Modular Monolith Report

Introduction

A modular monolith is an architecture where a large application is divided into smaller, relatively independent modules. Unlike a traditional monolith, which is often a single, tangled codebase, a modular monolith promotes better organization and maintainability. We choose this approach because it allows us to achieve many benefits of microservices like maintainability and scalability, but with a single code base that is easier to manage and deploy.

Current Architecture

Our current project consists of these projects:

`UniversityRegistration.Core`: This project holds all our domain logic, including entities (`Student`, `Course`, `College`, `Registration`, `Rule`), interfaces (repositories, Unit of Work, Mediator), implementations (in-memory repositories, Unit of Work, `CollegeRuleService`), handlers, commands and queries.
`UniversityRegistration.Desktop`: This Windows Forms application acts as a presentation layer and provides an entry point to our application.
`UniversityRegistration.Tests`: This project includes our xUnit tests for the rules validation.

We are using patterns like:

Repositories:** For abstracting data access.
Unit of Work:** For managing transactions.
Mediator:** For decoupling the requests from their handlers.

Modular Monolith Report

Proposed Modular Structure

We can divide `UniversityRegistration.Core` project into the following modules:

`UniversityRegistration.Core.Registration`: This module would be responsible for managing course registrations. It will contain our handlers, commands, and queries related to course registration.
`UniversityRegistration.Core.Colleges`: This module would be responsible for managing colleges and their rules. It will contain our `College` entity, our `IRule` implementations and our validation service.
`UniversityRegistration.Core.Courses`: This module would handle all things related to courses, including managing their properties and pre-requisites.
`UniversityRegistration.Core.Students`: This module will handle everything related to student information and their data.

By grouping related functionality into these modules, we will improve the system's organization, increase the maintainability and testability.

Steps for Modularization

To implement this modular structure, I'd take these steps:

1. **Create new Projects**: Create new class library projects for each of the mentioned modules.
2. **Move classes**: move the existing classes in `UniversityRegistration.Core` to the new projects based on their responsibilities.
3. **Public Interfaces**: Define clear public interfaces between these modules, so they can communicate and depend on each other in an orderly way.
4. **Dependency Injection:** Use dependency injection to make the modules loosely coupled, this will make the system more flexible to change.

Benefits of Modularization

By refactoring our application to follow a modular monolith architecture, we can achieve many benefits, including:

- Improved maintainability since it is easier to find, modify and debug code.
- Increased code reuse, as we can use modules in multiple places.
- Better testability, as modules can be tested independently.
- Reduced complexity and improved organization.
- Improved team collaboration as different teams can work on different modules.

Considerations and Challenges

Some challenges to consider are:

- Defining clear boundaries between modules, and what should each be responsible for.
- Managing dependencies between different modules and having a clear plan for each module to depend on others.
- Ensuring that existing code and tests are still working after the refactoring.
- Carefully planning and executing the steps to guarantee backwards compatibility for all our modules.

Conclusion

This plan will transform our single `UniversityRegistration.Core` project into a modular monolith that will allow us to scale, maintain, and test our application. This new architecture will make it easier for future developers to understand, modify and maintain this system.