ASP.NET Core

Web API

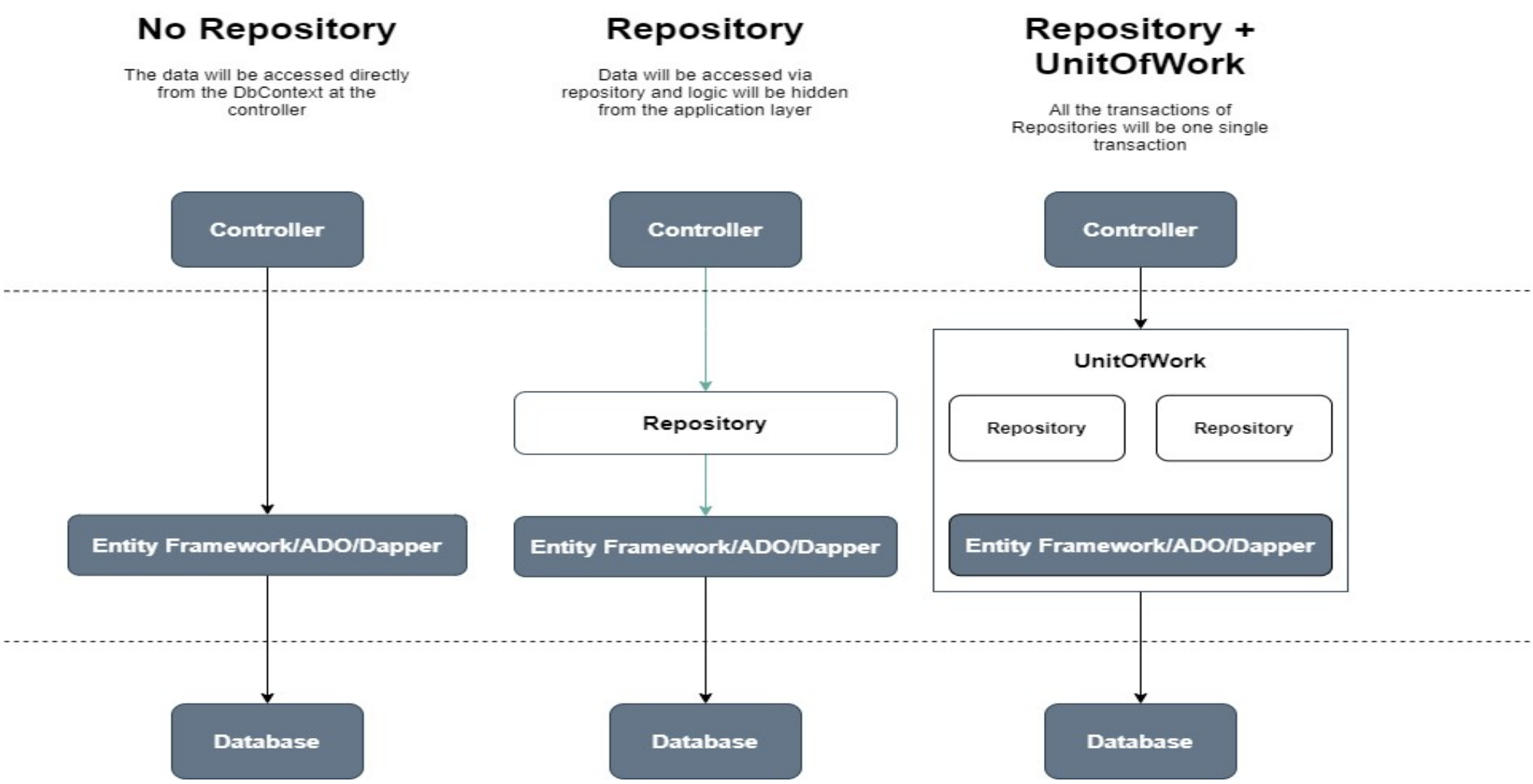
Mohamed ELshafei

Implementing the Repository and Unit of Work Patterns

The Repository and Unit of Work Patterns

The repository and unit of work patterns are intended to create an abstraction layer between the data access layer and the business logic layer of an application. Implementing these patterns can help insulate your application from changes in the data store and can facilitate automated unit testing or test-driven development (TDD).

The unit of work class coordinates the work of multiple repositories by creating a single database context class shared by all of them.


No Repository
The data will be accessed directly from the DbContext at the controller
Repository
Data will be accessed via repository and logic will be hidden from the application layer
Repository + UnitOfWork
All the transactions of Repositories will be one single transaction
Controller
Controller
Controller
Repository
UnitOfWork
Repository
Repository
Entity Framework/ADO/Dapper
Entity Framework/ADO/Dapper
Entity Framework/ADO/Dapper
Database
Database
Database

Repository Pattern In ASP.NET Core MVC And Entity Framework Core

The repository pattern is intended to create an Abstraction layer between the Data Access layer and Business Logic layer of an Application. It is a data access pattern that prompts a more loosely coupled approach to data access. We create a generic repository, which queries the data source for the data, maps the data from the data source to a business entity and persists changes in the business entity to the data source

Prerequisite - Dependency Inversion Principle (DIP)

- **The Dependency Inversion Principle (DIP)** states that high level modules should not depend on low level modules; both should depend on abstractions (typically interfaces). Abstractions should not depend on details.
- Details should depend upon abstractions which are provided to them when the class is constructed.

WHY?

- We implement repository pattern to develop a loosely coupled application. It makes the code more testable. It creates an abstraction layer between ORM and business logic layer of the application.
- the repository mediates between the data source layer (Entity Framework) and the business layer (Controller) of the application. It performs operations as following way.
  - It queries the underlying data source (database) for the data.
  - It maps the data from the data source to a business entity that uses to create the database.
  - It persists changes in the business entity to the data source.

Advantages of Repository Pattern

It has some advantages which are as follows:

- An entity might have many operations which perform from many locations in the application, so we write logic for common operations in the repository. These operations might be Create, Read, Update, Delete, Search, Filter, Sort, Paging, and Caching etc.
- The Entity Framework is not testable out of the box. We have to create mock classes to test it. Data access logic can be tested using repository pattern.
- As business logic and data access logic separate in the repository pattern that’s why it easy to manage and readable. It reduces development as well me, as common functionality logic writes once in the application.

How? “Employee Entity”

- ASP.NET Core is designed to support dependency injection. So, we create a repository interface named IEmployeeRepository for Employee .
- This interface has definitions of all methods to perform CRUD operations on the Employee entity.
- let’s implement the preceding interface on a class named EmployeeRepository.
- This class has implemention of all methods to perform CRUD operations on the Employee entity. It has a parameterized constructor that accepts Context as a parameter. It passes at a time of repository instance creates by dependency injection.

How? “Employee Entity” cont..

- web application communicates to data access layer via an interface so we register repository to the dependency injection during the application start up.
- **Create Application User Interface**
- create a view model named EmployeeViewModel for application UI and CRUD operations. This model strongly binds with a view.

Implement a Generic Repository and a Unit of Work Class

- Creating a repository class for each entity type could result in a lot of redundant code, and it could result in partial updates. For example, suppose you have to update two different entity types as part of the same transaction. If each uses a separate database context instance, one might succeed and the other might fail. One way to minimize redundant code is to use a generic repository, and one way to ensure that all repositories use the same database context (and thus coordinate all updates) is to use a unit of work class.

Implement a Generic Repository and a Unit of Work Class

- The unit of work class serves one purpose: to make sure that when you use multiple repositories, they share a single database context. That way, when a unit of work is complete you can call the SaveChanges method on that instance of the context and be assured that all related changes will be coordinated. All that the class needs is a Save method and a property for each repository. Each repository property returns a repository instance that has been instantiated using the same database context instance as the other repository instances.