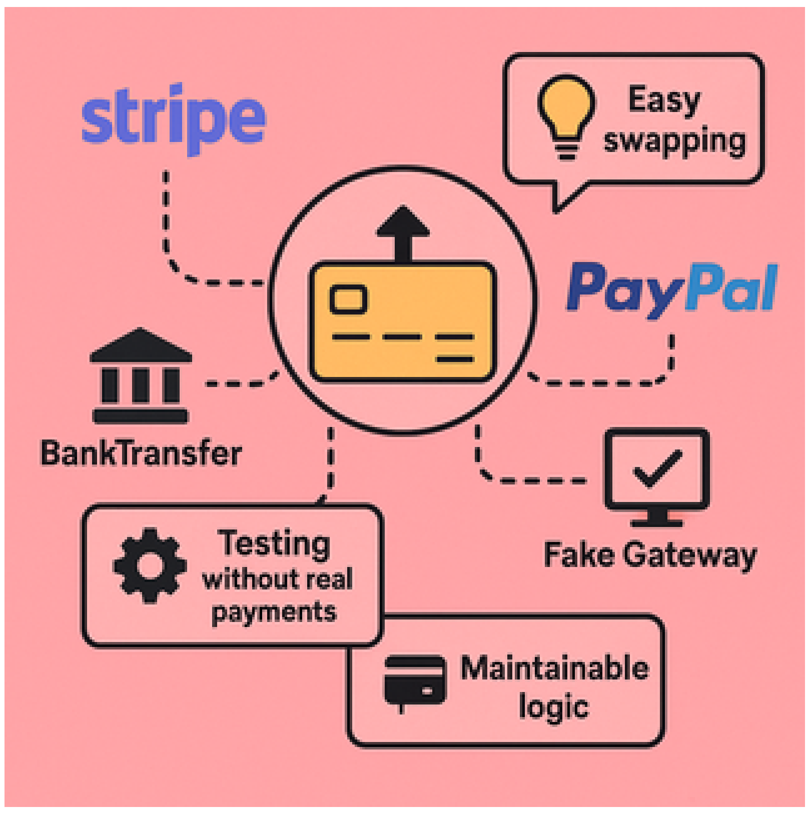
Topic 6: Dependency Injection in TypeScript (Constructor Injection, Interfaces for Contracts, Basic IoC)

1. Problem Statement

Scenario: Modern Payment Processing System

You're building a payment processing platform that must:

- Support different payment gateways (Stripe, PayPal, BankTransfer).
- Allow easy swapping or upgrading of gateways (e.g., for new regions).
- Enable testing with fake gateways (no real transactions).
- Keep the payment logic focused and maintainable.



The problem:
How do you provide the payment module with the right gateway, swap gateways easily, and test without real payments **while keeping your payment logic decoupled and flexible**?

2. Learning Objectives

By the end of this lesson, you'll be able to:

- Understand what Dependency Injection (DI) and Inversion of Control (IoC) are.
- Use constructor injection to supply dependencies.
- Define interfaces as contracts for dependencies.
- Build flexible, testable, and maintainable TypeScript systems using DI.

3. Concept Introduction with Analogy

Analogy: Plug-and-Play Power Sockets

Imagine a payment terminal (like a card reader) in a store:

- The terminal doesn't care what kind of plug (gateway) is used; it just needs a compatible socket.

- You can plug in a Stripe adapter, a PayPal adapter, or a test adapter for training.
- If you upgrade to a faster plug, you don't need to change the terminal-just swap the plug.

Dependency Injection is like using a universal socket:
You can swap adapters (dependencies) without changing the device (business logic).

What is Dependency Injection?

- **Dependency Injection (DI)** is a design pattern where an object's dependencies are provided ("injected") from outside, rather than hardcoded inside the object.
- **Inversion of Control (IoC):** The control of creating and supplying dependencies is inverted-handled by an external entity, not the object itself.

Why Use DI?

- **Decoupling:** The main logic doesn't depend on specific implementations.
- **Testability:** You can inject mocks or fakes for testing.
- **Flexibility:** Swap implementations without changing business logic.
- **Maintainability:** Changes to dependencies don't ripple through the codebase.

DI in TypeScript: How It Works

- **Interfaces** define contracts for dependencies.
- **Constructor injection** is the most common DI method.
- **IoC containers** (advanced) can automate dependency resolution (not covered in depth here).

4. Step-by-Step Data Modeling & Code Walkthrough

Define a Contract (Interface) for Payment Gateways
Interfaces define the expected behavior without implementation details.

```
interface PaymentGateway {
  processPayment(amount: number): Promise<boolean>;
}
```

This interface ensures any payment gateway class implements the `processPayment` method.

Implement Concrete Payment Gateways
Each gateway implements the interface with its own logic.

```
class StripeGateway implements PaymentGateway {
  async processPayment(amount: number): Promise<boolean> {
    console.log(`Processing payment of $${amount} via Stripe.`);
    // Simulate API call...
    return true;
  }
}

class PaypalGateway implements PaymentGateway {
  async processPayment(amount: number): Promise<boolean> {
    console.log(`Processing payment of $${amount} via PayPal.`);
    // Simulate API call...
    return true;
  }
}
```

Create the Payment Processor Using Constructor Injection

The payment processor receives the gateway via its constructor.

```
class PaymentProcessor {
  constructor(private gateway: PaymentGateway) {}

  async pay(amount: number): Promise<void> {
    const success = await this.gateway.processPayment(amount);
    if (success) {
      console.log("Payment successful!");
    } else {
      console.log("Payment failed.");
    }
  }
}
```

- The processor **does not create** the gateway; it receives it.
- This allows any class implementing `PaymentGateway` to be used.

Using Different Gateways

```
const stripeGateway = new StripeGateway();
const paypalGateway = new PaypalGateway();

const processor1 = new PaymentProcessor(stripeGateway);
processor1.pay(100); // Uses Stripe

const processor2 = new PaymentProcessor(paypalGateway);
processor2.pay(200); // Uses PayPal
```

Testing with Mock Gateways

For testing, inject a mock gateway that simulates payment without real transactions.

```
class MockGateway implements PaymentGateway {
  async processPayment(amount: number): Promise<boolean> {
    console.log(`Mock processing payment of $${amount}.`);
    return true;
  }
}

const mockGateway = new MockGateway();
const testProcessor = new PaymentProcessor(mockGateway);
testProcessor.pay(50); // Uses mock gateway for testing
```

5. Challenge

- Implement a new gateway class `BankTransferGateway` that logs payment processing.
- Use it with `PaymentProcessor` to process a payment.
- Write a mock gateway that simulates failure (`return false`) and test error handling.

6. Quick Recap & Key Takeaways

- **Dependency Injection** means supplying dependencies from outside, not creating them inside.
- **Constructor Injection** is the most common DI method in TypeScript.
- **Interfaces** define contracts that enable swapping implementations.
- DI improves **flexibility**, **testability**, and **maintainability**.

7. (Optional) Programmer's Workflow Checklist

- Identify dependencies your class needs.
- Define interfaces for those dependencies.
- Inject dependencies via constructors.
- For testing, inject mocks or stubs.
- Avoid creating dependencies inside business logic classes.

8. Coming up

You've mastered Dependency Injection basics!
Next, explore **IoC Containers** (like InversifyJS) for automatic dependency management and advanced DI features.