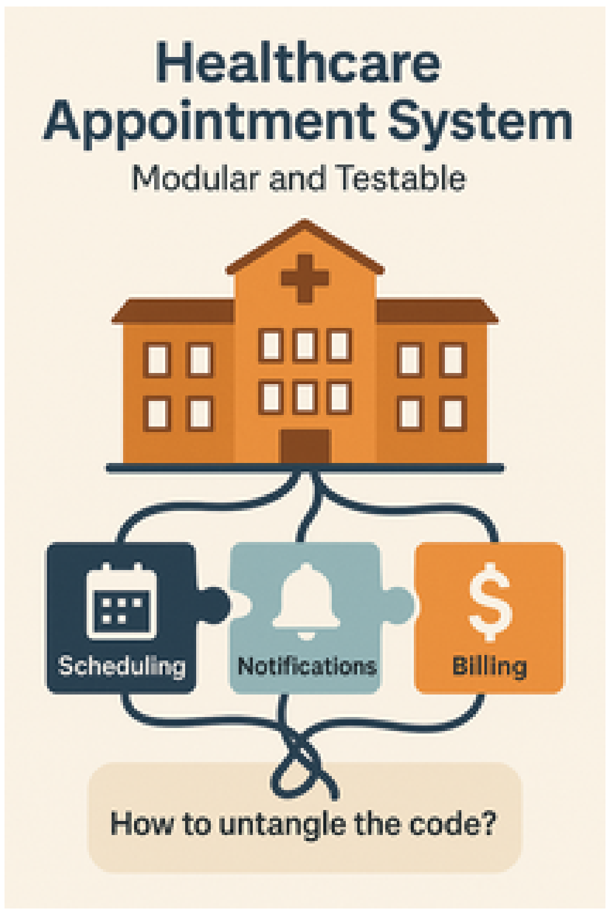
1. Problem Statement

Making a Healthcare Appointment System Modular and Testable; Growing Clinic's Scheduling Headaches

Sunrise Family Clinic started with a single doctor and a handwritten appointment book.
Now, with multiple doctors, online booking, reminders, and insurance checks:

- The code for scheduling, notifications, and billing is tangled together.
- Every time a new feature (like SMS reminders) is added, it's hard to test or swap out without breaking existing code.
- When the clinic wants to try a new SMS provider or add email reminders, developers must rewrite large parts of the system.
- Testing is risky, since real notifications go out during every test run.



The challenge:
How can you design your appointment system so each part (scheduling, notifications, billing) can be developed, tested, and swapped out independently-without breaking the rest of the system?

2. Learning Objectives

By the end of this tutorial, you will:

- Understand Dependency Injection (DI) and Inversion of Control (IoC) concepts.
- Use a DI container (TypeDI) to manage dependencies in a Node.js/Express app.
- Write modular, testable code by injecting services (e.g., notification, billing) instead of hardcoding them.
- Swap implementations (e.g., SMS vs. email) without changing business logic.
- Write unit tests with mock dependencies.

3. Concept Introduction with Analogy

Analogy: The Clinic’s Reception Desk and Service Providers

Imagine the clinic’s receptionist:

- Needs to schedule appointments, send reminders, and process payments.
- Instead of calling each service directly, the receptionist uses a **service directory** (like a switchboard or concierge).
- When the clinic changes SMS providers, the receptionist just updates the directory, not their entire workflow.

Dependency Injection is like this service directory:

- Each part of your system asks for the service it needs, not caring who provides it.
- You can swap services (real or mock) easily, for new features or testing.

---

4. Technical Deep Dive

A. What Is Dependency Injection (DI)?

Dependency Injection is a programming technique where an object receives the objects (dependencies) it needs from an external source, rather than creating them itself

- **Inversion of Control (IoC):** Instead of your class controlling how dependencies are created, an external system (the IoC container) takes over that responsibility.
- **Why?**
  - Makes code loosely coupled: your classes depend on *interfaces* (contracts), not concrete implementations.
  - Enables easy swapping of implementations (e.g., switch from SMS to Email notifications, or from a fake billing service to a real one).
  - Supports different configurations for different environments (e.g., test, development, production).

Benefits:

- Decouples components (e.g., scheduling doesn’t care how notifications are sent).
- Makes code easier to test (inject mocks).
- Enables swapping implementations with minimal code changes.

B. Types of Dependency Injection

- **Constructor Injection (most common and recommended):**
  Dependencies are passed as parameters to the class constructor.
  - **Advantages:**
    - Makes dependencies explicit (you know what’s required).
    - Enforces required dependencies at creation time (no runtime surprises).
    - Simplifies mocking for testing (just pass a mock in the constructor).
- **Setter Injection:**
  Dependencies are set via setter methods after object creation.

- **Property Injection:**
  Dependencies are set directly on public properties (less common in TypeScript/Node).

Constructor injection is preferred for clarity, safety, and testability.

C. How Does an IoC Container Work?

An **IoC container** (like TypeDI) manages the creation, configuration, and lifecycle of your services and their dependencies.

- **Registration:** You register classes (services) with the container, often with decorators like `@Service()`.
- **Resolution:** When you ask for a service, the container:
  1. Reads the constructor to see what dependencies are needed.
  2. Finds or creates those dependencies (recursively).
  3. Injects them into your class.
- **Lifecycle Management:**
  - Singleton: One instance for the whole app (default in TypeDI).
  - Transient: New instance every time.
  - Scoped: Instance per request/context.

D. Advanced Usage with TypeDI

- **Service Tokens:**
  Instead of using classes directly, you can use tokens (symbols or strings) for even looser coupling.
- **Factory Functions:**
  Register services using factories for dynamic creation or configuration.
- **Scoped Containers:**
  Create per-request or per-user containers for request-specific dependencies (e.g., user context, transactions).
- **Service Groups:**
  Inject arrays of services (e.g., multiple notification handlers).
- **Resetting and Cleaning Up:**
  Use `Container.reset()` to clear registrations between tests or for hot-reloading.

E. Why DI/IoC is Essential for Scalable Systems

- As your system grows, manual wiring of dependencies becomes error-prone and unmanageable.
- DI containers automate this, making your codebase modular, testable, and adaptable to change

F. Using TypeDI for DI in Node.js/Express

1. Install Prerequisites

```
npm install typedi reflect-metadata
```

In your `tsconfig.json`:

```
"experimentalDecorators": true,
"emitDecoratorMetadata": true
```

At the top of your entry file:

```
import "reflect-metadata";
```

2. Define Service Interfaces and Implementations

```
// notifications/NotificationService.ts
export interface NotificationService {
 send(to: string, message: string): Promise<void>;
}

// notifications/SMSService.ts
import { Service } from "typedi";
import { NotificationService } from "./NotificationService";

@Service()
export class SMSService implements NotificationService {
 async send(to: string, message: string) {
   console.log(`SMS sent to ${to}: ${message}`);
 }
}

// notifications/EmailService.ts
import { Service } from "typedi";
import { NotificationService } from "./NotificationService";

@Service()
export class EmailService implements NotificationService {
 async send(to: string, message: string) {
   console.log(`Email sent to ${to}: ${message}`);
 }
}
```

3. Inject Dependencies

```
// appointments/AppointmentService.ts
import { Service, Inject } from "typedi";
import { NotificationService } from "../notifications/NotificationService";

@Service()
export class AppointmentService {
 constructor(
   @Inject(() => SMSService) private notifier: NotificationService
 ) {}

 async bookAppointment(patient: string, time: string) {
   // ...scheduling logic...
   await this.notifier.send(patient, `Your appointment is booked for ${time}`);
   return { status: "confirmed" };
 }
}
```

4. Swap Implementations Easily

```
// main.ts
import "reflect-metadata";
import { Container } from "typedi";
import { AppointmentService } from "./appointments/AppointmentService";
import { EmailService } from "./notifications/EmailService";
import { NotificationService } from "./notifications/NotificationService";
```

```
// Use EmailService instead of SMSService
Container.set(NotificationService, new EmailService());

const appointmentService = Container.get(AppointmentService);
appointmentService.bookAppointment("alice@example.com", "Monday 10am");
```

- No change to `AppointmentService` code is needed!

5. Test with Mocks

```
// tests/AppointmentService.test.ts
import { Container } from "typedi";
import { AppointmentService } from "../appointments/AppointmentService";
import { NotificationService } from "../notifications/NotificationService";

class MockNotifier implements NotificationService {
 messages: string[] = [];
 async send(to: string, message: string) {
   this.messages.push(`${to}: ${message}`);
 }
}

test("should send notification on booking", async () => {
 const mock = new MockNotifier();
 Container.set(NotificationService, mock);

 const service = Container.get(AppointmentService);
 await service.bookAppointment("bob@example.com", "Tuesday 2pm");

 expect(mock.messages).toContain("bob@example.com: Your appointment is booked for Tuesday 2pm");
});
```

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

A. Setting Up for Flexibility and Growth

Clinic Scenario:
The clinic knows it will need to add new notification types (SMS, Email, automated calls) and billing providers (Stripe, insurance, etc.) as it grows. They want to avoid rewriting appointment logic every time.

How DI/TypeDI Solves This:
By setting up TypeDI and using interfaces, the clinic can register any notification or billing provider and swap implementations with a single line of code.

Implementation:

1. Install TypeDI and reflect-metadata:

```
npm install typedi reflect-metadata
```

2. Add at the top of your main file:

```
import "reflect-metadata";
```

3. Enable decorators in `tsconfig.json`:

```
"experimentalDecorators": true,
"emitDecoratorMetadata": true
```

B. Defining Interfaces: Contracts for Clinic Services

Clinic Scenario:
The clinic wants to ensure that all notification and billing services follow a standard contract, so they can add new providers or test with mocks easily.

How DI/TypeDI Solves This:
By defining interfaces, the appointment logic doesn't care how a notification is sent or a bill is processed—only that it happens.

Implementation:

```
// notifications/NotificationService.ts
export interface NotificationService {
 send(to: string, message: string): Promise<void>;
}

// billing/BillingService.ts
export interface BillingService {
 charge(patient: string, amount: number): Promise<void>;
}
```

C. Creating Injectable Implementations: Real-World Providers

Clinic Scenario:
The clinic starts with SMS for reminders, but wants to add Email later, and maybe swap in a mock for testing.

How DI/TypeDI Solves This:
Each provider is a class marked with `@Service()` , making it injectable and swappable.

Implementation:

```
// notifications/SMSService.ts
import { Service } from "typedi";
import { NotificationService } from "./NotificationService";

@Service()
export class SMSService implements NotificationService {
 async send(to: string, message: string) {
   console.log(`SMS sent to ${to}: ${message}`);
 }
}

// billing/StripeBillingService.ts
import { Service } from "typedi";
import { BillingService } from "./BillingService";

@Service()
export class StripeBillingService implements BillingService {
 async charge(patient: string, amount: number) {
   console.log(`Charged $${amount} to ${patient} via Stripe`);
 }
}
```

- If the clinic wants to add Email or insurance billing, they just add new classes.

D. Injecting Services into Appointment Logic

Clinic Scenario:
The appointment scheduler shouldn't care how reminders are sent or how billing is processed—it should just ask for those services.

How DI/TypeDI Solves This:
The `AppointmentService` receives its dependencies via constructor injection, making it easy to swap them for different environments or tests.

Implementation:

```
// appointments/AppointmentService.ts
import { Service, Inject } from "typedi";
import { NotificationService } from "../notifications/NotificationService";
import { BillingService } from "../billing/BillingService";
import { SMSService } from "../notifications/SMSService";
import { StripeBillingService } from "../billing/StripeBillingService";

@Service()
export class AppointmentService {
 constructor(
   @Inject(() => SMSService) private notifier: NotificationService,
   @Inject(() => StripeBillingService) private billing: BillingService
 ) {}

 async bookAppointment(patient: string, time: string, amount: number) {
   await this.billing.charge(patient, amount);
   await this.notifier.send(patient, `Your appointment is booked for ${time}`);
   return { status: "confirmed" };
 }
}
```

- If the clinic wants to use Email instead of SMS, or a different billing provider, they only change what's injected, not the business logic.

E. Swapping Implementations for Growth or Testing

Clinic Scenario:
The clinic wants to use Email reminders for some patients, or a mock billing service for testing, without touching appointment logic.

How DI/TypeDI Solves This:
You can register any implementation with the container at runtime.

Implementation:

```
import { Container } from "typedi";
import { AppointmentService } from "./appointments/AppointmentService";
import { EmailService } from "./notifications/EmailService";
import { NotificationService } from "./notifications/NotificationService";

// Swap to EmailService for notifications
Container.set(NotificationService, new EmailService());

const appointmentService = Container.get(AppointmentService);
appointmentService.bookAppointment("alice@example.com", "Monday 10am", 50);
```

- For tests, you can inject a mock service that just logs messages.

F. Testing with Mocks: Safe, Isolated, and Reliable

Clinic Scenario:
The clinic wants to test appointment booking without sending real notifications or charging real credit cards.

How DI/TypeDI Solves This:
Inject a mock service that records calls instead of performing real actions.

Implementation:

```
class MockNotifier implements NotificationService {
 messages: string[] = [];
 async send(to: string, message: string) {
   this.messages.push(`${to}: ${message}`);
 }
}

class MockBilling implements BillingService {
  charges: string[] = [];
  async charge(patient: string, amount: number) {
    this.charges.push(`${patient}: $${amount}`);
  }
}

// In your test setup
Container.set(NotificationService, new MockNotifier());
Container.set(BillingService, new MockBilling());

const service = Container.get(AppointmentService);
await service.bookAppointment("bob@example.com", "Tuesday 2pm", 75);

// Assert that the mocks recorded the expected actions

Container.reset(); // Clean up after test
```

- No real SMS or billing happens during tests—just logs in memory.

6. Challenge

Your Turn!

- Add a `BillingService` interface and a `StripeBillingService` implementation.
- Inject `BillingService` into `AppointmentService` to charge patients when booking.
- Write a test using a mock billing service to verify the charge is made.

7. Common Pitfalls & Best Practices

Pitfall	Best Practice
Hardcoding dependencies in services	Always inject dependencies via constructor
Not using interfaces for services	Depend on abstractions, not implementations
Forgetting to reset the container in tests	Clean up after each test to avoid side effects
Over-injecting (too many dependencies)	Keep service responsibilities focused

8. Optional: Programmer’s Workflow Checklist

- Define interfaces for all service dependencies.
- Use `@Service()` and `@Inject()` decorators for DI.
- Register implementations in the DI container.
- Swap implementations for testing or new features.
- Write tests using mock services.
- Reset the container between tests to avoid leaks