Dependency Injection

Breaking the Chains of Tight Coupling

What is Dependency Injection?

Design pattern that removes hard-coded dependencies

Dependencies are provided externally rather than created internally

Goal: Loose coupling, better testability, flexibility

The Problem

Tight Coupling - Hard to Test & Change

```
import { EmailService } from './EmailService';

export class UserService {
  private emailService = new EmailService(); // Hard dependency!

  async registerUser(email: string, name: string) {
    const user = { email, name, id: Math.random() };

    // This will always send real emails, even in tests!
    await this.emailService.sendWelcomeEmail(user);

    return user;
  }
}

// Hard to test - always uses real email service
const userService = new UserService();
```

The Solution

Dependency Injection - Flexible & Testable

```
interface EmailService {
    sendWelcomeEmail(user: any): Promise<void>;
}

export class UserService {
    constructor(private emailService: EmailService) {}  // Injected!

    async registerUser(email: string, name: string) {
        const user = { email, name, id: Math.random() };

        // Uses the injected email service
        await this.emailService.sendWelcomeEmail(user);

        return user;
    }
}
```

Types of Dependency Injection

Benefits

1. Testability

```
import { describe, it, expect, vi } from 'vitest';

// Easy to mock dependencies for testing
const mockEmailService: EmailService = {
    sendWelcomeEmail: vi.fn()
};

const userService = new UserService(mockEmailService);

// Test only the UserService logic
await userService.registerUser('test@example.com', 'John Doe');
expect(mockEmailService.sendWelcomeEmail).toHaveBeenCalled();
```

2. Flexibility

```
// Switch implementations easily
const prodUserService = new UserService(
    new SMTPEmailService()  // Real email service
);

const testUserService = new UserService(
    new MockEmailService()  // Fake email service for testing
);

const devUserService = new UserService(
    new ConsoleEmailService()  // Log emails to console
);
```

3. Single Responsibility

```
// Classes focus on their core responsibility
class UserService {
    // Only handles user registration logic
    // Doesn't worry about how emails are sent
    async registerUser(email: string, name: string) { /* ... */ }
}

class EmailService {
    // Only handles email sending
    async sendWelcomeEmail(user: any): Promise<void> { /* ... */ }
}
```

DI Containers

Manual DI

```
const emailService: EmailService = new SMTPEmailService(config.smtp);
const loggerService: Logger = new FileLogger();
const userService = new UserService(emailService);
const orderService = new OrderService(emailService, loggerService);
const paymentService = new PaymentService(emailService, loggerService);
const notificationService = new NotificationService(emailService);

// Can get hard to manage as app grows
```

DI Container - Automatic Wiring

More popular in Java and C# ecosystems (enterprise applications)

Example: InversifyJS

```
import { Container, injectable, inject } from 'inversify';

@injectable()
class EmailService {
    async sendWelcomeEmail(user: any) { /* implementation */ }
}

@injectable()
class UserService {
    constructor(@inject('EmailService') private emailService: EmailService) {}

    async registerUser(email: string, name: string) {
        const user = { email, name, id: Math.random() };
        await this.emailService.sendWelcomeEmail(user);
        return user;
    }
}

const container = new Container();
container.bind('EmailService').to(EmailService);
container.bind('UserService').to(UserService);

const userService = container.get<UserService>('UserService');
```

Best Practices

Keep Dependencies Minimal

Summary

Key Takeaways

Benefit	Description
Testability	Easy to mock dependencies
Flexibility	Swap implementations easily
Maintainability	Loose coupling, clear dependencies
Single Responsibility	Classes focus on core logic

Thank You!

Questions?