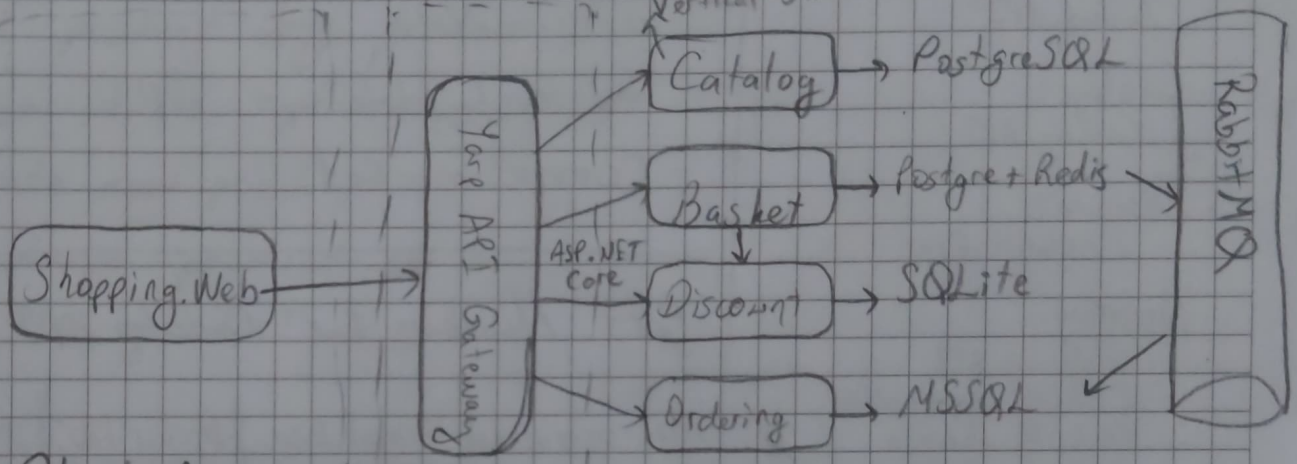
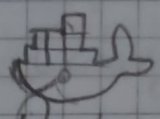
- Carter will be used for implementing minimal API endpoint definition into Catalog microservice - Mapster will be used for mapping DTO's

What will be used in Catalog.API → will use PostgreSQL but will act as Document DB using Marten library

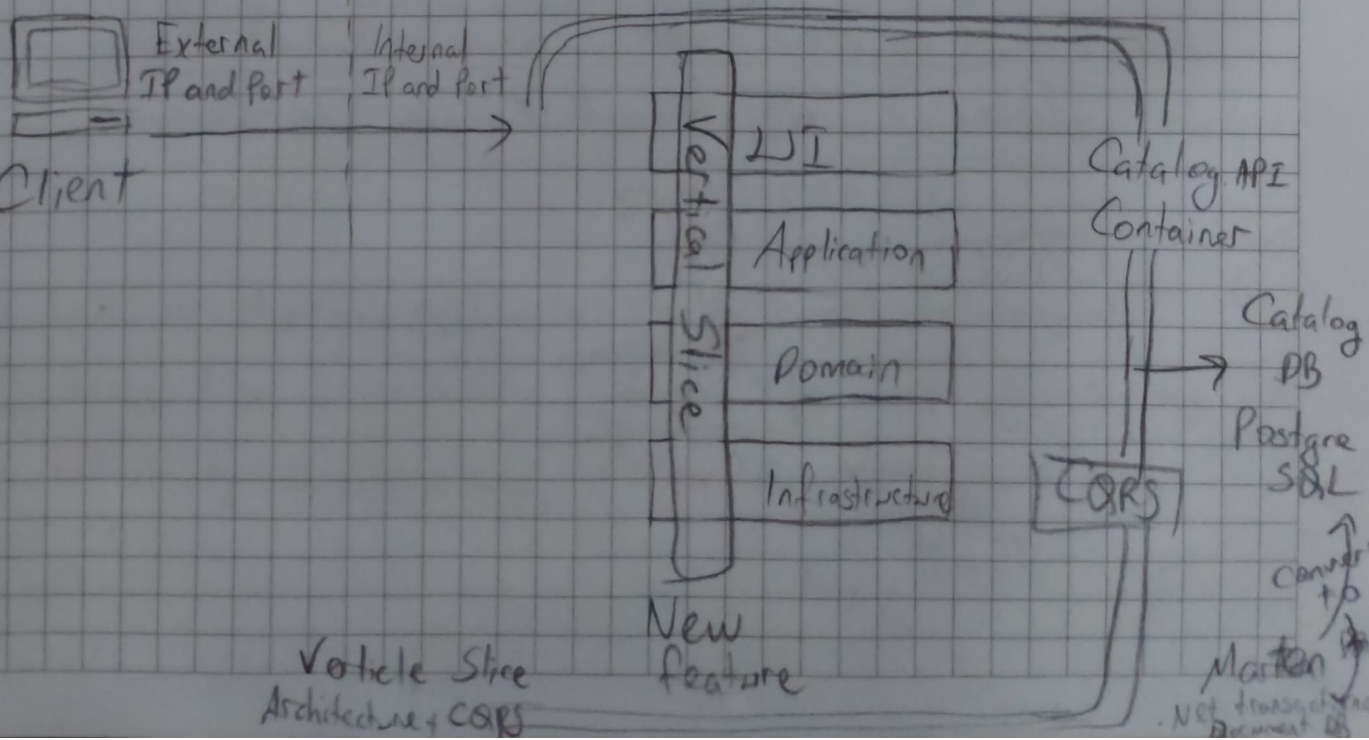
- Minimal API
- Vertical Slice with feature folders
- CQRS with MediatR - Fluent Validation
- Marten library for .Net transactional Document DB on PostgreSQL





Catalog Microservice

Internal "Catalog" Microservice Architecture



Overview to the Solution

- Api Gateways
  - YarpApi Gateway
- Building Blocks
- Services
  - Basket
    - Basket.API
  - Catalog
    - Catalog.API
  - Discount
    - Discount.Grpc
  - Ordering
    - Ordering.API
    - Ordering.Application
    - Ordering.Domain
    - Ordering.Infrastructure
  - WebApps
    - Shopping.Web

Phase 1 → Creat Folder Services

Create folder named Catalog and or

Add project ASP.NET Core Emty Catalog.API

don't forget to add folders

enable docker Linux

configure HTTPS

top level statements will be used

- remove map Get

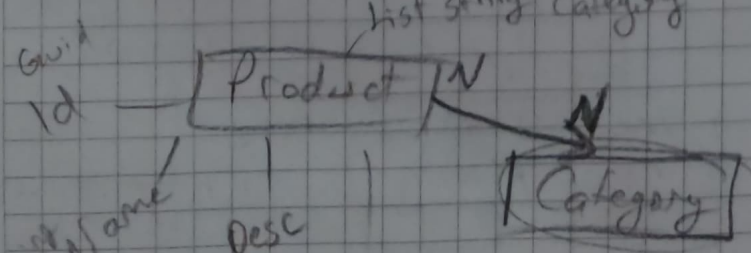
Port Numbers for Catalog.API

Microservices	Local Env	Docker Env.	Docker Invoke
Catalog	5000-5050	6000-6060	8000-8050
Basket	5001-5051	6001-6061	8080-8081
Discount	5002-5052	6002-6062	8050-8051
Ordering	5003-5053	6003-6063	8080-8051

listed as http/https

modify launch settings

Domain Analysis of Catalog Microservices



Application Use Cases of Catalog Microservices

CRUD operations:

List products and Categories
Get product by product id
Get products by category
Create new product
update product
delete product

GET	/products
GET	/products/{id}
GET	/products/category
POST	/products
PUT	/products/{id}
DELETE	/products/{id}

Underlying Data Structures of Catalog Microservices

- Document Database with Store Catalog JSON data
- There are 2 options: MongoDB No-SQL database, PostgreSQL DB JSON columns

Chosen → PostgreSQL with the Marten Library
Combines the flexibility of a document database with the reliability of relational PostgreSQL database

Initial Folder Structure for Catalog Microservice

```
Catalog
  Catalog.API
    Data
      CatalogInitialData.cs
    Exceptions
      ProductNotFoundException
    Models
      Product.cs
    Products
      CreateProduct
        CreateProductEndpoint.cs
        CreateProductHandler.cs
      DeleteProduct
      GetProductByCategory
      GetProductById
      GetProducts
      UpdateProduct
```

Organized into

- Model
- Features
- Data
- Abstractions

---

restart ↑

1. Code your app
2. (Once) Add docker support to projects
3. Run/Debug containers / Compose app
4. Test your app or microservices
5. Push or continue developing → git push

Vertical Slice vs Clean Architecture

* VSA emphasizes organizing software development around features, cutting through all layers
* CA focuses on separation of concerns and dependency rules. It organizes code into layers
* VSA development teams concentrate on delivering complete features, each potentially touching all layers of the stack.
* CA more structured approach, ensuring that business logic is decoupled from external concerns

* VSA well suited for agile teams working on complex apps with numerous features

* CA ideal for large-scale applications where long-term maintenance, scalability, and the ability to adapt to changing business requirements

When to Choose Vertical Slice Over Clean Arc?

Rapid Development & deployment: When the priority is to develop and deploy features rapidly and independently.

Agile and Scrum Teams: For teams practicing Agile or Scrum that needed to deliver complete features in short cycles.

Microservices: In a microservice architecture, whose each service is often responsible for a distinct feature or business capability.

CQRS - Comman Query Responsibility Segregaton

* Used in order to avoid complex queries to get rid of inefficient Joins

* Seperates read and write operations with seperating databases.

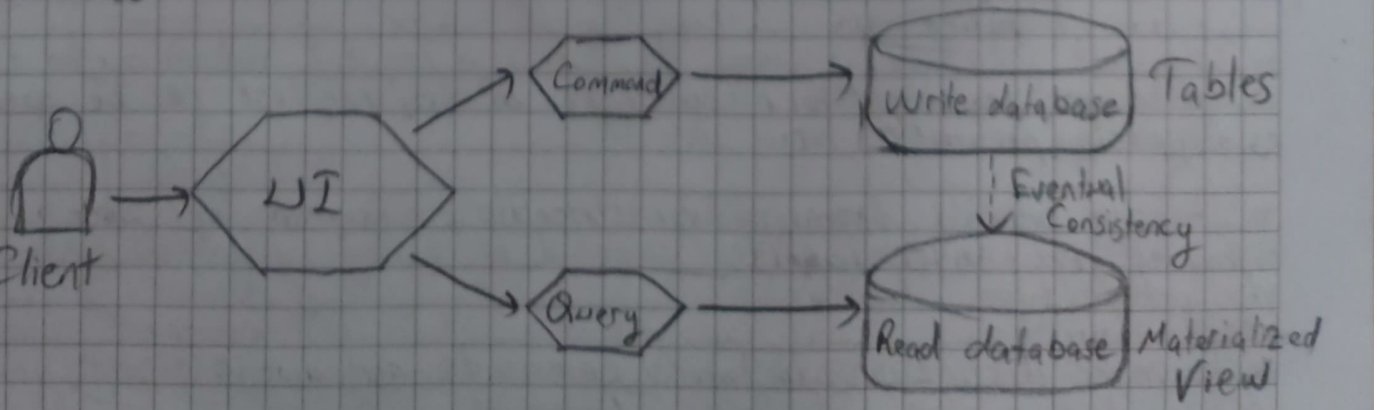
Commands: Changing the state of the data into application

Queries handling complex Join operations and return a result and don't change the state of data into application

* Large-scaled microservices architectures needs to manage high volume data requirements.

* Single database for services can cause bottlenecks.

* Uses both CQRS and Event Sourcing patterns to improve application performance



To study later → Database Normalization.

* Monolitic has single database is both working for complex Join queries, and also perform CRUD operations

* Application required som query that needs to Join more than 10 table, will lock the database due to latency of query computation

* Performing CRUD operations need to make complex validations and process long business logic, will cause to lock database operations

* Reading and writing database has different approaches, define different strategy

* Seperation of concerns >> principles: seperate reading database and the writing database with 2 database

- Read Database uses No-SQL databases with denormalized data
- Write Database uses Relational databases with fully normalized and supports strong data consistency

Logical and Physical Implementation of CQRS

Logical Implementation of CQRS: Splitting Operations Not Databases. Seperate the read (query) operations from the write (command) operations at the code level, but not necessarily at the database level

Even though the same database is used, the paths for reading and writing data are distinct.

Physical Implementation of CQRS: Seperate databases. Splitting the read and write operations not just at the code level but also physically using seperate databases.

* Introduces data consistency and synchronization problems.

CQRS Design Pattern With MediatR Library

* MediatR is a popular .Net library used to implement the mediator pattern, which is a great fit for CQRS.

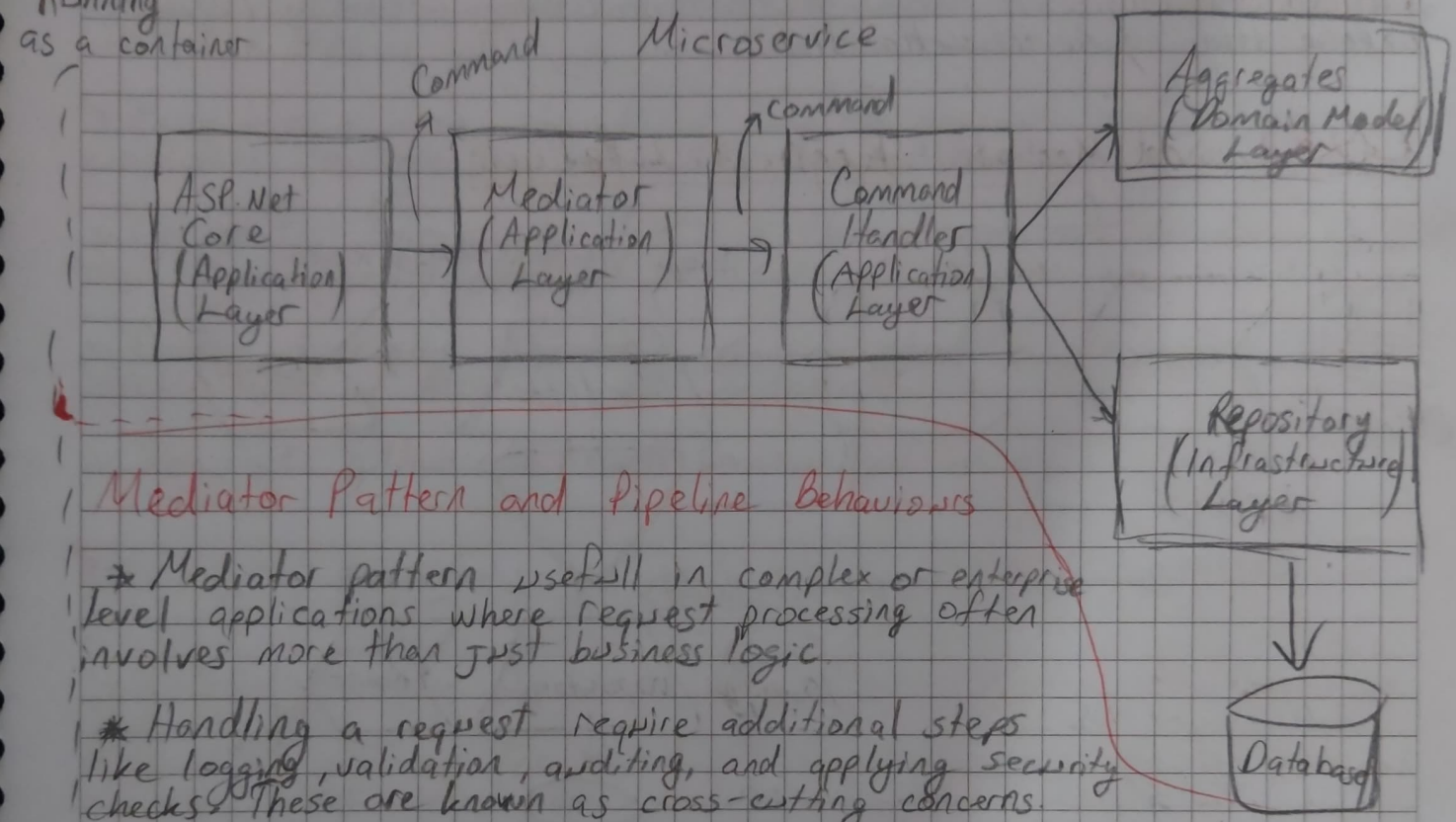
* IRequest interface is used to define a request which can be either command or a query. The return type of the request can be specified as a generic parameter.

* Handler inherits from IRequestHandler<TRequest, TResponse>, where TRequest is the type of the command or query, and TResponse is the return type

* To make distinction between commands and queries clearer, you can define two custom interfaces: ICommand, IQuery

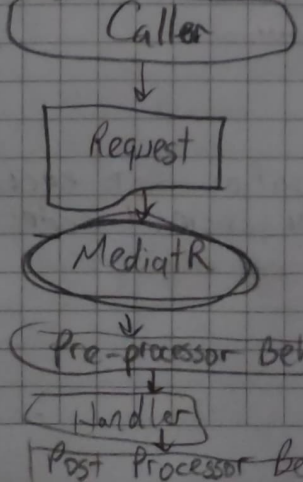
```
public interface ICommand<TResult> : IRequest<TResult>{}
public interface IQuery<TResult> : IRequest<TResult>{}
```

ASP.Net Core using MediatR minimal APIs act as the entry point for handling HTTP requests. Instead of implementing the business logic directly, these actions delegate the responsibility to MediatR.



Mediator Pattern and Pipeline Behaviours

* Mediator pattern usefull in complex or enterprise level applications where request processing often involves more than just business logic.
* Handling a request require additional steps like logging, validation, auditing, and applying security checks. These are known as cross-cutting concerns.
* MediatR provides a mediator pipeline where these cross-cutting concerns can be inserted transparently.
* Pipeline coordinates the request handling, ensuring that all necessary steps are executed in the right order.
* In MediatR, pipeline behaviors are used to implement cross-cutting concerns.
* Wrap around the request handling, allowing you to execute logic before and after the actual handler is called.



The behaviors will execute in the order they are registered

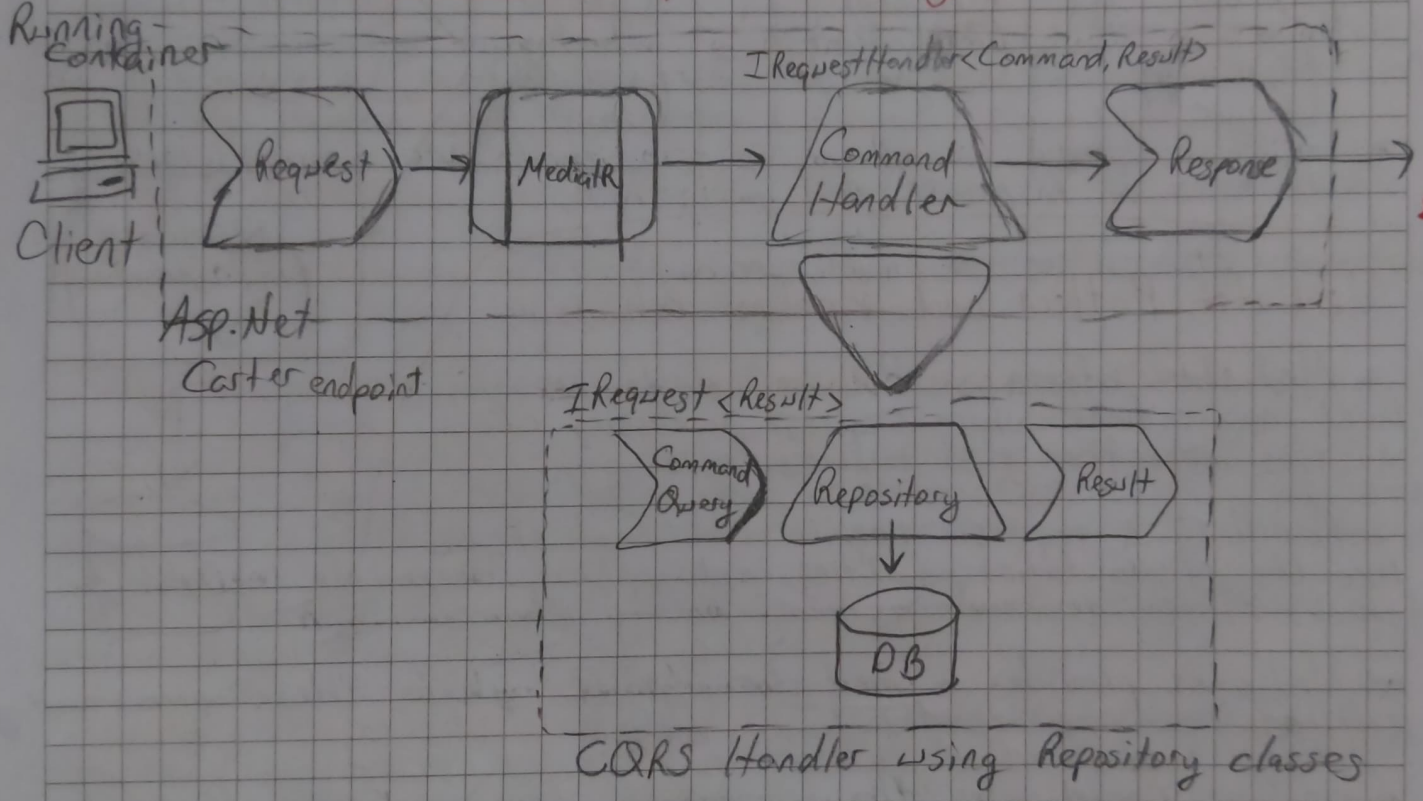
1. Logging
2. Validation
3. Caching
4. Retry

EShop Microservices Pipeline Behaviors

LogBehavior A behavior that logs details about the handling of a request

ValidatorBehavior A behavior that validates incoming requests before they reach the handler

CQRS and MediatR Request LifeCycle



Create Abstraction on MediatR for CQRS

IRequest → IQuery

IRequest → ICommand

IRequestHandler → ICommandHandler

IRequestHandler → IQueryHandler

BuildingBlocks

BuildingBlocks encapsulate our common operation for each microservices Install all common NuGet packages and codes into BuildingBlocks so that microservices can use it by referencing this class library

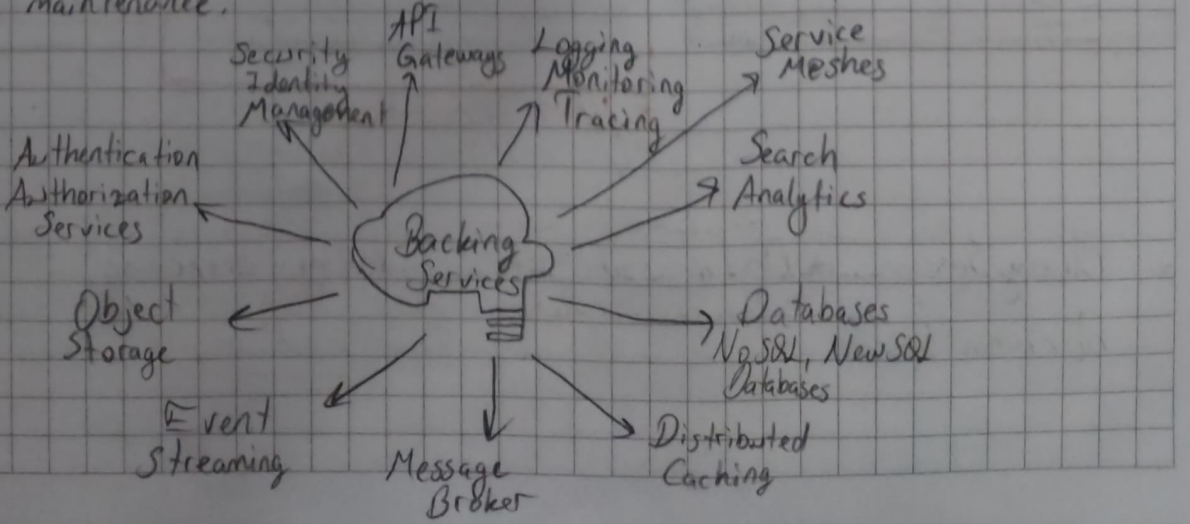
Carter Library for Minimal Api Endpoint Definition

* With Asp.Net Core 8, Minimal APIs have become a popular way to build lightweight and performant microservices and web API's
* Carter is a library that extends the capabilities of ASP.NET Core's Minimal APIs
* Provides a more structured way to organize our endpoints and simplifies the creation of HTTP request handlers.
* Carter is framework that is a thin layer of extension methods and functionality over ASP.NET core.
* Carter especially beneficial in minimal APIs, simplify the development of Minimal API endpoints.
* Go visit github.com/CarterCommunity/Carter for more

For mapping operations go visit github.com/MapsterMapper/Mapster

What is Backing Services for Cloud-Native Microservices

* Backing services for cloud-native microservices are the external components that microservices depend on for their operation.
* Provide support for various functionalities, such as data storage, messaging, caching, and authentication.
* Backing services like databases, messaging systems, and caching services are treated as attached resources.
* Backing services are external to the microservice and can be swapped or replaced without changing the microservices core logic.
* Decoupled from the microservices themselves, promoting flexibility, scalability, and easier maintenance.



Underlying Data Structures of Catalog Microservices

* Marten is an ORM (Object Relational Mapper) that leverages PostgreSQL's JSON capabilities.

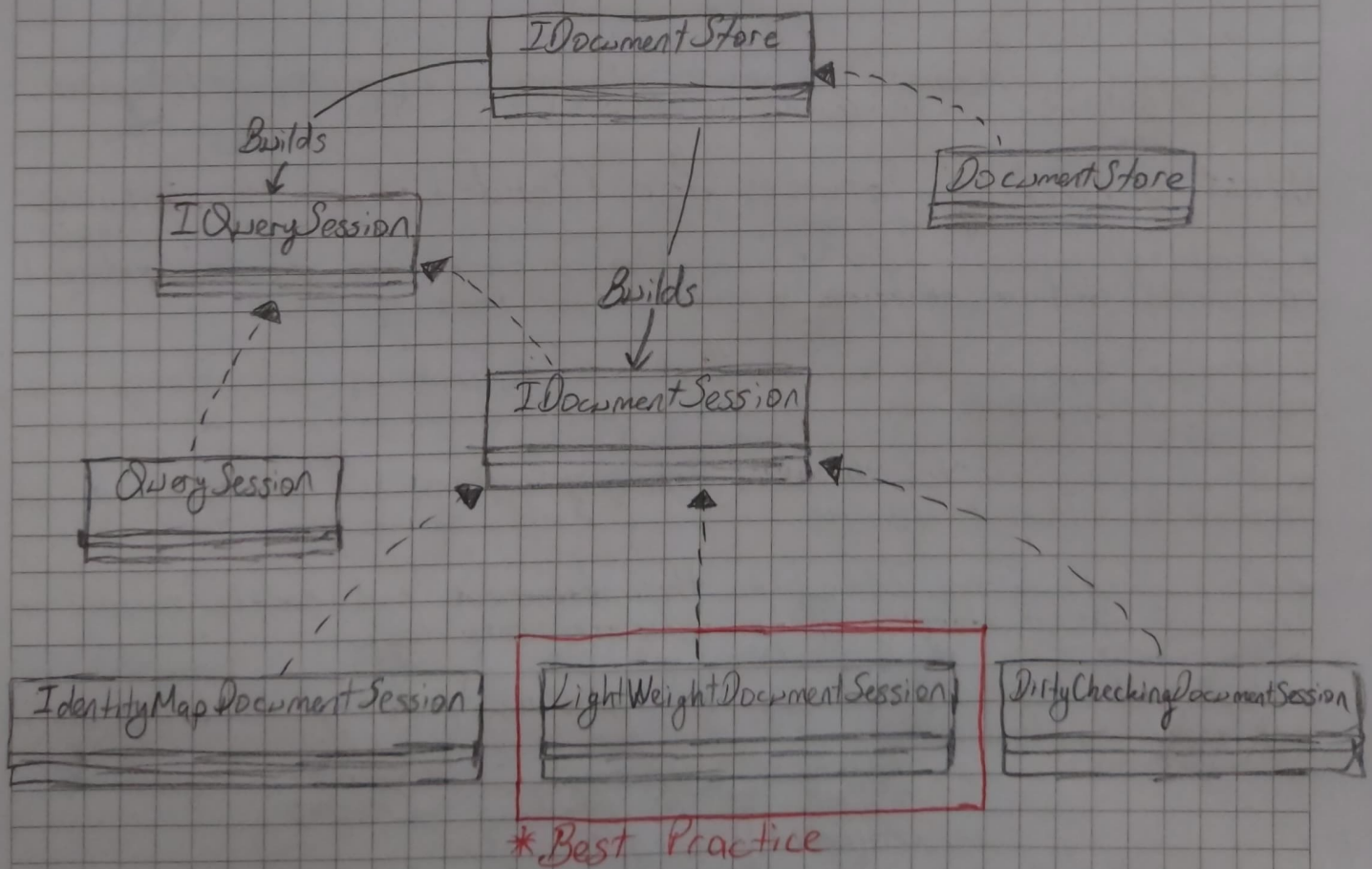
* Marten is a powerful library that transforms PostgreSQL into .NET Transactional Document DB

* PostgreSQL's JSON column features, allowing us to store and query our data as JSON documents.

* Combines the flexibility of a document database with the reliability of relational PostgreSQL database

* Catalog service use Marten for PostgreSQL Interaction as a Document DB

Opening Sessions in Marten as Document DB



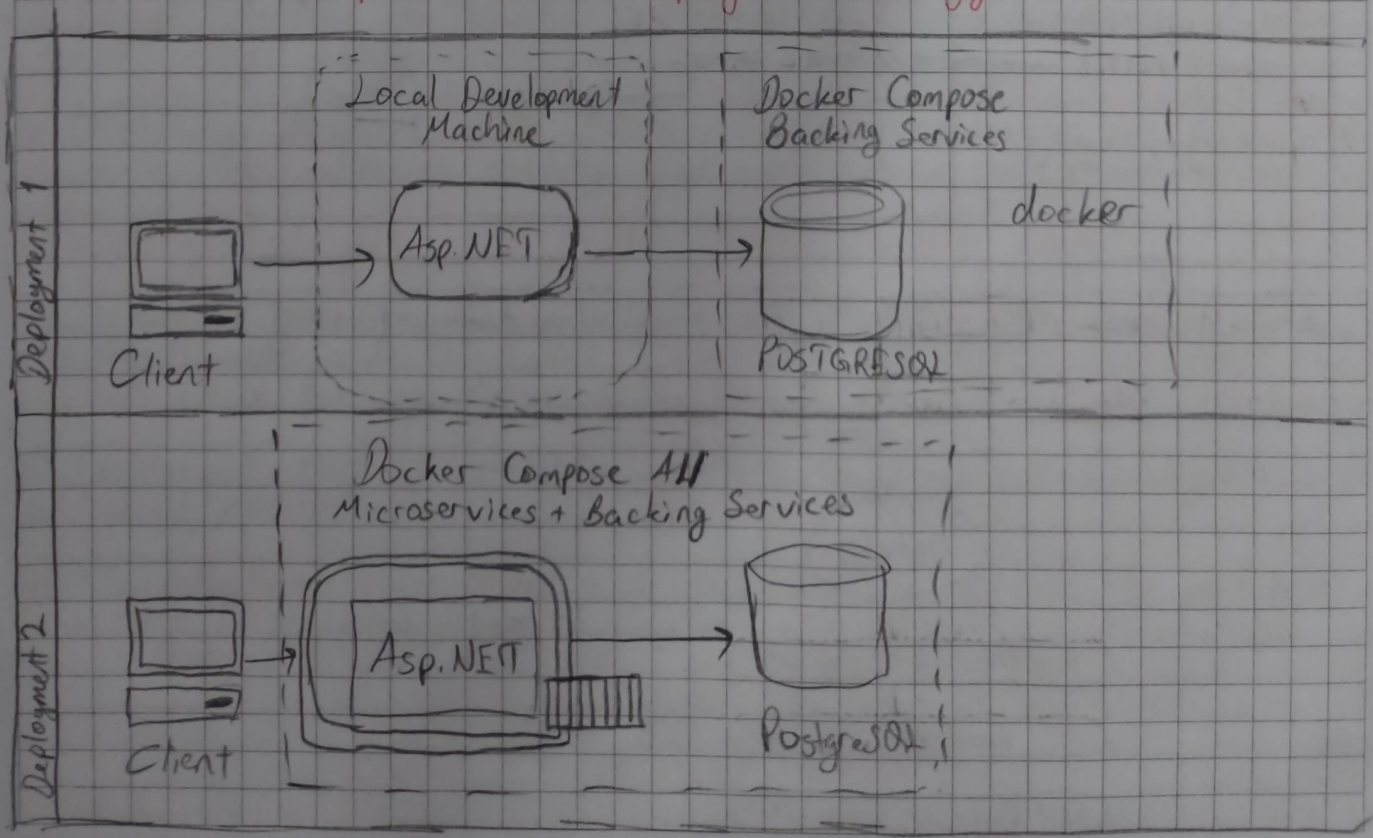
QuerySession → Optimized for readonly scenarios

DocumentSession → Optimized for read and write scenarios

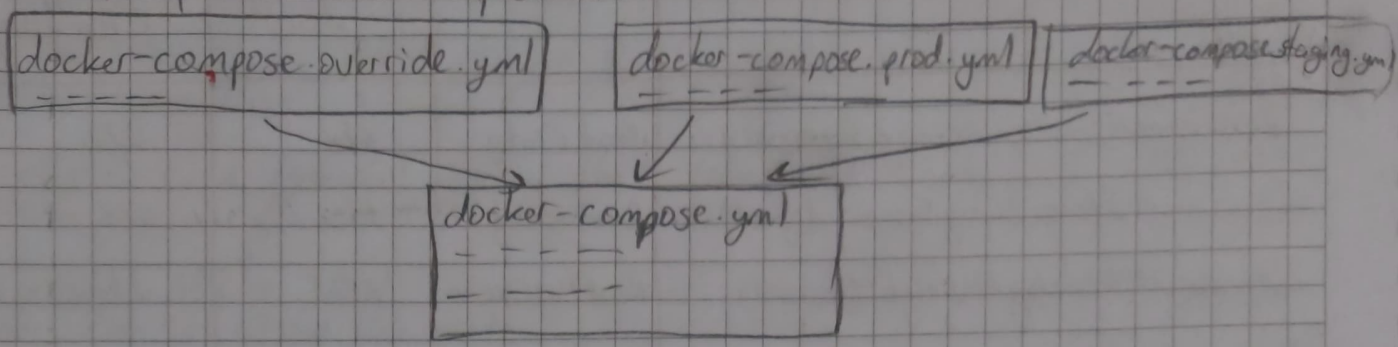
Creation	Read/Write	Identity Map	Dirty Checking
IDocumentStore.QuerySession()	Read Only	No	No
IDocumentStore.QuerySessionAsync()	Read Only	No	No
IDocumentStore.LightWeightSession()	Read/Write	No	No
IDocumentStore.LightWeightSerializableSessionAsync()	Read/Write	No	No
IDocumentStore.IdentitySession()	Read/Write	Yes	Yes
IDocumentStore.IdentitySerializableSessionAsync()	Read/Write	Yes	Yes
IDocumentStore.DirtyTrackedSession()	Read/Write	Yes	Yes
IDocumentStore.DirtyTrackedSerializableSessionAsync()	Read/Write	Yes	Yes
IDocumentStore.OpenSession()	Read/Write	Yes	No
IDocumentStore.OpenSerializableSessionAsync()	Read/Write	Yes	No

* IDocumentSession object is already an abstraction of the database operations So, we don't need any additional abstraction or unnecessary code like repository patterns.

E-Shop Microservices Deployment Strategy



Multiple Docker Compose Files



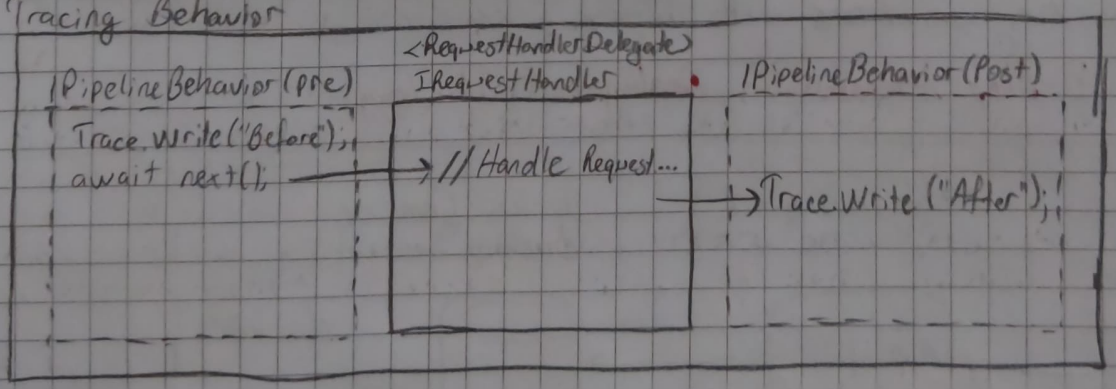
MediatR Pipeline Behaviors

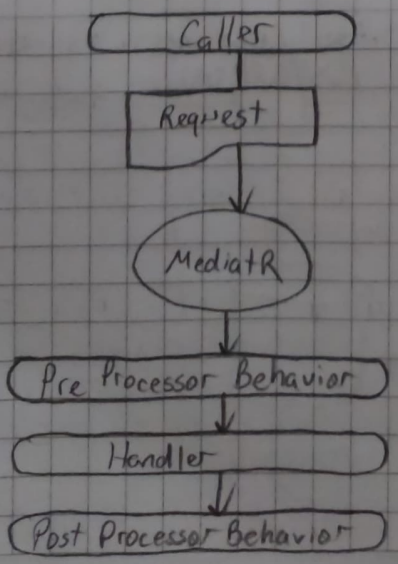
* MediatR powerful feature is 'Pipeline Behaviors' allows to additional logic into the request handling process: validation, logging, exception handling and performance tracking

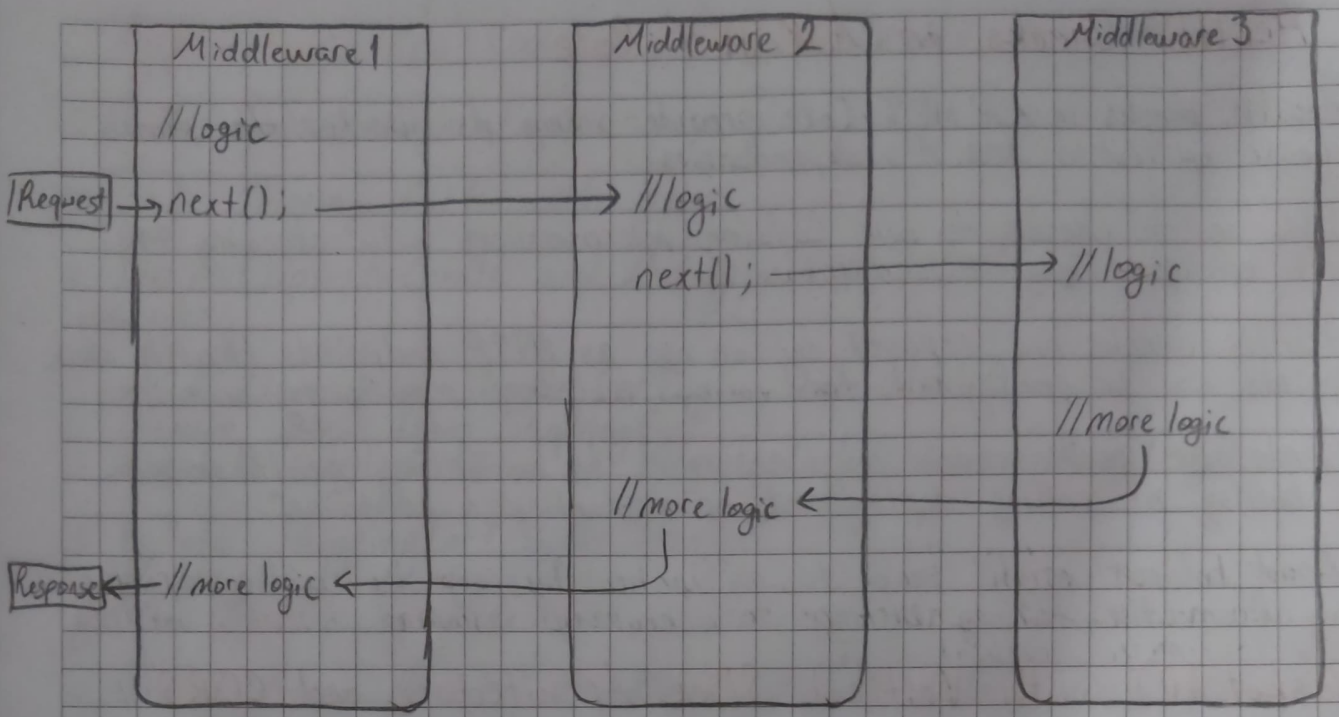
* Pipeline behaviors act as middleware in the MediatR library, wrap around the request handling process, enabling to implement cross-cutting concerns

* A class that implements IPipelineBehavior<TRequest, TResponse>

* Handle method where you can execute code before and after the next delegate is invoked







MediatR Pipline Behaviors and Fluent Validation Library

* Fluent Validation is a .Net library for building strongly-typed validation rules.

* Integrate Fluent Validation with MediatR to validate requests before they reach the actual handler within a pipeline behavior.

* Fluent Validation define validation rules in seperate classes using a fluent interface to specify conditions that each property of your models must satisfy.

* Combining MediatR with Fluent Validation centralizes our cross-cutting concerns like validation making our code cleaner and more maintainable

Seeding CatalogDb with Marten Initial Baseline Data

* Seeding is essential for initializing the database with baseline data. Marten provides a feature called IInitialData for this purpose. (Visit the martendb website for the documents)

* Implement the IInitialData interface to seed our CatalogDb with predefined products

* This interface allows Marten to automatically populate the database when it's first Initialized

Health Checks in ASP.NET Core

* Health checks in ASP.NET Core provide a way to monitor the status of your application and it's dependancies.
* Add health checks to our Catalog microservice with checking the health of our PostgreSQL database.
* Health checks are exposed by an app as HTTP endpoints. Health check endpoints can be configured for various real-time monitoring scenarios.
* Container orchestrator may respond to a failing health check by halting or rolling deployment or restarting a container.
* Load balancer might react to an unhealthy app by routing traffic away from the failing instance to a healthy instance

Basket API with Vertical Slice Architecture and CQRS

* ASP.NET Core Minimal APIs for building fast HTTP APIs - fully functioning REST endpoints and top-level Program.cs
* Vertical Slice Architecture implementation with feature folders and single .cs file includes different classes in one file
* CQRS implementation using MediatR library
* Marten library for .NET Transactional Document DB on PostgreSQL
* Repository Pattern Implementation over Marten and Redis Cache
* Carter for Minimal API endpoint definition
* FluentValidation to validate inputs and add MediatR validation pipeline
* Dockerfile and docker-compose file for running Basket microservice and PostgreSQL database in docker environment

Domain Models of Basket Microservices

* Primary domain model is 'ShoppingCart', 'ShoppingCartItem' and 'BasketCheckout'
* Consider Domain event for basket checkout, leading to integration events.

Application Use Cases of Basket Microservices

CRUD Basket Operations:

* Get Shopping Cart with items
* Store (Upsert-Create and Update) ShoppingCart with Items that includes;
  - Add Item into ShoppingCart
  - Remove Item from ShoppingCart
  - Update ShoppingCartItem in ShoppingCart (Increase Quantity)
* Delete ShoppingCart with Items

gRPC Basket Operations:

* When Store Basket: GetDiscount and deduct discount coupon from Item Price

Async Basket Operations:

* Checkout Basket and Publish event to RabbitMQ Message Broker

Rest API Endpoints of Basket Microservices

Method	Request URI	Use Cases
GET	/basket/{userName}	Get basket w/username
POST	/basket/{userName}	Store basket (insert-update)
DELETE	/basket/{userName}	Delete basket w/ username
POST	/basket/checkout	Checkout basket

Underlying Data Structures of Basket Microservices

Basket Microservice has 2 Datastore:

1. Marten Document Database
2. Redis Distributed Cache

Redis is a powerful in-memory data store and distributed cache which is good fit for microservices architecture

Patterns and Principals of Basket Microservices

Repository Pattern: DDD pattern to keep persistence concerns outside of the system's domain model. Provides an abstraction of data that app can work with a simple abstraction interfaces.

CQRS Pattern: Command Query Responsibility Segregation divides operations into commands (write) and queries (Read).

Mediator Pattern: Facilitates object interaction through a mediator reducing direct dependencies and simplifying communications.

DI in ASP.NET Core: Dependency Injection is a core feature, allowing us to inject dependencies.

Minimal APIs and Routing in ASP.NET 8: ASP.NET 8's Minimal APIs simplify endpoint definitions, providing lightweight syntax for routing and handling HTTP requests

Libraries Nuget Packages of Basket Microservice

- MediatR for CQRS
- Carter for API Endpoints
- Marten for PostgreSQL Interaction
- Mapster for Object Mapping
- FluentValidation for Input Validation
- Next Sections: Scrutor, gRPC, StackExchange.Redis, MassTransit

Project Folder Structure of Basket Microservices

* The project is organised into Model, Features, Data, and Abstractions
* Features like GetBasket and StoreBasket have dedicated handlers and endpoint definitions
* Feature folder will be Basket
* Data folder and Repository objects manages database and cache interactions
* Containerize our service with Docker, ensuring deployment and integration with PostgreSQL
* Implement Dockerfile and docker-compose file for running Catalog microservice and PostgreSQL dabase and Redis

- Basket.API
  - Basket
    - CheckoutBasket
    - DeleteBasket
    - GetBasket
      - GetBasketEndpoints.cs
      - GetBasketHandler.cs
    - StoreBasket
  - Data
    - BasketRepository
    - IBasketRepository
    - CachedBasketRepository
  - Dtos
  - Exceptions
  - GrpcServices
  - Models > ShoppingCart.cs, ShoppingCartItem.cs

Why Choose Redis for Distributed Caching

* Redis is an advanced key-value store known for it's high performance
* It is often used for caching, session storage, pub/sub systems, and more
* Redis offers in-memory data storage, resulting in fast data access
* It's support for various data structures makes it versatile for different use cases.
* Redis is an excellent choice for microservices architecture primarily due to it's inherent distributed characteristics.
* Enabling services to access shared data quickly and reducing the load on databases

Distributed Caching With Redis in Basket Microservices

* Implement Proxy Pattern, Decorator Pattern and Scrutor library
* Implement Cache-aside Pattern / Cache invalidation
* Develop CachedBasketRepository and decorate w/ Scrutor library
* Setup Redis as a distributed cache using docker-compose file for multi-container docker environment
* Integrate Redis into our Basket microservice to cache user data
* Boost performance by reducing the load on our database and speeding up data retrieval

Cache Aside Pattern for Microservices

(1) When a client needs to access data, it first checks to see if the data is in the cache

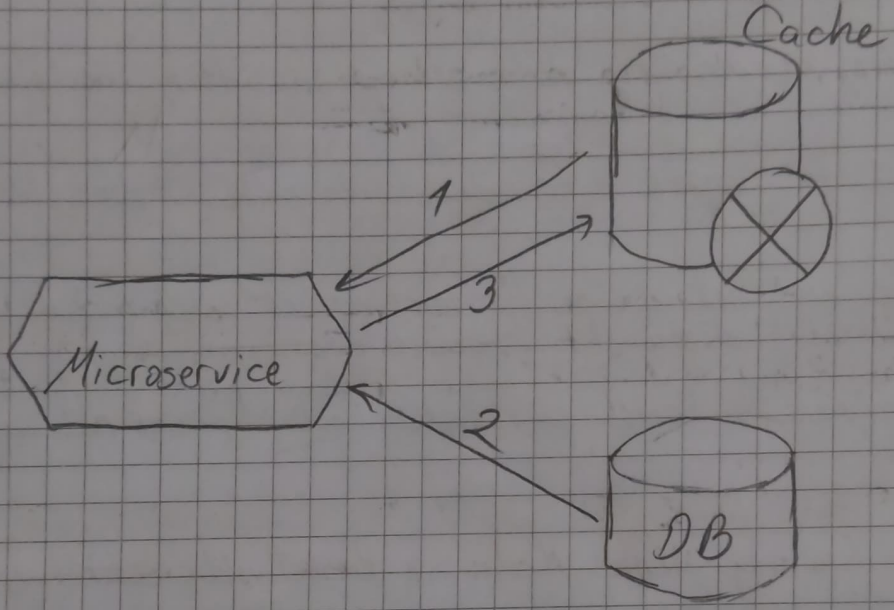
(2) If the data is in the cache, the client retrieves it from the cache and returns it to the caller

(3) If the data is not in the cache, the client retrieves it from the database, stores it in the cache, and then returns it to the caller

* Some of caching systems provide read-through and write-through / write-behind operations. In these systems, client application retrieves data over by the cache.

* For not supported caches, it's the responsibility the application use the cache and update the cache if there is a cache-miss.

* Microservices good example to implement Cache-aside Pattern, it is common to use a distributed cache that is shared across multiple services.



* Cache-aside pattern can improve performance of microservices architecture, by reducing the number of expensive database calls.

* To use the cache-aside pattern in a microservice, need to implement a cache layer in your service

* Involve using a cache library or framework, such as Redis or Memcached, or implementing a custom cache solution.

Drawbacks of Cache-Aside Pattern for Microservices

* Cache can introduce additional complexity and may not be suitable for all situations.

* The cache may need to be invalidated or refreshed when data is updated in the database or data store

* This can require additional coordination between the microservices.

* The cache may introduce additional latency if it is located remotely from the microservices that are using it.

What is Proxy and Decorator Patterns

Proxy Pattern:

* Provides a placeholder for another object to control access to it. This pattern creates a proxy object that serves as an intermediary for requests intended for the original object.

* Lazy loading, controlling access, logging... It's like having a gatekeeper; add extra behavior or checks before accessing the actual object

Decorator Pattern:

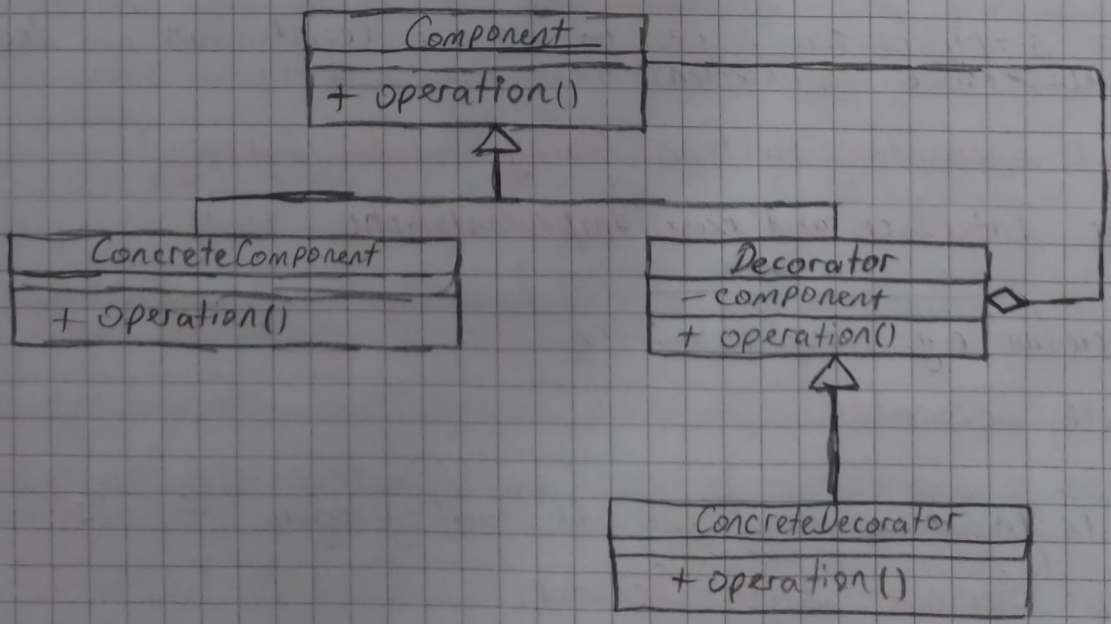
* Dynamically adds behavior to an object without altering it's structure. It involves a set of decorator classes that are used to extend the functionality of the original class without changing it's code.

* Useful for adding functionalities to objects at runtime.

* For example, enhancing a window object with additional features like borders, scrollbars dynamically.

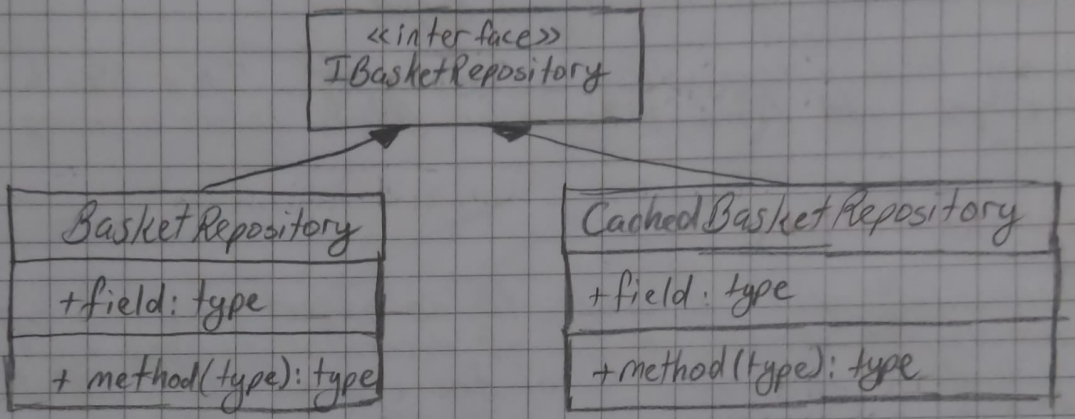
Implementation

* Create abstract decorators that implement the same interface as the object they will extend. Then, concrete decorator classes add additional behavior.



Example

* If you have a BasketRepository class for database operations, you can create a CachedBasketRepository that extends the BasketRepository with caching capabilities.



What is Scrutor Library? Why We Use It?

Scrutor Library:

* .NET library that extends the built in IOC container of ASP.NET Core. It provides additional capabilities to scan and register services in a more flexible way.

Usage:

Implementing patterns like Decorator in a clean and manageable way. It simplifies the process of service registration and decoration in ASP.NET Core applications.

Implement Decorator Pattern Using Scrutor Library

1. Define Interface and Base Implementation:

* Start with an interface, e.g., IBasketRepository, and a base implementation, e.g., BasketRepository

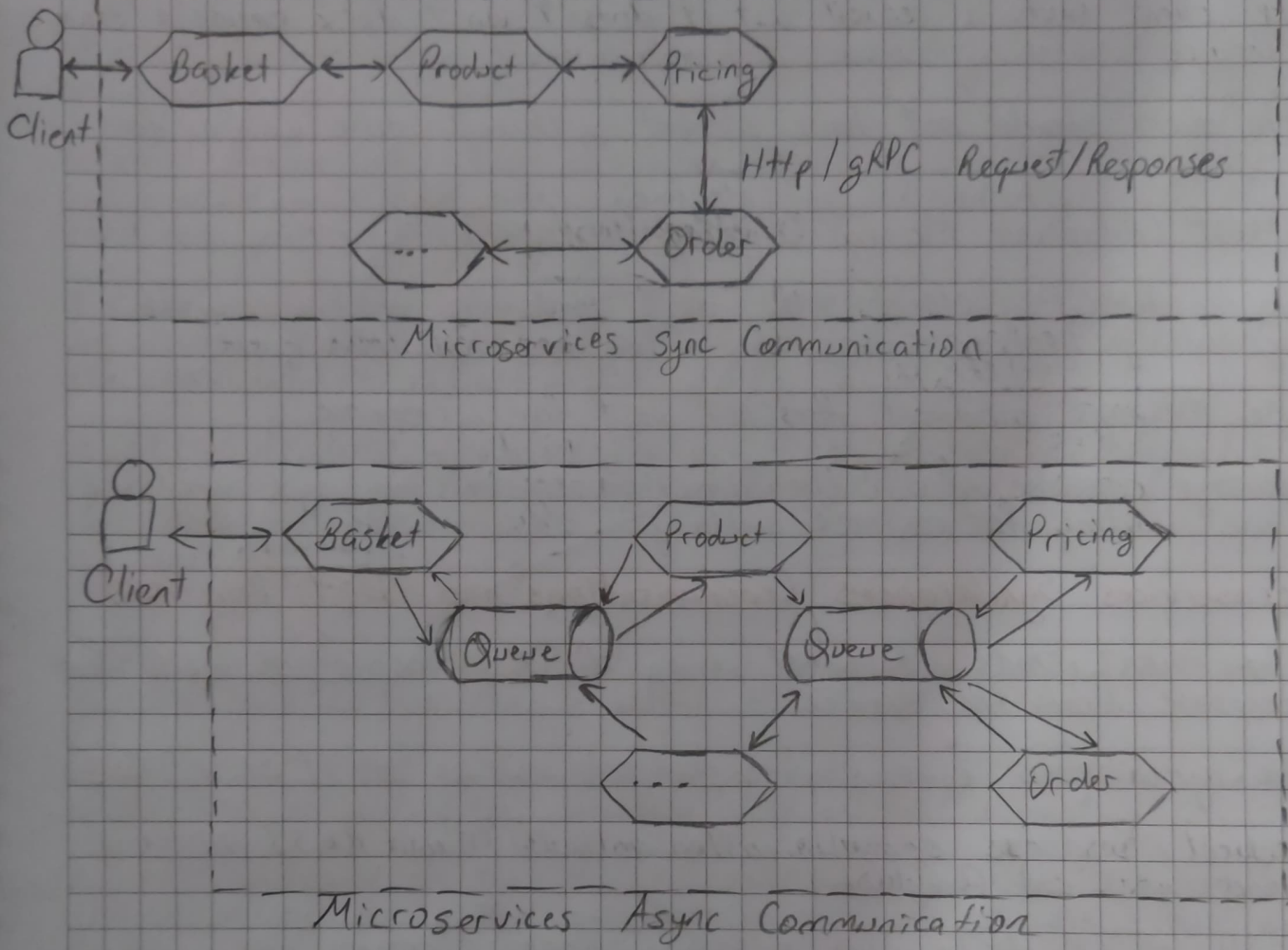
2. Create Decorator:

* Create a decorator class like CachedBasketRepository that also implements IBasketRepository but adds caching logic.

3. Register Services With Scrutor:

* Use Scrutor in Program.cs to first register the base service (BasketRepository) and then apply the decorator (CachedBasketRepository).

Microservices Communication Types - Sync or Async



Microservices Synchronous Communication

* Synchronous communication is using HTTP or gRPC protocol for returning synchronous response
* The client sends a request and waits for a response from the service
* The client code block their their thread, until the response reach from the server
* The synchronous communication protocols can be HTTP or HTTPS
* The client sends a request with using http protocols and waits for a response from the service
* The client call the server and block client their operations
* The client code will continue it's task when it receives the HTTP server response

Microservices Asynchronous Communication

* The client sends a request but it doesn't wait for a response from the service
* The client should not have blocked a thread while waiting for a response
* AMQP (Advanced Message Queuing Protocol)
* Using AMQP protocols, the client sends the message with using message broker systems like Kafka and RabbitMQ queue
* The message producer does not wait for a response
* Message consume from the subscriber systems in async way, and no one waiting for response suddenly

Microservices Synchronous Communications and Best Practices

* The client sends a request using HTTP protocols and waits for a response from the service
* The synchronous communication protocols can be HTTP or HTTPS
* Request / Response communication with HTTP and REST protocol (extends gRPC and GraphQL)
* REST HTTP APIs when exposing from microservices
* gRPC APIs when communicate internal microservices
* GraphQL APIs when structured flexible data in microservices
* WebSocket APIs when real-time bi-directional communicaton

gRPC: High Performance Remote Procedure Calls

* gRPC is an open source remote procedure call (RPC) system developed by Google
* gRPC is framework to efficiently connect services and build distributed systems
* It is focused on high performance and uses the HTTP/2 protocol to transport binary messages
* It relies on the Protocol Buffers language to define service contracts
* Protocol Buffers (Protobuf), allow to define the interface to be used service-to-service communication regardless of the programming language

* It generates cross-platform client and server bindings for many languages

* Most common usage scenarios include connecting services in micro-services style architecture

* gRPC framework allows developers to create services that can communicate with each other efficiently and independently with their preferred programming language

* Once you define a contract with Protobuf, this contract used by each service to automatically generate the code that sets up the communication infrastructure

* This feature simplifies the creation of service interaction and together with high performance, makes gRPC the ideal framework for creating microservices.

How gRPC Works?

* RPC is a form of client-server communication that uses a function call rather than a usual HTTP call

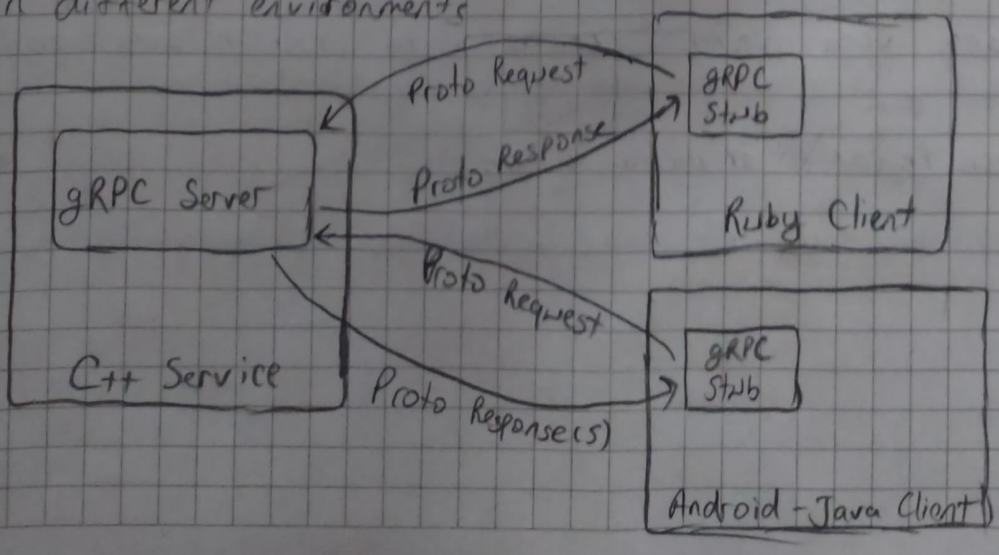
* In gRPC, client application can directly call a method on a server application on a different machine like a local object.

* gRPC is based on the idea of defining a service and methods, and these can be called remotely with their parameters and return types

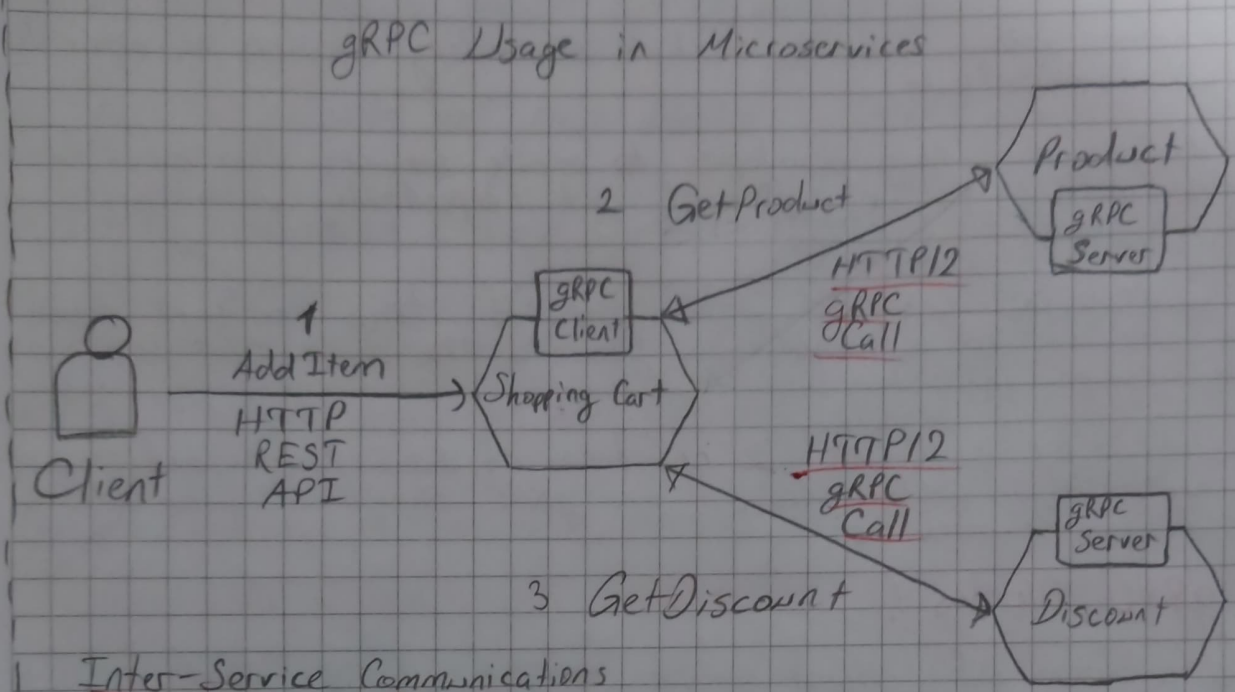
* On the server side, the server implements this interface and runs a gRPC server to handle client calls

* On the client side, the client has a stub that provides the same methods as the server

* gRPC clients and servers can work and talk to each other in different environments



gRPC Usage in Microservices Communication



Inter-Service Communications
Backend microservices requires performance with gRPC

Developing Discount.Grpc Microservice

* ASP.NET gRPC API application
* Build highly performant inter-service gRPC communication with Discount and Basket microservice
* gRPC Communications, Proto files CRUD operations
* Exposing gRPC services creating Protobuf message
* SQLite database connection and containerization
* Entity Framework Core ORM - SQLite Data Provider and migrations to simplify data access and ensure high performance
* N-layer architecture implementation
* Containerize Discount microservice with SQLite database using docker-compose

Main Idea of Discount.Grpc Microservice

Use Case: Add Item to Shopping Cart

* When client add item into shopping cart, Basket microservice will consume this discount grpc services, to get latest discounts on that product item

Main Idea: Be performant as much as possible

* Achive maximum performance and reduce service invocation time
* Ensuring the Discount microservice responds quickly to synchronous calls from the Basket service.