Multi-layered Architectures in .Net

Kristijan Horvat
Software Architect
kristijan@mono-software.com

mono.track

Table of Contents

What is multi-layered architecture ?

A multi-layered software architecture is a software architecture that uses many layers for allocating the different responsibilities of a software product.

Wikipedia

Advantages and disadvantages

Advantages

- Increases flexibility, maintainability, and scalability
- Multiple applications can reuse the components
- Enables teams to work on different parts of the application
- Enables develop loosely coupled systems
- Different components of the application can be independently deployed and maintained
- Helps you to test the components independently of each other

Disadvantages

- Longer implementation period
- Possible negative impact on the performance
- Tends to become very complex
- Adds unnecessary complexity to simple applications

Layer or Tier ? What's the difference ?

Layers are logical separation

- Logical layers are merely a way of organizing your code. Typical layers include Presentation, Business and Data

Tiers are physical separation

- Defines a place where the code runs. Specifically, tiers are places where layers are deployed and where layers run. In other words, tiers are the physical deployment of layers.

Layered, Onion & Hexagonal architectures

Layered

- An architecture in which data moves from one defined level of processing to another

Onion

- An architecture that has layers defined from core to Infrastructure and code can depend on layers more central, but code cannot depend on layers further out from the core.

Hexagonal

- Hexagonal Architecture is an architecture defined by establishing a perimeter around the domain of your application and establishing adapters for input/output interactions. By establishing this isolation layer, the application becomes unaware of the nature of the things it's interacting with.

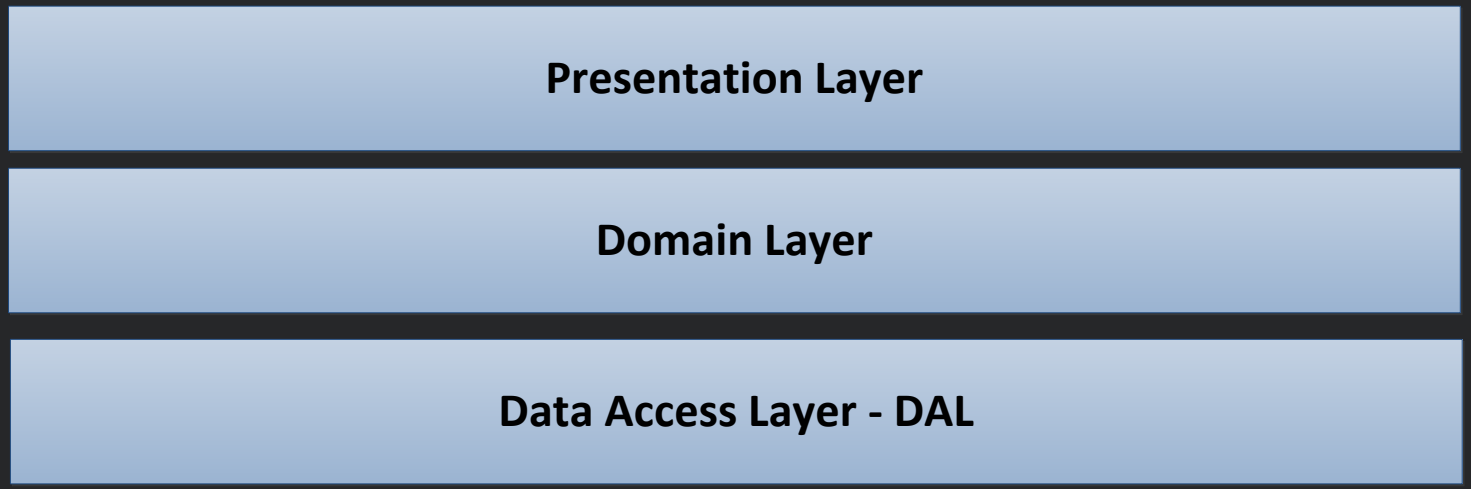
Layered architecture

Layered architecture is an architecture in which data moves from one defined level of processing to another

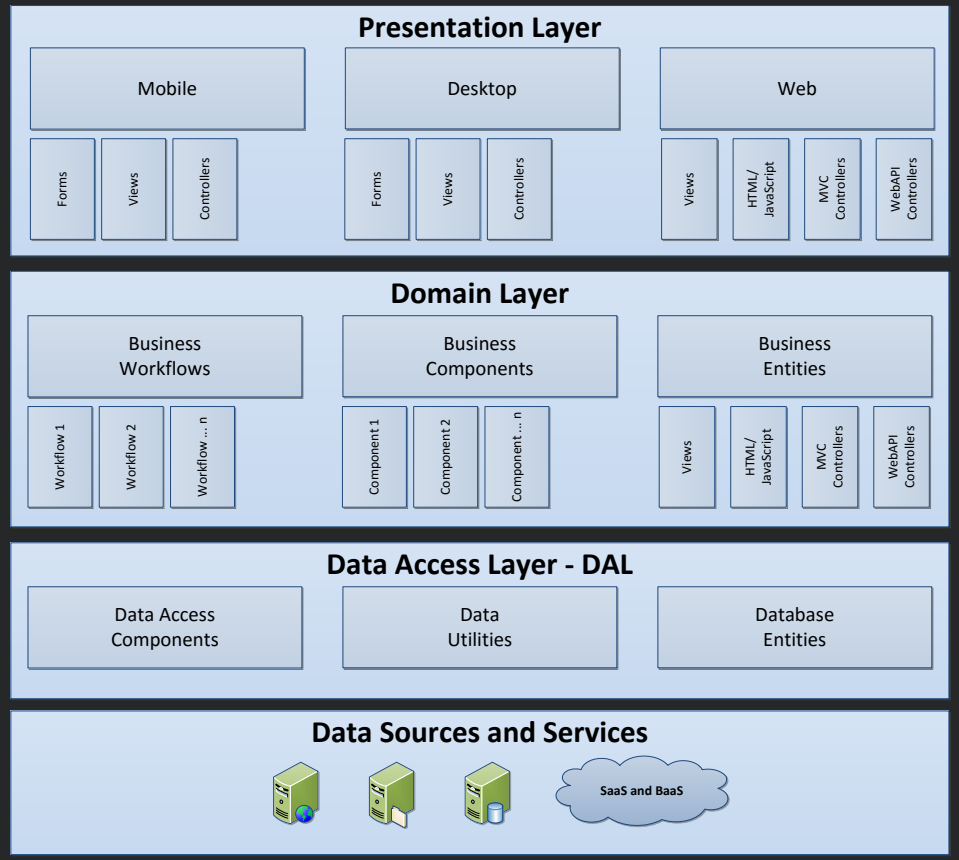
Layered - Involved Layers

- Presentation layer
  - Also know as Front-End UI
  - Platforms - Mobile, Desktop, Web, etc.
  - Technologies - HTML5/JavaScript, MVC, WebForms, WPF, etc.
- Domain Layer
  - Also know as Business Layer, BLL, Service Layer
  - Contains business logic and entities
- Data Access Layer
  - Also know as DAL, Data Layer
  - Contains database models or entities

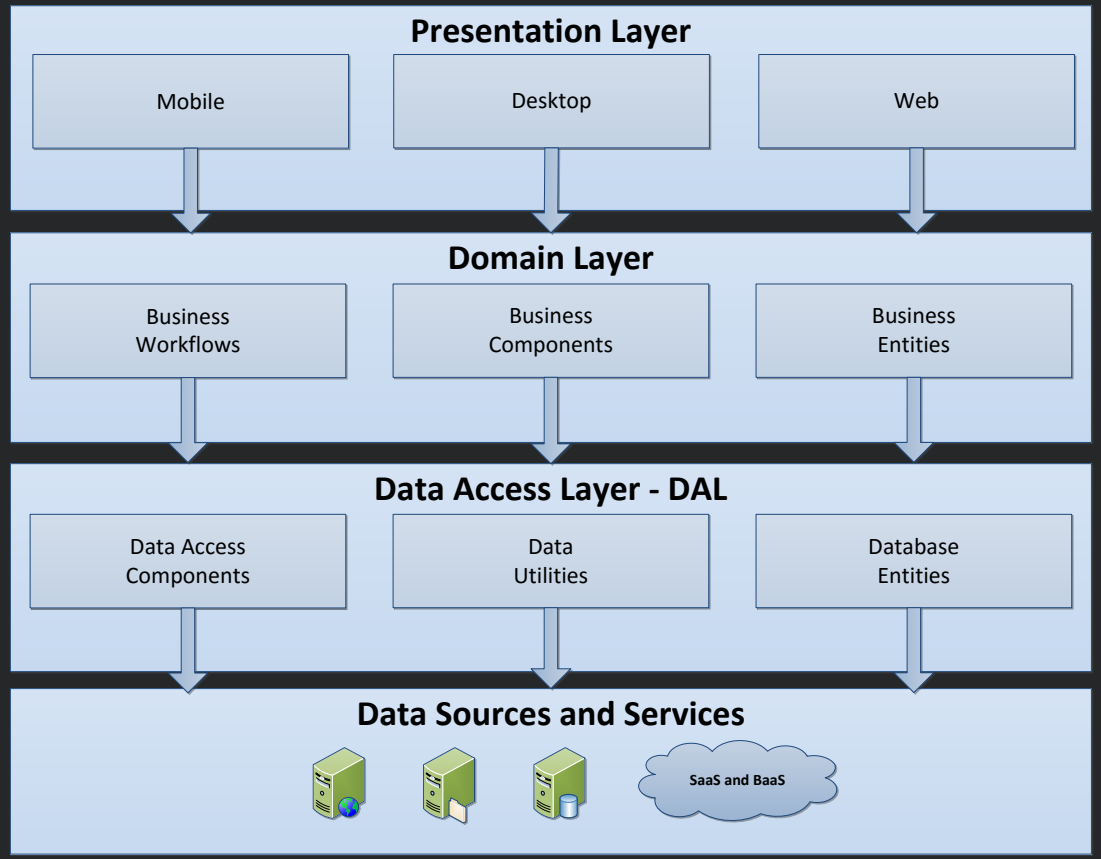
Layered architecture - main diagram



Layered architecture - complex diagram



Layered architecture - layer coupling diagram



Layered architecture

Advantages

- Increases flexibility to a certain level
- Reuse the components
- Teams can work in parallel

Disadvantages

- Harder to determine appropriate layer for functionality at hand
- Harder to introduce new team members - learning curve
- Tends to expose database models to top layers

DIP, DI & IoC, anybody ?

In order to work with Onion and Hexagonal architectures one should be familiar with the dependency inversion principle (DIP), dependency injection (DI), inversion of control (IoC) as they heavily depend on it.

Dependency Injection - DI

- In case where service depends on some other services, dependencies are injected into target service rather then instantiated inside the service.
- On of the best practices and most used technique is constructor injection
- Key features you should know when working with DI
  - constructor injection
  - factory pattern
  - facade services

Inversion of Control - IoC

- Design in which parts of application receive a flow of control from generic reusable library.
- In practice IoC will come down to substitution of one abstraction with another, or substitution of one implementation layer with another
- Key features you should know when working with IoC
  - decoupling
  - increased modularity
  - increased extensibility

Dependency Inversion Principle - DIP

- To apply DIP your architecture should follow these simple rules:
  - Common classes depend on nothing
  - Domain classes depend only on Common classes and each other
  - Service classes depend only on Domain classes
  - Infrastructure classes depend only on Service and Domain classes

DI has important role in the architecture

- Unit testing made easier
- Architecture layers can be replaced using IoC
- Glue everything together - Onion & Hexagonal

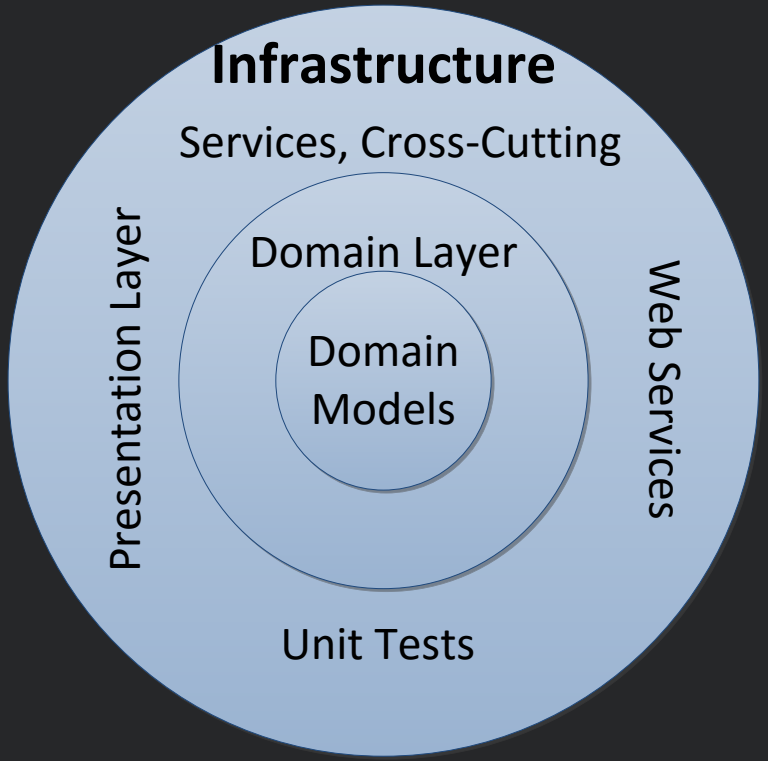
Onion architecture

Onion is an architecture that has layers defined from core to Infrastructure and code can depend on layers more central, but code cannot depend on layers further out from the core.

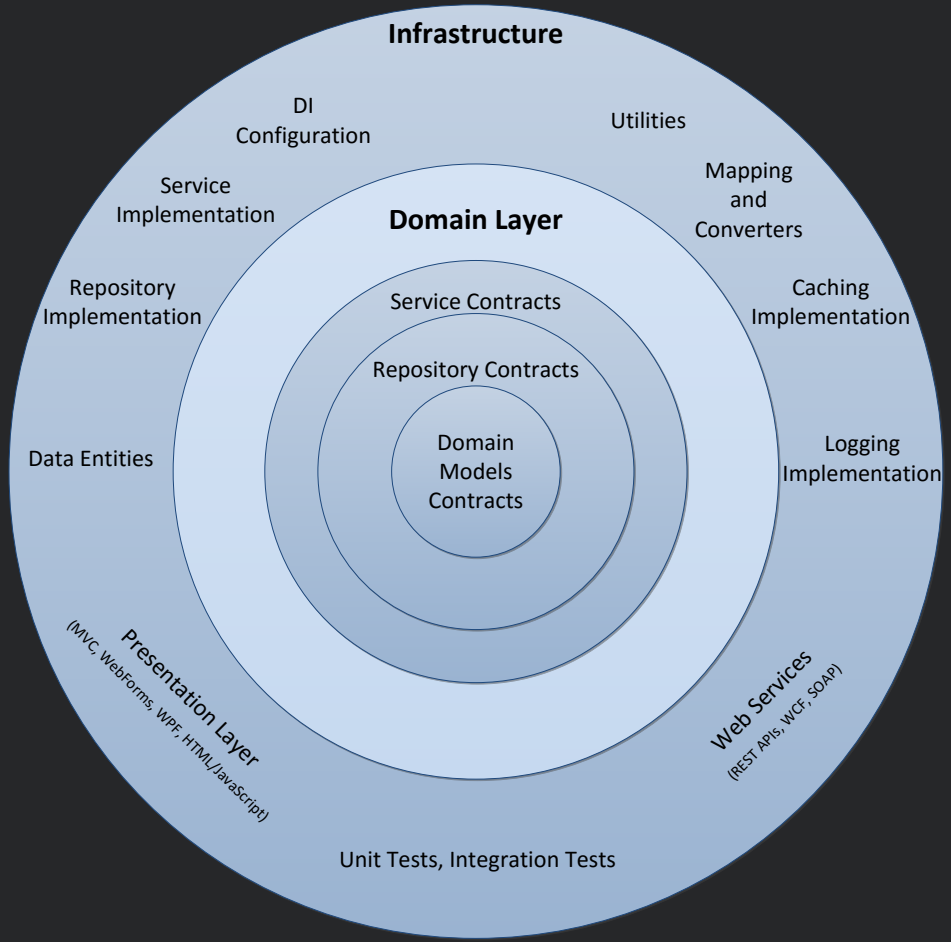
Onion - Involved Layers

- Presentation Layer (Front-End UI)
  - Platforms - Mobile, Desktop, Web, etc.
  - Technologies - HTML5/JavaScript, MVC, WebForms, WPF, etc.
- Web Service Layer (REST APIs, SOAP, etc.)
- Domain Layer
  - Business classes or BLLs, Service classes
- Domain Model Layer
- Infrastructure Layer
  - Repository layer
  - Data Access Layer (DAL, Data Layer)
    - contains database models or entities

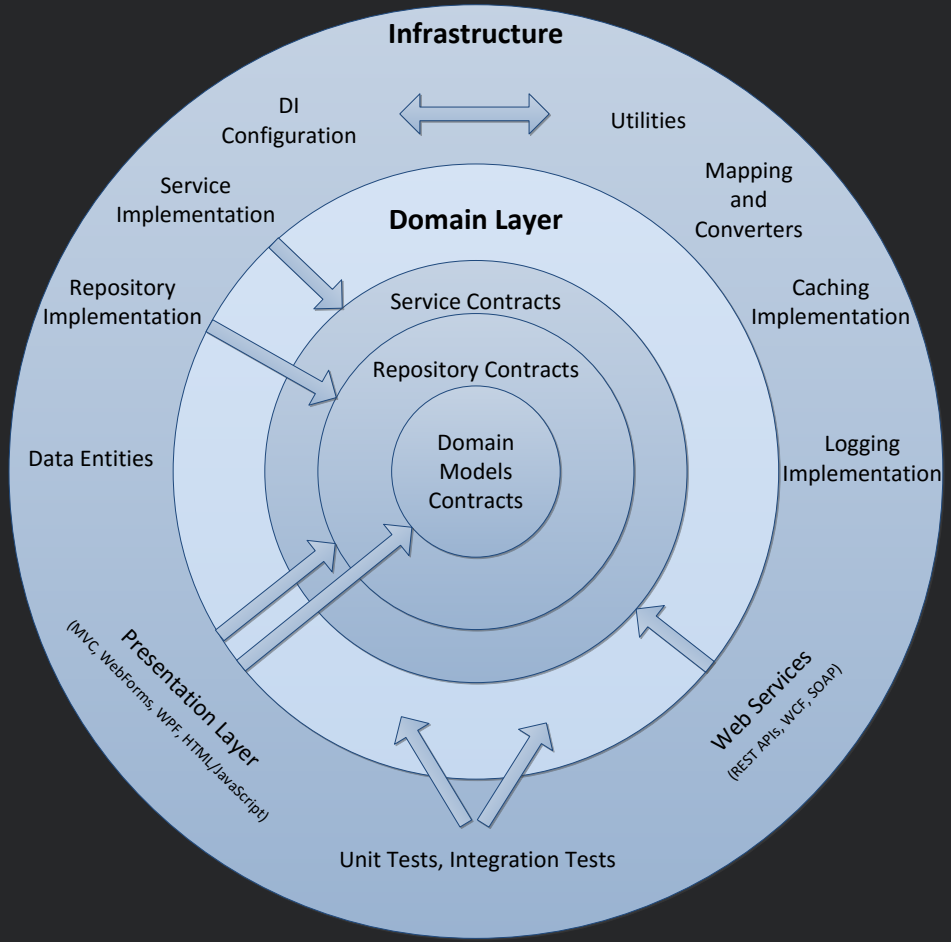
Onion architecture - main diagram



Onion architecture - complex diagram



Onion architecture - layer coupling diagram



Onion vs Layered architecture

Onion

- Domain model is in the middle of the architecture
- Outer layers can communicate only with inner layers
- Layers can communicate with multiple inner layers
- More complex
- Requires many elements to be abstracted
- Heavily depends on DIP (DIP/DI/IoC)
- Easily decoupled

Layered

- Database models are at the bottom of the architecture
- Layers can communicate only with layers one level beneath
- Less complex
- Only parts of application needs to be abstracted
- Uses DI and IoC where needed
- Easy/Hard to decouple

Onion architecture

Advantages

- Increases flexibility, maintainability, and scalability
- Multiple applications can reuse the components
- Enables teams to work on different parts of the application
- Enables develop loosely coupled systems
- Different components of the application can be independently deployed and maintained
- Helps you to test the components independently of each other

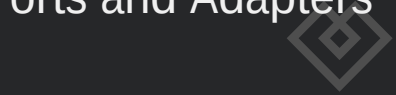
Disadvantages

- Longer implementation period
- Easier, than in layered architecture, to introduce new team members – learning curve
- Possible negative impact on the performance
- Tends to become very complex
- Adds unnecessary complexity to simple applications

Hexagonal architecture

Hexagonal Architecture is an architecture defined by establishing a perimeter around the domain of your application and establishing adapters for input/output interactions. By establishing this isolation layer, the application becomes unaware of the nature of the things it's interacting with.

Also know as Ports and Adapters



Hexagonal Onion architecture

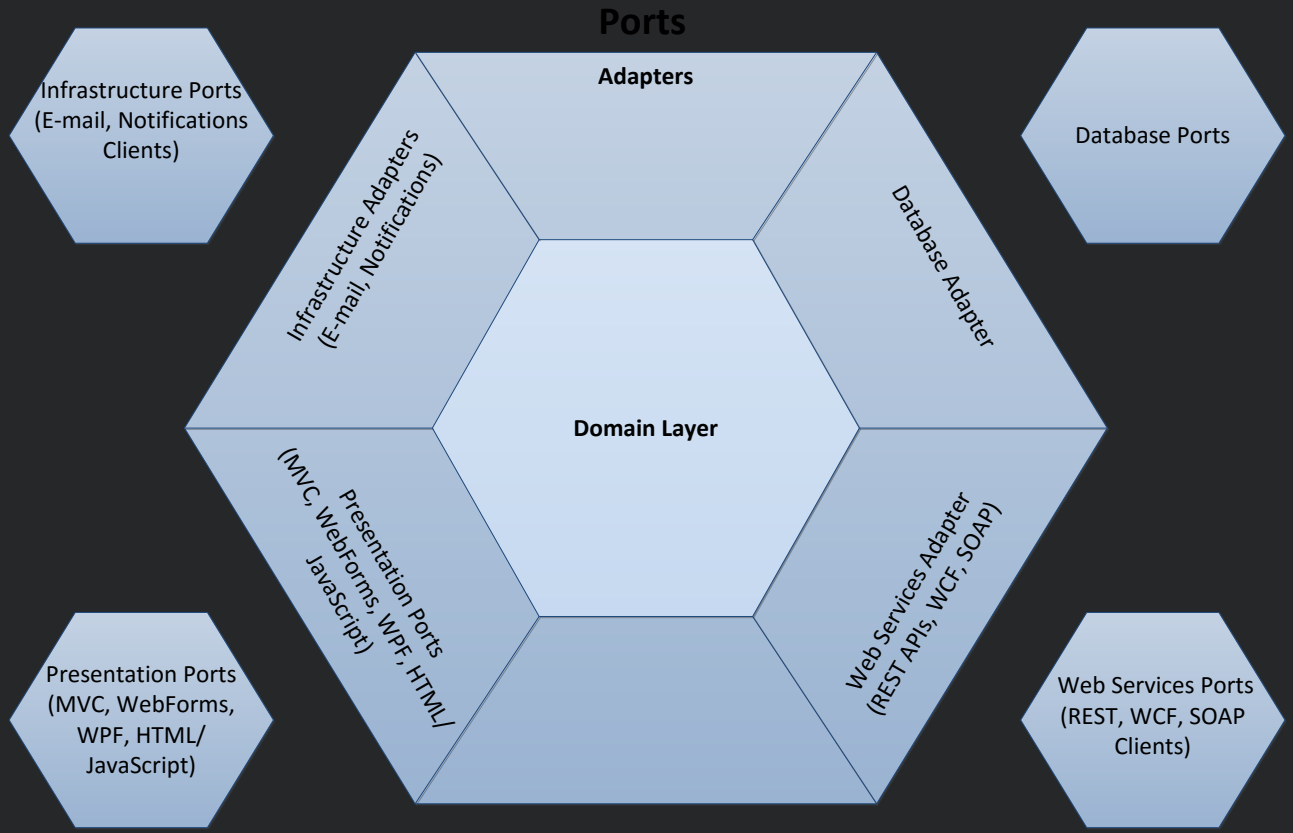
Hexagonal architecture only addresses how external dependencies connect with the application, while Hexagonal Onion applies Onion structuring and Hexagonal principles of establishing isolation layer towards outside world.

Hexagonal Onion - Involved Layers

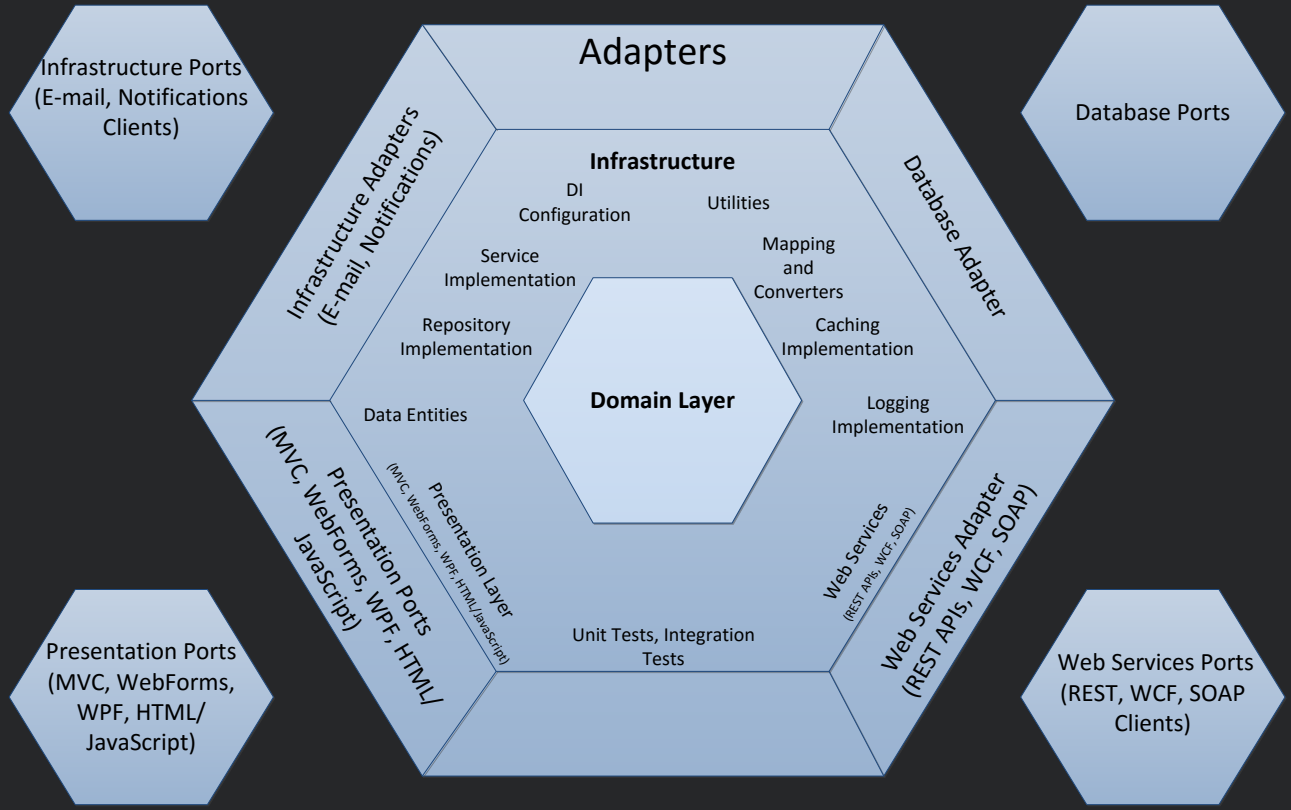
Almost same layers as Onion but different coupling

- Ports and Adapters
  - Web Services (REST APIs, SOAP, Message Bus, etc.)
- Domain Layer
  - Business Layer, BLL, Service Layer
- Domain Models
- Infrastructure Layer
  - Repository Layer
  - Data Access Layer (DAL, Data Layer)

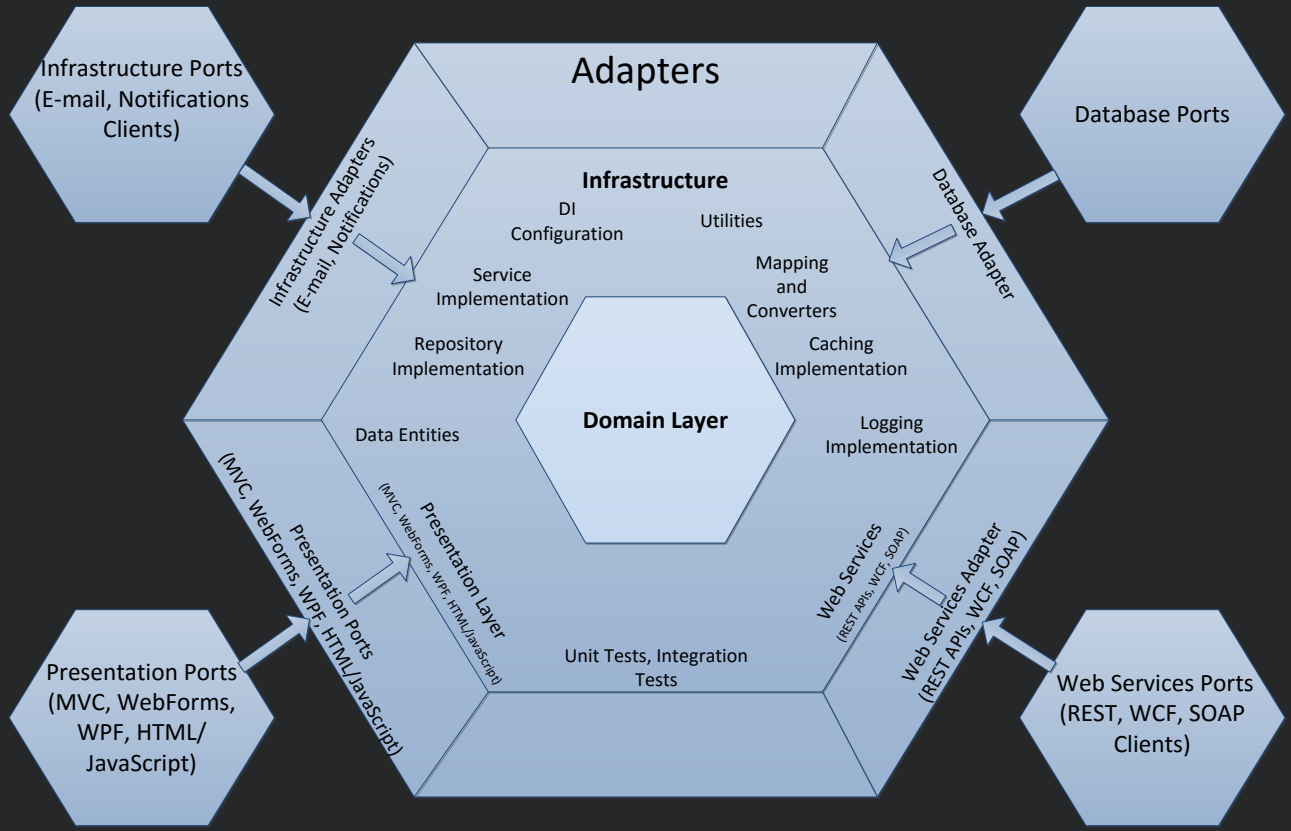
Hexagonal architecture - main diagram



Hexagonal architecture - complex diagram



Hexagonal architecture - layer coupling diagram



Onion vs Hexagonal Onion architecture

Onion

- *Domain model is in the middle of the architecture*
- *Outer layers can communicate only with inner layers*
- *Same complexity*
- *Requires same level of abstraction*
- *Heavily depends on DIP (DIP/DI/IoC)*
- *Easily decoupled*

- Layers can communicate with multiple inner layers
- Infrastructure layers can communicate with each other

Hexagonal Onion

- *Domain model is in the middle of the architecture*
- *Outer layers can communicate only with inner layers*
- *Same complexity*
- *Requires same level of abstraction*
- *Heavily depends on DIP (DIP/DI/IoC)*
- *Easily decoupled*

- Layers can communicate only with immediate neighbor layer
- Ensure domain logic is not bypassed in infrastructure layer
- Ensure infrastructure components are not coupled together
- Hexagonal doesn't address how the application is structured. It only addresses how external dependencies connect with the application.

Hexagonal Onion architecture

Advantages

- Same advantages as Onion and
- Multiple applications and services can reuse the components
- Teams can develop stand alone services
- Truly loosely coupled systems
- Domain logic is not bypassed in infrastructure layer
- Infrastructure components are not coupled together

Disadvantages

- Same disadvantages as Onion
- Tends to become even more complex than Onion

Architecture in Practices

Common challenges when designing architecture

- Choose appropriate architecture for your project
- Layers Demystified and Component placement
- Naming conventions
- SRP "all the way" (Single Responsibility Principle)
- Security handling
- Caching through layers
- Exception handling
- Package handling
- Do not go overboard

Choose Appropriate Architecture

- Layered
  - Small projects
  - Small teams
  - Slow evolving projects
- Onion
  - Mid to Large projects
  - Large teams
  - Fast evolving projects
- Hexagonal Onion
  - Mid to Large projects
  - SaaS/BaaS solutions
  - Large, Decentralized teams
  - Fast evolving projects

Common - Architecture layers demystified

Present in almost all projects, any layer can depend on common

- Utility classes
  - Converters, Helpers
- Cross-Cutting abstractions
- Data structures
- Custom exceptions

- Cross-Cutting Attributes
- Event arguments
- Parameters

Domain - Architecture layers demystified

Depends only on other domain classes and common classes

- Business domain classes
  - Model Contracts/Abstractions
- Business domain logic
  - Services (BLLs) contracts
  - Business Validation
  - Lookups contracts
  - Membership contracts
  - ACL (Access Control List) Services

- Can use cross-cutting abstractions
  - Logging (ILogger)
  - Caching (ICacheProvider)
  - Mapping (IMapper)
- Exposes operations needed by upper layers
  - In Layered arch. - WebAPI, MVC, etc.
  - In Onion arch. - WebAPI, MVC, etc.
  - In Hexagonal arch. - Ports and Adapters

Infrastructure - Architecture layers demystified

Can depend on any component as required

- Contains components that connect domain and services to outside world
  - Controllers, MBus, Database, etc.
  - In Hexagonal they are called Ports and Adapters
- Contains layer implementations and sub-components
  - I/O components
  - ASP.NET WebAPI or MVC
  - Repositories
  - Domain models - implementation
  - Dependency Injection configuration

- Services used to interact with domain layer
- Validation
  - REST model validation
  - ViewModel validation
- Security
  - Authentication
  - Authorization

Naming Conventions – Domain Layer

- Project.Models.Common
  - Domain Model Contracts
  - *IUser*, *IUserModel*
- Project.Repository.Common
  - Repository contracts - specialized contracts used for data access operations
  - *IUserRepository*, *ICompanyRepository*
- Project.Service.Common
  - Service contracts - business logic contracts
  - *IUserService*, *ICompanyService*, *IUserBLL*

Naming Conventions – Infrastructure Layer

- Project.Web
  - HTML/JavaScript, MVC, WebForms, etc.
  - DI Configuration
- Project.REST or WebService
  - WebAPI, SOAP, etc.
  - DI Configuration
- Project.Service
  - Service/BLL implementations
- Project.Repository
  - Repository implementations
- Project.DAL
  - Database entities
  - *UserEntity*, *CompanyEntity*
- Project.Common
  - Utilities, Mapping, Converter
- Cross-Cutting Components
  - Caching, Logging, etc.

SRP "all the way" (Single Responsibility Principle)

It is important to follow the SRP principle in all elements of application design. You should be carefully when deciding responsibility of:

- layers
- contracts and classes
- cross-cutting libraries

Security Handling

There are few approaches when handling security, handling in Infrastructure layer (WebAPI, MVC) or in Domain layer (Business classes)

- Authentication in Infrastructure layer (WebAPI, MVC)
  - faster, but not reusable
- Authentication in Domain layer (Business classes)
  - reusable, slower
- Authorization in Infrastructure layer (WebAPI, MVC)
  - can be fast, not reusable, business rules can be violated
- Authorization in Domain layer (Business classes)
  - slower, reusable, business rules are not violated

Caching through layers

Data and output caching are just some of caching mechanisms you can use.

Data caching

- mostly used in domain layer
- boost performance
- offload database servers

Output caching

- used in infrastructure layer
  - MVC, WebForms, WebAPI
- boost performance
- offload web servers

Keep in mind

- Caching invalidation is very complex
- Distributed cache may be introduced
  - Redis, Azure In-Role, CouchBase, MemCache, etc.

Exception handling

There are many debates online on how and where to handle exceptions.

How do you od it ?

- Throw exception in domain layer, catch exception in infrastructure layer
- Infrastructure layer catch all approach
- or

Package handling

When dealing with large projects it is important to decouple everything. Decoupling services, tools and cross-cutting functionality will bring a large number of smaller libraries that you need to maintain. In order to do it successfully one should use package managers like NuGet.

Package handling

Common pitfalls

- Large number of libraries
- Naming issues
- Version mismatch
- Dependency issues

How to avoid them

- Follow the SRP practice and you will handle large number of libraries with ease
- Try to categorize your package names
  - *Solution.Project.Functionality.Name*
- Properly define the dependency package version
- Use developmentDependency if needed

Readings

- Architectural Patterns and Styles
  - https://msdn.microsoft.com/en-us/library/ee658117.aspx
- Layered Application Guidelines
  - https://msdn.microsoft.com/en-us/library/ee658109.aspx
- The Onion Architecture
  - http://jeffreypalermo.com/blog/the-onion-architecture-part-1/
- Hexagonal architecture
  - http://alistair.cockburn.us/Hexagonal+architecture
- The Clean Architecture
  - http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html
- Layers, Onions, Ports, Adapters: it's all the same
  - http://blog.ploeh.dk/2013/12/03/layers-onions-ports-adapters-its-all-the-same/
- Onion-izing your multi tier architecture
  - http://www.incredible-web.com/blog/the-onion-architecture/

Next Mono.Tracks on CodeCamp

- 13.11. - Multi-layered Architectures - Workshop
- 16.12. - NodeJS Introduction & Workshop

We are hiring

Facebook: mono.software

Twitter: @monosoftware

E-mail: careers@mono.software

Thank you!

Questions?

mono.track