Design Patterns

October 29, 2017

In this document we describe briefly the design patterns we found to be useful in our application and are compatible with the application's architecture.

Microservices Pattern

"Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of loosely coupled services, which implement business capabilities. The microservice architecture enables the continuous delivery/deployment of large, complex applications. It also enables an organization to evolve its technology stack." - http://microservices.io/

Our application is integrating, for now, three functionalities. But our goal is to create an application which can support various types of functionalities. Diverse functionalities can add to our project diverse types of dependencies. To solve this issue, we use this architectural pattern. Thus, we can build a module (microservice) for each individual functionality and because we are communicating through HTTP, we allow each module to have the dependencies it needs (maybe it is a programming language dependency, for example, if we'll need a tool for natural language processing, we will use Python) without complicating the rest of the application.

Repository Pattern

Both the API Controller and the modules for login and courses need to access their own database. If this interaction is done directly the following can result:

- Duplicate code
- A higher potential for programming errors
- Difficulty in centralizing data-related policies

Using the Repository pattern the business logic will be separated from the data access logic, thus enabling a lower coupling and improving the code's readability and maintainability.

The implementation chosen for the current project involves creating a generic repository interface with the CRUD methods and then implementing it with repositories for each entity mapped in the database. This approach allows accessing the data from various locations, but in a consistent and centrally managed manner.

Component Architecture (Composite pattern)

The web application which will consume the application endpoints exposed by the server, aside from the other design patterns that will be used, it will feature a component based architecture. This will decrease decoupling and increase the cohesion.

Alongside the component based architecture the Composite pattern will be used to treat components with different behaviour the same (e.g modules).

The following 2 patterns are usually found inside applications working together, and they are based on the **Dependency Inversion** principle (SOLID): Abstractions should not depend upon details, details should depend upon abstractions. These two patterns greatly help us in creating a decoupled application. The base principle is the following: high-level modules should not depend on the low-level modules, they should both depend on abstractions. Abstractions should not depend on details, the details should depend on abstractions.

Inversion of Control

Inversion of Control is a software development principle, in which a class states at initialization all the components it needs for proper functioning. The framework itself has a container where all the components reside, as singletons, and when it is “called”, it will give to our class the instance it requires.

To summarize, a class does not instantiate dependent components itself, but instead calls the IoC container to give it the needed components. This way, the control is inverted, being held by the class itself.

.NET Core provides a built-in IoC container, which we will use to register components inside our application. When our application is initialized, all the registered components will be instantiated as Singleton instances, per application scope.

Dependency Injection

Dependency Injection comes as a natural extension to the Inversion of Control pattern. It is basically the technique in which an object supplies dependencies to another object. In our case, the IoC container will *Inject* the required components to the objects that needed them. Most of the times, these objects will declare their dependencies via constructor. This is known as constructor injection.

We will be using Dependency Injection to achieve a loose coupling between our objects/components and their collaborators or dependencies.

Proxy

In short, a proxy is a wrapper or agent object that is being called by the client to access the real serving object behind the scenes. In our case the proxy design pattern is used for the controller interfaces to access each modules functionalities. It will act as a proxy to every request to a specific module and it will manage all the calls to internal services.

Observer Pattern

The Observer defines a one-to-many relationship so that when one object changes state, the others are notified and updated automatically. It is helpful when implementing distributed event handling systems and C# already has an “event” construct which implements the Observer pattern components.

This pattern will be user for the updating of the timetable of a user when they add more courses to their profile. When the list of courses that the student enrolled in changes, the timetable module will be notified and it will retrieve the data for the newly added courses, thus keeping the user’s timetable updated.

Interpreter

The definition of interpreter, at it’s core, is a translation, that allows people to understand a foreign language or code. Think a musical score, and the musician that interprets it and transformed it to song.

In our case the interpreter is to be used for a number of small tasks, such as informing the user the number of hours it has in a schedule, or later on to convert grade values to whether a student passed an exam or failed it.

Chain of Responsibility

The Chain of Responsibility pattern allows for multiple objects to handle a request, without the need to couple the class to a concrete receiver. The number and type of handler objects isn't known beforehand so they can be configured dynamically. Oftenly it is used in built-in .NET features, such as, generics, reflection, object initializers, automatic properties, etc.

An example for what this will basically be used for is to build a chain of middleware objects and check on the next object in chain or traverse it if we're in last object in chain, till it checks all of them and solves them. As such it is used to check email and password fields, handling login and authorization, or grant further access into different zones of the application - depending on the user's access rights. Further uses, it could also allow the secretary to handle certain requests from users/students, or pass them on to other admins.

Iterator

Iterator is a behavioral design pattern that lets you access the elements of an aggregate object sequentially without exposing its underlying representation. It provides multiple, and possibly parallel, ways to traverse the same data structure, all the while simplifying the codes collections.

Basically the iterator, though a tad overkill for what we have initially, will be used to act as guides over our collection of courses when creating the schedule, from the total amount of courses we provide. With this the students can quickly fetch all of the existing schedule for a certain class, checking duration, location etc without the need of going deeply into details such as for instance, authentication or sending requests.

Decorator

This design pattern allows you to add extra functionality to some objects. We can use it to create a communication data structure that can expose if the operations inside the module are okay or faulted.