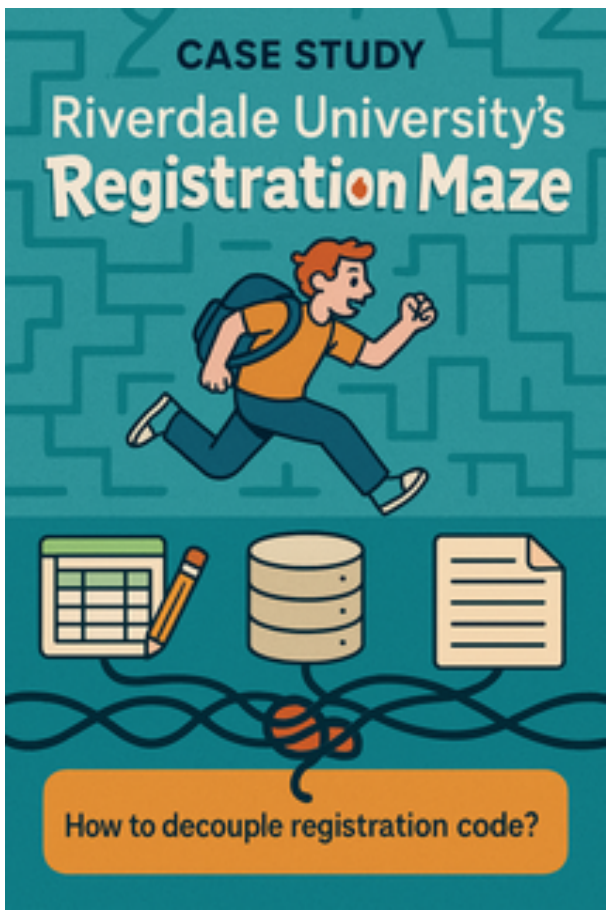
1. Problem Statement

Case Study: Riverdale University's Registration Maze

At Riverdale University, students race to register for classes each semester:

- Some departments use spreadsheets, others use databases, and a few still keep paper records.
- When a student tries to enroll, their request is sometimes lost, or two systems accidentally double-book a seat.
- If the university wants to upgrade from spreadsheets to a new database, they worry about rewriting all their registration logic.
- Testing new features is risky, because the code is tightly coupled to the current storage method.



The challenge:
How can you build a course registration system where the rules and features work no matter how or where the data is stored-so you can upgrade, test, and scale without rewriting everything?

---

2. Learning Objectives

By the end of this tutorial, you will:

- Understand the Repository Pattern and its purpose.
- Create repository interfaces to abstract data access.
- Implement repositories for different storage types (memory, file, database).
- Swap storage backends without changing business logic.
- Write tests using mock repositories.
- Apply best practices and avoid common pitfalls.

---

3. Concept Introduction with Analogy

Analogy: The University Records Office

Imagine the records office:

- Professors and students submit requests (“Add me to Physics 101!”).
- The office staff don’t care if records are in filing cabinets, spreadsheets, or a fancy database-they just use a standard form.
- If the university upgrades to a new system, the process for students and staff stays the same.

The Repository Pattern is like this standardized records office:

- All requests go through a single interface, no matter where the data lives.
- The rest of the university never worries about how records are stored or retrieved.

---

4. Technical Deep Dive

A. What is the Repository Pattern?

The Repository Pattern is a design pattern that centralizes data access logic in a single place, separating it from business logic.

- **Purpose:**
  - Abstracts the details of data storage and retrieval from the rest of the application.
  - Provides a collection-like interface for accessing domain objects.
  - Makes it easy to swap storage backends (e.g., memory, file, database) without changing business logic.
- **Benefits:**
  - Centralized, maintainable data access code.
  - Business logic is decoupled from storage details.
  - Easier testing (mock repositories).
  - Reduces code duplication and errors.

B. Repository Pattern in Practice

Key Components:

- **Repository Interface:**
  - Defines the operations for a resource (CRUD, custom queries).
  - Example: `ICourseRepository` with methods like `findAll`, `findById`, `save`, `enrollStudent`.
- **Repository Implementation:**
  - Handles the actual data access logic (memory, file, database).
  - Can be swapped out without changing the rest of the app.
- **Domain Model:**

    - The data structure (e.g., `Course`) the repository manages.

- **Business Logic Layer (Service):**
    - Uses the repository interface, not the implementation, to enforce rules and policies.

C. Why Not Access Data Directly?

If you mix data access and business logic:

- Every change in storage (e.g., new database) forces you to rewrite all logic.
- Testing is hard-you need a real database for every test.
- Code is harder to read, debug, and maintain.

D. Best Practices

- Define repository interfaces in the domain layer
- Create one repository per aggregate root (main entity, e.g., Course, Student).
- Never expose storage-specific types or queries (e.g., SQL, ORM objects) to business logic.
- Use dependency injection to provide repository implementations.
- Test business logic with mock or in-memory repositories.

---

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

Let's build a robust course registration system using the Repository Pattern.

A. Define the Domain Model

```
// models/Course.ts
export interface Course {
  id: string;
  name: string;
  capacity: number;
  students: string[];
}
```

B. Create the Repository Interface

```
// repositories/interfaces/ICourseRepository.ts
export interface ICourseRepository {
  findAll(): Promise<Course[]>;
  findById(id: string): Promise<Course | null>;
  save(course: Course): Promise<void>;
  enrollStudent(courseId: string, studentId: string): Promise<void>;
  findByStudentId(studentId: string): Promise<Course[]>;
}
```

- **Why?**
    - This interface is the “standard form” for the records office.
    - Business logic only talks to this interface, never to storage details.

C. Implement an In-Memory Repository

```
// repositories/InMemoryCourseRepository.ts
import { ICourseRepository } from './interfaces/ICourseRepository';
import { Course } from '../models/Course';

export class InMemoryCourseRepository implements ICourseRepository {
  private courses: Course[] = [];

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

  async findById(id: string): Promise<Course | null> {
    return this.courses.find(course => course.id === id) || null;
  }

  async save(course: Course): Promise<void> {
    const idx = this.courses.findIndex(c => c.id === course.id);
    if (idx >= 0) {
      this.courses[idx] = course;
    } else {
      this.courses.push(course);
    }
  }

  async enrollStudent(courseId: string, studentId: string): Promise<void> {
    const course = await this.findById(courseId);
    if (course && !course.students.includes(studentId)) {
      course.students.push(studentId);
      await this.save(course);
    }
  }

  async findByStudentId(studentId: string): Promise<Course[]> {
    return this.courses.filter(course => course.students.includes(studentId));
  }
}
```

- **Why?**
- All data access is here.
- If you switch to a database, only this file changes.

D. Implement a Database Repository (Example Outline)

```
// repositories/DatabaseCourseRepository.ts
import { ICourseRepository } from './interfaces/ICourseRepository';
import { Course } from '../models/Course';

export class DatabaseCourseRepository implements ICourseRepository {
  // Assume db is a connected database client
  constructor(private db: any) {}

  async findAll(): Promise<Course[]> {
```

```
    // Use real database queries here
    return this.db.query('SELECT * FROM courses');
  }

  async findById(id: string): Promise<Course | null> {
    // ...
    return null; // Example
  }

  // ...implement other methods
}
```

- **Why?**

- You can now swap between in-memory and database storage without changing business logic.

E. Use the Repository in a Service

```
// services/CourseService.ts
import { ICourseRepository } from '../repositories/interfaces/ICourseRepository';

export class CourseService {
  constructor(private courseRepo: ICourseRepository) {}

  async enroll(courseId: string, studentId: string) {
    const course = await this.courseRepo.findById(courseId);
    if (!course) throw new Error('Course not found');
    if (course.students.length >= course.capacity) throw new Error('Course full');
    await this.courseRepo.enrollStudent(courseId, studentId);
    return { message: 'Enrolled successfully' };
  }

  async getStudentCourses(studentId: string) {
    return this.courseRepo.findByStudentId(studentId);
  }
}
```

- **Why?**

- The service only knows about the repository interface, not how data is stored.

F. Hook Up in Your App

```
// app.ts
import express from 'express';
import { InMemoryCourseRepository } from './repositories/InMemoryCourseRepository';
import { CourseService } from './services/CourseService';

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

const courseRepo = new InMemoryCourseRepository();
const courseService = new CourseService(courseRepo);

app.post('/courses/:id/enroll', async (req, res) => {
  try {
    const result = await courseService.enroll(req.params.id, req.body.studentId);
    res.json(result);
  } catch (e) {
    res.status(400).json({ error: e.message });
  }
});
```

```
app.get('/students/:id/courses', async (req, res) => {
  const courses = await courseService.getStudentCourses(req.params.id);
  res.json(courses);
});

app.listen(3000, () => console.log('Server running on port 3000'));
```

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 everywhere	Depend on interfaces, not implementations
Not testing with mocks	Use fake repositories for unit tests
Exposing storage-specific types to logic	Only return domain models from repositories
Not updating the repository interface	Keep interfaces up to date with business needs

8. Optional: Programmer's Workflow Checklist

- Define repository interfaces for all major resources.
- Implement repositories for each storage type (memory, file, database).
- Never access storage directly from services or controllers-use repositories.
- Swap repository implementations easily for testing or upgrades.
- Write unit tests with mock repositories.
- Don't expose storage-specific types or queries to business logic.
- Keep repository interfaces in the domain layer.

---

9. Coming up Next

Learn how to use Dependency Injection to provide repositories to your services and controllers automatically-making your app even more modular, testable, and ready for growth!
