1. Problem Statement

Chaos at "Maplewood Public Library"

Maplewood Public Library's software system is in disarray:

- Rules like "members can't borrow more than 5 books" are buried in code that handles website requests. When policies change, updates risk breaking other parts of the system.
- The team wants to add new features (e.g., online reservations, late fines) but struggles because data access and business logic are tangled together.
- Testing is nearly impossible-every small change requires rewriting large parts of the code.

The challenge:
How do you redesign the system so that:

- Rules and policies are clearly separated and easy to update?
- Data storage details don't interfere with how books are managed or loans are processed?
- New features can be added without fear of breaking existing functionality?

2. Learning Objectives

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

- Structure code using **MVC (Model-View-Controller)** to separate rules, data, and user interactions.
- Use the **Repository Pattern** to isolate data storage details from business logic.
- Apply **Dependency Injection** to create modular, testable components.

3. Concept Introduction with Analogy

Analogy: The Library’s Organizational Chart

Imagine the library has three teams:

1. **Front Desk (Controller):** Handles member requests (checkouts, returns).
2. **Policy Team (Service):** Manages rules (loan limits, fines).
3. **Archives Team (Repository):** Stores and retrieves books from the shelves.

Each team works independently:

- The front desk doesn’t care *how* books are stored-it just asks the archives team to fetch them.
- The policy team enforces rules without knowing *where* books are kept.
- If the archives team reorganizes the shelves (changes storage systems), the other teams keep working as usual.

This separation prevents chaos and lets the library adapt to changes smoothly.

A. MVC Pattern: Separating Concerns

Layer	Responsibility	Example in Library
Model	Data structure and validation	Book, Member classes
View	Presentation (JSON, HTML)	API responses
Controller	Handles user requests/responses	BookController routes
Service	Business logic (e.g., loan rules)	Calculating fines, reservations
Repository	Data storage/retrieval	Fetching books from storage

B. Repository Pattern: Isolate Storage Details

- **Repository Interface:** Defines *what* operations are available (e.g., `findAllBooks()` , `saveBook()`).
- **Implementation:** Handles *how* data is stored (e.g., in-memory, files, databases).

Why?

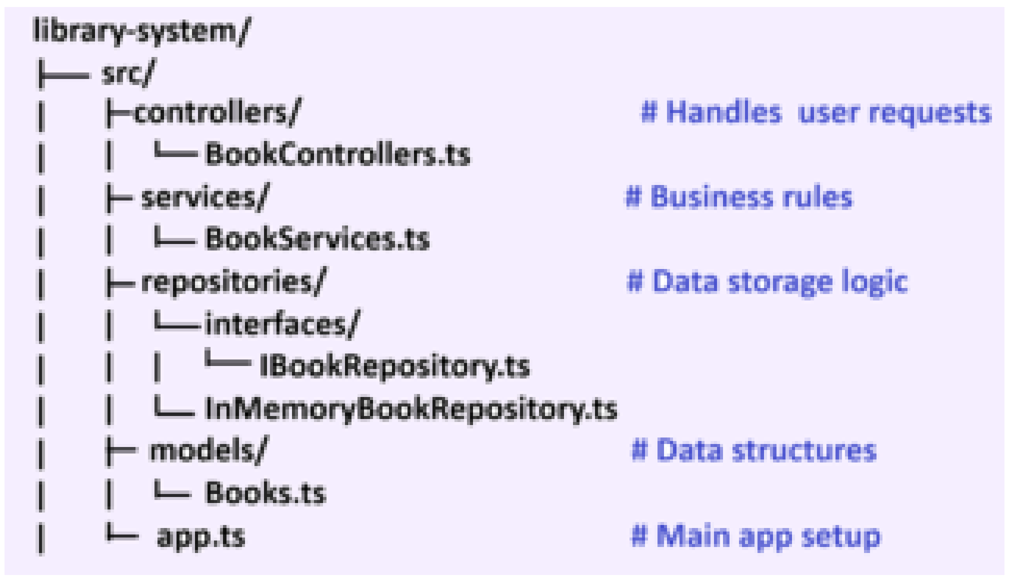
- Change storage systems without rewriting business logic.
- Test services with fake repositories (no real storage needed).

C. Dependency Injection: Building Modular Systems

- **Dependency Injection (DI):** Provide components with their dependencies (e.g., services, repositories) instead of letting them create their own.
- **Benefits:**
  - Swap components easily (e.g., switch storage systems for testing).
  - Avoid tight coupling between layers.

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

A. Project Structure



```
library-system/
├── src/
|     ├controllers/                          # Handles  user requests
|     |   └── BookControllers.ts
|     ├ services/                           # Business rules
|     |   └── BookServices.ts
|     ├ repositories/                       # Data storage logic
|     |   └──interfaces/
|     |   |    └── IBookRepository.ts
|     |   └── InMemoryBookRepository.ts
|     ├ models/                             # Data structures
|     |   └── Books.ts
|     └─ app.ts                             # Main app setup
```

B. Model: Define the Book Structure

```
// src/models/Book.ts
export interface Book {
id: string;
title: string;
author: string;
isBorrowed: boolean;
}
```

C. Repository: Separate Storage Logic

```
// src/repositories/interfaces/IBookRepository.ts
export interface IBookRepository {
findAll(): Promise<Book[]>;
findById(id: string): Promise<Book | null>;
save(book: Book): Promise<void>;
}

// src/repositories/InMemoryBookRepository.ts
import { IBookRepository } from './interfaces/IBookRepository';
import { Book } from '../models/Book';

export class InMemoryBookRepository implements IBookRepository {
private books: Book[] = [];

async findAll(): Promise<Book[]> {
  return this.books;
}

async findById(id: string): Promise<Book | null> {
  return this.books.find(book => book.id === id) || null;
}

async save(book: Book): Promise<void> {
  this.books.push(book);
}
}
```

D. Service: Enforce Business Rules

```
// src/services/BookService.ts
import { IBookRepository } from '../repositories/interfaces/IBookRepository';
import { Book } from '../models/Book';

export class BookService {
```

```
constructor(private bookRepository: IBookRepository) {}

async borrowBook(bookId: string): Promise<Book> {
  const book = await this.bookRepository.findById(bookId);
  if (!book) throw new Error('Book not found');
  if (book.isBorrowed) throw new Error('Book already borrowed');

  const updatedBook = { ...book, isBorrowed: true };
  await this.bookRepository.save(updatedBook);
  return updatedBook;
}
}
```

E. Controller: Handle User Requests

```
// src/controllers/BookController.ts
import { Request, Response } from 'express';
import { BookService } from '../services/BookService';

export class BookController {
constructor(private bookService: BookService) {}

async borrowBook(req: Request, res: Response): Promise<void> {
  try {
    const book = await this.bookService.borrowBook(req.params.id);
    res.json(book);
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
}
}
```

F. Dependency Injection Setup

```
// src/app.ts
import express from 'express';
import { BookController } from './controllers/BookController';
import { BookService } from './services/BookService';
import { InMemoryBookRepository } from './repositories/InMemoryBookRepository';

const app = express();
app.use(express.json());

// Initialize components
const bookRepository = new InMemoryBookRepository();
const bookService = new BookService(bookRepository);
const bookController = new BookController(bookService);

// Routes
app.post('/books/:id/borrow', (req, res) => bookController.borrowBook(req, res));

const port = 3000;
app.listen(port, () => {
console.log(`Library system running on port ${port}`);
});
```

6. Challenge

Your Turn!

- Implement a `delete(courseId: string)` method in the repository.
- Add a service and route to allow admins to delete a course.

7. Common Pitfalls & Best Practices

Pitfall	Best Practice
Mixing data access in business logic	Always use repositories for storage access
Hardcoding storage details	Depend on interfaces, not implementations
Not testing with mocks	Use fake repositories for unit tests

8. Quick Recap & Key Takeaways

- **Separate layers** (controllers, services, repositories) for clarity.
- **Repository Pattern** hides storage details from business rules.
- **Dependency Injection** makes components reusable and testable.

9 Optional: Programmer's Workflow Checklist (MVC Pattern)

- **Identify the main resources** in your application (e.g., Book, User, Event).
- **Define Models** for each resource (data structure, validation rules).
- **Create Controllers** for each resource to handle user/API requests and responses.
- **Write Services** to contain business logic (rules, calculations, policies).
- **Keep Controllers thin**-they should only coordinate input/output and call services.
- **Never access data storage directly from controllers**-always go through services.
- **If using repositories:** Only services should interact with repositories for data access.
- **Organize code into folders:** `/models` , `/controllers` , `/services` (and `/repositories` if needed).
- **Document the responsibility of each layer** so team members know where to add new logic.
- **Test each layer independently** (e.g., unit test services without controllers).
- **Review regularly:** Refactor if business logic creeps into controllers or data access leaks into services.

10. Coming up next

Learn to use automated dependency injection tools (like `tsyringe`) to manage complex systems effortlessly!