Software Architecture

THIS DOCUMENT COVERS

- Introduction
- S.O.L.I.D.

Introduction

Architecture consists broadly of two parts. The first part takes a set of requirements and creates from them a high-level conceptual architecture. The high-level architecture is then decomposed into smaller subsystems and components. The architecture defines the interfaces, responsibilities and interactions of these components. The architect will specify logical layers and services and the interfaces between them. Non-functional requirements influence the choice of technology. The second part involves making the hard decisions early on in a project which are hard to change later. For example, deciding to use KDB and switching later to Sybase is not an easy change. In order to produce an architectural solution, the software architect considers

- Use cases – how the application will be used
- Functional and non-functional requirements
- Technology requirements
- How the application will be deployed and managed

The architect must decide on the most appropriate application type, architectural style(s), technologies and deployment methods. Each draft should be tested against scenarios, known constraints, and quality attributes. The process of creating a candidate architecture might be described as

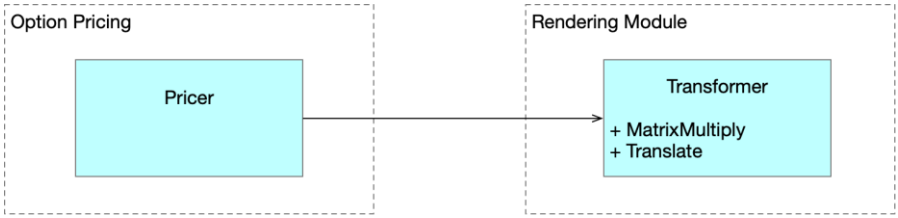
	Description
1. Goal	A goal might be to create a prototype or a complex systems architecture
2. Key Scenarios and Use Cases	These will be used to evaluate candidate architecture
3. Application Type	Are we building a thick client, web application etc.
4. Deployment Constraints.	Corporate policy might restrict the protocols or network topology on which the application will be deployed. Attributes such as security and reliability will also impact the deployment
5. Architectural Styles	Come up with architectural styles such as message bus, SOA etc.
6. Decide on technologies	The choice of technology will be influenced by the application type, deployment constraints and architectural styles.
7. Whiteboard the architecture	If you cant it is sure you don't understand it
8. Quality attributes	Consider Usability, performance, reliability, security
9. Cross-cutting concerns	Authentication and authorization, caching, configuration, Logging

Design Principles

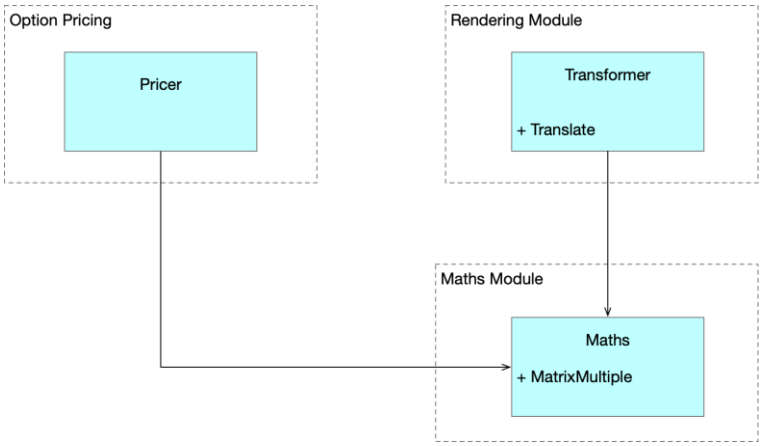
S.O.L.I.D.

Single Responsibility Principle

A single class should have only one responsibility. If a class has multiple responsibilities, then changes to one responsibility can break the logic of the other responsibilities thus increasing brittleness. In cases where a module exposes a class that provides multiple responsibilities, clients of that module may be forced to recompile and redeploy when a responsibility on which they have no logical dependency changes.



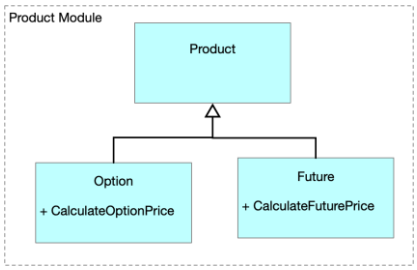
We can fix this as follows



Mixing business rules and persistence is a classic example of breaking the Single Responsibility Principle. Patterns such as Façade, Proxy and DAO patterns can be used in such situations

Open Closed Principle

Types should be open for extension and closed for modification. Consider the following simple type hierarchy and a piece of client code which breaks the Open/Closed Principle

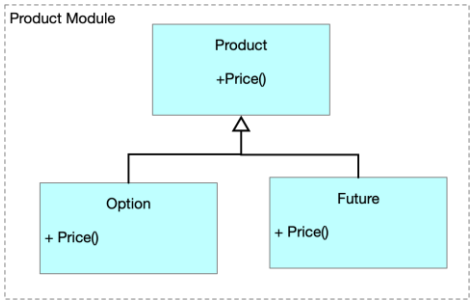



```
double PriceProducts(IEnumerable<Product> products)
{
    double total = 0.0;

    foreach (var product in products)
    {
                if (product is Option)
                total += ((Option)product).PriceOption();
          if (product is VarSwap)
                total += ((VarSwap)product).PriceVarswap();
    }

    return total;
}
```

The use of abstractions and polymorphism enable us to modify our hierarchy and `PriceProducts` method to obey the open closed principle.




```
double PriceProducts(IEnumerable<Product> products)
{
    double total = 0.0;

    foreach (var product in products)
          total += product.Price();

    return total;
```

Liskov Substitution Principle

Replacing objects of a base-class with object of a subclass should not impact the correctness of the program.

Interface Segregation Principle

Multiple interfaces are better than one big interface

Dependency Inversion Principle

Code against abstractions rather than concrete implementations

Questions – Design Principles

What is SOLID?

	Column Header
Single Responsibility	A class should have one responsibility
Open Closed Principle	Open for extension, closed for modification
Liskov Substitution Principle	Replacing objects with sub-type instances should not break the code
Interface Segregation	Multiple specific interfaces are better than one large one
Dependency Inversion	One should depend on abstractions, not concrete types

Why is SRP important?

If a class has multiple responsibilities, then changes to one responsibility can break the logic of the other responsibilities thus increasing brittleness.

Architectural Styles

There are many architectural styles. The following lists some.

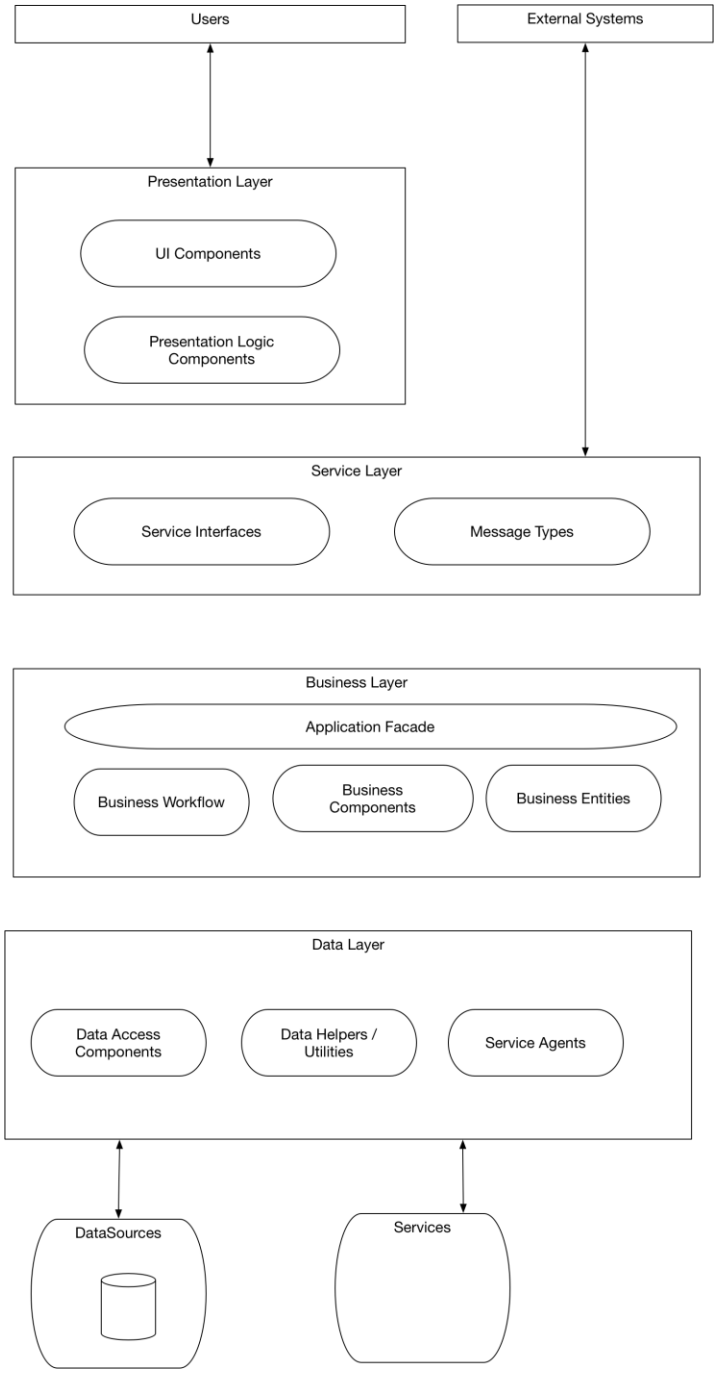
Type	Description
Client-Server	Presentation and data access
Multi-layer	Presentation, Business and Data Layers. Application layer can exist between presentation and business
Multi-tier	Use Tiers instead of layers
Domain Model	Layered architecture that uses presentation layer, application layer, domain layer and infrastructure layer.
CQRS	A Layered architecture with two columns; one for commands and one for queries. Each column can have its own architecture
EventSourcing	Evolved from CQRS with focus on events rather than data
Monolithic	Enough said
Microservices	https://martinfowler.com/articles/microservices.html
Hexagonal	
Service Oriented Architecture	
Partitioning	

Layered Architecture

Layers determine division of components into logical groups. Layers say nothing about where components are deployed. Layers can be on the same physical tier or on different tiers. Tiers describe the physical location of component on computers, networks and servers. The architect should consider whether layers exist purely to provide logical separation or to also provide physical separation. Reasons to use physical separation are shared business logic, scalability or security. The components within each layer can be further decomposed into sub layers. One set of layers might be

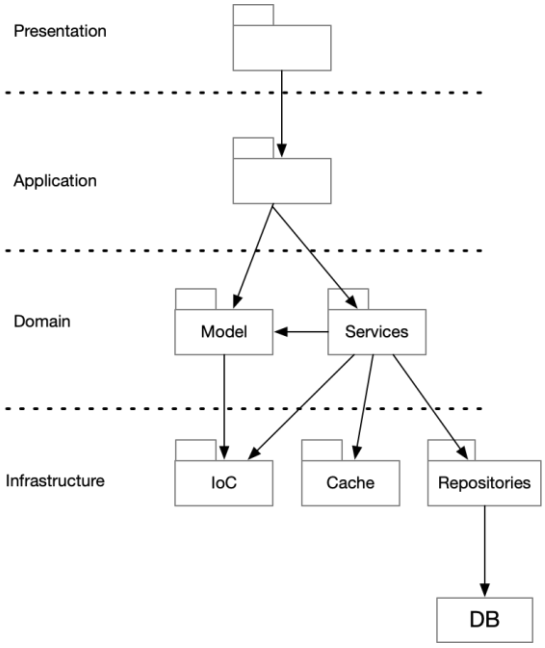
- Presentation
- Business Layer
- Data Layer
- Services Layer

Crossing layers incurs a cost, especially if physically located on different tiers. The benefits of increased scalability, maintainability, extensibility and flexibility often outweighs the performance costs. Where layers are in the same process we must take care to use encapsulation and loose coupling. Where layers are on different tiers, we must take into consideration latency.



The interface between layers is loosely coupled and explicitly. In a strict system each layer can only access the layer directly beneath it. In more relaxed implementations layers can communicate with any other layers lower in the stack. The different layers can be deployed on different tiers if necessary.

A modern layered architecture can look as follows



Presentation layer

Anything that populates the presentation layer is known as the view model. The input model consists of actions originating in the UI that cause back end actions.

Application layer

Helps to separate presentation and domain. Takes care of use cases by executing and coordinating work done in lower layers of the stack.

Domain layer

All the business logic. An Entity should provide data and behaviour. In addition to entities the domain model provides domain services. Domain services operate on multiple entities to achieve some task.

Infrastructure layer

Data access, IoC containers, logging, caching

Microservices Architecture

Each service is responsible for a specific capability. Data ownership is decentralized. Decoupling

Domain Driven Design

Domain Driven Design is a software development methodology that describes a set of practices for processing the requirements from business domains. The requirements processing leads to the identification of subdomains each of which has its own ubiquitous language. Each subdomain in the problem space maps to a bounded context in the solution space. It is worth emphasising that the domain defines the problem to be solved and the bounded contexts are the solution to that problem. At a more granular level subdomains are subsets of the domain and bounded contexts are subsets of the solution.

A context map shows the relationships between bounded contexts. The two contexts in each relationship are often marked as u and d. U is the upstream context and d is the downstream context. U can force change on d, but d cannot force change on u. Context mapping is high level design and doesn't produce code or other artefacts.

The second part takes the bounded contexts and selects for each an appropriate architecture. The Evans book recommends a layered architecture and a domain model. The domain model or entity model consists of an object model and a set of domain service classes.

Questions DDD

What are the disadvantages of multi-tier versus multi-layer architecture?

Increases latency and decreased performance.

What does the repository belong?

The interface belongs in the domain layer and the implementation belongs in the infrastructure layer.

Architectural Patterns

Transaction Script

Each user request originating in the presentation layer is processed by a single procedure known as a transaction script. Typically, a transaction script executes as a single database transaction.

TS is suited to simple use cases. Ideally the logic is unlikely to change much. Might be used for a web portal onto an existing back end.

In more complex cases can lead to code duplication without careful refactoring.

Domain Model

The domain model is a layered architecture where the layers are presentation, application, domain and infrastructure. The domain layer consists of entities and value objects and their state and behaviour. Entities have identity whereas value objects do not. In order to make a more manageable design entities can be grouped together into aggregates.

DOMAIN LAYER

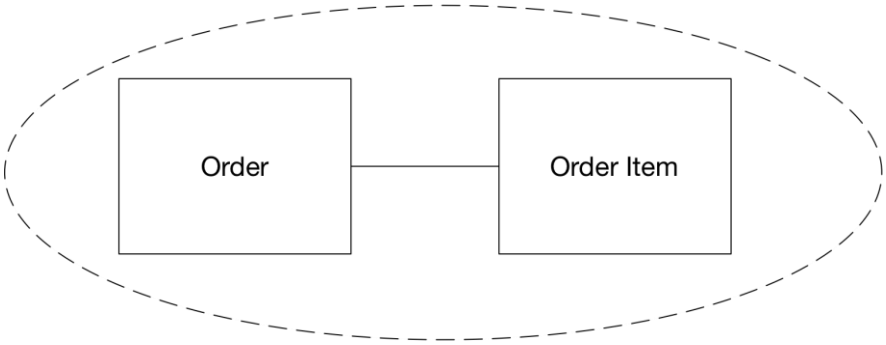
A domain layer focusses on behaviour rather than data. It consists of entities, their behaviours and the relationships between them. Each entity represents important entity in the domain. In addition to entities we have value objects and services. Services are classes utilise multiple entities and are known as Entity Services. The objects in the domain layer are grouped into modules where each module is a .NET namespace.

- One bounded context maps to one domain model
- One bounded context can consist of multiple modules
- One domain model can consist of multiple modules
- Each module is a .NET namespace.

An Entity has identity. Two objects with the same attributes are not necessarily the same. Objects that need and Id are entities in the domain layer. In contrast a value object is just data grouped together.

Aggregates

Where entities are often being used together, we can treat them as being logically grouped together into an aggregate. An aggregate has a root. Consider an order and its order items. The order items have identity but most clients won't need direct access to them. The order becomes the root of an aggregate. Entities should only below in one aggregate. Value objects can be in multiple different aggregates



An aggregates root entity can reference its own child entities or the root entities of other aggregates. A child entity can only access root entities of other aggregates. The root of the aggregate is visible to other objects in the model. Aggregate child entities have identity and life but cannot be directly referenced outside the aggregate. If a root provides reference to child items the receiver should treat them as transient.

Services

Implement logic that utilise multiple entities and aggregates to carry out a business action. The actions of domain services are defined in the requirements. A repository is a domain service

Repositories

There should be one repository per aggregate root.

Domain model classes should not have direct knowledge of how to persist and load objects from the persistence store.

POCO means Plain Old CLR Object. Persistence is not a domain model responsibility. The domain model will define a repository interface and the infrastructure layer will implement it. The infrastructure level implementation can be injected in using Inversion of Control.

Risk and Pricing Solutions

Transaction Script

Lazy Load

Query Object

Questions Architectural patterns

In a domain model what it the difference between an entity and a value object?

An entity has an ID that identifies it through its life cycle. A value object is just an aggregation of data.

What is the difference between domain logic and application logic?

The application layer contains the implementation of use cases.

How are queries persisted?

Via components known as repositories.

What are the two kinds of consistency?

Transactional and Eventual

What are the differences?

Transactional consistency is guaranteed after every transaction

Eventual is guaranteed at some point but may not be guaranteed after every transaction

What are aggregates?

Logical grouping of entities.

What is the benefit

Increases abstraction. Enables one to work with fewer, coarser objects.

Design Patterns

Command

Encapsulate a request as an object.

This pattern enables us to queue requests on a thread pool and have user interface components carry out different actions in response to clicks without knowing what those actions are.

Questions – Design Patterns

Which design pattern encapsulates a request as an object?

Command

What are the uses of the command pattern?

Queue requests on a thread pool

Allow user interface components to carry out different actions in response to gestures without actually knowing what those actions are

Questions - General

When use multiple tiers?

Increase scalability. If certain components can be remoted and duplicated the system will have increased scalability.

Why avoid tiers?

Cross process calls on the same machine are 100 times slower than calls within a single process. Calls that must cross networks to reach endpoints are slower still.

Why is CRUD?

Create, Read, Update and Delete are the four basic operations of a persistence store.

What are cross cutting concerns?

Persistance, logging, caching security

Transactions

In a monolithic system with a single database ensuring system consistent outcomes when operations involve multiple disparate entities is simple. The database provides support for ACID transactions that can be started, committed and rolled back.

In a microservices architecture ensuring consistent outcomes is more complex. Rather than joins on a single database we have calls to multiple services. If any of the services fail the application as a whole can be left in an inconsistent state. Consistency must be dealt with at the application level rather than at the level of individual services.

Two phase commit and transaction manager

A simple solution is to use a transaction manager and a 2PC protocol. Operations on multiple resources are performed in two phases: prepare and commit. Resources are locked introducing contention and the risk of deadlock in long running operations. This can reduce throughput (Pessimistic?)

Event Based

Services in a microservices architecture typically use events for communication. Events lead to decoupling. Event based approaches to synchronization tend to focus on eventual consistency where the eventual consistent state is achieved via a number of independent local transactions.

Another alternative is to use events, choreography and optimistic approach. Each service listens and publishes events to know what to do. Services can also emit failure events which enable rollback. If a service goes down it can process any backlog of events when it comes back up. Each individual piece only knows what events it consumes and publishes decoupling services from each other.

Saga pattern.

Saga is a series of co-ordinated local transactions. Successful previous steps trigger subsequent steps. This prevents using locking in long lived transaction which reduces availability.

A developer must design the system such that the system is eventually consistent even in the case where individual transactions fail.

One approach is to use events

In microservices the availability is the product all microservices involves in any given operation. The more services we add the more chance of failure and lower reliability.

Eventual Consistency.

Misc

What are two times we can cache?

On demand first time it is retrieved

Seeding when an application starts.

What is the problem with seeding?

Impose high load on original data store when the application starts up

How should one determine whether to use seeding or on demand loading?

Performance testing and usage analysis

What kind of data lends itself well to caching?

Immutable data or data that changes infrequently.

Give examples

Reference info such as product data

Immutability

There is an argument that mutable objects are harder to understand because when we invoke a method it is not always clear what internal state changes.

Immutable types make parallel execution easy because we don’t have to worry about concurrent access to an object’s fields.