Repository Pattern

.NET

Conceptually, a repository encapsulates operations that can be performed on a Database. Repositories also support the purpose of separating, clearly and in one direction, the dependency between the work domain and the data allocation or mapping.

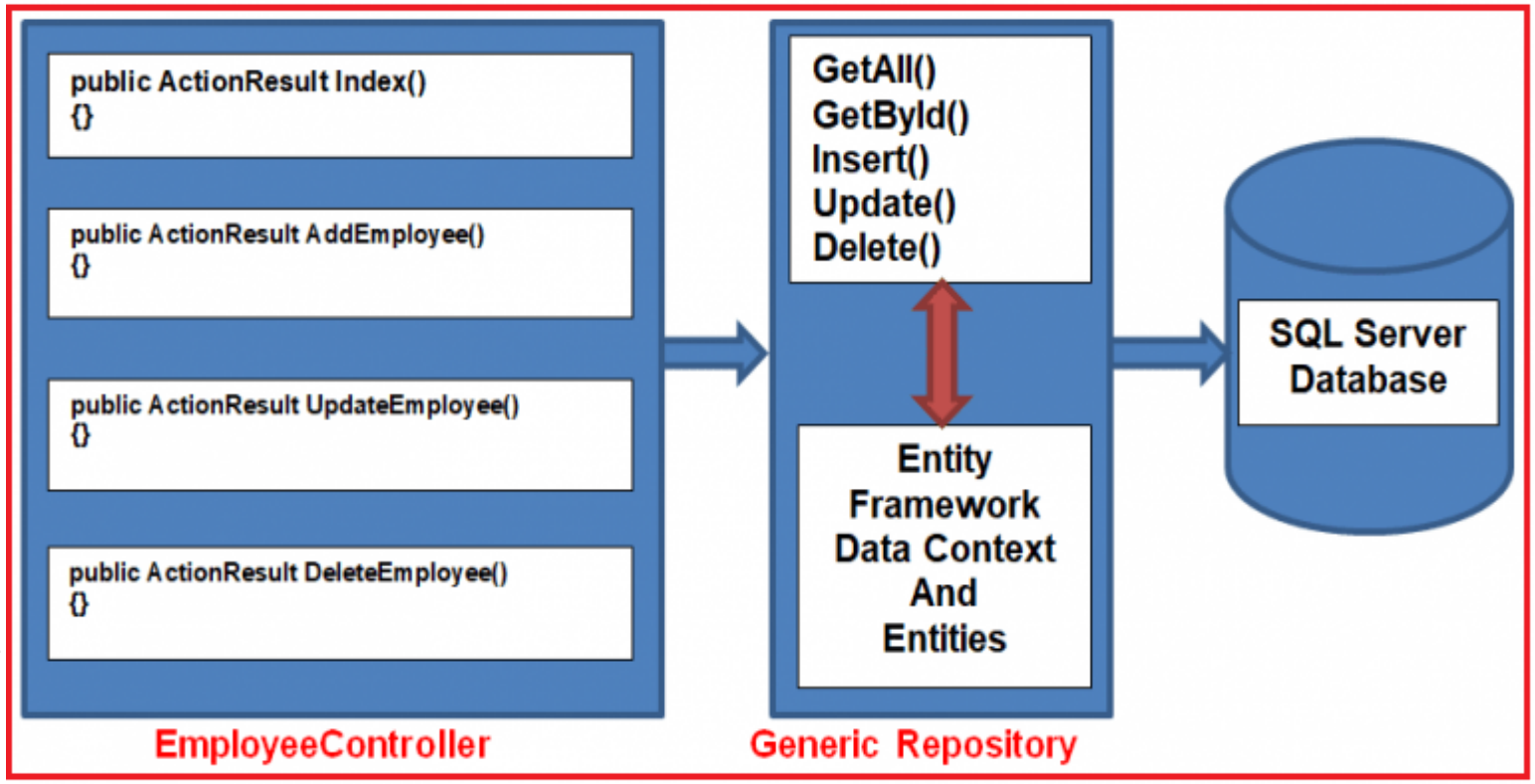
- Martin Fowler

https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design

The Repository Pattern

https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design#the-repository-pattern

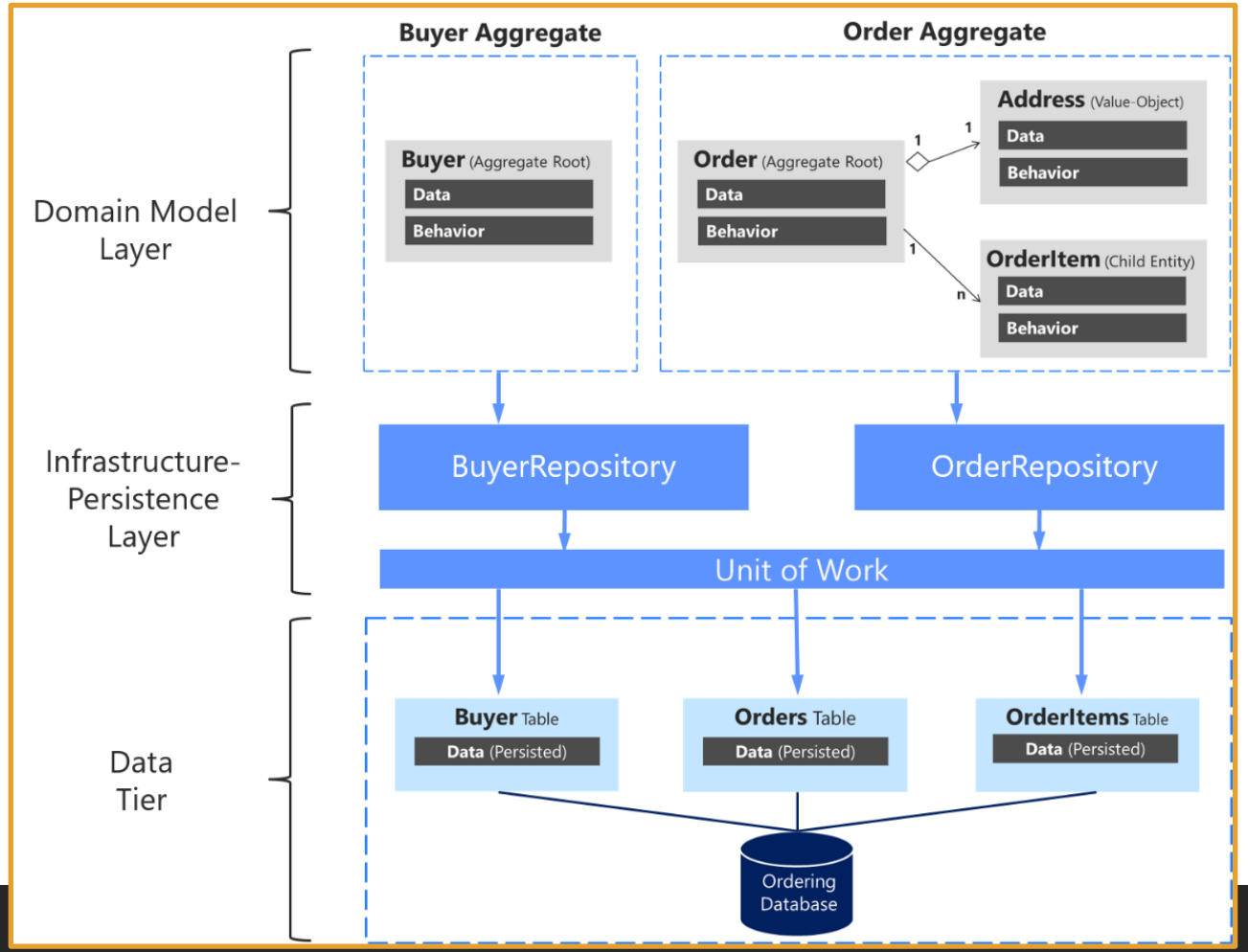
Repositories are classes that encapsulate the logic required to access data sources. They provide better maintainability and decouple the infrastructure or technology used to access databases from the domain model layer



Repository Pattern Best Practices

https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design#define-one-repository-per-aggregate

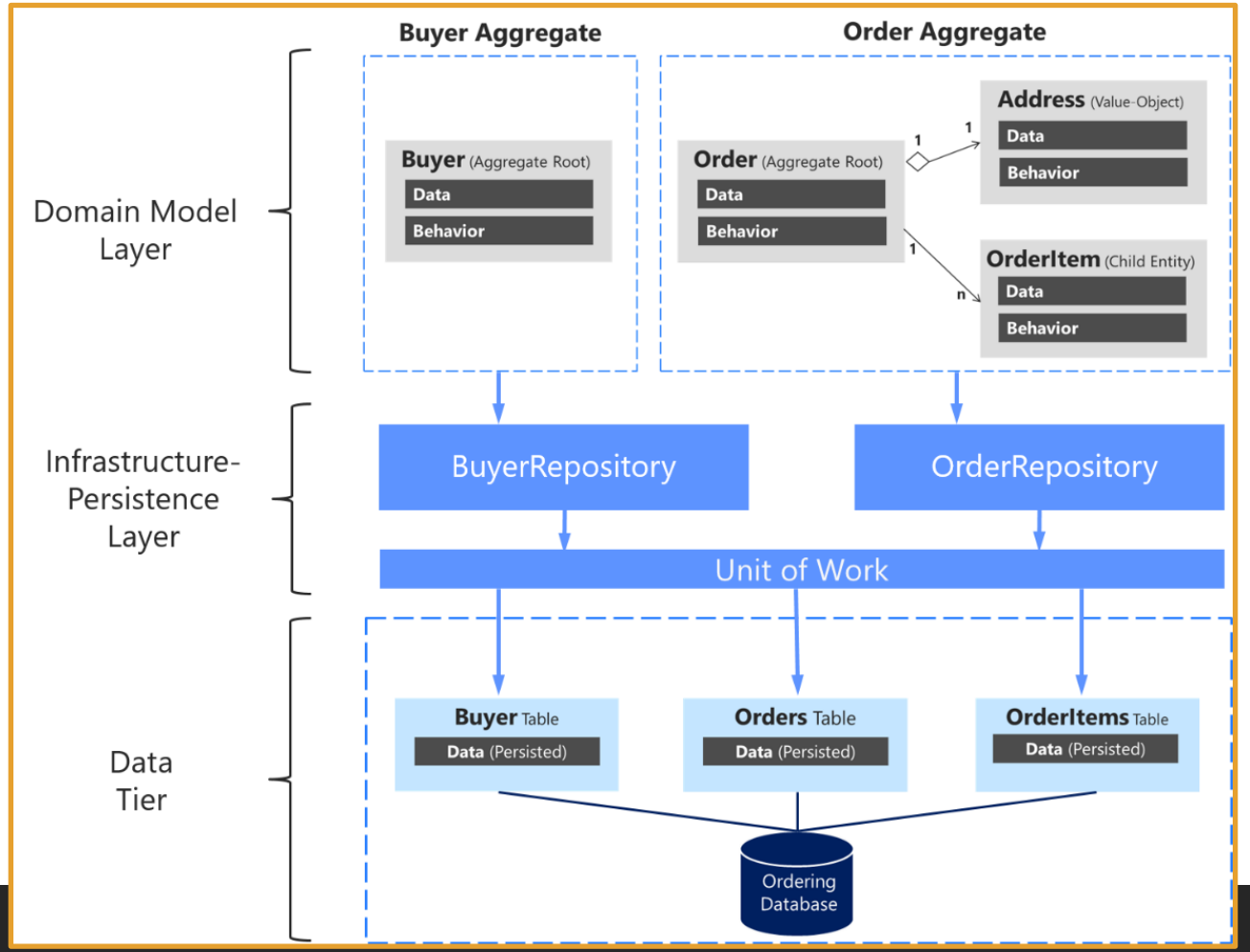
- EF Core forces the use of the Repository Pattern.
- For each aggregate or aggregate root (like a microservice or API), you should create one repository class. This is referred to as Domain-Driven Design (DDD)
- The only channel you should use to update the database should be the repositories because they have a one-to-one relationship with the aggregate root.
- It's okay to query the database through other channels because queries don't change the state of the database.
- The transactional area (updates) must always be controlled by the repositories.



Repository Pattern Best Practices

https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design#define-one-repository-per-aggregate

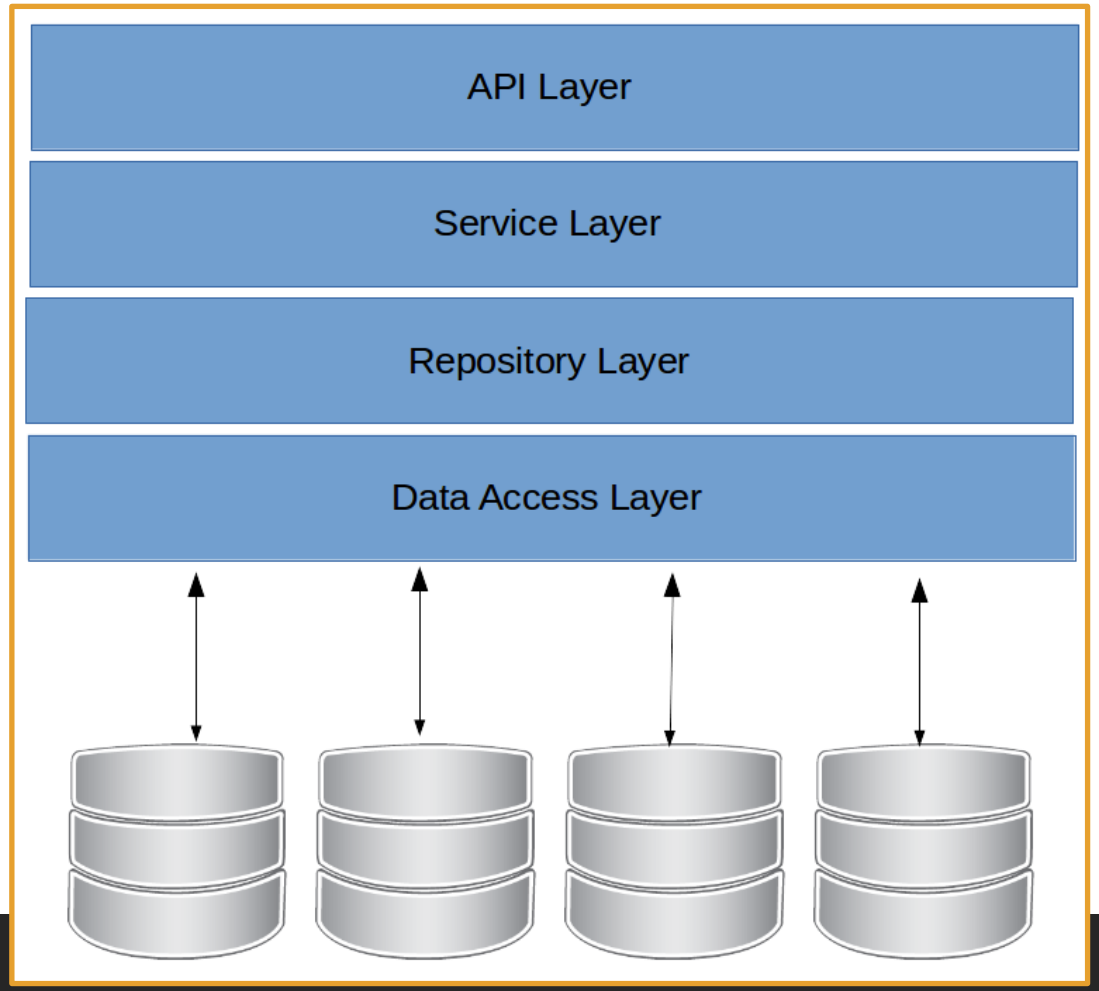
- A repository allows you to populate data in memory that comes from the database in the form of the domain entities. Once the entities are in memory, they can be changed and then persisted back to the database through transactions.
- Data to be updated comes from the client app or presentation layer to the application layer (such as a Web API service).
- Use repositories to get the data you want to update from the database. Update it in memory with the data passed with the commands, and you then add or update the data (domain entities) in the database through a transaction.
- Only define one repository for each aggregate. To maintain transactional consistency between all the objects within the aggregate, never create a repository for each table in the database.



Repository Pattern Benefits

https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design#the-repository-pattern-makes-it-easier-to-test-your-application-logic

- Unit tests only test your code, not infrastructure, so the repository abstractions make it easier to achieve that goal.
- When you define and place the repository interfaces in the domain model layer, the application layer, (API microservice) doesn't depend directly on the infrastructure layer
- Using Dependency Injection in the controllers of your Web API, you can implement mock repositories that return fake data instead of data from the database.
- Using Dependency Injection in your API Controllers decouples your layers and allows you to create and run unit tests that focus the logic of your application without requiring connectivity to the database, which are, by definition, Integration Tests).



Repository Pattern Structure

https://garywoodfine.com/generic-repository-pattern-net-core/