# Project

Software Architecture

# MEMBER

| | |
|---|---|
| Hà Hoàng Duy | 22011288 |
| Trần Thành Huy | 22003768 |
| Bùi Phúc Hậu | 22012201 |

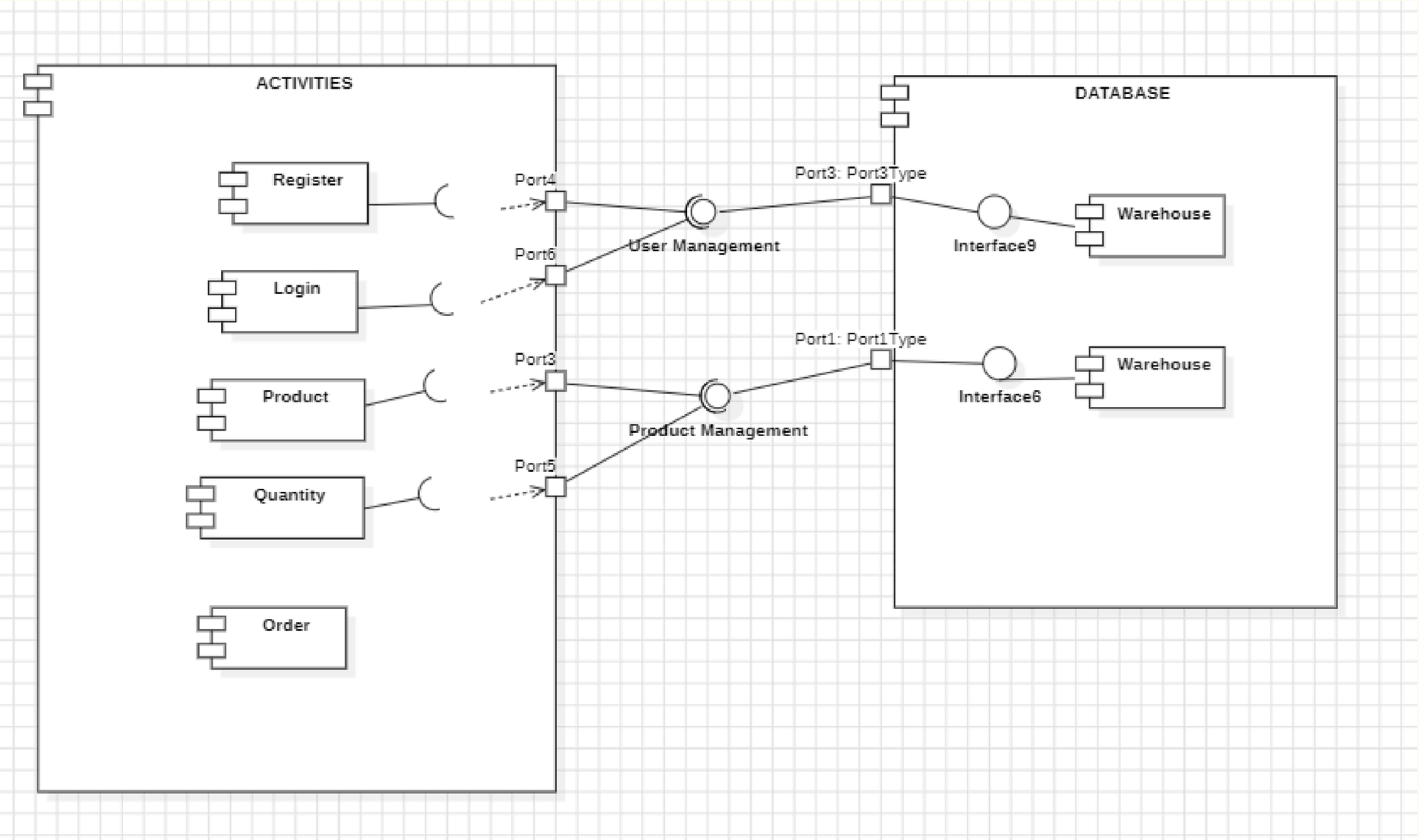
# Problem Statement

The Car Management System (CMS) is designed to sell and manage car to improve business and management efficiency.

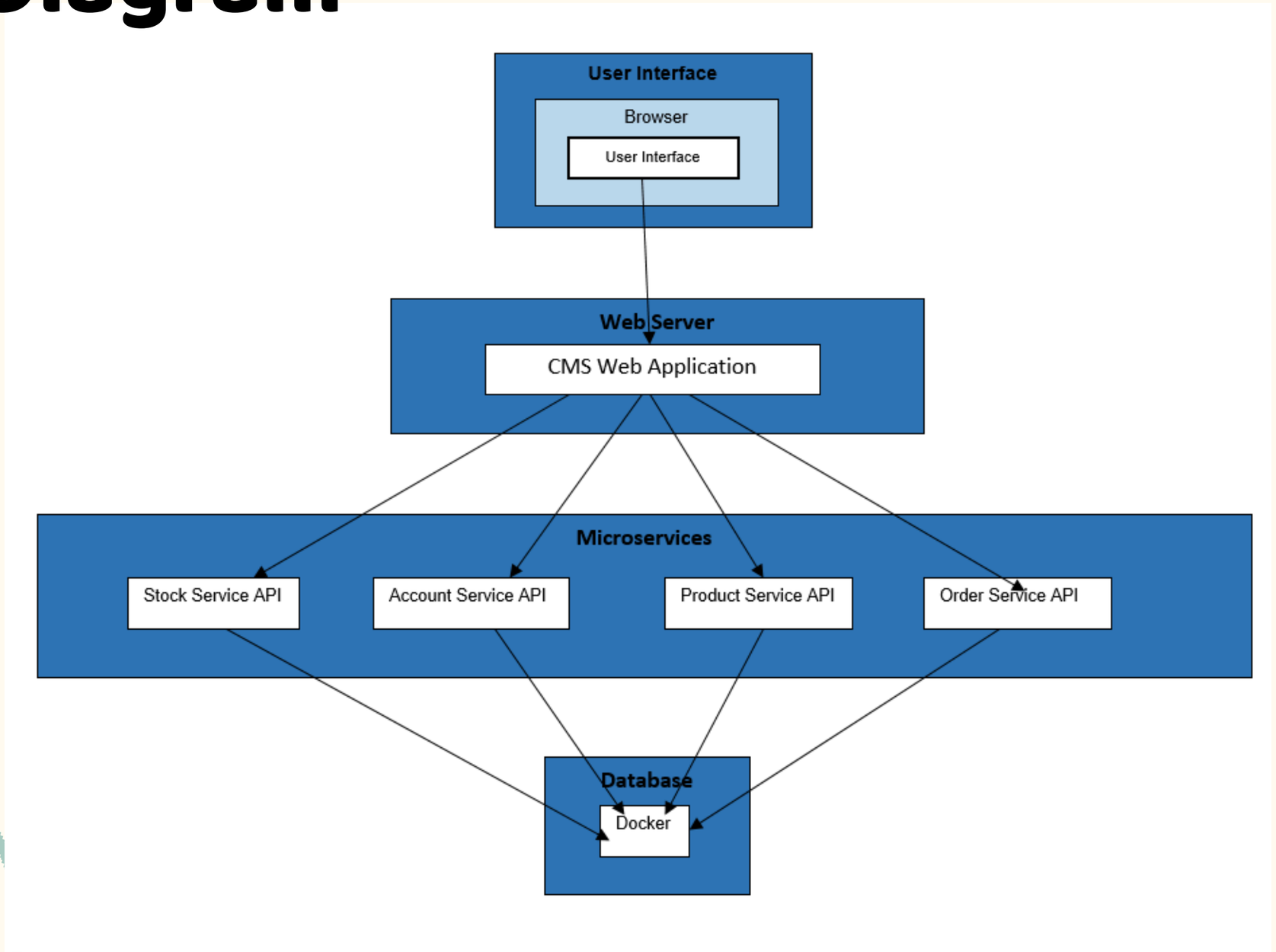
Car management and sales application with the following main functions

- Login
- Add products
- Inventory management
- Inventory management
- Sell

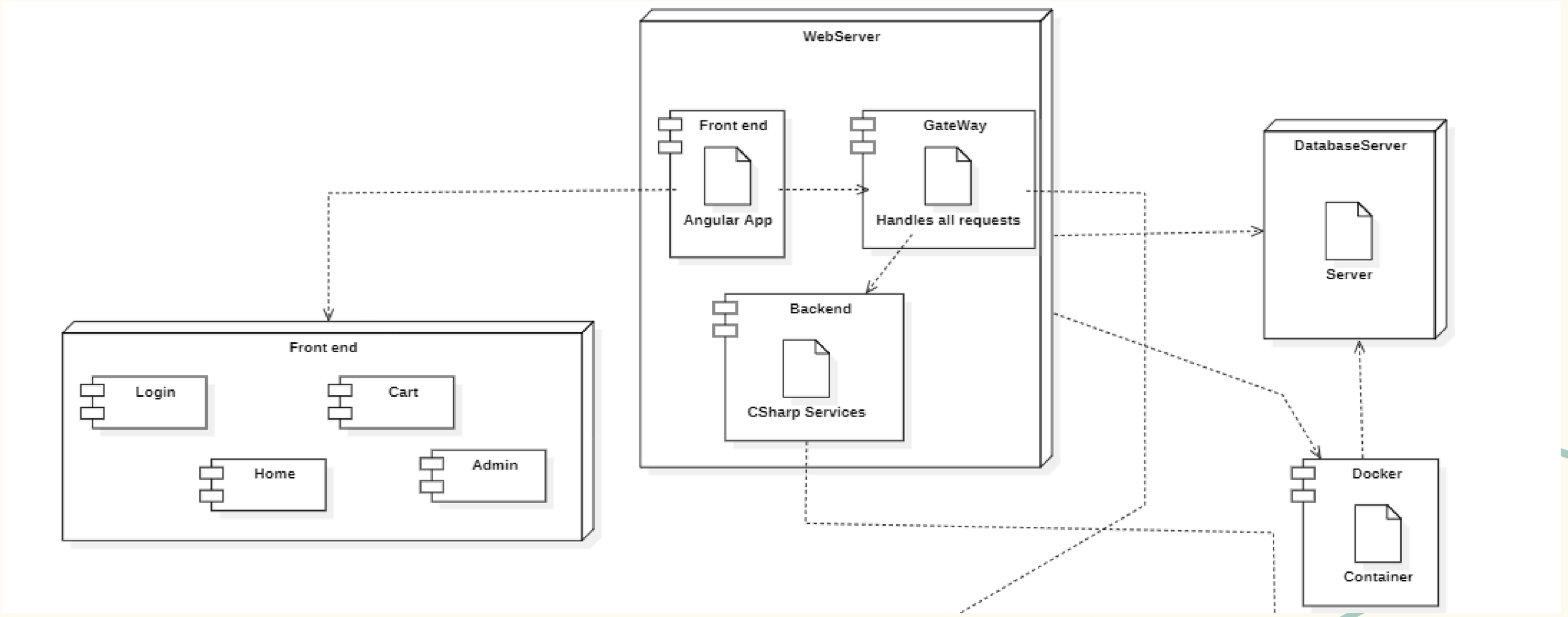
# Component Diagram



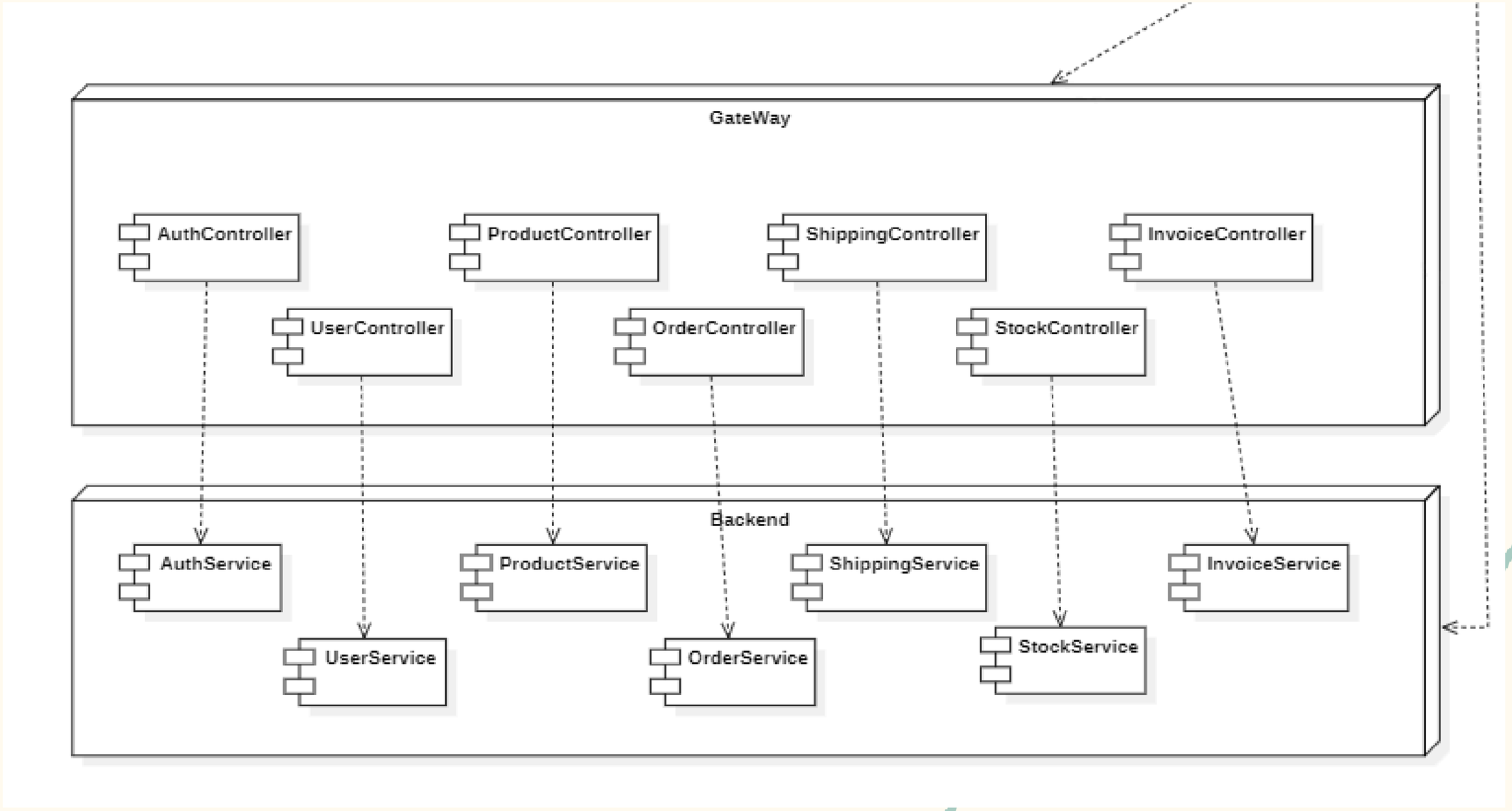
# Block Diagram



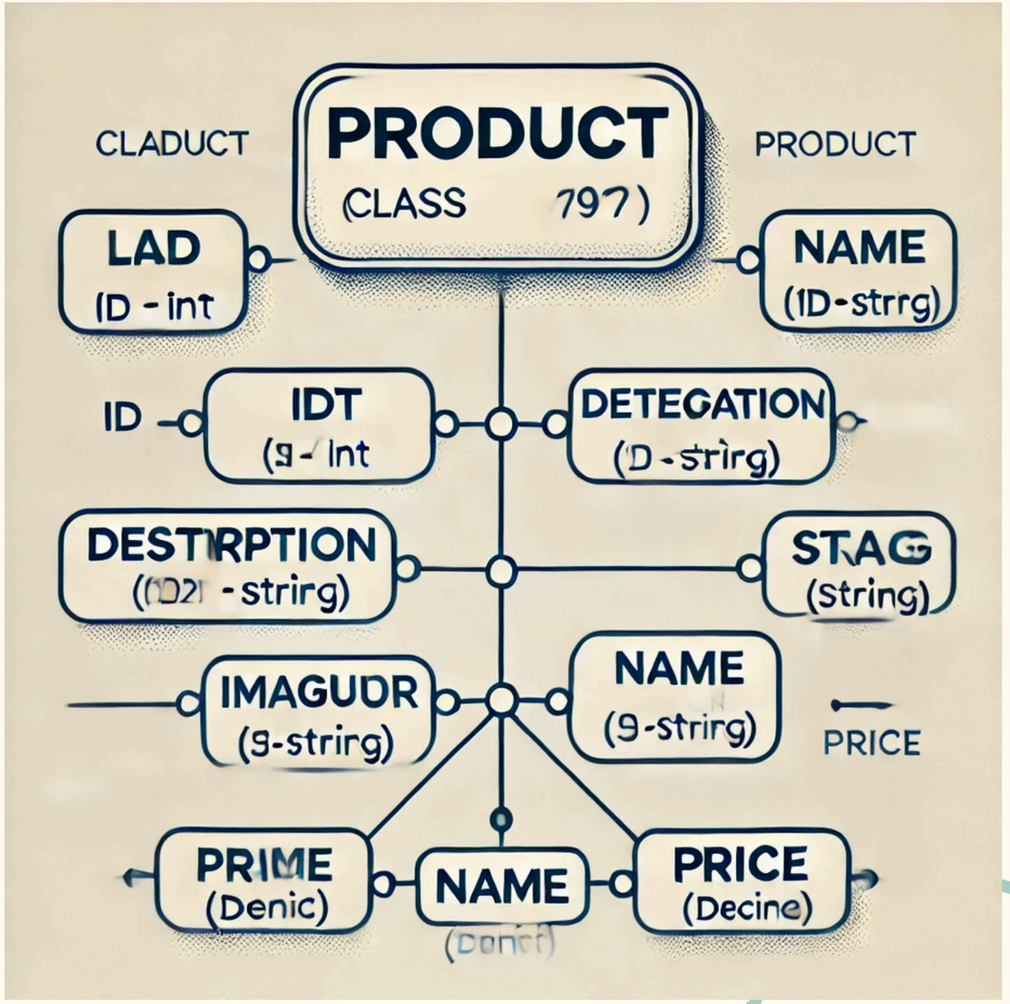
# Deployment Diagram



# Deployment Diagram



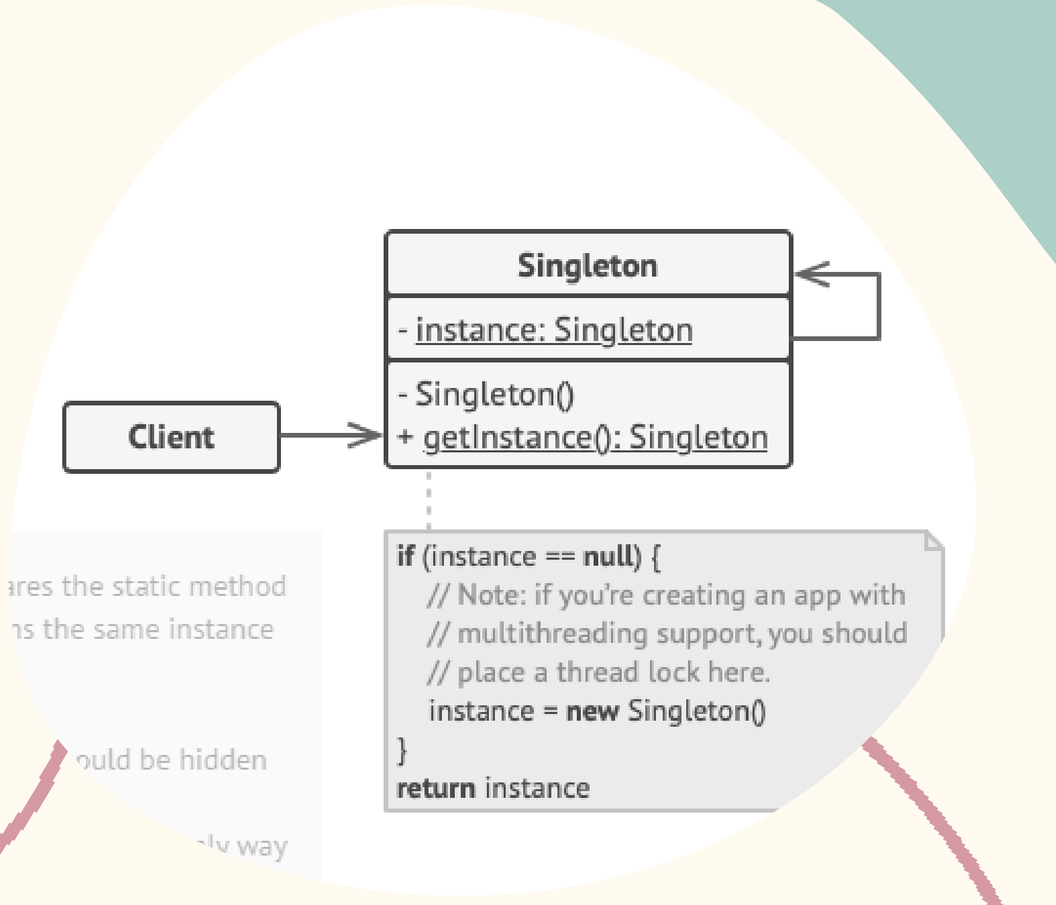
# Class Diagram 1st



# Singleton

## Definition

The Singleton design pattern is a creational pattern used to ensure that a class has only one instance while providing a global point of access to that instance. This pattern is particularly useful when exactly one object is needed to coordinate actions across the system.



# Singleton

## Pros

- You can be sure that a class has only a single instance.
- You gain a global access point to that instance.
- The singleton object is initialized only when it’s requested for the first time.

# Singleton

## Cons

Violates the Single Responsibility Principle. The pattern solves two problems at the time

The pattern requires special treatment in a multithreaded environment so that multiple threads won’t create a singleton object several times.

The Singleton pattern can mask bad design, for instance, when the components of the program know too much about each other.

Many test frameworks rely on inheritance when producing mock objects. Since the constructor of the singleton class is private and overriding static methods is impossible in most languages

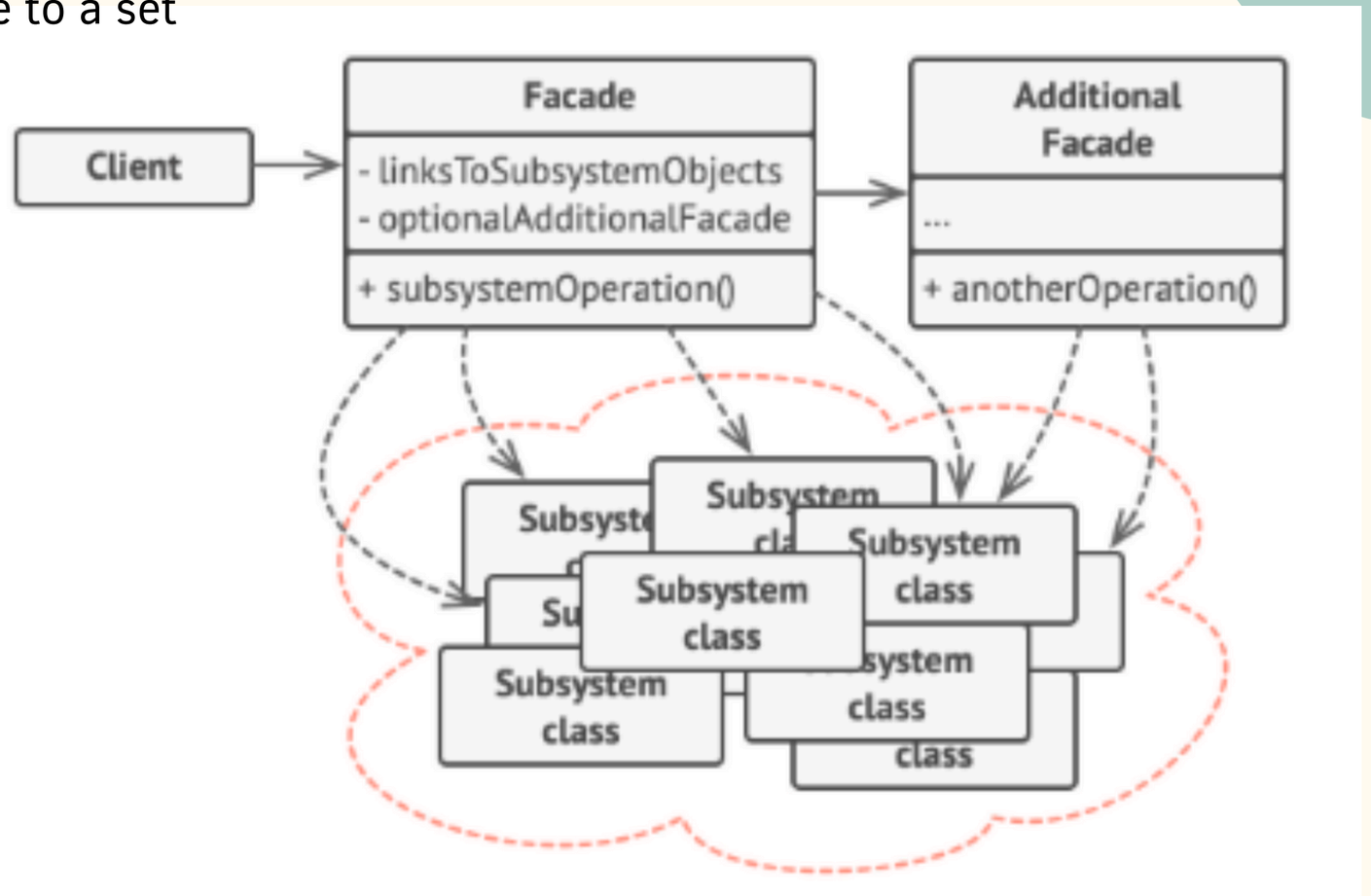
# Singleton

Objects such as UserClient, AuthClient, ProductClient, StockClient, OrderClient, ShippingClient, and InvoiceClient will be started only once during the life of the application, and the same instance will be used each time they are requested.

```
builder.Services.Add(ServiceDescriptor.Singleton(typeof(UserService.User.UserClient), userClient));
builder.Services.Add(ServiceDescriptor.Singleton(typeof(AuthService.Auth.AuthClient), authClient));
builder.Services.Add(ServiceDescriptor.Singleton(typeof(ProductService.Product.ProductClient), productClient));
builder.Services.Add(ServiceDescriptor.Singleton(typeof(StockService.Stock.StockClient), stockClient));
builder.Services.Add(ServiceDescriptor.Singleton(typeof(OrderService.Order.OrderClient), orderClient));
builder.Services.Add(ServiceDescriptor.Singleton(typeof(ShippingService.Shipping.ShippingClient), shippingClient));
builder.Services.Add(ServiceDescriptor.Singleton(typeof(InvoiceService.Invoice.InvoiceClient), invoiceClinet));
```

# Facade Pattern

The Facade pattern is a structural design pattern in programming that helps you simplify the interface of a complex system by providing a unified interface to a set of subclasses.



# Facade

## Cons

### Increase simplicity

Facade simplifies the use of a complex system by providing an easy-to-use interface

### Increased flexibility

Facade makes it easier for the system to change without affecting users

### Increase maintainability

Facade makes the system easier to maintain by concentrating complexity in a single place

### Reduce dependencies

Facade helps reduce dependencies between interface classes. This makes the system easier to change and expand.

# Facade

## ProductService

Each method like AddProduct, DeleteProductu, UpdateProduct, etc., performs complex operations (updating the database, querying data, and handling exceptions) but is exposed through a simple API.

In Case of ProductService:

- Like Facade: ProductService provides a simple interface to interact with the database through gRPC methods, helping to hide details about how to retrieve and manage data in the database. It helps the client not have to worry about Entity Framework details or SQL queries.

- Differences from Facade: While Facade is often about providing a single interface to a complex system or multiple subsystems, ProductService not only simplifies communication but also actively participates in processing it. Manage business logic, control data, and execute specific product-related tasks.

# Facade

## ProductService

```
3 references
public override Task<CreateProductReply> AddProduct(CreateProductRequest request, ServerCallContext context)
{
    _context.Products.Add(new Models.Product()
    {
        Name = request.Data.Name,
        Category= request.Data.Category,
        Description= request.Data.Description,
        ImageUrl= request.Data.ImageUrl,
        Price= request.Data.Price,
    });
    _context.SaveChanges();
    return Task.FromResult(new CreateProductReply
    {
        Message = "Created"
    });
}
```

# Facade

## ProductService

```
3 references
public override Task<UpdateProductReply> UpdateProduct(UpdateProductRequest request, ServerCallContext context)
{
    var prod = (from p in _context.Products
                where p.Id == request.Data.Id
                select p).SingleOrDefault();

    if (prod == null)...
    else
    {
        prod.Name = request.Data.Name;
        prod.Price = request.Data.Price;
        prod.ImageUrl = request.Data.ImageUrl;
        prod.Description = request.Data.Description;
        prod.Category = request.Data.Category;
        _context.SaveChanges();
    }
    return Task.FromResult(new UpdateProductReply { IsSuccess = true });
}
```

# Facade

## ProductService

```
3 references
public override Task<DeleteProductReply> DeleteProduct(DeleteProductRequest request, ServerCallContext context)
{
    var prod = (from p in _context.Products
                where p.Id == request.Id
                select p).SingleOrDefault();
    if (prod == null)
    {
        return Task.FromResult(new DeleteProductReply { IsSuccess = false });
    }
    else
    {
        _context.Products.Remove(prod);
        _context.SaveChanges();
    }
    return Task.FromResult(new DeleteProductReply { IsSuccess = true });
}
```

# Facade

## ProductService

```
3 references
public override Task<GetProductPaginateReply> GetProductPaginate(GetProductPaginateRequest request, ServerCallContext context)
{
    _context.Products.Load();
    var prods = (from prod in _context.Products
                 where prod.Id > request.AfterID
                 select new ProductData...).Take(request.Limit);
    var result = new GetProductPaginateReply();
    foreach (var prod in prods)...
    return Task.FromResult(result);

}

3 references
public override Task<GetNumOfProductReply> GetNumOfProduct(GetNumOfProductRequest request, ServerCallContext context)
{
    _context.Products.Load();
    var result = new GetNumOfProductReply();
    result.Total = _context.Products.Count();
    return Task.FromResult(result);
}
3 references
public override Task<GetProductPriceReply> GetProductPrice(GetProductPriceRequest request, ServerCallContext context)
{
    var price = (from p in _context.Products
                 where p.Id == request.Id
                 select p.Price).SingleOrDefault();
    return Task.FromResult(new GetProductPriceReply { Price = price });
}
```

# Facade

## ProductController

The ProductController in the gateway is a typical example of the Facade Pattern. It provides a simple interface to interact with product and warehouse services, hiding the complexity of gRPC communication and business logic details.

- **Simplified Access**: Controller simplifies access to complex services such as ProductService and StockService. Programmers don't need to worry about the details of how gRPC communicates or what parameters are needed for each call.
- **Centralized Logic**: All product-related requirements are centralized through a single layer, making code management and maintenance easier.
- **Hides the Complexity**: The controller hides all the complex details of product service interactions, providing users with just a simple interface to work with.

# Facade

## ProductController

```
[ApiController]
[Route("v1/[controller]")]
1 reference
public class ProductController : ControllerBase
{
    private ProductService.Product.ProductClient _productClient;
    private StockService.Stock.StockClient _stockClient;

    0 references
    public ProductController(ProductService.Product.ProductClient productClient, StockService.Stock.StockClient stockClient)
    {
        _productClient = productClient;
        _stockClient = stockClient;
    }

    [HttpPost]
    0 references
    public async Task<ProductService.CreateProductReply> CreateProduct(ProductService.CreateProductRequest createProductRequest)
    {
        var result = await _productClient.AddProductAsync(createProductRequest);
        return result;
    }
```

# Facade

## ProductController

```
[HttpGet("paginate")]
0 references
public async Task<ProductService.GetProductPaginateReply> GetProductPaginate([FromQuery]long afterID, [FromQuery]int limit)
{
    return await _productClient.GetProductPaginateAsync(new ProductService.GetProductPaginateRequest { AfterID = afterID, Limit = limit });
}

[HttpGet("total")]
0 references
public async Task<ProductService.GetNumOfProductReply> GetNumOfProduct()
{
    return await _productClient.GetNumOfProductAsync(new ProductService.GetNumOfProductRequest { Message = "" });
}

[HttpGet("search")]
0 references
public async Task<ProductService.GetProductByIdReply> GetProductByIdReply([FromQuery]long id)
{
    return await _productClient.GetProductByIdAsync(new ProductService.GetProductByIdRequest { Id = id });
}
```

# Facade

## ProductController

```
[HttpDelete("delete")]
0 references
public async Task<ProductService.DeleteProductReply> DeleteProduct([FromQuery]long id)
{
    return await _productClient.DeleteProductAsync(new ProductService.DeleteProductRequest { Id = id });
}

[HttpGet("all")]
0 references
public async Task<Models.ProductResponse> GetALLProductDetail()
{
    var stocks = new List<StockService.StockData>();
    var total = await _productClient.GetNumOfProductAsync(new ProductService.GetNumOfProductRequest { Message = "" });
    var products = await _productClient.GetProductPaginateAsync(new ProductService.GetProductPaginateRequest { AfterID = 0, Limit = (int)total.Total });
    foreach(var item in products.ProductList)
    {
        var stock = await _stockClient.GetStockByIdAsync(new StockService.GetStockByProdIdRequest { ProdID = item.Id });
        stocks.Add(stock.Data);
    }
    return new Models.ProductResponse { stocks = stocks, products = products.ProductList.ToList() };
}
```

# Builder Pattern

## ProductContext

The Builder Design Pattern in DbContext configuration helps create a visual, structured, and easily adaptable way to define work and manage data models in application.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Product>(entity =>
    {
        entity.ToTable("product");

        entity.Property(e => e.Id).HasColumnName("id");
        entity.Property(e => e.Category).HasColumnName("category");
        entity.Property(e => e.Description).HasColumnName("description");
        entity.Property(e => e.ImageUrl).HasColumnName("imageURL");
        entity.Property(e => e.Name).HasColumnName("name");
        entity.Property(e => e.Price).HasColumnName("price");
    });

    OnModelCreatingPartial(modelBuilder);
}
```

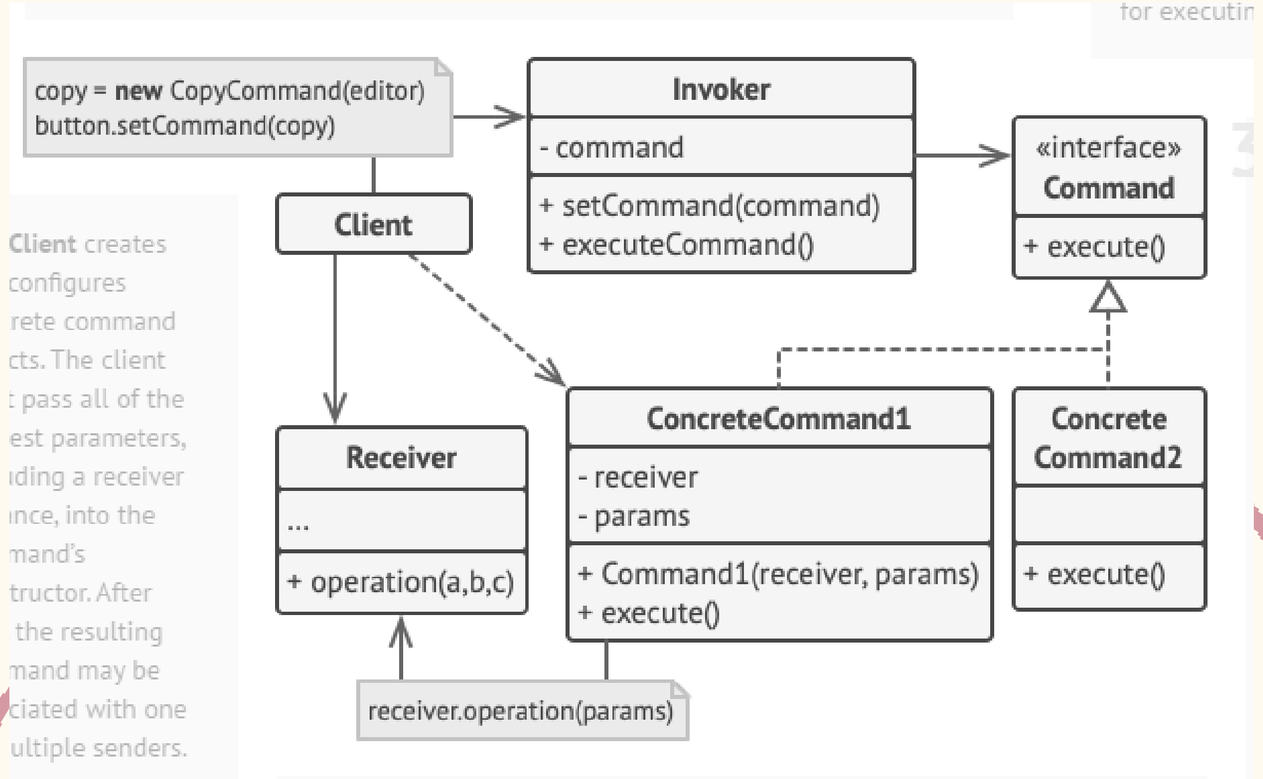
# Builder Pattern

## ProductContext

- **Method Chaining:** ModelBuilder allows you to define a chain of configurations without having to rename modelBuilder variables. Each method on ModelBuilder typically returns a ModelBuilder or a child builder, allowing you to continue performing configurations without interruption.
- **Entity Configuration:** In this section, entity is a specific builder for the Product object. It is used to configure more details for each Product attribute, such as table name, column name and other attributes. Each call to Property or ToTable is part of the configuration build process.
- **The Sophistication of the Builder Pattern:** Using a builder makes the complex configuration of Entity Framework entities clear and easy to manage. It also reduces errors due to configuration complexity and makes the code easier to maintain.
- **Extension:** OnModelCreatingPartial is a method that you can override in a separate file, allowing you to separate the model configuration from the main context file, making the code more manageable and extensible.

# Command Pattern

Command is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method arguments, delay or queue a request's execution, and support undoable operations.



# Command

## Single Responsibility Principle

You can decouple classes that invoke operations from classes that perform these operations.

## Open/Closed Principle

You can introduce new commands into the app without breaking existing client code.

## Cons

- You can implement undo/redo.
- You can implement deferred execution of operations.
- You can assemble a set of simple commands into a complex one.

# Command Pattern

## ProductService

In ProductService.cs, each method handles an action on the product, similar to the Command pattern, where each command encapsulates a specific operation

```
3 references
public override Task<DeleteProductReply> DeleteProduct(DeleteProductRequest request, ServerCallContext context)
{
    var prod = (from p in _context.Products
                where p.Id == request.Id
                select p).SingleOrDefault();
    if (prod == null)...
    else
    {
        _context.Products.Remove(prod);
        _context.SaveChanges();
    }
    return Task.FromResult(new DeleteProductReply { IsSuccess = true });
}
```

Each method like AddProduct, UpdateProduct, DeleteProduct, etc., acts as a command that handles a specific request. Helps increase modularity, maintainability and extensibility of code, while reducing dependencies and increasing abstraction in the application.

# Command Pattern

## ProductService

In ProductService.cs, each method handles an action on the product, similar to the Command pattern, where each command encapsulates a specific operation

```
3 references
public override Task<DeleteProductReply> DeleteProduct(DeleteProductRequest request, ServerCallContext context)
{
    var prod = (from p in _context.Products
                where p.Id == request.Id
                select p).SingleOrDefault();
    if (prod == null)...
    else
    {
        _context.Products.Remove(prod);
        _context.SaveChanges();
    }
    return Task.FromResult(new DeleteProductReply { IsSuccess = true });
}
```

Each method like AddProduct, UpdateProduct, DeleteProduct, etc., acts as a command that handles a specific request. Helps increase modularity, maintainability and extensibility of code, while reducing dependencies and increasing abstraction in the application.