Solution- Architecture in ASP.Net Core

Ahmed Othman Mohammed
Department of Computer Science
Faculty of Science
Minia University, El-Minia, Egypt

Moheb R. Girgis
Department of Computer Science
Faculty of Science
Minia University, El-Minia, Egypt

ABSTRACT

In software development process, the development team should focus on business development solutions not on technical issues. This means that in order to increase the developers' productivity, they must be assisted to focus only on how to implement business issues. This can be achieved by building a good solution in maintainable and reliable architecture to avoid any critical problems. This paper presents proposed solution architecture in ASP.Net Core that can handle the business logic of multiple domains for the same organization in one technical solution. This solution uses multiple design patterns such as Repository, Unit of Work, and Inversion of Controls, to solve common problems that face developers and help them to increase business productivity. Also, this solution provides an easy way to integrate and facilitate the communication between domains. This solution leverages some characteristics from Microservices architecture and Domain-Driven Design approach. Also, it includes some features to fit the problem solution, and handles some common technical issues such as authentication, authorization, caching, exception, tracing, etc. The paper also presents a simple case study to show how the proposed solution architecture enables multiple domains to interact with each other to serve organization business needs.

General Terms

Solution Architecture, Software Development.

Keywords

Solution Architecture, Architecture Patterns, Design Patterns, ASP.NET Core, Domain-Driven Design.

1. INTRODUCTION

There are common behaviors which the developer needs to write to start in working or in implementing the business. In real world, from business perspective, the main concern of a business owner is achieving the client requirements not the type of technical solutions that the developer has been used. The main problem which faces the developer is the type of architecture that he/she will use to achieve business requirements. This means that the developer needs to consider some factors like Memory and CPU utilities, Business rules management, Data storage and other factors. Unfortunately, good solution architecture is very expensive, and the developer should be familiar with.

The solution architecture should guide the developers and restrict their mistakes, which may cause application crash. Also, it should be simple in working, and should handle more assets like Dependency Injection, Authorization, Authentication, Cashing, Database migrations, Security, Mapping, Auditing, Rule base engine, Unit Testing … etc., which are very expensive.

ASP.NET Core [1], the next generation of ASP.NET, is a cross-platform, high-performance, open-source framework, developed by Microsoft, for building modern, cloud-based, Internet-connected applications. It is a modular framework that runs on Windows or any operating system (OS).

This paper presents proposed solution architecture in ASP.Net Core (SAIA) that utilizes the power of design patterns such as Repository, Unit of Work, and IoC (Inversion of Controls) to solve common problems that face developers and help them to increase business productivity. This solution provides a best practice at each system component then packages all of them in one solution architecture that will help developers and put them on the right way to achieve their goals.

The paper is organized as follows: Section 2 presents some related work that use similar concept but different implementation. Section 3 presents the problem definition and solution. Section 4 describes the high-level design of the proposed solution architecture, and its benefits, then gives summery of this solution. Section 5 presents a case study. Finally, Section 6 presents experiments and results, and Section 7 presents the conclusion of the presented work.

2. RELATED WORK

There are some solutions available in the market, which have been implemented for the problem under consideration. This section presents some examples of such solutions.

2.1 Clean Architecture

Clean Architecture [2] is a pattern, practice and principle, which implements some concepts from Domain-Driven Design (DDD). This solution is considered domain centric architecture, which divides the solution architecture into several components as shown in Figure 1. These components are:

- **Presentation Layer:** represents the user interface.
- **Application Layer**: is a core layer that depends on the Domain Layer.
- **Domain Layer**: is domain context with no dependency.
- **Persistence Layer**: handles database logic, depends on the Application Layer, and implements IoC concept.
- **Infrastructure Layer**: depends on the Application Layer and handles services which are related to OS and external services.
- **Common Layer:** considered as cross cutting layer.

This solution architecture solves the problem under consideration but the maintainability and readability of the project will take more time because this solution handles all domains as a parent domain and sub-domains, which will make the project complex and hard to maintain, and increase the dependency [2]. This kind of architecture is used for small scope projects that contain some core user stories need to focus on. [3]

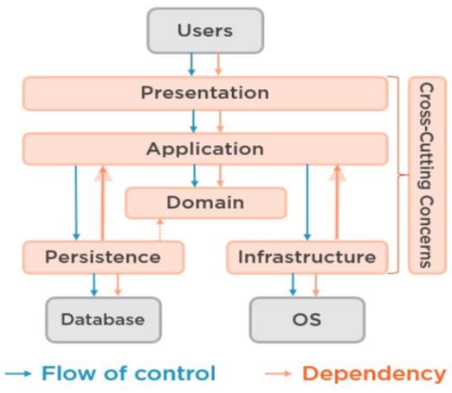

Fig 1: The Clean Architecture Components

2.2 Microsoft Unit of Work Architecture

The Repository and Unit of Work patterns [4] 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 isolate the application from changes in the data store and can facilitate automated unit testing or Test-Driven Development (TDD). This solution depends on TDD not DDD. Some architects consider it as a database centric architecture. It implements N-Tiers kind of solution architectures, which divide the solution into several parts as shown in Figure 2.

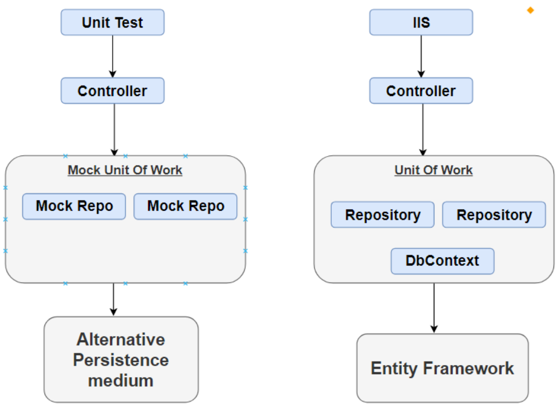

Fig 2: Microsoft Unit of Work Solution Architecture

2.3 ASP.Net Boilerplate

ASP.NET Boilerplate (ABP) [5] is an open source and well-documented application framework. It's not just a framework, it also provides a strong architectural model based on DDD, with all the best practices in mind. ASP.Net Boilerplate framework provides many features such as Dependency Injection, Repository pattern, Authorization, Validation, Audit Logging, Unit of Work, Exception Handling, Logging, Localization, Auto Mapping, Dynamic API Layer, and Dynamic JavaScript AJAX Proxy. But, as the Clean Architecture, it deals with the problem as a main domain and subdomains. ABP is good solution architecture but it needs a huge learning curve from developers to understand. Figure 3 shows the components of ABP architecture.

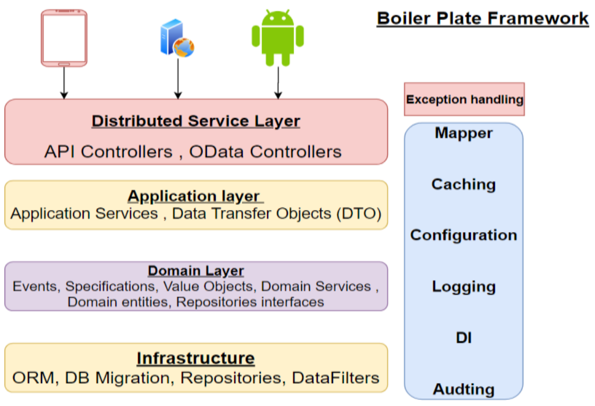
Boiler Plate Framework
Distributed Service Layer
API Controllers , OData Controllers
Application layer
Application Services , Data Transfer Objects (DTO)
Domain Layer
Events, Specifications, Value Objects, Domain Services , Domain entities, Repositories interfaces
Infrastructure
ORM, DB Migration, Repositories, DataFilters
Exception handling
Mapper
Caching
Configuration
Logging
DI
Auditing

Fig 3: Boilerplate Architecture Components

3. PROBLEM DEFINITION AND SOLUTION

Assume that there is a big organization that needs management system, which has a large-scale business logic divided into multiple domains, such as Accounts, Accountant, Sales, etc. The organization needs one solution that handles all business logic across these domains in the same project.

The problem here is so difficult because one solution is needed for all organization logic that should be maintainable, readable, extensible, and provide ability of integration and communication between domains.

Microservice architecture integrated with DDD can solve this problem but not in the same solution, which means that each domain will be separated from other domains and this will consume more time for integration and communication between these domains.

The solution architectures available in the market are either very costly to license them or provide difficult solution. So, this paper presents proposed solution architecture in ASP.Net Core (SAIA) to solve the above-mentioned problems by integrating some design patterns to develop a good model that helps developers to implement and focus only on business requirements. SAIA uses best practices from each field and tried to eliminate the mentioned problems for helping developers to be more productive. The proposed solution provides a simple way to do all required tasks such that multiple teams can collaborate with each other smoothly. This solution will cover common functionalities such as Authentication, Authorization, Security, User Management, Service Locator, Rule-base engine, Notification engine, Mapping, Exception Handling, Exception Logging, Email Service, SMS Service, Caching, Project Template, and Localization.

4. THE PROPOSED SOLUTION ARCHITECTURE (SAIA)

The proposed solution architecture SAIA depends on some design patterns and architecture patterns, such as DDD, Microservices, Repository Pattern, Unit of Work, IoC, Dependency Injection, Service Locator Pattern, Factory.

4.1 SAIA Components

Figure 4 depicts the components of the proposed solution architecture. These components are described in the following subsections.

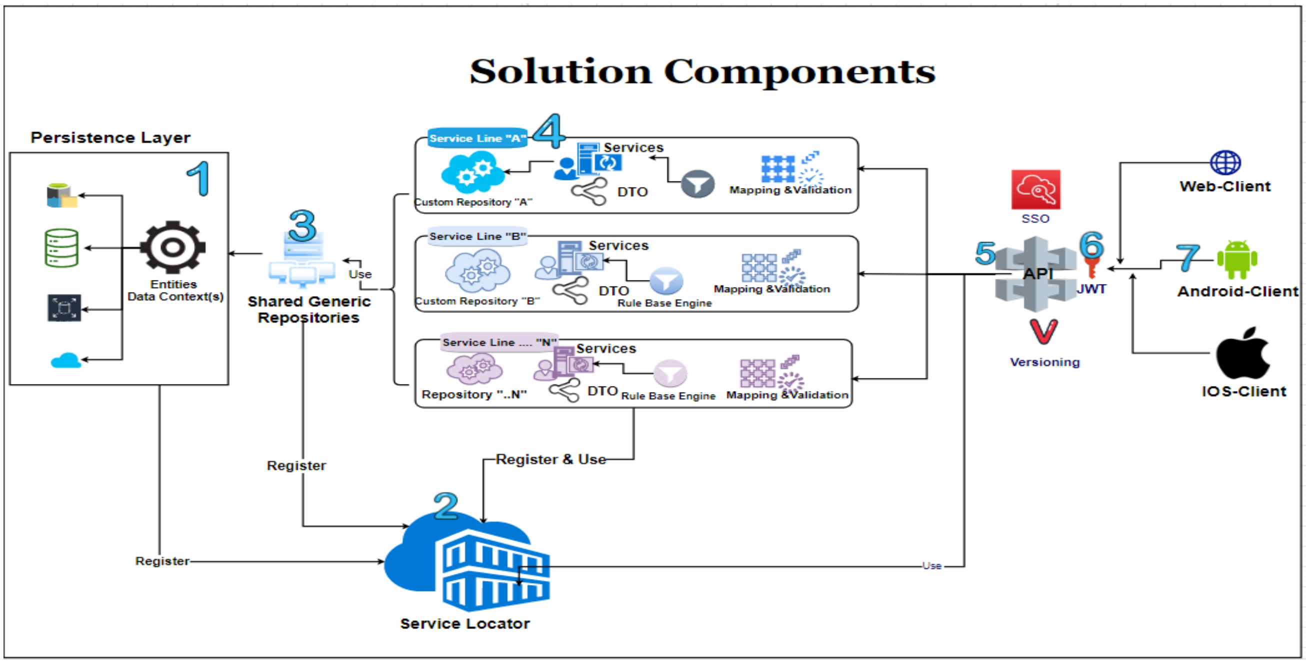

Fig 4: The components of the proposed solution architecture (SAIA)

4.1.1 *Infrastructure Layer (Data Store Layer)*

Data Store Layer is responsible for storing and retrieving data. It can be SQL-Server, Oracle or any other data store type. This layer is divided into five key elements:

- **Database Management System**, which will be SQL Server.
- **Database Entities**, which play as object relational mapping (ORM) for database tables or database collections in case of using No-SQL database type.
- **ORM Engine**, like Entity Framework in our case, which will be responsible for storing and retrieving data from data source.
- **Database Context Object**, which will be managed by Entity Framework to store and retrieve data.
- **Data Filters**.

4.1.2 *Service Locator*

The Service Locator is the most important object in our solution architecture. This layer is responsible for resolving any required dependency. This requires each domain to register its own objects in it and decide whether they will be shared between other domains or not. If any service in our application needs a dependency, it doesn't instantiate the dependency explicitly from the respective class; instead it will get it through the Service Locator. This will enable us to avoid the overhead of creating unnecessary objects. Figure 5 shows how the Service Locator resolves needed dependencies.

The pattern seeks to establish a level of abstraction via a public interface and to remove dependencies on components by supplying 'plug-in' architecture, i.e. individual components are tied together by the architecture rather than being linked together by themselves.

The main difference between shared services and local services is that shared services need extra implementation. In order to register any service as a shared domain service, developer should add its interface to the Service Locator layer to be accessible by other domains.

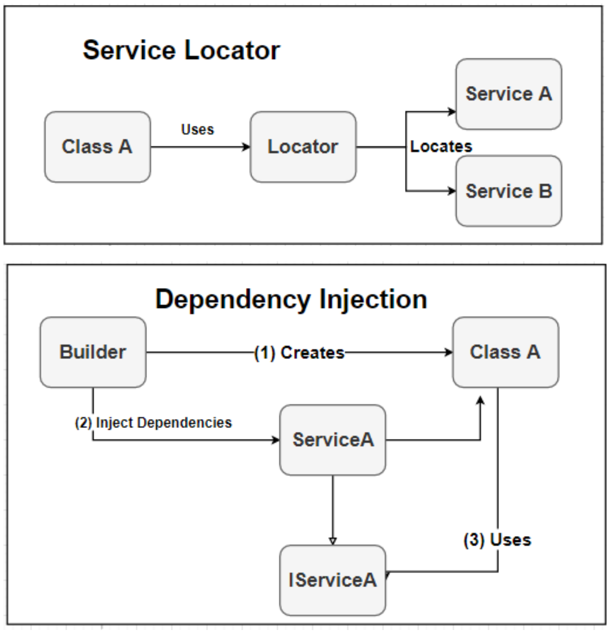

Fig 5: Service Locator Pattern and Dependency Injection

As shown in Figure 6, if any domain needs to register a domain service as a shared service, it must share implementation contract at the Service Locator, and data transfer objects should be implemented in a shared layer. By this way SAIA can isolate each domain and mark only services that team needs to be shared as shared services to interact with other domains. Service Locator layer plays a big role in this case as it restricts sharing only for domain services that can be shared not core objects or repositories.

4.1.3 *Generic Repositories*

The third component of the proposed solution architecture is a generic repository for database root aggregate entities. The generic repository is a class that can be used generically with any entity of type DB entity and implements the main methods that will be used to store and retrieve data.

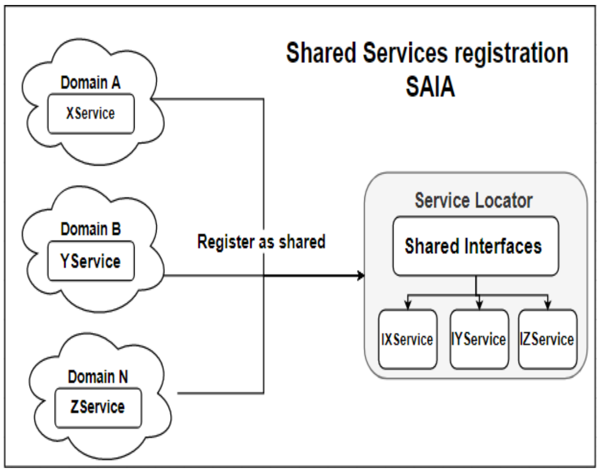

Fig 6: Shared Services Registration

A generic repository will be used only on each bounded context (domain) to get/restore data from database generically.

4.1.4 Bounded Contexts (Domains)

This part will be the first part of isolation; SAIA will be divided into multiple domains that collaborate with each other using the Service Locator.

Each domain will follow the Onion Architecture [6] implemented by DDD [7], which consists of some layers and main Core layer at the center, as shown in Figure 7. It is similar to classic layer architecture, except that Onion Architecture and the Core part of domain can't be dependent on any other part. This means that the core elements of our domain model should act in isolation from each other.

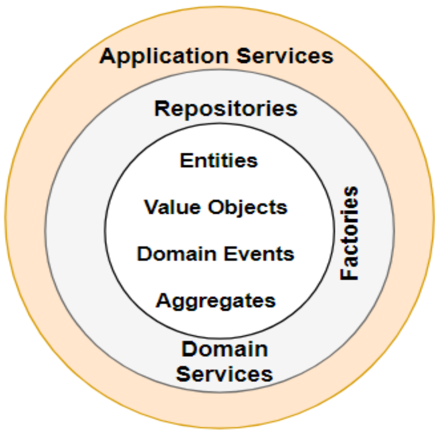

Fig 7: The Onion Architecture [6]

The Repository Layer will be responsible for dealing with the generic repository and map database entities into domain entity using Auto mapper and fluent APIs. Each domain in our solution will be responsible for registering its objects on the Service Locator layer and mark these objects as either sharable objects or internal objects.

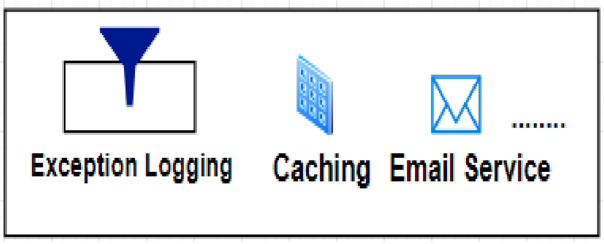

Fig 8: Examples of Cross-Cutting Services

4.1.5 Cross-Cutting service

Cross-Cutting services are consumed across all the solution layers. A cross-cutting service can be a web service or NuGet package or external repository, i.e. Notification Gateway, Rabbit MQ, Exceptions, etc. These services can be used as external services. Figure 8 shows some examples of these services.

4.1.6 API Gateway

API Gateway is considered the end point for any consumer that will call or deal with our service. It will present the consumer to all domains (services) and the container that will hold all references and support the required configuration. Also, it will be Restful API (Service Oriented Architecture) that can be consumed from Android, IOS, Angular or any other consumer. Our solution will use OAuth concept to secure this API; it will implement JSON Web Token (JWT) to authenticate the users and give them the authorization.

JWT [8] is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret key (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA techniques. [8]

As Figure 9 shows, using JWT token to authenticate clients and assign authorization to them each time they call the server. Although JWTs can also be encrypted to provide secrecy between parties, it is considered a signed token. Signed tokens can verify the integrity of the claims contained within it, while encrypted tokens hide those claims from other parties. When tokens are signed using public/private key pairs, the signature also certifies that only the party holding the private key is the one that signed it.

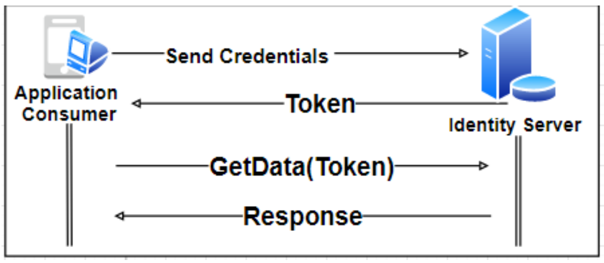

Fig 9: JWT Auth

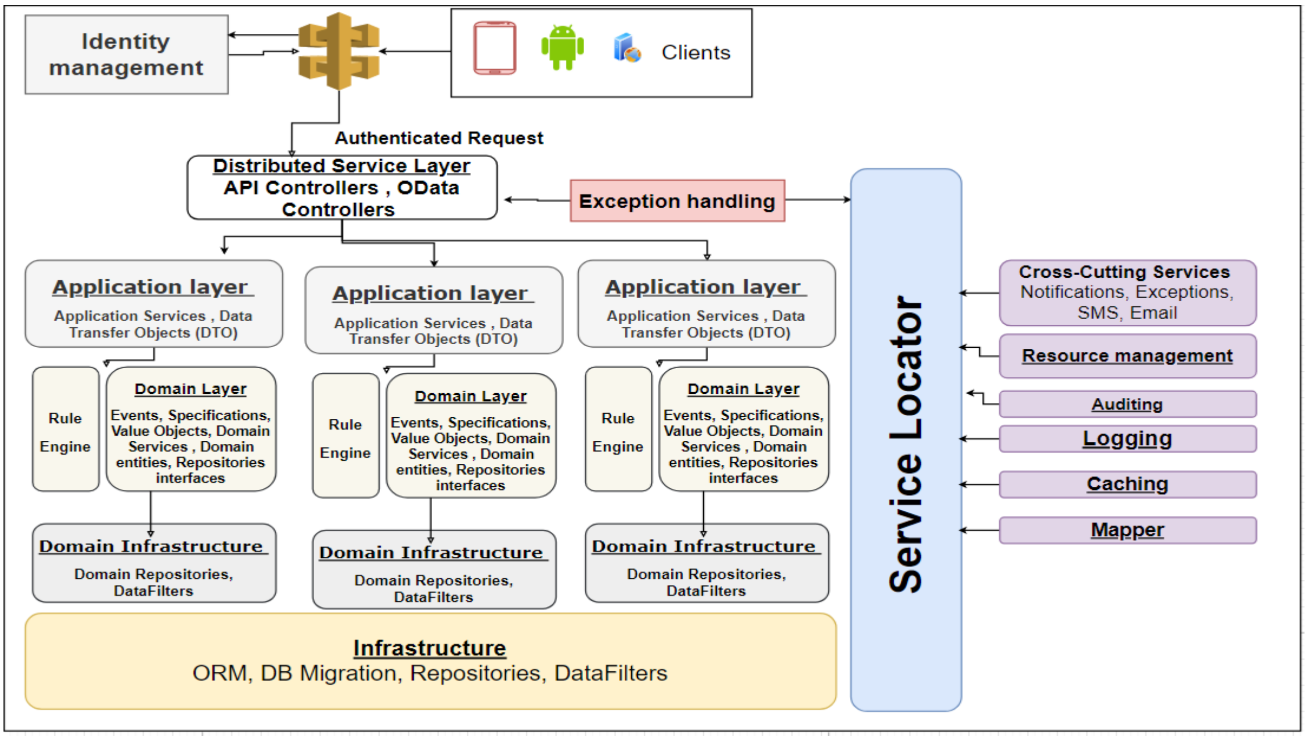
Identity management
Clients
Authenticated Request
Distributed Service Layer
API Controllers , OData Controllers
Exception handling
Application layer
Application Services , Data Transfer Objects (DTO)
Rule Engine
Domain Layer
Events, Specifications, Value Objects, Domain Services , Domain entities, Repositories interfaces
Domain Infrastructure
Domain Repositories, DataFilters
Infrastructure
ORM, DB Migration, Repositories, DataFilters
Service Locator
Cross-Cutting Services
Notifications, Exceptions, SMS, Email
Resource management
Auditing
Logging
Caching
Mapper

Fig 10: Detailed components of SAIA

4.2 Benefits of SAIA

Section 2 presented some alternatives to the proposed solution but each of these solutions fits a part of business. Also, all these solutions except Boilerplate Architecture don't fit the problem under consideration. But, Boilerplate architecture deals with a problem as a single domain with sub-domains not as multiple domains. SAIA divides the problem into multiple domains that can be dealt to with using the Service Locator.

The benefits of the proposed solution are:

- Like Microservice Architecture, SAIA can handle multiple domains in one solution architecture.
- Communications and interactions between these domains will be so simple and don't need extra work to do.
- SAIA can handle most common problems that face the developers, and accordingly increase their productivity.
- It guarantees memory optimization because no one can create objects anywhere (DI Benefits).
- It implements DDD concepts, which guarantee code maintainability and reusability.
- Identity management operation and external identity providers integration will be implemented in a single layer that will deal with API gateway, and applications will receive only authenticated requests.

4.3 Summary of SAIA

SAIA uses several design patterns that help developers to do best practice to solve their problems. Integration of these patterns with each other will help the developers to do only business tasks and increase the productivity. Figure 10 shows the detailed components of SAIA.

As Figure 10 shows, the proposed solution can be summarized in the following points:

- API gateway will be responsible for filtering and auditing and logging requests that will fire our application. This allows us to do pricing, auditing, and logging for our APIs.
- Identity Management Layer will be responsible for authenticating and authorizing request plus handling user management APIs.
- API Controllers and OData controllers hold solution domains.
- The problem will be divided into multiple domains, which will be integrated with each other through the Service Locator.
- The Service Locator Layer will be responsible for resolving all required objects across the project.
- Exception Handling Layer will handle any fired exception and log it.
- Each domain will contain the following layers:
  - Application Layer that will hold domain use cases plus data transfer objects and parameter validations.
  - Core Domain Layer that will contain domain entities, events, value objects, repositories interfaces and domain business.
  - Rule Engine Layer that will be injected into the Domain Layer and hold business rules.

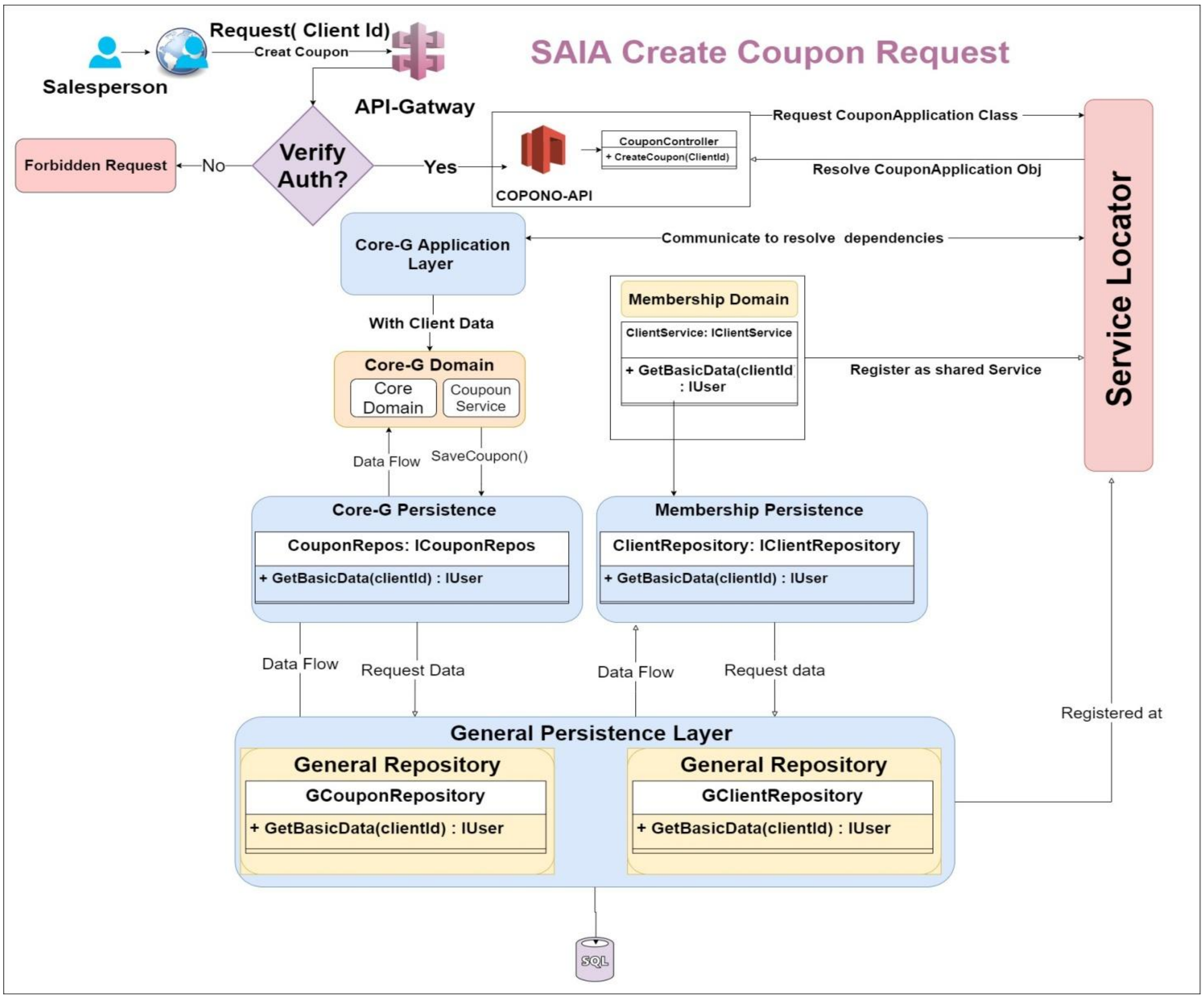

Fig 11: Create Coupon request

- Domain Infrastructure that will hold domain repositories and data filters.

- Cross Cutting Services will register themselves into the Service Locator Layer and used across all solution parts.

- Infrastructure Layer will be used by domains infrastructure and its responsibilities are ORM, Db Migrations, Database Repositories, and Data Filters

5. CASE STUDY

As mentioned in the previous section, SAIA helps developers to build an extensible solution for large scale projects that share same data and infrastructures. This section presents a small example that will show how SAIA enables multiple domains to interact with each other to serve organization business needs.

Let COPONO be a big organization that deals with generating discount business coupons for clients. COPONO should have a big database for its users and user's clients. Also let COPONO users be other organizations interested in different business models. COPONO organization needs technical solution that gathers its own business in one solution and serves its business needs. COPONO organization will be taken as the case study.

A small feature of COPONO business model will be taken as an example to implement it across system layers to demonstrate how SAIA implements the interactions between its components.

Assume that COPONO includes two domains: Membership domain and Cor-G domain. *Membership domain* will encapsulate the business of users and clients and will be responsible for supporting all services including storing and retrieving all data that are related to COPONO users and clients. Here, "users" refers to organizations that deal with COPONO and "clients" refers to end-users who deal with these organizations. *Core-G domain* will be responsible for creating discount coupons for clients and validating these coupons during their usage by client.

In this example, creating a coupon for a specific client will be demonstrated. Core-G domain needs information about this client. It will get this information from the Membership domain, which exposes *ClientService* class that encapsulates data about a client.

Figure 11 shows how SAIA deals with requests that require an interaction between multiple domains in our organization. SAIA forces each domain to implement its own logic and share it to other domains through the Service Locator layer, which plays a main role for aggregation and interaction between layers.

In this case, SAIA will take the following steps to handle requests and interactions between domains.

Step 1: Initiate request
As shown in Figure 11, the salesperson opens his/her web-portal and login to the system then selects the client that needs a coupon.

As shown in Figure 12, the receiver of salesperson request is SAIA's API-Gateway (Firewall), which will be public for all clients (Mobile, Web, and Desktop).

API-Gateway layer will be responsible for filtering the requests and doing statistics and dealing with identity server for checking authentication and authorization logic. If the request is authorized, the system will send it to COPONO API URL which will be responsible for handling the rest of the request. If the request is not authorized the API-Gateway will log it and return forbidden response to the salesperson to take an action, such as re-login or refresh the authorization token.

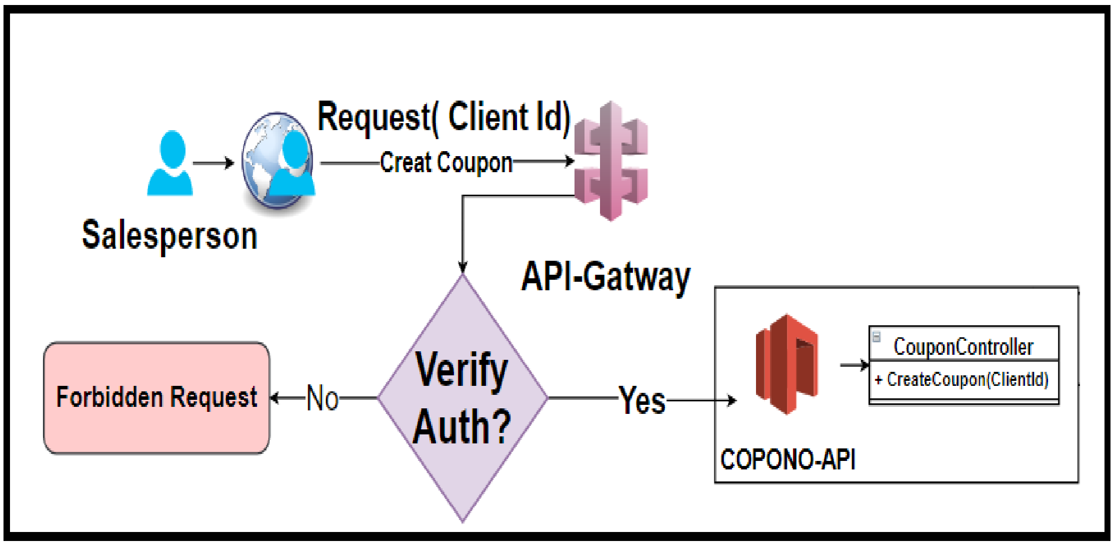

Fig 12: Request Initiation

Step 2: Request handling
COPONO API is a non-public API that can be accessed only from the API-Gateway, which make it highly secure. It receives the authorized salesperson request. By implementing this model, no one can reach our organization logic without visiting the API-Gateway, which permits only authorized requests to be processed.

As shown in Figure 13, COPONO-API includes **CouponController** class, which is the API controller that is responsible for handling coupon requests logic. The **CouponController** receives Create Coupon request from API-Gateway. **CouponController** has multiple dependencies such as **CouponApplicationService**, which is responsible for encapsulating the creation of coupons and other business services.

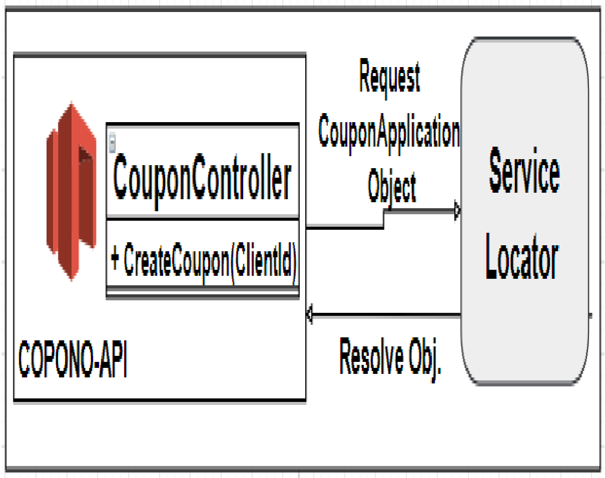

Fig 13: COPONO API

In SAIA, the Service Locator layer is responsible for creating dependencies to avoid any mistakes from non-experience developers, and this will be the only way for resolving dependencies. So, the **CouponController** will get its own dependencies from the Service Locator using the power of dependency injection.

CouponController uses property injection technique to get an object from **ICouponApplicationService**, and then call **CreateCoupon()** method which is responsible for consuming domain service to do the required logic.

Step 3: Get client data through Membership domain
The Membership domain, in this case, registers a **ClientService** as shared domain service to be used by other domains. **ClientService** encapsulates some logic for the client such as **GetBasicData()** which return some basic data about this client. In this case, using this method to get client data by passing a **ClientId** as a parameter and get Data transfer object (Dto) that encapsulates client's information.

Figure 14 describes how the data will be extracted using **GetBasicData**() on **ClientService** class. Also, the Service Locator is responsible for getting a repository object of type **IClientRepository** for **ClientService** class. **GetBasicData()** calls a repository to get the required data.

ClientRepository class here doesn't call a data source directly but calls **General Persistence** layer class, which is responsible for storing/retrieving data to/from data store. General Persistence layer contains general repository classes, which will be used by each domain repository or persistence. It includes **GClientRepository** class, generic repository for client, which is responsible for getting data from data store and returning it to Membership domain's repository class.

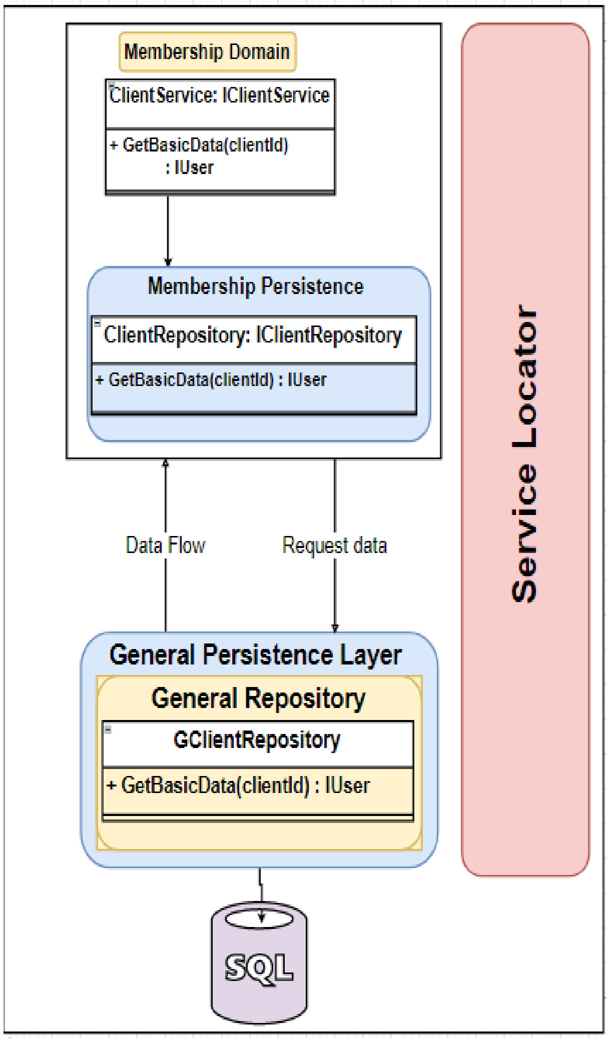
Membership Domain
ClientService: IClientService
+ GetBasicData(clientId) : IUser
Membership Persistence
ClientRepository: IClientRepository
+ GetBasicData(clientId) : IUser
Data Flow
Request data
General Persistence Layer
General Repository
GClientRepository
+ GetBasicData(clientId) : IUser
SQL
Service Locator

Fig 14: Membership Domain layer

Step 4: Create the coupon by the Core-G Domain

Core-G domain is responsible for creating a coupon for the specified client. **ApplicationService** layer in Core-G domain is responsible for interacting with other domains to get all required data.

In this case, **CouponApplicationService** class asks the Service Locator to get an instance of **ClientService**, which was implemented in the Membership domain, then the Service Locator resolves the dependency and returns the required **ClientService** object. Also, **CouponApplicationService** calls **GetBasicData()** method to get required client basic data.

As shown in Figure 15, **CouponApplicationService** class gets the client data from the Membership domain using **ClientService** object.

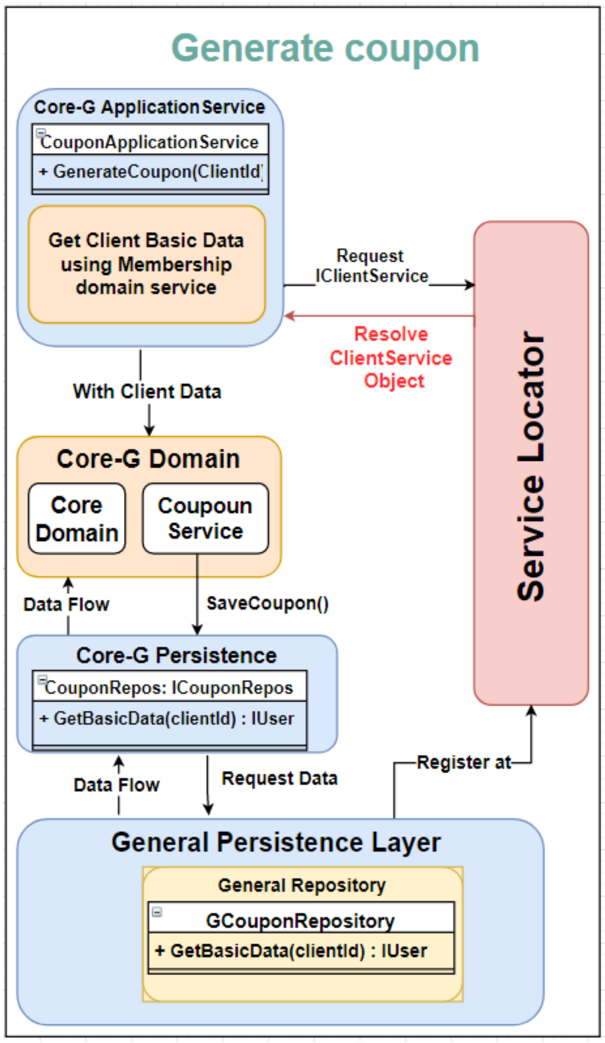
Generate coupon
Core-G Application Service
CouponApplicationService
+ GenerateCoupon(ClientId)
Get Client Basic Data using Membership domain service
Request IClientService
Resolve ClientService Object
With Client Data
Core-G Domain
Core Domain
Coupoun Service
Data Flow
SaveCoupon()
Core-G Persistence
CouponRepos: ICouponRepos
+ GetBasicData(clientId) : IUser
Data Flow
Request Data
Register at
General Persistence Layer
General Repository
GCouponRepository
+ GetBasicData(clientId) : IUser
Service Locator

Fig 15: Coupon Generation

Marking **ClientService** as a shared service allowed the **CouponApplicationService** class to communicate with **Membership domain** and get the required data.

By following this process, it can be said that "***Each business domain is responsible for deciding which services it will share"*** and this provides easy communication between domains.

Figure 15 shows the steps of generating coupons:

1. Application Service class (**CouponApplicationService**) will get the client basic data from **Membership domain.**

2. Pass these data to **CouponService** class, which is responsible for implementing the business of generating a coupon.

3. **CouponService** calls its domain repository object to save the new coupon and return its id. Then the repository object calls the **General Repository** class to save the new coupon.

4. **General Repository** will save the coupon data and return coupon id to above layers.

This model will guarantee isolation between domains, and each domain will be responsible for implementing its logic. Also, each domain can contain multiple bounded contexts which can communicate with each other in a simple way. The Service Locator layer will be responsible for setting up the communication channels between these domains and guarantee the way for resolving all dependencies.

API Gateway will be responsible for filtering the requests and pass only authorized requests to the application layer.

6. EXPERIMENTAL RESULTS

As the case study demonstrated, SAIA provides a development strategy that helps developers and increases their productivity. It also adds a value from business perspective by eliminating the time that developers take to handle common problems, such as authentication, authorization, exception logging, caching and sharing services.

After assessment of a problem that has two domains, the times required to handle and maintain common problems, with and without using SAIA, are shown in Table 1.

Table 1. A comparison between development time with and without using SAIA

Problem	Time Needed (in man-days)	
	with SAIA	**without SAIA**
Architecture Assets		
Project Template	0	10
Folder and Domain Structure	1	10
Maintaining and enforcing development rules and technical architecture rules	0	> 5 / Main change request
Implementing DDD	0	> 5/ each domain.
Integrating domains	0	2 / deploying and extracting services if microservices are used
Deployment	1	1/ Microservice
Technical Tools		
Authentication	0	>5
Authorization	1	>10
Caching	0	>5
Notifications (Firebase)	0	>20 for web and mobile platforms
Exception Handling and Logging	0	> 10

Table 1 shows part of SAIA features that help the developer to minimize the time taken to solve common problems. For example, Project template feature does not take time because SAIA provides a template that can be used directly, and Authorization problem takes only 1 man-day because SAIA provides the developer with a designed solution that handles authentication and authorization using OAuth and JWT technologies.

According to this evaluation, the business productivity chart at the first 8 development weeks will be as showm in Figure 16. SAIA increases business productivity from the first day of the development cycle because it handles all common technical problems that face the devloper. It also adds a great value for business team by increasing productivity; and for development team by organizing a solution into multiple domains and facilitating the communication between these domains.

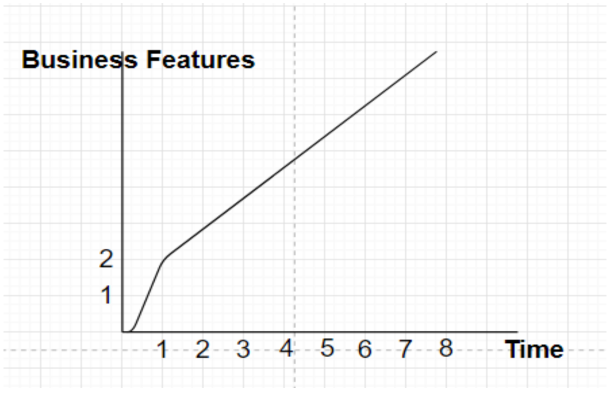

Fig 16: SAIA productivity chart

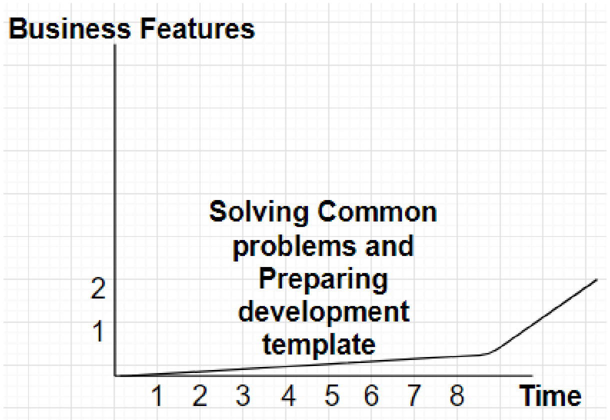

Fig 17 : Microservices (without SAIA) productivity chart

As shown in Figure 17, the used microservices architecture without SAIA does not cover common problems, which makes the developer reinvent the wheel. This will cause a delay in delivery and business acheievement.

7. CONCLUSION

This paper presented proposed solution architecture in ASP.Net Core, called SAIA, and described its components. It is a good architecture that will solve and handle the huge organizations business. It provides a simple way to do all required tasks such that multiple teams can collaborate with each other smoothly. It will handle multiple domains in one solution architecture with easy communications and interactions between domains. It guarantees memory optimization and implements DDD concepts which guarantees code maintainability and reusability. In addition, it provides secured API gateway.

Also, the paper presented a small case study that showed how SAIA enables multiple domains to interact with each other to serve organization business needs.

Finally, the paper presented comparisons between development time and team productivity with and without using the proposed SAIA solution architecture. The results indicated that using SAIA significantly reduces the development time, and increases the team productivity.

8. REFERENCES

[1] Roth D., Anderson R., and Luttin S. 2019. Introduction to ASP.NET Core. https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-3.0.

[2] Vegliach G. 2018. Clean Architecture by Uncle Bob: Summary and review. https://clevercoder.net/ 2018/09/08/clean-architecture-summary-review/.

[3] Clean-architecture-patterns-practices-principles, https:// app.pluralsight.com/library/courses/clean-architecture-patterns-practices-principles/table-of-contents, Last Accessed: 1/1/2019

[4] Dykstra T. 2013. Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application. 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.

[5] The ASP.NET Boilerplate. https://aspnetboilerplate.com /Pages/ Documents/Introduction, Last accessed 4/1/2020.

[6] Palermo J. 2013. Onion Architecture: Part 4 – After Four Years. https://jeffreypalermo.com/tag/onion-architecture/.

[7] Millett S. and Tune N. 2015. Patterns, Principles, and Practices of Domain-Driven Design, John Wiley & Sons, Inc.

[8] Peyrott S. E. 2016-2017. The JWT Handbook, Auth0 Inc. Version 0.12.0.