Sample House Info API

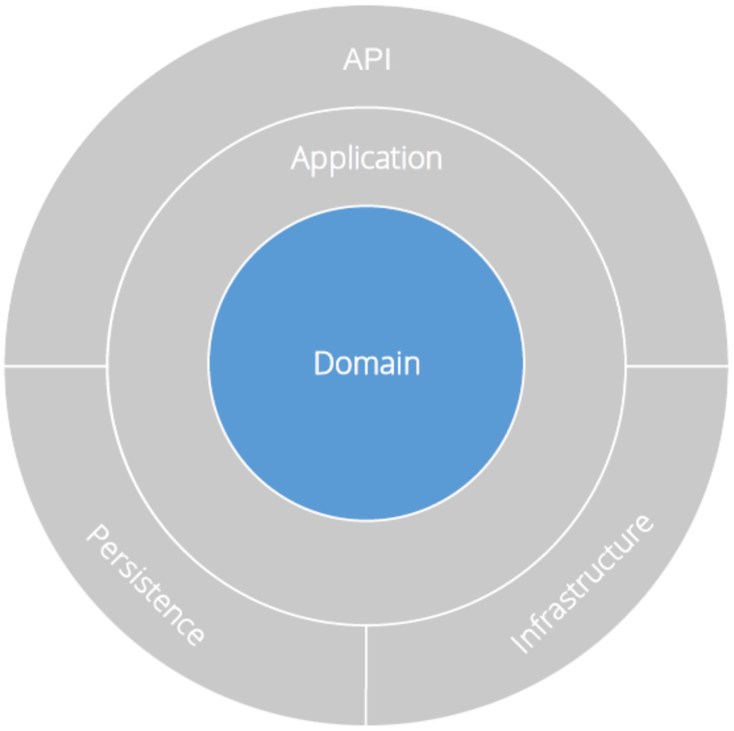
Git repository: https://github.com/Ibrahim5aad/sample-house-info

Azure app service: https://sample-house-info.azurewebsites.net

Available endpoints:

- ***/api/rooms*** Gets all the rooms from the IFC file.
- ***/api/rooms/{roomId}*** Gets a room by its id from the IFC file.
- ***/api/summary*** and ***/api/summary/elementsCount*** Gets the elements count summary.

The approach I took to design the API is what I most of the time do when approaching a new API development, which is based my grasping of the clean architecture standards. This design will allow a project to scale easily and accommodate for architectural changes.



Ibrahim Saad

The design is what is often called an onion architecture and the major guideline is that nothing in an inner circle can know anything at all about something in an outer circle. The intent behind this is that you don't want your domain and application logic to depend/rely on the implementation details of the infrastructure i.e., the database, third party providers, IO operations.

I used project structure to implement those layers as it gives me strict and clear boundaries between the layers.

The Domain

Here lives the domain/business entities.

The Application Layer

The application layer describes all the use cases that your application can perform. The application layer should not define/implement any I/O operations whether in database or files. Those implementation details should always be provided by the upper infrastructure layers. The application layer only providing interfaces for what operations are expected.

This is why I like to use the CQRS pattern. The CQRS pattern allow me to implement what I usually call a **feature-based application layer**. Usually, your use cases are exposed in terms of CRUD operations carried out by the repositories and unit of works. But when thinking in terms of CQRS you would define some **query** and **command** objects each of them acts as a feature of the application, an encapsulated feature that gives more intent about what this piece of application logic can do.

And with the help of a mediator like MediatR you could implement those features to be reactive in some sense. This will make a more verbose and elegant application layer.

CQRS is of course exists for more sophisticated design problems, but I'm just giving my thoughts about why you should start with CQRS + MediatR even in small business scenarios.

In our application for example we only define simple retrieval operations from the IFC file but with the meaningful definition of them as query objects that invoke some handler in the application layer to react is a more elegant way, and with the help of MediatR the application layer query handler doesn't need to know about his clients, the handler only listens to clients that are broadcasting the query objects. This also helps of keeps the API controllers clean of any application logic.

The Persistence Layer:

Our data is persisting in an IFC file, but with the application of repository pattern and respecting the onion architecture control flow, the repositories abstract away any I/O details, the application layer doesn't need to know how the repositories are getting data, they just deal communicate through the declared interfaces/contracts. This allows a loosely coupled persistence layer that can be replaced easily at any time.

In our case you can find that the references to the Xbim toolkit only resides in the persistence layer. However, there are some abstraction leaks inside the layer itself that I think can be done better.

Tests:

Unit tests are carried out for application layer and integration tests are carried out for the API.

Ibrahim Saad