Design Patterns

.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

Repository vs Unit-of-Work Patterns

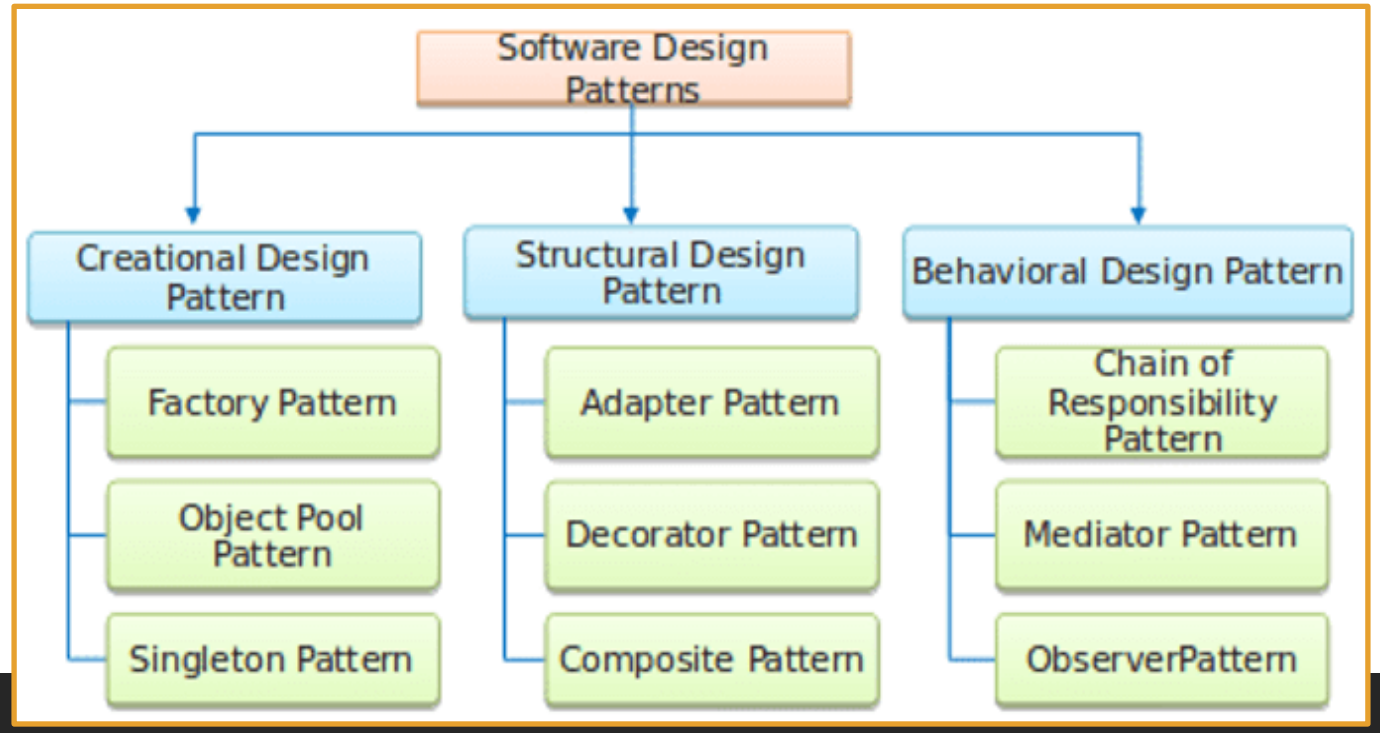
https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application#:~:text=Dan%20Wahlin%27s%20blog.-,Note,-There%20are%20many

---

There are many ways to implement the ***Repository*** and ***Unit-of-Work*** patterns.

- You can use ***repository*** classes with or without a ***Unit-of-Work*** class.
- You can implement a single repository for each entity type.
  - When implementing one entity class for each type, separate classes, a generic base class with derived classes, or an abstract base class and derived classes can be used.
- You can include business logic in your ***repository*** or restrict it to data access logic.
- You can build an abstraction layer into your database ***context*** class by using IDbSet interfaces there instead of DbSet types for your entity sets.

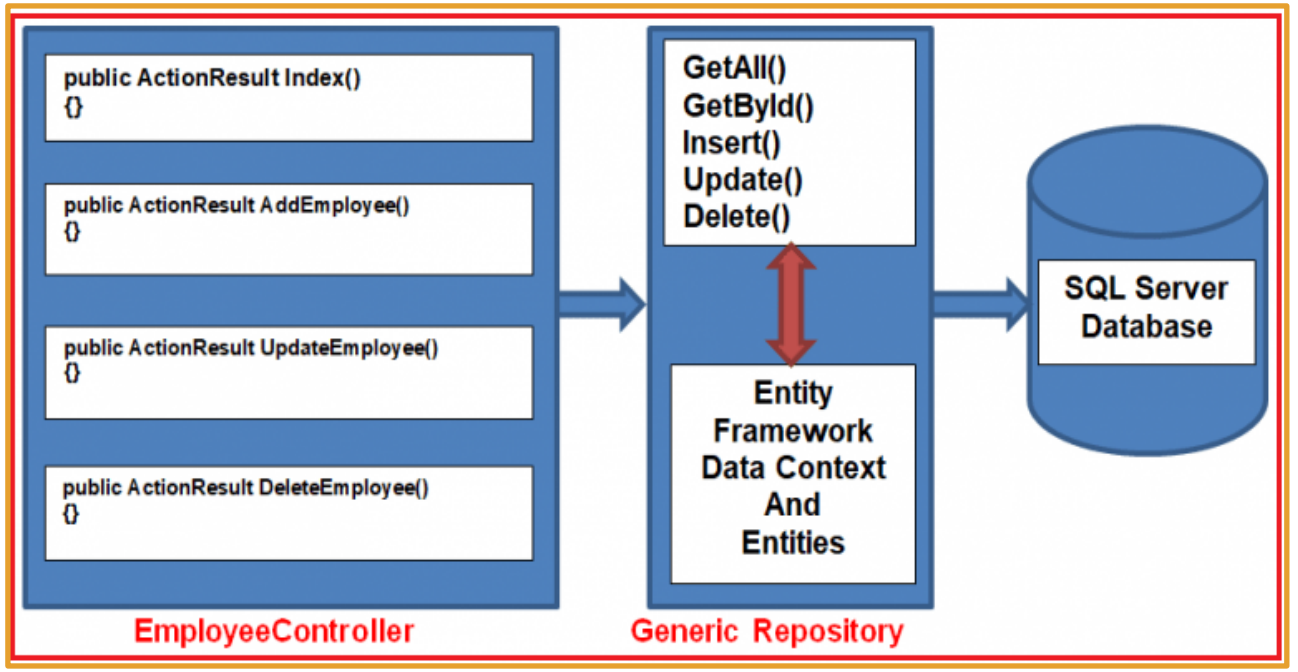
Every approach to implementing an abstraction layer is just one option to be considered when creating an application. Choose which pattern is right for your project.



The Repository Pattern

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/previous-versions/msp-n-p/ff649690(v=pandp.10)?redirectedfrom=MSDN

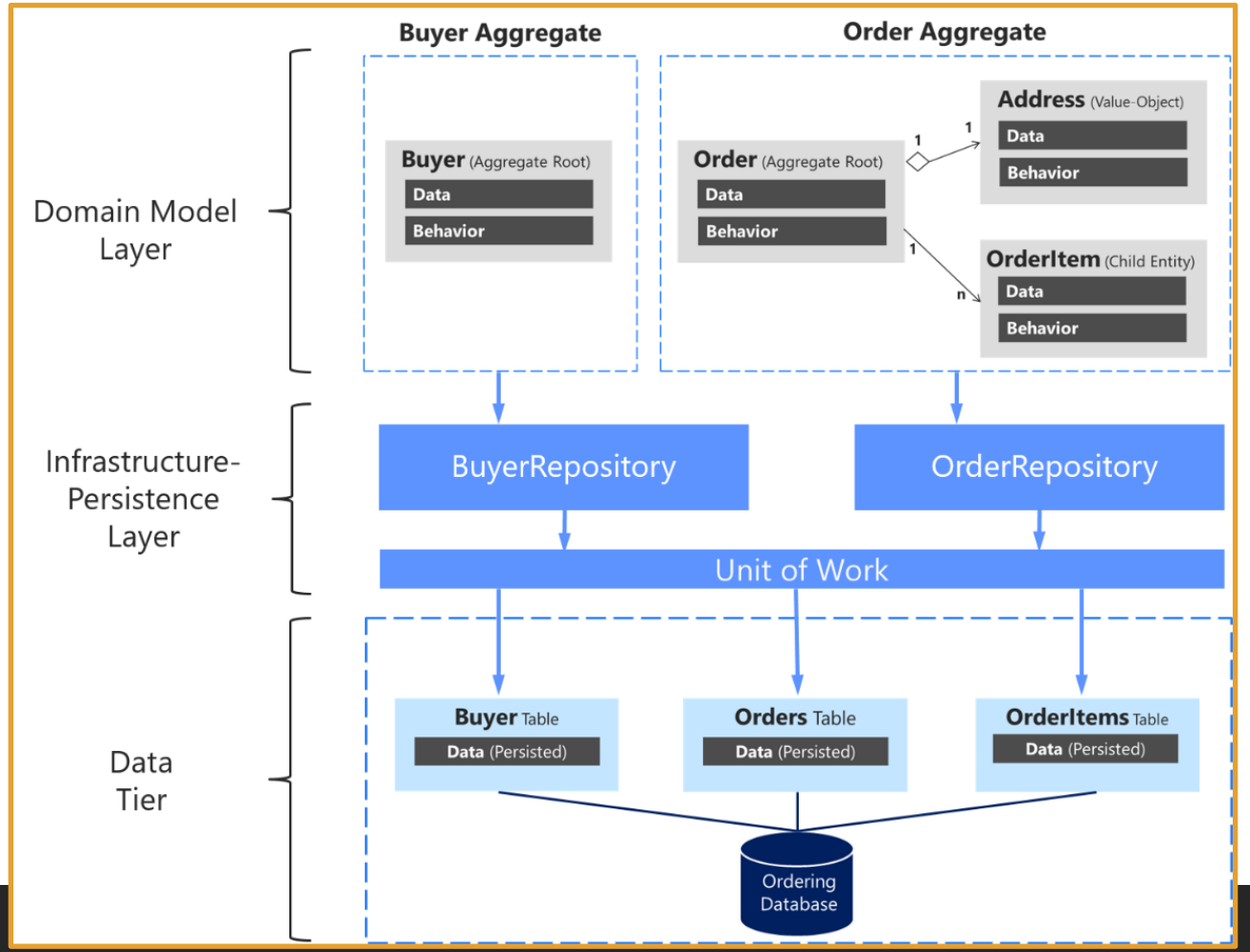
Repositories are classes that encapsulate the logic required to access data sources. They provide better maintainability and decouple the infrastructure or technology used to access databases from the domain model layer



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

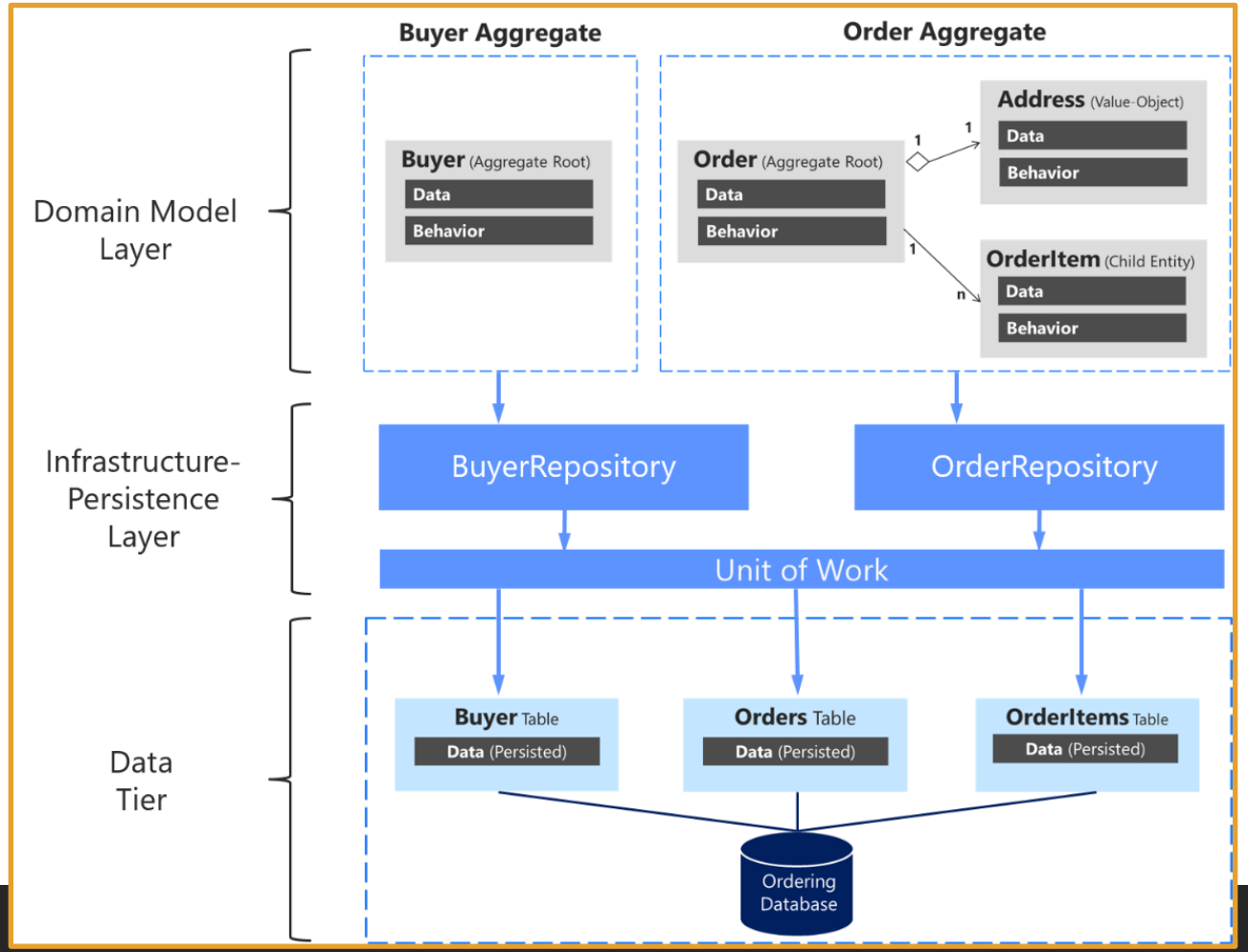
- EF Core forces the use of the ***Repository Pattern***.
- For each aggregate or aggregate root (a microservice or API), you should create one repository class. This is referred to as Domain-Driven Design (DDD)
- The only channel you should use to update the database should be the repositories because they have a one-to-one relationship with the aggregate root.
- It's okay to get data from 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

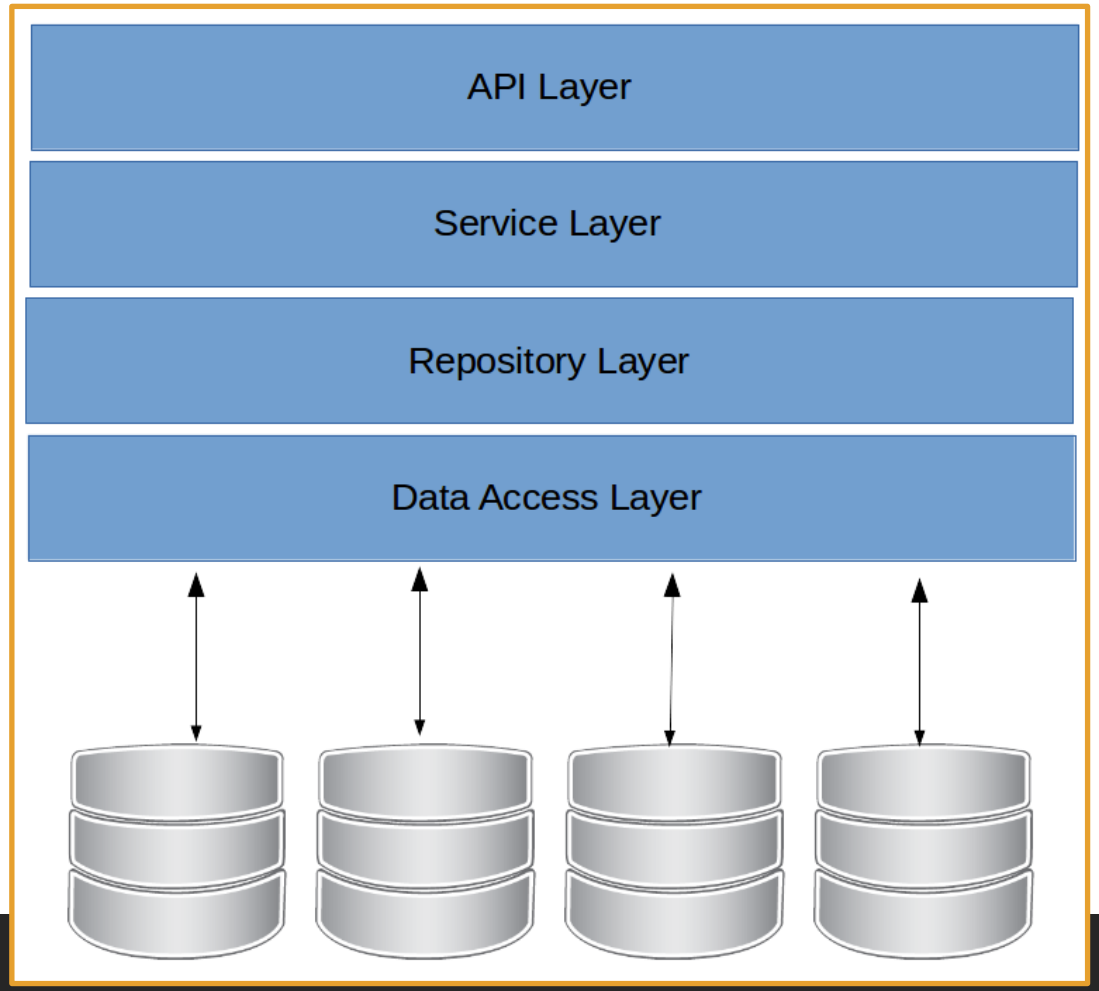
- A ***repository*** allows you to populate data in memory that comes from the database in the form of the domain ***entities***. Once the ***entities*** are in memory, they can be changed and then persisted back to the database through ***transactions***.
- Data to be updated comes from the client app or presentation layer to the application layer (a WebAPI).
- 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 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

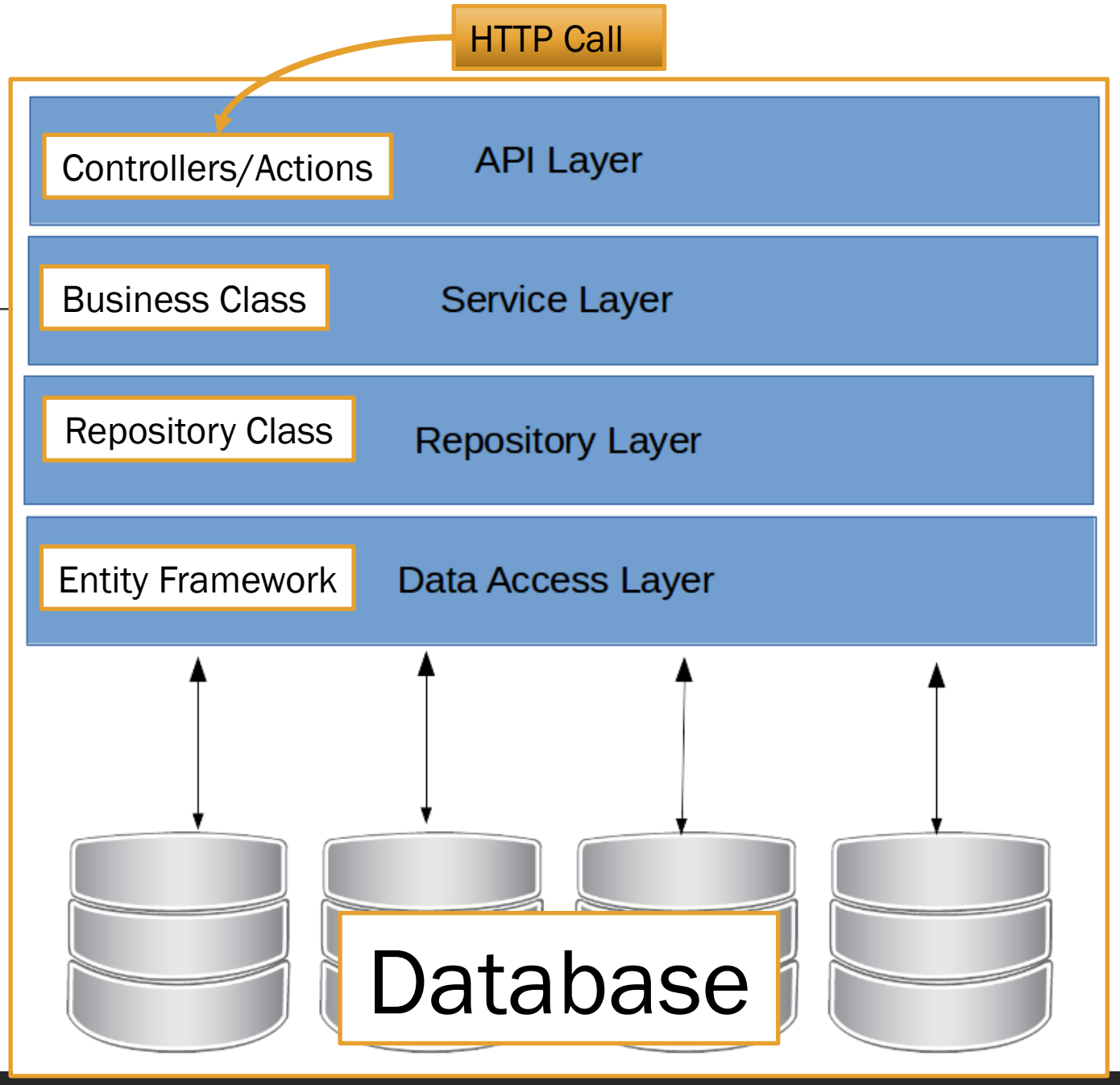
https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design#the-repository-pattern-makes-it-easier-to-test-your-application-logic

- Unit tests only test code, not infrastructure, so the repository abstractions make it easier to achieve that goal.
- When repository interfaces are defined and placed in the domain model layer, the application layer (API service), doesn't depend directly on the infrastructure layer.
- ***Dependency Injection*** in the ***Controller*** of a Web API allows mock ***repositories*** to return mocked (staged) data instead of data from the database.
- ***Dependency Injection*** in ***Controllers*** decouples the layers and allows unit tests that focus on the logic of an application without requiring connectivity to a database.



Repository Pattern Structure

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



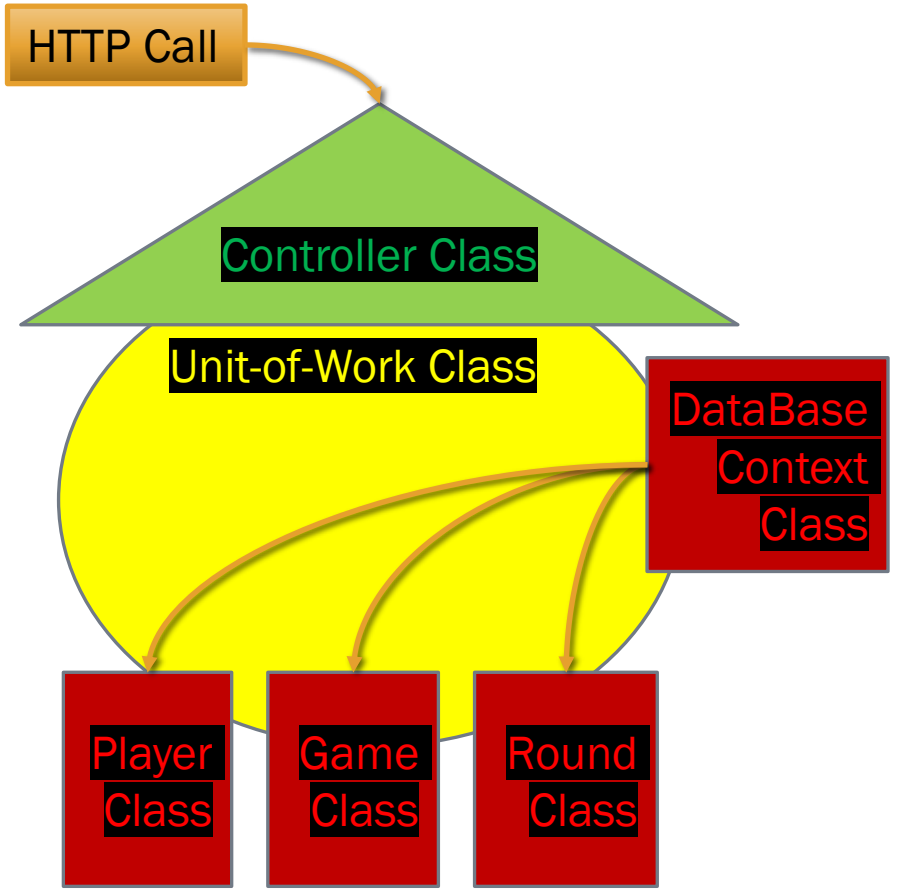
Unit of Work Design Pattern

https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application

---

The ***Unit-of-Work*** Design Pattern helps to minimize redundant code and ensures that all repositories use the same database ***context.*** The ***Unit-of-Work*** *Design Pattern* uses a ***Unit-of-Work*** class along with a ***generic*** repository. The ***Unit-of-Work*** class instantiates the generic repository multiple times with different ***entities*** to perform the actions needed in one work action (unit of work).

The ***Unit-of-Work*** class coordinates the work of multiple repositories by first creating a single database ***context*** class. Then it instantiates the generic repo classes shared by all of them.



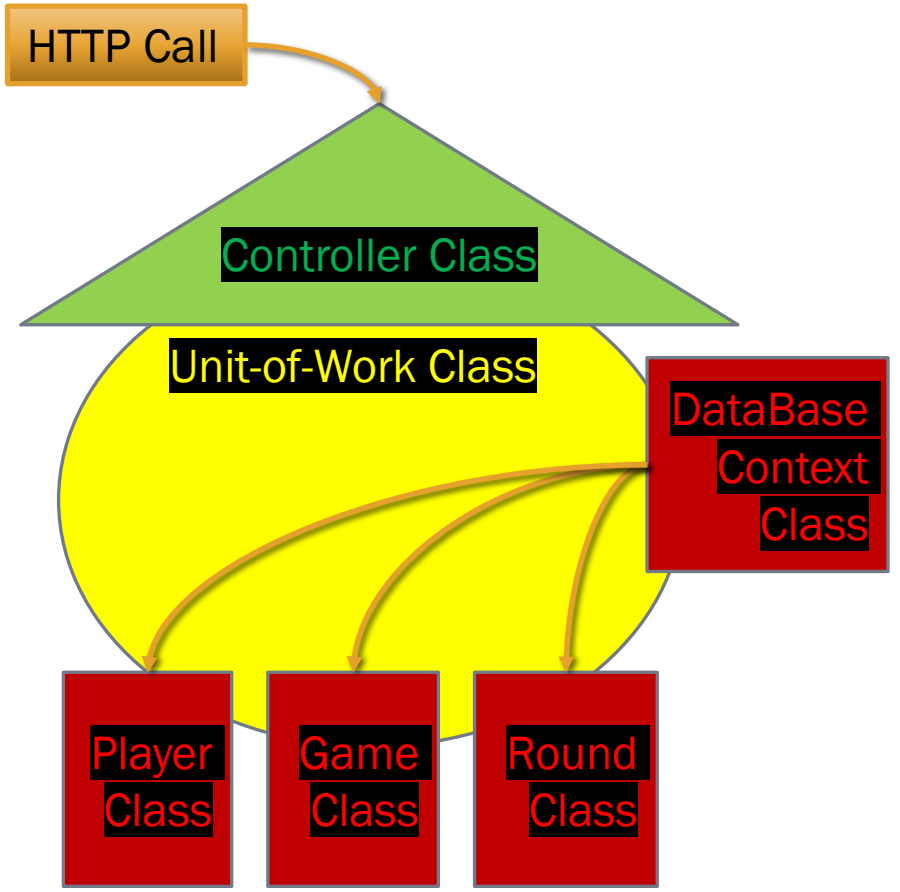
Unit of Work Design Pattern

https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application

---

The unit of work class serves one purpose: to make sure that all repositories share a single database ***context***. When a unit of work (a single action) is complete, call .SaveChanges() on the ***Db context*** instance so all related changes are persisted simultaneously.

The ***Unit-of-Work*** class needs a Save() method and a ***property*** representing an instance of each repository. Each repository ***property*** is a repository instance that has been instantiated using the same database ***context*** instance as the other repository instances.

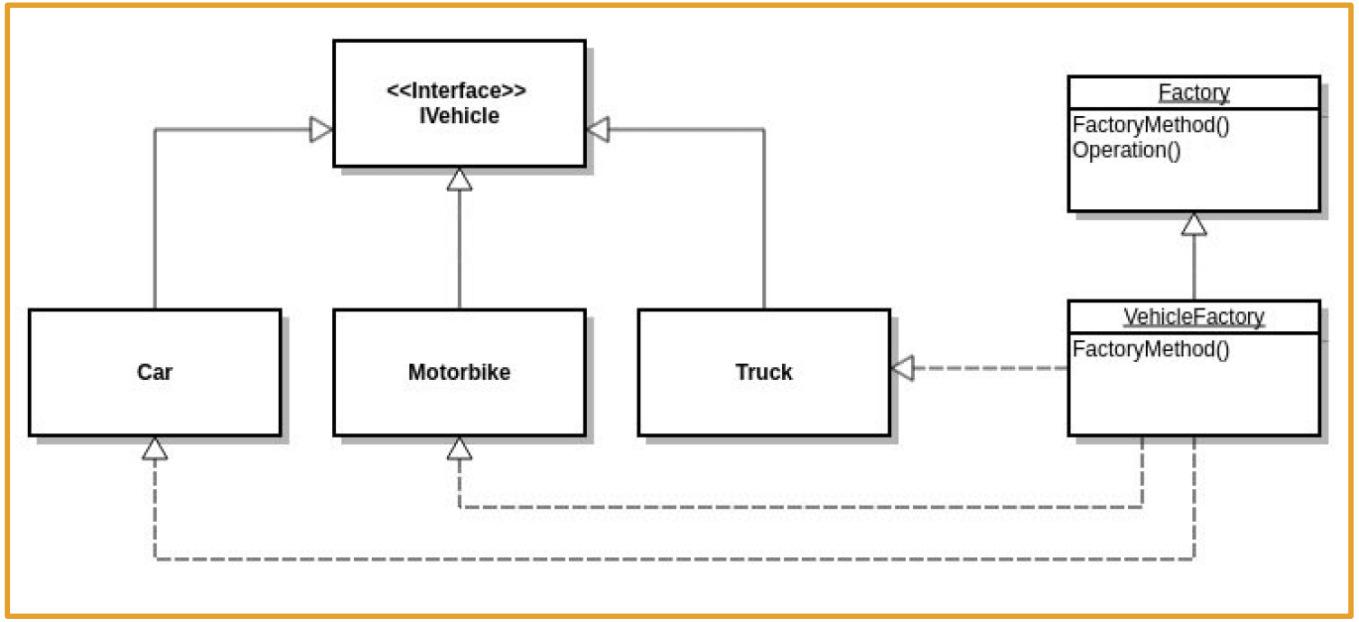


Factory Design Pattern

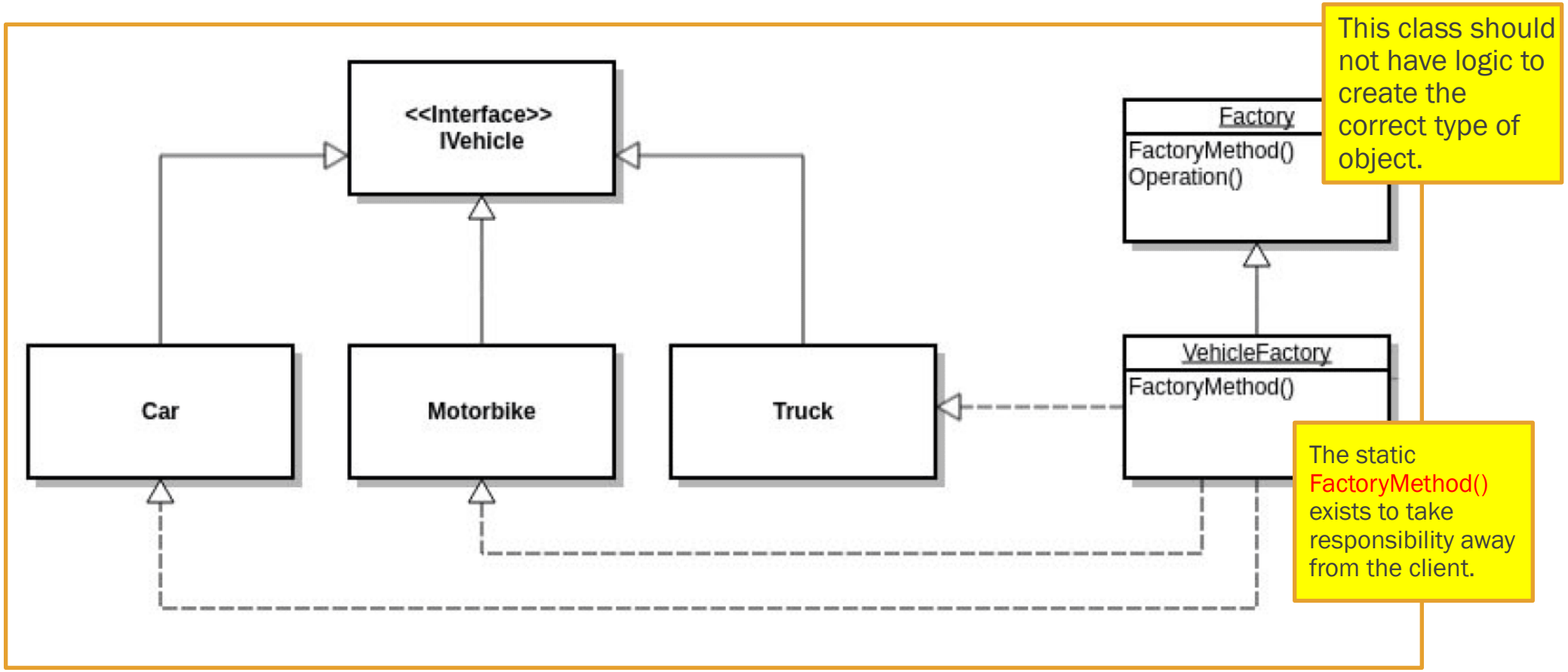
In ***Factory pattern***, objects are created without exposing the creation logic to the client. The client uses the same interface (UI) to create each new type of object.

The idea is to use a ***static*** member-function (***static factory method***) that creates & returns instances, hiding the details of class modules from the user.

A factory pattern is one of the core design principles to create an object, allowing clients to create objects of a library(explained below) in a way such that it doesn't have tight coupling with the class hierarchy of the library.



Factory Design Pattern



Singleton Design Pattern

https://en.wikipedia.org/wiki/Singleton_pattern

The ***Singleton*** pattern restricts the instantiation of a class to one single instance. This is useful when exactly one object is needed to coordinate actions across the system.

The singleton design pattern solves problems by allowing it to:

- Ensure that a class only has one instance
- Easily access the sole instance of a class
- Control its instantiation
- Restrict the number of instances
- Access a global variable