**Context and Problem Statement**

The AML system will receive requests to API endpoints held within ASP.NET controllers. A pattern must be implemented which will allow for scalability and maintainability.

**Decision Drivers**

* There must be an efficient and easily maintainable pattern which can handle how these requests are consumed and responded to.
* Separation of Concerns is important as we need to ensure that we separate the business layer from the data layer
* Must be easy to implement unit tests

**Considered Options**

* CSR (Controller service repository) pattern
* Clean Architecture (Onion Architecture)
* CQRS

**Pros and Cons of the Options**

**Controller service repository pattern**

* Good: Testing is easier as each layer can be tested individually due to the availability of mocking a layer (Collings, 2021)
* Good: Separation of Concerns is inherent to the pattern as services handle business logic and repositories handle database access
* Good: methods held in Repositories and Services can be reused, avoiding code duplication
* Bad: More overhead to implement the code
* Bad: Testing will take longer to implement due to more methods being implemented requiring testing
* Bad: Potential for some methods in the service class to become overly complex and unmanageable

**Clean Architecture**

* Good: Promotes loose coupling as layers only depend on layers below them
* Good: Separation of concerns supported as business logic is handled by the core
* Bad: Complex to implement, would be better suited for medium to large scale applications (Dor Lugasi-Gal, 2024)
* Bad: Possible over-engineering for the scope of the project at hand
* Bad: Performance may take a hit due to the layers and multitude of abstract classes that need to be created (Patil, 2024)

**CQRS**

* Good: Separation of concerns addressed by having different read and write classes. Business logic mainly handled by the write class (EdPrice-MSFT, n.d.)
* Good: Can use separate technologies for read and write databases. Using no-sql for the read database would make querying faster (*The Pros and Cons of the CQRS Architecture Pattern*, 2021)
* Neutral: Using different database technologies may eventually lead to technical debt
* Bad: Possible to achieve a mismatch between the read and write models if not handled correctly
* Bad: More points of failure as there are multiple databases and database models (*The Pros and Cons of the CQRS Architecture Pattern*, 2021)
* Bad: Complex to implement due to previously mentioned point of having different read and write models

**Decision Outcome**

The Architecture pattern chosen is ‘Controller Service Repository’ pattern due to the separate layers implementing separation of concerns by default. Separating the business logic from database access objects also allows for easier testing and limits code duplication.

Furthermore, the pattern is far more simple to implement without much overhead, and is much more suited to the small scope we are trying to achieve

References

Collings, T. (2021, August 10). *Controller-Service-Repository*. Medium. https://tom-collings.medium.com/controller-service-repository-16e29a4684e5

Dor Lugasi-Gal. (2024, June 14). *Next-Level Boilerplate: An Inside Look Into Our .net Clean Architecture Repo - ISE Developer Blog*. ISE Developer Blog. <https://devblogs.microsoft.com/ise/next-level-clean-architecture-boilerplate/>

Patil, H. (2024, February 19). *Clean Architecture is a software design approach introduced by Robert C. Martin, also known as Uncle Bob.* Linkedin.com. <https://www.linkedin.com/pulse/exploring-depths-clean-architecture-its-pros-cons-net-hanumant-patil-oay8f>

EdPrice-MSFT. (n.d.). *CQRS pattern - Azure Architecture Center*. Learn.microsoft.com. <https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs>

*The pros and cons of the CQRS architecture pattern*. (2021). Redhat.com. https://www.redhat.com/en/blog/pros-and-cons-cqrs