.NET Backend Bootcamp

Modular Monoliths (Modulith)

Vertical Slice Architecture (VSA)

Domain-Driven Design (DDD)

Command Query Responsibility Segregation (CQRS)

Outbox Pattern for Reliable Messaging

Hands-on experience building a complete EShop Modular Monolith

Mehmet Ozkaya

Big Picture – Course Project

Catalog Module Internal Architecture

Basket Module Internal Architecture

Sync Communications with In-process Method Calls

Async Communications w/ RabbitMQ & MassTransit

Identity Module Internal Architecture

Ordering Module Internal Architecture

Outbox Pattern For Reliable Messaging

Migrate to Microservices: EShop Modules to Microservices

Big Picture – Course Project

.NET Backend Bootcamp: Modulith, VSA, DDD, CQRS and Outbox

Modular Monoliths on .NET platforms which used Asp.Net Web API, PostgreSQL, Redis, RabbitMQ, Keycloak, Seq, Docker, MassTransit, Entity Framework Core, CQRS, MediatR, Carter, DDD, Vertical Slice Architecture and Outbox Pattern implementation.

Mehmet Ozkaya

Traditional Monoliths

- **Monolithic** architectures were **the standard**.
- **All components** in **single**, **indivisible unit**, everything is **centralized**.
- As **apps** grew in **size** and **complexity,** the **challenges multiplied.**
- Updating a single part requires **redeploying** the **entire application**.
- Scaling requires **the entire application**.

Rise of Microservices

- **Microservices** architecture **emerged** as a **paradigm shift.**
- **Independently deployable services**, run own process and **communicate** via **lightweight mechanisms**.
- **Easier** to **update** and **scale parts** of an app **independently** without impacting **the whole.**
- **Leads** to **complexities** in managing **network latency**, **data consistency**, and **fault tolerance**.
- **Complicating** the **development environment** and **operational overhead**, including **monitoring**, **logging**, and **security**.

The Bridge: Modular Monoliths Architecture (Modulith)

- Emerged as a **compelling middle path**, Harmonizes **benefits** of **monolithic** structure and **microservices agility**.
- **Monolith** into **well-defined**, **loosely coupled** modules, that are **easier to develop** and **maintain**, **simplicity** of **deploying** a single app.
- **Simplifies data management** and **transaction integrity**, Eliminates **network communication** complexity.
- **Start** with a **modular monolith** and **gradually evolve** into **microservices.**
- **The rise of modular monoliths boost** in new software projects for **optimal software architectures.**

Big Picture – Course Project

Catalog Module Internal Architecture

Basket Module Internal Architecture

Sync Communications with In-process Method Calls

Async Communications w/ RabbitMQ & MassTransit

Identity Module Internal Architecture

Ordering Module Internal Architecture

Outbox Pattern For Reliable Messaging

Migrate to Microservices: EShop Modules to Microservices

Big Picture – Course Project

Eshop Modulith Project

Architectures

- DDD
- CQRS
- Vertical Slide Architecture
- Modular Monoliths Architecture

Patterns&Principles

- SOLID
- Dependency Injection
- Mediator, Decorator, Options
- Pub/Sub, Caching
- Outbox

Backing Services

- Postgres
- Redis
- RabbitMQ
- Keycloak
- Seq

Libraries

- Carter
- MediatR
- Mapster
- MassTransit
- FluentValidation
- EF Core, Serilog

Authentication

- Securing Apis with Keycloak OpenID Connect with Bearer Tokens

Communications

- Sync: In-process Method Calls (Public APIs)
- Async: Publish/Subscribe pattern w/ MassTransit and RabbitMQ

Course Target

- To be a **master of .NET Backend Development** with **patterns, principles** and best **practices.**

- Designed for **software developers** and **architects**
- **%95 of Hands-on Design and Development** Activities
- Apply **best practices** with **cloud-native design patterns** and **principles**
- **Latest features** of **C# 12, .Net 8** and **Asp.Net 8** like Minimal Apis, primary constructors and so on.

- **Solid understanding** of the **.NET Modular Monoliths** and will be ready to **design, develop,** and **deploy** your own **enterprise applications.**

Prerequisites

- .Net 8
- Visual Studio
- Docker Desktop
  - At least 7 GB of free space
- Postman

- Install & Download before starting course

Source Code on GitHub

- **GitHub EShop Modulith Repository**
  - **https://github.com/mehmetozkaya/EshopModularMonoliths**
- **Section by section GitHub EShop Modulith Repository**
  - **https://github.com/mehmetozkaya/EshopModularMonoliths-Sections**

Course Slides

- Download Course Slides from **This Lecture Resources**

Big Picture – Course Project

DEMO: Run Final Application

- Clone Microservices Repository: https://github.com/mehmetozkaya/EshopModularMonoliths

- Docker-compose up
- Follow steps on GitHub documentation
- Run the Project Section
- DEMO

Way of Development: Shared -> Modules -> API

Development Path

Module Development Path

Project Code Structure

External vs. Internal Architecture

EShop Modules Internal Architecture

Architectures, Patterns, Libraries and Best Practices

Architectures

- DDD
- CQRS
- Vertical Slide Architecture
- Modular Monoliths Architecture

Patterns&Principles

- SOLID
- Dependency Injection
- Mediator, Decorator, Options
- Pub/Sub, Caching
- Outbox

Backing Services

- Postgres
- Redis
- RabbitMQ
- Keycloak
- Seq

Libraries

- Carter
- MediatR
- Mapster
- MassTransit
- FluentValidation
- EF Core, Serilog

Authentication

- Securing Apis with Keycloak OpenID Connect with Bearer Tokens

Containerization

- Dockerfiles and Docker Compose

Communications

- Sync: In-process Method Calls (Public APIs)
- Async: Publish/Subscribe pattern w/ MassTransit and RabbitMQ

How to Follow the Course (The Course Flow)

- **Creating the Solution Structure**
  - Wiring up Dependencies Between Modules
- **Developing Modules**
  - Catalog, Basket, Identity and Ordering Module Developments
- **Communications Between Modules**
  - Sync Communications
  - Async Communications

Way of Learning (For Each Modules)

Traditional Monoliths (Monolithic Architecture)

Whats wrong with Traditional Monoliths ?

Mehmet Ozkaya

What is Monolithic Architecture ?

- **Traditional way** of building applications.
- All components and functionalities are **packaged into a single**, **cohesive unit**. (user interface, business logic, and data access)
- Build **everything together** and **deploy** it on the server: **tightly coupling.**
- **Legacy applications** are mainly implemented as a **monolithic architecture**.
- All application concerns are **contained** in a **single big deployment.**
- Can design in **different layers** like **presentation**, **business** and **data layer.**
- Some cases Monolithic **still best option** to design your app, especially when you **required** to **strong consistency**.

Characteristics of Monolithic Architecture

Single Codebase

- Easier to manage and deploy initially.
- No separation between different parts of the app.

Single Deployment Unit

- Entire app is packaged and deployed as a single unit.
- Simplify deployment processes, especially in the early stages of development.

Tight Coupling

- Changes in one part of the application can affect others.
- Increased complexity and difficulty in making changes or scaling specific parts of the application.

Shared Database

- Uses a single, shared database for all its data needs.
- Challenges in maintaining and scaling the database as the application grows.

When to use Monolithic Architecture

- If you are **building small application**, still **monolithic architecture** is **one of the best architecture**.

They're straightforward to:

- Build
- Test
- Deploy
- Troubleshoot
- Scale vertically (scale up)

- **Simple to develop** relative to microservices
- **Easier to deploy** as only a single jar/war file

When to use Monolithic Architecture

- **Small team at Founding Stage**
  If you are a startup and your team is small like 2 to 5 members, don't need to deal with the complexity of the high-overhead microservices architecture.
- **Simple application with Predictable Scale and Complexity**
  If your app doesn’t require advanced scalability and the complexity is manageable, then a monolith architecture is the best option to start.
- **Proof of Concept and Quick Launch**
  Build a proof of concept, like testing a new idea on market, that means your new products will pivot and evolve a lot over time, when you figure out what will be useful to your users.
- **No Microservices Expertise**
  If your team has no prior experience with microservices architecture, that will really hard to ship your application effectively and timely.

Request per Second and Acceptable Latency

Concurrent Users	Requests/second	Latency (Expected)
2K	0.5K	
20K	12K	
100K	80K	<= 2 sec
500K	300K	?

What's wrong with traditional monolitic architecture?
Lose control monolitic architecture
Lose control to manage , scale, and update
Hard to manage to manage, and update.

Scalability Issues of Traditional Monolithic

Difficult to Scale

- Entire application is a single unit.
- When one part of the app needs to scale, the entire app must be scaled.
- Can't scale components independently, the only option is the scaling the whole app.
- If the user authentication component get high load, you can't scale just that component. Scaling the entire app leading to inefficient resource utilization.
- Scaling a monolithic app can be resource-intensive and costly. End up using more resources than necessary.

Development Bottlenecks of Traditional Monolithic

Complex Codebase: Become Complex over time

- Becomes too large with time and difficult to manage.
- App grows with adding new functionalities, codebase can become extremely large and complex.
- Becomes too complicated to understand, developers working on the individual codebase expands.
- Become so complicated that no single person understands it.

Tight Coupling

- Changes in one part of app can have unintended consequences in other parts.
- Difficult to make changes, as need to thoroughly test the entire app to ensure nothing breaks.

Development Bottlenecks of Traditional Monolithic

Hard to Making New changes and Slower development speed:

- Any code change affects the whole system.
- Makes you fear when making new changes. And these changes brings costly side effects.
- Makes the overall development process much longer.
- Develop new features become time-consuming and expensive to implement.
- Even the smallest change requires full deployment of the entire app - it is expensive and risky.
- Not reliable because a single bug can bring down the whole monolithic app.

Deployment Risks of Traditional Monolithic

Single Point of Failure

- Entire app is deployed as a single unit, any issue during deployment can bring down the entire system.

Difficult Rollbacks

- If a deployment fails, you must roll back the entire app, which can be a complex and risky process.

Frequent Downtime

- Deploying updates to a monolithic app often requires downtime.
- This can impact the user experience, especially if updates are frequent.

Technology Lock-In : Barrier to new technology adoption

- It is extremely **problematic** to **apply a new technology** in a monolithic app.
- The entire app **has to be re-development** due to the **interlocking dependencies** found in a monolith.
- If you use a **different database** technology for a component, it is **difficult to make this change** without affecting the entire app.
- **Any changes** in the framework or language affects the entire app, making changes often **expensive** and **time-consuming**.
- We can say this condition as the **Fear Cycle**. It's stressful to change your full code base with newest technology.
- Instead of building new or innovative solutions, most of your time is **spent maintaining legacy apps**.
- Responding to **new business requirements** or market changes **can be slow**.

Maintenance Challenges of Traditional Monolithic

Technical Debt

- As the application grows, it’s easy to accumulate technical debt. This includes outdated code, poor documentation, and quick fixes that compromise code quality.

Bug Fixes and Enhancements

- The risk of introducing new bugs while fixing existing ones is high, requiring extensive testing.

Team Coordination

- Merging changes from different developers can lead to conflicts and delays.

Big Ball of Mud

- Software system with a **messy**, **poorly structured** architecture. The code is so entangled that it’s hard to understand, maintain, or extend.
- **No clear structure** or **organization**. Everything is mixed together, making it hard to figure out where one part ends and another begins.
- Over time**, quick fixes** and **shortcuts add up**, creating **technical debt**. This makes the code messy and difficult to work with.
- In a **rush to deliver features quickly**, best practices are often ignored. This leads to a **messy codebase** full of shortcuts.
- **Without proper design** and **planning**, the system evolves haphazardly. Design decisions are made on the fly, leading to inconsistencies.
- Over time, **code needs to be cleaned up** and **improved**. If this doesn’t happen regularly, the codebase becomes hard to manage.

Microservices Architectures

Why it is not fit for all projects ?

What’s wrong with Microservices Architecture ?

Mehmet Ozkaya

What are Microservices ?

- **Microservices** are **small**, **independent**, and **loosely coupled** services that can work together.
- Each service is a **separate codebase**, which can be managed by a small development team.
- Microservices **communicate** with each other by using **well-defined APIs.**
- Microservices can be **deployed independently** and **autonomously**.
- Microservices can work with many different technology stacks which is **technology agnostic**.
- Microservices has its **own database** that is not shared with other services.

What is Microservices Architecture ?

- From Martin Fowlers Microservices article;

*The microservice architectural style is an approach to developing a single application as a **suite of small services**, each running in **its own process** and **communicating** with lightweight mechanisms, often an **HTTP** or **gRPC API**.*

- Microservices are **built around business capabilities** and **independently deployable** by **fully automated** deployment process.
- Microservices architecture **decomposes** an application into **small independent services** that communicate over **well-defined APIs**. Services are owned by **small**, **self-contained teams**.
- Microservices architecture is a **cloud native architectural approach** in which services composed of many **loosely coupled** and **independently deployable** smaller components.
- Microservices have their **own technology stack**, communicate to each other over a combination of **REST APIs**, are organized by **business capability**, with the **bounded contexts**.
- Following **Single Responsibility Principle** that referring separating responsibilities as per services.

Microservices Characteristics

- From Martin Fowlers Microservices article;
- Componentization via Services
- Organized around Business Capabilities
- Products not Projects
- Smart endpoints and dumb pipes
- Decentralized Governance
- Decentralized Data Management
- Infrastructure Automation
- Design for failure

When to Use Microservices Architecture

- **Make Sure You Have a “Really Good Reason” for Implementing Microservices**
  Check if your application can do without microservices. When your application requires agility to time-to-market with zero-down time deployments and updated independently that needs more flexibility.
- **Iterate With Small Changes and Keep the Single-Process Monolith as Your “Default”**
  Sam Newman and Martin Fowler offers Monolithic-First approach. Single-process monolithic application comes with simple deployment topology. Iterate and refactor with turning a single module from the monolith into a microservices one by one.
- **Required to Independently Deploy New Functionality with Zero Downtime**
  When an organization needs to make a change to functionality and deploy that functionality without affecting rest of the system.
- **Required to Independently Scale a Portion of Application**
  Microservice has its own data persistence. Data consistency can be a challenge. Follow eventual consistency where possible.

When to Use Microservices Architecture -2

- **Data Partitioning with different Database Technologies**
  Microservices are extremely useful when an organization needs to store and scale data with different use cases. Teams can choose the appropriate technology for the services they will develop over time.
- **Autonomous Teams with Organizational Upgrade**
  Microservices will help to evolve and upgrade your teams and organizations. Organizations need to distribute responsibility into teams, where each team makes decisions and develops software autonomously.

When Not to Use Microservices
Anti-Patterns of Microservices

- **Don't do Distributed Monolith**
  Make sure that you decompose your services properly and respecting the decoupling rule like applying bounded context and business capabilities principles.
- Distributed Monolith is the worst case because you increase complexity of your architecture without getting any benefit of microservices.
- **Don't do microservices without DevOps or cloud services**
  Microservices are embrace the distributed cloud-native approaches. And you can only maximize benefits of microservices with following these cloud-native principles.
  - CI/CD pipeline with devops automations
  - Proper deployment and monitoring tools
  - Managed cloud services to support your infrastructure
  - Key enabling technologies and tools like Containers, Docker, and Kubernetes
  - Following asnyc communications using Messaging and event streaming services

When Not to Use Microservices

- **Limited Team sizes, Small Teams**
  If you don't have a team size that cannot handle the microservice workloads, This will only result in the delay of delivery.
- For a small team, a microservice architecture can be hard to justify, because team is required just to handle the deployment and management of the microservices themselves.
- **Brand new products or startups**
  If you are working on a new startup or brand new product which require significant change when developing and iterating your product, then you should not start with microservices.
- Microservices are so expensive when you re-design your business domains. Even if you do become successful enough to require a highly scalable architecture.
- **The Shared Database anti-pattern**

Monolithic vs Microservices Architecture Comparison

- **Application Architecture**
  Monolith has a simple straightforward structure of one undivided unit. Microservices have a complex structure that consists of various heterogeneous services and databases.
- **Scalability**
  Monolithic application is scaled as a whole single unit, but microservices can be scaled unevenly. encourages companies to migrate their applications to microservices.
- **Deployment**
  Monolithic application provides fast and easy deployment of the whole system. Microservices provides zero-downtime deployment and CI/CD automation.
- **Development team**
  If your team doesn't have experience with microservices and container systems, building a microservices-based application will be difficult.

Architecture Comparison

Deployment Comparison

Monolithic

Microservices

Increased Complexity of Microservices Architecture

Distributed System

- Managing and coordinating multiple services can be complex.
- There is more moving parts than the monolithic application.
- Each service is simpler, but the entire system is more complex.

Network Latency and Failures

- Communicate over the network, which introduces latency. Network failures can disrupt communication between services.
- Handling network issues requires robust retry and fallback mechanisms, adding to the complexity.

Data Management of Microservices Architecture

Data Consistency

- Ensuring data consistency across multiple services and databases is difficult.
- Traditional ACID transactions are hard to implement in a distributed system.
- Eventual consistency and distributed transactions (e.g., using Saga patterns) are often necessary but add complexity.

Data Duplication

- Data may need to be duplicated across services but this can lead to data inconsistency and increased storage costs.
- Keeping duplicated data in sync requires additional mechanisms, such as event sourcing or change data capture.

Operational Overhead of Microservices Architecture

Deployment and Management

- Deploying multiple services is more complex than handling a single monolithic application.
- CI/CD pipelines need to be set up for each service, increasing maintenance efforts.

Monitoring and Logging

- Become more complex as you need to aggregate data from multiple services.
- E2E visibility and tracing requests across services requires sophisticated monitoring tools.

DevOps Skills

- Invest in quite a lot of devops automation processes and tools.
- DevOps expertise is essential for automating deployments, scaling services, and maintaining infrastructure.

Inter-Service Communication of Microservices Architecture

Communication Protocols

- Choosing the right communication protocol (HTTP, gRPC, messaging) is critical but each protocol has its trade-offs.
- Chain of services calls increase latency problems, need correct design to APIs for proper communication.

Service Discovery

- Service discovery mechanisms is necessary but adds complexit
- Consul, Eureka, and Kubernetes' built-in service discovery, but they require proper configuration and management.

Fault Tolerance

- Handle failures gracefully is crucial. Develop circuit breakers, retries, and fallbacks adds to the development effort.
- Service failures requires careful planning and robust infrastructure.

Testing and Debugging of Microservices Architecture

Integration Testing

- Hard to develop and testing these E2E processes that touch on several microservices.
- Ensuring that all services work together as expected requires extensive integration testing.
- Automated tests and staging environments are necessary to catch issues before they reach production.

Debugging Distributed Systems

- Tracing a problem across multiple services and logs requires good observability tools.
- Distributed tracing tools like Jaeger and Zipkin can help, but they need to be properly integrated and maintained.
- Remote debugging through your local IDE isn't an option and it won't work across dozens or hundreds of services.

Team Coordination of Microservices Architecture

Team Isolation

- Teams focusing only on their own services can lead to a lack of understanding of the overall system and potential integration issues.
- Encouraging cross-team communication and collaboration is essential to avoid isolation.

Coordination Overhead

- Changes to shared contracts or APIs require careful planning and communication.
- Managing dependencies between services and ensuring backward compatibility adds to the coordination overhead.

Fallacies of Distributed Computing

Set of **common misconceptions** that developers and architects may have about the **behavior** of **distributed systems**.

First identified by Peter Deutsch and others at Sun Microsystems.

8 Fallacies of Distributed Computing

- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology doesn't change
- There is one administrator
- Transport cost is zero
- The network is homogeneous

Fallacy 1 - The Network is Reliable

- **Assumption:** The network will always be reliable.
- **Reality:** Networks are inherently unreliable.
- Packets can be lost, delayed, or corrupted. Network partitions can occur, causing services to become temporarily unreachable.
- Microservices rely heavily on network communication. Assuming the network is always reliable can lead to serious issues, such as lost messages or timeouts.
- Implement robust error handling, retries, and circuit breakers. Use monitoring and alerting to detect network issues early.

Fallacy 2 - Latency is Zero

- **Assumption:** Network latency is negligible.
- **Reality:** Network latency can be significant where multiple services communicate over a network that can affect the performance and responsiveness of the system.
- High latency can slow down interactions between microservices, impacting the overall performance of the application.
- Optimize communication patterns, minimize the number of remote calls, and use caching to reduce the impact of latency.

Fallacy 3 - Bandwidth is Infinite

- **Assumption:** Bandwidth is unlimited.
- **Reality:** Network bandwidth is limited and can be a bottleneck, especially when transferring large amounts of data between services.
- Limited bandwidth can cause delays and affect the performance of data-intensive microservices.
- Compress data, use efficient serialization formats, and minimize the amount of data transferred over the network.

Fallacy 4 - The Network is Secure

- **Assumption:** The network is inherently secure.
- **Reality:** Networks are vulnerable to various security threats, such as eavesdropping, man-in-the-middle attacks, and unauthorized access.
- Assuming network security can expose microservices to data breaches and attacks.
- Implement encryption (e.g., TLS) for data in transit, use secure authentication and authorization mechanisms, and regularly update security practices.

Fallacy 5 - Topology Doesn’t Change

- **Assumption:** The network topology is static and doesn’t change.
- **Reality:** Network topology can change due to scaling, failures, and reconfiguration. Services can move, IP addresses can change, and new nodes can be added or removed.
- Changes in network topology can disrupt communication between microservices.
- Use service discovery mechanisms to dynamically locate services, and design microservices to handle changes in network topology gracefully.

Fallacy 6 - There is One Administrator

- **Assumption:** There is a single administrator managing the network.
- **Reality:** In large distributed systems, multiple administrators and teams are involved, leading to potential miscommunication and misconfiguration.
- Coordination between different teams managing various microservices can be challenging, leading to configuration and deployment issues.
- Establish clear communication channels, use centralized configuration management, and automate deployment processes.

Fallacy 7 - Transport Cost is Zero

- **Assumption:** The cost of transporting data is negligible.
- **Reality:** Transporting data over the network incurs costs, including bandwidth usage and latency, which can impact performance and expenses.
- Ignoring transport costs can lead to inefficient communication patterns and higher operational costs.
- Optimize data transfer, use efficient communication protocols, and monitor network usage to manage costs effectively.

Fallacy 8 - The Network is Homogeneous

- **Assumption:** The network is uniform and consistent.
- **Reality:** Networks are heterogeneous, with varying performance, protocols, and configurations across different segments.
- Variability in network performance and configuration can lead to inconsistent behavior and performance issues in microservices.
- Design microservices to be resilient to network variability, use abstraction layers to hide network differences, and test services in different network conditions.

Microservices Antipattern: The Distributed Monolith

- **Distributed monolith** is an **antipattern** where a supposedly microservices-based system **retains the drawbacks** of a **monolithic** architecture.
- **Services** are **highly dependent** on each other. Changes in one service often **require changes in others**, making **independent deployments** nearly impossible.
- **Synchronized Deployments:** Because of the interdependencies, all services need to be deployed together. Negates benefits of microservices, which is the ability to deploy independently.
- **Shared Databases:** Multiple services share the same database schema, leading to tight coupling at the data layer. Difficult to change the database schema without affecting all the services.
- **Communication Overload:** Frequent synchronous calls between services can lead to performance bottlenecks.

Why The Distributed Monolith Happen ?

Inadequate Service Boundaries

- Poorly defined service boundaries often lead to services that are too interdependent.

Lack of Domain-Driven Design

- Without understanding of the domain and its subdomains, it’s hard to define services that are cohesive and loosely coupled.

Database Coupling

- Sharing a single database schema across services leads to tight coupling.

Legacy Code and Practices

- Migrating from a monolithic application to microservices without changing legacy code and practices can result in a distributed monolith.

Why The Distributed Monolith Happen ?

Amazon Case: Rearchitecting from a Distributed Microservices to a Monolith

Video Streaming

Scaling up the Prime Video audio/video monitoring service and reducing costs by 90%

The move from a distributed microservices architecture to a monolith application helped achieve higher scale, resilience, and reduce costs.

https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90

The Bridge: Modular Monolith (Modulith) Architectures

Gateway between Monolithic and Microservices

Mehmet Ozkaya

Modular Monolithic (Modulith) Architecture

- App is developed as a **single**, **cohesive unit** but is **internally divided** into **well-defined**, **loosely coupled** modules.
- Maintain the **simplicity** of a **monolith** also **benefits** of **modularity** typically associated with microservices.
- Breaks up the code into **independent modules** and each **module encapsulates** their **own features** needed in your app.
- **Still build** and **deploy** a **single app**, breaking up the code into **independent modules**.
- **Modules** are represents **Bounded Context** of our application domain and **group features** of **Domain contexts**.
- **Reduces** the **dependencies** of a **module** and we can **develop** or **modify** a **module** without it **effecting other modules**.

Characteristics of Modular Monolithic (Modulith)

Modularity and Bounded Contexts

- App is divided into several modules, each encapsulate specific business domain. Communicate well-defined interfaces, reducing dependencies and enhancing maintainability.

Simplified Deployment and Operations

- Unlike microservices, MMA are deployed as a single unit, reduces operational complexity and eliminates issues arising from service interdependencies.

Inter-Module Communication

- Communicate via direct method calls or event mechanisms within the same application process. Eliminates the network latency like microservices.

Unified Data Management

- Segmented by schemas or tables specific to each module, enabling ease of data management while maintaining the integrity.

EShop Modular Monoliths
API
Modules
Catalog
Public API
Basket
Public API
Public API
Ordering
Public API
Identity
PostgreSQL
Class Library
Modules
Catalog
Basket
Ordering
Shared
Shared
.NET 8
Module A
In-process Method Calls
Module B - Public API
Public API
Module B Contracts
Module B

Current Trends: The Resurgence of the Modular Monolith

- Industry continues to wrestle with the **trade-offs** between **monolithic** and **microservices**, a **middle path** has started to gain favour: **the modular monolith (modulith)**.
- The **need** to **balance** the **simplicity** of **monoliths** with the **flexibility** of **microservices**.
- **Resurgence** as businesses seek **efficient** and **scalable** solutions **without** the **complexities** of **microservices**.
- **Modular monoliths** serve as a **gateway** between **traditional monoliths** and **microservices**.
- **Combine** the **best aspects** of both architectures, offering a **balanced approach** to modern application development.
- Harmonize the **benefits** of a **monolithic structure** with the **agility** of **microservices**.
- Each **module** is **designed around** a **business domain**.

Advantages Over Microservices

- **Eliminates** much of the complexity associated with **network communications** in microservices, as all components run within the **same process**.
- **Coordinated database schema** simplifies data management and ensures **transaction integrity**.
- Ensures **data consistency** without the need for distributed **transaction management**.
- While not as granular as in microservices, **scalability** can still be effectively managed through load balancers, and by **optimizing individual modules**.
- Allows for **targeted scaling** of **specific modules** that require more resources.
- **Running** all **components** within the **same process** avoids the complexities of managing a distributed system, such as service discovery, network latency, and inter-service communication.

Monolith First Approaches, Martin Fowler

- Martin Fowler: Monolithic First, 2015
- https://martinfowler.com/bliki/MonolithFirst.html
- **Successful microservice** stories have **started with a monolith**
- A system that was **built as a microservice** system from scratch, it has ended up in **serious trouble.**
- **Microservices** are a useful architecture, but it incurs a **significant complexity**, which means they are only **useful with more complex systems.**
- Consider **Yagni**. When you begin a new application, how sure are you that it will be useful to your users?
- Starting with microservices works well if you come up with good, stable boundaries between the services which is **BoundedContexts**.

Monolith First Approaches, Sam Newman

- **Sam Newman — Building Microservices, 2nd Edition**
- https://samnewman.io/books/building_microservices/
- He agrees with the Martin Fowler approach which is recommend **beginning development** of new projects and systems as a **single deployable unit — the monolith.**
- Leveraging microservices only if you can **become convinced** of the benefits for your system, not as a default for every project.
- *A monolithic architecture is a choice, and a valid one at that. I'd go further and say that in my opinion it is the sensible default choice as an architectural style. In other words, I am looking for a reason to be convinced to use microservices, rather than looking for a reason not to use them.*

Monolith First Approaches, Mehmet Ozkaya

- Monolithic architecture does not means that the system is **poorly designed or bad.** Instead
- If you follow Modular approach, It will comes with lots of similar benefits with microservices like re-usability, easy to Refactor, Better organized dependencies and teams.
- So Should we always start with Monolith First Approaches ? **– NO**
- How we can decide ? If
- **Strong Consistency is Mandatory && Independent Scale and Deploy is not Required = Modular Monolithic**
- **Strong Consistency is not Mandatory, Eventual Consistency is OK && Independent Scale and Deploy is Required = Microservices**

Starting Point for Microservices: Modular Monoliths

- **Modular monoliths** provide a **more evolutionary** approach to software architecture.
- Organizations can **start with a modular monolith**, as needs change, **gradually spin off certain modules** as full-fledged **microservices**.
- Significant advantage for businesses **navigating uncertain markets** or those in the midst of digital transformation.
- Companies can **incrementally refactor** parts of the **monolith** into **microservices**, reducing risk and disruption.
- Focuses on **critical areas** that benefit most from microservices, ensuring efficient **resource utilization**.

Gradually Transition with Incremental Refactoring

Avoiding the Big Bang Approach

- Transitioning directly from a monolith to microservices can be risky and complex. It can disrupt the entire system.

Incremental Refactoring

- Gradually break down the monolith into microservices.
- This reduces risk, allows for continuous delivery, and ensures that the system remains stable throughout the transition.

Risk Mitigation

- Smaller, manageable changes reduce the risk of introducing critical issues by focusing on one module or service at a time.

Continuous Improvement

- Allows for ongoing improvements and adaptations based on feedback and changing requirements.

Steps for Incremental Refactoring: Modules to Microservice

Compare of Monolith, Microservices, Modular Monolith

	Architectures	Communications	Data Management	Transaction Management
Monolith	Traditional, single, unified application	Same process (In-process Method Calls)	Single Database	Single Transaction Scope
Microservices	Distributed architecture independent services	Inter-Service Communication over the network	Polyglot Persistence	Distributed Transactions, Eventual Consistency
Modular Monolith	Well-defined, loosely coupled modules	Same process (In-process Method Calls) – Public API	Shared or Coordinated Database – DB Schemas	Single Transaction Scope

Compare of Monolith, Microservices, Modular Monolith - 2

	Deployments	Development and Maintenance	Use Cases and Suitability
Monolith	Single Deployment Unit	Single Codebase, Tight Coupling	Small to Medium Applications
Microservices	Independent Deployment, Operational Complexity	Independent Codebases, Loose Coupling, Complexity in Coordination	Large, Complex Applications, Independent Teams
Modular Monolith	Single Deployment Unit, Modular Updates, Balanced Approach	Modular Codebase, Loose Coupling, Efficient Development	Growing Applications, Transitional Architecture, Balanced Approach

Shopify Modular Monolithic Architecture

- **Shopify** is one of the **largest e-commerce** platforms, especially during **peak events** like **Cyber Monday**.
- The core application, built on **Ruby on Rails**, has over **2.8 million lines** of code and **500,000 commits**, demonstrating its massive scale.
- As the **application grew**, the **complexity** and **feature set expanded** making it harder to manage.
- **Shopify formed a team** over three years ago to make their **Rails monoliths more modular** by creating smaller, **independent units** of code called components.
- **Shopify Engineering Teams** published articles:
- [Deconstructing the Monolith: Designing Software that Maximizes Developer Productivity]
- [Under Deconstruction: The State of Shopify's Monolith]

Shopify Modular Monolithic Architecture

Embracing the Majestic Monolith

- The **main application**, is a single large **Ruby on Rails monolith**.
- Internally, it is **split into multiple components** focused on **different business domains**.
- **Shopify** fully embraces the idea of a **Majestic Monolith**—most user-facing functionality by a single large **Ruby on Rails** application called "**Shopify Core.**"
- Shopify uses a **multi-layered testing approach**, starting with unit and integration tests.
- **Canary testing** is used during deployment, where changes are first deployed to a small subset of production instances.
- Shopify's **deployment pipeline** involves building images, deploying to **canaries**, **monitoring metrics**, and rolling out changes across the global fleet.

Decomposing Modules with Bounded Context Pattern

Bounded Context Pattern

- Bounded Context is a central pattern in Domain-Driven Design-DDD.
- Helps in identifying the boundaries within which a particular domain model applies.

Identifying Boundaries

- Each module in our system will represent a distinct Bounded Context, encapsulating specific business logic and data.
- These boundaries ensure that each module operates independently, reducing complexity and enhancing maintainability.

Identifying Eshop Boundaries

- Need to understand the core business domains of our EShop system.
- What are the main functionalities?
- How do different parts of the system interact?
- What are the key entities and processes?

Identifying EShop Modules

Create EShop Modular Monolith Project Solution Structure

Create GitHub Repository

Create New Blank Solution: eshop-modular-monoliths

MMA Solution Folder Structure: Bootstrapper API, Modules, Shared

Create Bootstrapper Api Project

Configure Empty Web Project to Bootstrapper API

Mehmet Ozkaya

Create EShop Modular Monolith Project Solution Structure

- Create GitHub Repository
- Create New Blank Solution: eshop-modular-monoliths
- MMA Solution Folder Structure: Bootstrapper, Modules, Shared
- Create Bootstrapper Api Project
- **Create Modules according to DDD Bounded Context Analysis**

Folder Structure: Bootstrapper, Modules, Shared

Identifying EShop Modules

Project Structure: Bootstrapper, Modules, Shared

External vs. Internal Architecture

EShop Modules Internal Architecture

Architectures, Patterns, Libraries and Best Practices

Architectures

- DDD
- CQRS
- Vertical Slide Architecture
- Modular Monoliths Architecture

Patterns&Principles

- SOLID
- Dependency Injection
- Mediator, Decorator, Options
- Pub/Sub, Caching
- Outbox

Backing Services

- Postgres
- Redis
- RabbitMQ
- Keycloak
- Seq

Libraries

- Carter
- MediatR
- Mapster
- MassTransit
- FluentValidation
- EF Core, Serilog

Authentication

- Securing Apis with Keycloak OpenID Connect with Bearer Tokens

Containerization

- Dockerfiles and Docker Compose

Communications

- Sync: In-process Method Calls (Public APIs)
- Async: Publish/Subscribe pattern w/ MassTransit and RabbitMQ

Vertical Slice Architecture

- Introduces by Jimmy Bogard offers this architecture **against to traditional layered/onion/clean architecture** approaches.
- Aims to **organize code around specific features** or **use cases**, rather than **technical concerns**.
- **Feature** is implemented **across all layers** of the software, from the user interface to the database.
- Often used in the development of **complex, feature-rich** apps
- **Divide** application into **distinct features** or **functionalities**, each of which cuts through all the layers of the application.
- **Contrast to traditional n-tier** or **layered architectures**, where the application is divided **horizontally** (e.g., presentation, business logic, data access layers).

Characteristics of Vertical Slice Architecture

- The application is divided **into feature-based slices**.
- **Each slice** is **self-contained** and **independent**.
- There are **reduced dependencies** between different parts of the application.
- It **promotes** the use of **cross-functional teams**.
- The architecture **supports scalability** and **maintainability**.
- It allows for **improved testing** and **deployment** processes.
- **Every microservice handles** a **specific piece of functionality** and communicates with other services through well-defined interfaces.

Benefits & Challenges of Vertical Slice Architecture

Benefits

- Focused development, teams can concentrate on one feature at a time.
- Simplifies refactoring and upgrades since changes in one slice usually don't affect others.
- Aligns well with Agile and DevOps practices, supporting incremental development and continuous delivery.

Challenges and Considerations

- Duplication of code across slices, particularly for common functionalities.
- Learning curve involved, especially for teams accustomed to traditional architectures.
- Design of each slice requires careful consideration to ensure independence and maintainability.

Clean Architecture

- Introduced by **Robert C. Martin**, also known as **Uncle Bob**.
- Software architecture design that aims to **separate concerns** and **create systems** that are **independent of frameworks**, **UI**, and **databases**.
- Leads to systems that are more **maintainable**, **flexible**, and **adaptable** to changes.

Key Principles of Clean Architecture

Independence of Frameworks:

- The system is not tightly coupled with a specific framework, which makes it adaptable to changes in frameworks and tools.

Testable:

- The business rules can be tested without the UI, database, web server or any external element.

UI Agnostic:

- The UI can change easily, without changing the rest of the system.

Database Agnostic:

- Business rules are not bound to the database, decoupled from the underlying data store.

External System Agnostic :

- Business rules don’t know anything about the outside world, isolated.

Clean Architecture Layers

Entities Layer (Domain Layer):

- Contains enterprise-wide business rules.
- Entities encapsulate the most general and high-level rules.
- Example: In an ordering microservice, entities might be Order, OrderItem, etc.

Use Cases Layer (Application Layer):

- Contains application-specific business rules.
- Encapsulates and implements all of the use cases of the system.
- Example: CreateOrder, CancelOrder, and UpdateOrderStatus use cases.

Clean Architecture Layers - 2

Interface Adapters Layer (Infrastructure Layer):

- Converts data from the format most convenient for the use cases and entities, to the format most convenient for external systems.
- Example: Mapping data from database models to domain entities.

Frameworks and Drivers Layer (Infrastructure/External Concerns):

- The outermost layer consisting of frameworks and tools such as the Database, the Web Framework, etc.
- Generally includes the UI, database, external interfaces, etc.
- Example: REST controllers, database repositories.

Clean Architecture

Clean Architecture: Modular, Adaptable, Testable and Maintainable

Domain Layer
Application Layer
Infrastructure Layer
Web UI Layer

Vertical Slice Architecture 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 applications** 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 Architecture

- **Rapid Development & Deployment:** When the priority is to develop and deploy features rapidly and independently.
- **Agile & Scrum Teams:** For teams practicing Agile or Scrum that need to deliver complete features in short cycles.
- **Microservices:** In a microservices architecture, where each service is often responsible for a distinct feature or business capability.

Project Structure: Bootstrapper, Modules, Shared

Shared Kernel Pattern in DDD

- **Small subset** of the **domain model** is **shared** between **multiple bounded contexts** or **modules**.
- Includes **common data structures**, **utility classes**, or services that **multiple parts of the system** need to use.
- The **Shared Kernel** ensures **consistency** and **reduces duplication** while **maintaining clear boundaries** between different modules.

Why Use the Shared Kernel Pattern ?

- **Consistency:** By sharing a common subset of the domain model, we ensure consistency across different modules.
- **Reduced Duplication:** Instead of duplicating common code in multiple modules, the Shared Kernel centralize this code in one place, reducing redundancy and the risk of inconsistencies.
- **Decoupling:** Enforces clear boundaries between modules. Only the minimal, necessary parts of the domain model are shared, keeping the rest of the module implementations decoupled.

Dependency Path of EShop Modular Monoliths

Solution Folder Structure of EShop Modular Monoliths

```
Solution
├── Bootstrapper
│   └── Api
├── Modules
│   ├── Catalog
│   │   └── Catalog.csproj
│   ├── Basket
│   │   └── Basket.csproj
│   └── Ordering
│       └── Ordering.csproj
└── Shared
    └── Shared.csproj
```

Wire-up Dependencies Between Modules (DI Management of Modules)

Asp.net Dependency Injection Management

Wiring up dependecies between modules

Create xxxModule classes

Api Project - Program.cs

Asp.net Middlewares and Http Request Pipeline

Mehmet Ozkaya

Wiring Up Dependencies Between Modules

Creating Module Classes

- Each module will have a class responsible for configuring its dependencies.

Registering Module Services

- Define extension methods to register services from each module in the DI container.

Configuring the API Project

- Call these extension methods in our Bootstrapper API project to wire up all dependencies.

Wiring Up Dependencies Between Modules

Asp.net Dependency Injection Management

What is Dependency Injection ?

- DI allows an object to receive its dependencies from an external source rather than creating them itself.
- Promotes loose coupling and enhances testability and maintainability.

Built-in DI Container in ASP.NET Core

- ASP.NET Core provides a built-in dependency injection container that supports the registration and resolution of services.
- DI container is configured in the Program.cs file.
- DI container handles the lifecycle of these services, ensuring they are instantiated and disposed of correctly.

Wiring Up Dependencies Between Modules

Adding Module Services to the DI Container

- Each module will have its own services that need to be registered in the DI container.
- Contain extension methods that encapsulate the registration logic for the module.
- Create extension methods for each module to encapsulate the registration logic

Accumulating Dependencies

- Accumulate dependencies from different module's DI class in the Program.cs
- All necessary services are registered and available throughout the application.

Catalog Module Extension Method

Catalog – CatalogModule.cs

```
public static class CatalogModule {
    public static IServiceCollection AddCatalogModule(this IServiceCollection services)    {
        services.AddTransient<ICatalogService, CatalogService>();
        services.AddScoped<ICatalogRepository, CatalogRepository>();
        return services;
```

Api Project - Program.cs

```
var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
builder.Services.AddControllers();
// Add services to the container.
builder.Services
    .AddCatalogModule(builder.Configuration)
    .AddBasketModule(builder.Configuration)
    .AddOrderingModule(builder.Configuration);
```

Dependency Path of EShop Modular Monoliths

Asp.net Middlewares and Http Request Pipeline

What is Middleware ?

- Assembled into an application pipeline to handle requests and responses.
- Each component in the pipeline can either handle a reques or pass it on to the next middleware component.
- Chain of delegates that can process an HTTP request on it in and an HTTP response on its way out.
- Authentication, logging, and error handling.

https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-8.0

ASP.NET Core HTTP Request Pipeline

- Sequence of middleware components that process incomir requests and outgoing responses.
- 1. Request Arrival
- 2. Middleware Processing
- 3. Response Creation
- 4. Response Processing

Asp.net Built-in Middlewares

https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-8.0

First Module: Catalog Module with Vertical Slice Architecture and CQRS

Domain Analysis of Catalog Module: Models, UCs, Rest Apis, Databases

Technical Analysis of Catalog Module: Architectures, Patterns, Libraries, Folders

Domain Driven Design - DDD

Vertical Slice Architecture

Mehmet Ozkaya

Big Picture – Course Project

Catalog Module Internal Architecture

Catalog Module with Vertical Slice Architecture and CQRS

- Vertical Slice Architecture implementation with Feature folders
- Domain Driven Design – DDD
- CQRS (Command Query Responsibility Segregation) Pattern
- Logical and Physical Implementation of CQRS Pattern
- CQRS Design Pattern With MediatR Library
- EF Core connection with PostgreSQL
- Minimal APIs for building fast HTTP APIs-fully functioning REST endpoints

Domain Analysis of Catalog Module

1. **Domain Models**
2. **Application Use Cases**
3. **Rest API Endpoints**
4. **Underlying Data Structures**

1. Domain Models of Catalog Module

- Primary domain model is '**Product**', associated with **'Category**'

```
public class Product
{
    8 references
    public Guid Id { get; set; }
    9 references
    public string Name { get; set; }
    10 references
    public List<string> Category { ge
    9 references
    public string Description { get;
    9 references
    public string ImageFile { get; se
    9 references
    public decimal Price { get; set;
}
```

- Consider Domain event for **ProductCreated** and **ProductPriceChanges**, leading to integration events.

2. Application Use Cases of Catalog Module

- Main use case are **Listing Products** and **Categories**, able to **search products**.

CRUD Operations:

- Listing Products and Categories
- Get Product with product Id
- Get Products with category
- Create new Product
- Update Product
- Delete Product
- If the price has changed, raise a ProductPriceChanged domain event that will lead to an integration event.
- *Along with this design our APIs according to REST perspective.

3. Rest API Endpoints of Catalog Module

Method	Request URI	Use Cases
GET	**/products**	List all products
GET	**/products/{id}**	Fetch a specific product
GET	**/products/category**	Get products by category
POST	**/products**	Create a new product
PUT	**/products/{id}**	Update a product
DELETE	**/products/{id}**	Remove a product

4. Underlying Data Structures of Catalog Module

- **Catalog Module** will be stored in a **PostgreSQL** relational database.
- Use a single **EShop** relational database, specifically **PostgreSQL**.
- Create a **Catalog schema** within the **PostgreSQL database** for **data isolation**.
- Use **Entity Framework Core's Code-First approach** for **database migrations** and **entity configurations**.
- **Configure** our **entities** and their **relationships** using **EF Core**.

Catalog Module Internal Architecture

Technical Analysis of Catalog Module

1. Application Architecture Style
2. Patterns & Principles
3. Libraries Nuget Packages
4. Project Folder Structure

1. Application Architecture Style of Catalog Module

Vertical Slice Architecture

- Organizes our code into feature folders, each feature encapsulated in a single .cs file

2. Patterns & Principles of Catalog Module

- **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.
- **ORM Pattern:** Object-Relational Mapping abstracts database interactions, work with database objects using high-level codes.

3. Libraries Nuget Packages of Catalog Module

- **MediatR for CQRS:** MediatR library simplifies the implementation of the CQRS pattern.
- **Carter for API Endpoints:** Routing and handling HTTP requests, easier to define API endpoints with clean and concise code.
- **EF Core for PostgreSQL Interaction:** EF Core provides a robust ORM for interacting with PostgreSQL, enabling us to manage database operations with ease.
- **Mapster for Object Mapping:** Mapster is a fast, configurable object mapper that simplifies the task of mapping objects.
- **FluentValidation for Input Validation:** For building strongly-typed validation rules, ensure inputs are correct before processed.

4. Project Folder Structure of Catalog Module

- The project is organized into **Data, Feature Folder-Products** and **Models**.
- **Features** like **CreateProduct** and **GetProduct** have dedicated handlers and endpoint definitions.
- **Feature** folder will be **Products**.
- **Data folder** and **Context objects** manages database interactions.

Way of Development: Shared -> Modules -> API

Way of Development: Shared -> Modules -> API

Development Path

Module Development Path

Develop Shared Domain for Tactical DDD Abstraction

Domain Driven Design - DDD

Tactical Domain-Driven Design: Entities, Value Objects, Aggregates, Aggregate Roots

Develop DDD Abstractions (Entity, Aggregate, DomainEvent)

Develop IDomainEvent interface using MediatR INotification

Develop Aggregate classes in Tactical DDD Abstraction

Mehmet Ozkaya

Big Picture – Course Project

Catalog Module with Vertical Slice Architecture and CQRS

- Domain Driven Design - DDD
- Tactical Domain-Driven Design: Entities, Value Objects, Aggregates, Aggregate Roots
- Develop DDD Abstractions (Entity, Aggregate, DomainEvent)
- Develop IDomainEvent interface in Tactical DDD Abstraction using MediatR INotification
- Develop Aggregate classes in Tactical DDD Abstraction

Domain Driven Design - DDD

- **Domain-Driven Design (DDD)** has been around since **Eric Evans** published his book about the subject in **2003**.
- **Domain Driven Design (DDD)** is not an improved technology or specific method.
- DDD is solving a **complex problem** is to **break the problem** int **smaller parts** and **focus on smaller problems** that are relative easy.
- A **complex domain** may **contain sub domains**. And some of **sub domains** can **combine** and **grouping** with each other for common rules and responsibilities.

DDD Types: Strategic and Tactical DDD

Strategic DDD:

- Understanding and modeling the business domain.
- It involves identifying the different domains, their subdomains, ar how they interact with each other.

Tactical DDD:

- Implementation details and provides design patterns.
- Including patterns like Entities, Value Objects, Aggregates, etc.

- We'll focus on Strategic DDD and its key concepts.

DDD Concepts

- **DDD Concepts:** Domain, SubDomain, Ubiquitous Language, Bounded Context, Context Mapping

Domain:

- The sphere of knowledge and activity around which the application logic revolves.
- Represents the problem space - the area of expertise or business need that the software addresses.

SubDomain:

- Represents a specific area of expertise within the overall domain.

Ubiquitous Language:

- Common language used by developers and domain experts to ensure clarity and consistency.
- Crucial for this language to be used in both discussions and in the codebase.

DDD Concepts - 2

Bounded Context:

- Grouping of closely related scopes that we can say logical boundaries.
- Logical boundary that represents the smaller problem particles of the complex domain that are self-consistent and as independent as possible.
- Each bounded context can have its database schema, its code models, and its team working on it.

Context Mapping Pattern:

- Identify the whole bounded contexts in the application and with their logical boundaries.
- A way to define logical boundaries between domains.

DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships.

http://martinfowler.com/bliki/BoundedContext.html

DDD Conclusion

- Depends on the Domain-Model Layer so it can:
  - Use entity objects
  - Use Repository Interfaces/Contracts
- Depends on the Infrastructure Layer (thru DI) so it can:
  - Use Repository implementation classes, ideally through DI

- Ideally, it must NOT take dependency on any other layer
- It implements:
  - Domain Entities, Aggregate-Roots and Value-Objects
  - Repository Contracts/Interfaces (to be used in DI)

- Depends on the Domain-Model Layer so it can:
  - Use entity objects.
    - Like EF updating a database through mapped entities
- Direct dependency on infrastructure frameworks like EF Core or any other database, cache or infrastructure API

https://learn.microsoft.com/en-us/dotnet/architecture/microservices

Bounded Context Pattern (Domain-Driven Design)

- **DDD - Bounded Context Pattern** which is one of the main pattern that we mainly use when **decomposing microservices**.
- **Domains** are require **high cooperation** and have a certain complexity by nature are called **collaborative domains**.
- DDD has **2 phases**, **Strategic** and **Tactical DDD**.
- **Strategic DDD**, we define the large-scale model of the system, defining to the business rules that allow designing loosely coupling units and the context map between them.
- **Tactical DDD** focuses on implementation and provides design patterns that we can use to build the software implementation.
- Include concepts such as **entity**, **aggregate**, **value object**, **repository**, and **domain service**.

Bounded Context Pattern (Domain-Driven Design)-2

- **DDD** domain defines its **own common language** and divides boundaries into specific, independent components. Common language is called **ubiquitous language**, and independent units are called **bounded context**.
- **DDD** is solving a **complex** problem is to break the problem into **smaller parts** and focus on smaller problems that are relatively easy.
- **A complex domain** may contain **sub domains**. And some of sub domains can **combine** and **grouping** with each other for **common rules** and **responsibilities**.
- **Bounded Context** is the grouping of closely related scopes that we can say **logical boundaries**.
- **Bounded context** is the **logical boundary** that represents the smaller problem particles of the complex domain that **are self-consistent** and as **independent** as possible.

DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships.

http://martinfowler.com/bliki/BoundedContext.html

DDD Types: Strategic and Tactical DDD

Strategic DDD:

- Understanding and modeling the business domain.
- It involves identifying the different domains, their subdomains, ar how they interact with each other.

Tactical DDD:

- Implementation details and provides design patterns.
- Including patterns like Entities, Value Objects, Aggregates, etc.

- We'll focus on Strategic DDD and its key concepts.

Tactical Domain-Driven Design

- **Tactical DDD** is a set of patterns and principles that help in modeling complex software systems.
- Focuses on the **domain** and **domain logic**, importance of a model that reflects the business and its rules.
- **Entities**, **Value Objects**, **Aggregates**, and **Aggregate Roots**

https://learn.microsoft.com/en-us/dotnet/architecture/microservices

Entities

- **Entity** is an object that is identified by its identity (ID), rather than its attributes.
- Identity **makes** each **entity unique**, even if other attributes are the same.
- **Entities** are used to represent objects in the system that have a **distinct identity** and lifecycle.
- **Example:** Ordering microservices, an Order can be an Entity.
- It is **uniquely identified** by an **OrderId**, even if other attributes (like date, total amount) are identical to another order.

```
public class Order : Entity
{
    0 references
    public OrderId Id { get; private set; }
    // Other properties and methods
}
```

https://learn.microsoft.com/en-us/dotnet/architecture/microservices

Value Objects

- **Value Object** is an object that describes some characteristic or attribute but carries no concept of identity.
- **Value Objects** are used to describe aspects of the domain with **no conceptual identity**.
- They are **immutable** and are often used to **encapsulate complex attributes**.
- **Example:** An **Address** used in an Order might be a Value Object, as it is important for the order but does not define the order's identity.

```
public class Address
{
    0 references
    public string Street { get; private set; }
    0 references
    public string City { get; private set; }
    // Other properties and constructor
}
```

https://learn.microsoft.com/en-us/dotnet/architecture/microservices

Aggregates

- **Aggregate** is a cluster of domain objects (Entities and Value Objects) that can be treated as a single unit.
- **Aggregate** will have one of its **components** be the **Aggregate Root**.
- Define a **boundary** around **related objects**. Operations within the boundary should maintain consistency of changes to data in an Aggregate.
- **Example: Order** can be an **Aggregate**, containing OrderItems, PaymentDetails, etc., within it. The consistency of an order (like total price calculation, stock validation) is maintained within the Aggregate boundaries.

https://learn.microsoft.com/en-us/dotnet/architecture/microservices

```
public class Order : Aggregate<OrderId>
{
    private readonly List<OrderItem> _orderItems = new();
    6 references
    public IReadOnlyList<OrderItem> OrderItems => _orderItems.AsReadOnly();
```

Aggregate Roots

- **Aggregate Root** is the main Entity in an Aggregate, through which external objects interact with the Aggregate.
- Provides a **gateway** to the **Aggregate**, ensuring that the **Aggregate's invariants** are enforced and consistency is maintained.
- **Example:** In the **Order Aggregate**, Order itself can be the Aggregate Root. External objects would interact with Order to affect changes within the Aggregate.

https://learn.microsoft.com/en-us/dotnet/architecture/microservices

```
public class Order : Entity // Aggregate Root
{
    1 reference
    private List<OrderItem> _orderItems;
    0 references
    public IReadOnlyCollection<OrderItem> OrderItems => _orderItems.AsReadOnly();
    // Other properties and methods
}
```

Tactical DDD Patterns in Catalog Module

Develop DDD Abstractions (Entity, Aggregate, DomainEvent) in Shared Library

IDomainEvent interface in Tactical DDD Abstraction using MediatR INotification

Domain Events

- Significant occurrences within the domain that you want to track and react to.
- For example, an event like ProductPriceChanged would be a domain event indicating that the price of a product has changed.

MediatR INotification Interface

- MediatR is a mediator library for .NET that allows to send messages (commands, events) within your app.
- INotification is an interface in MediatR used to define events that can be published and handled asynchronously.
- Raising and Dispatching Domain Events

Develop Aggregate classes in Tactical DDD Abstraction

Develop Catalog Domain with Product Domain Entity Models

Develop Product Domain Entity Models in Catalog Module

Anemic-domain vs Rich-domain Model Entities Entities in DDD

Develop DDD Rich-domain model Entity for Product Entity

Domain Events in DDD and Domain vs Integration Events

Develop Product Created and Updated Domain Event

Mehmet Ozkaya

Big Picture – Course Project

Catalog Module Internal Architecture

Develop Catalog Domain w/ Product Domain Entity Models

- Develop **Product Domain Entity** Models in Catalog Module
- **Anemic-domain** vs **Rich-domain** Model Entities Entities in DDD
- Develop **DDD Rich-domain** model Entity for Product Entity
- **Domain Events** in DDD and Domain vs Integration Events
- Develop **Product Created** and **Updated Domain Event** in Catalog Module
- **Raise ProductCreated** and **ProductPriceChanged Domain Event** in Catalog Domain Layer

1. Domain Models of Catalog Module

- Primary domain model is '**Product**', associated with **'Category**'

```
public class Product
{
    8 references
    public Guid Id { get; set; }
    9 references
    public string Name { get; set; }
    10 references
    public List<string> Category { ge
    9 references
    public string Description { get;
    9 references
    public string ImageFile { get; se
    9 references
    public decimal Price { get; set;
}
```

- Consider Domain event for **ProductCreated** and **ProductPriceChanged** that leading to integration events.

4. Project Folder Structure of Catalog Module

- The project is organized into **Data, Feature Folder-Products** and **Models**.
- **Features** like **CreateProduct** and **GetProduct** have dedicated handlers and endpoint definitions.
- **Feature** folder will be **Products**.
- **Data folder** and **Context objects** manages database interactions.

Anemic-Domain Model Entity

- **Entity** have **little** or **no business logic** (domain logic).
- Essentially **data structures** with **getters** and **setters**.
- But the **business rules** and **behaviors** are typically **implemented outside the entity**, often in service layers.

```
public class Order
{
    0 references
    public List<OrderItem> OrderItems { get; set; }
    // Other properties...
}
```

- **Order class** is **anemic** because it **only contains data** and lacks any domain logic or behaviors.

Rich-Domain Model Entity

- **Entities encapsulate** both **data** and **behavior**.
- This model enriches entities with methods that **embody business rules** and **domain logic**.

```
public class Order : Aggregate<Guid>
{
    1 reference
    private readonly List<OrderItem> _orderItems = new();
    0 references
    public IReadOnlyList<OrderItem> OrderItems => _orderItems.AsReadOnly();
    0 references
    public void AddOrderItem(OrderItem item)
    {
        // Domain logic to add an item
    }
    0 references
    public void RemoveOrderItem(OrderItemId itemId)
    {
        // Domain logic to remove an item
    }
    // Other properties...
}
```

- **Order** is a **rich domain model** as it includes methods **AddOrderItem** and **RemoveOrderItem** which **encapsulate the business logic** for manipulating the order items.

Anemic-domain vs Rich-domain Model Entities

Anemic Domain Model:

- Business logic is scattered across the application, often in services.
- Entities are simple data containers.
- Lead to issues with maintainability and understanding the business logic.

Rich Domain Model:

- Encapsulates business logic within the entities.
- More closely represents the real-world business domain.
- Can be more complex to design but leads to a more maintainable and cohesive model.

Order Example with Rich Domain Model

```
public class Order : Aggregate<OrderId>
{
    3 references
    private readonly List<OrderItem> _orderItems = new();

    // Encapsulated method to add an item
    0 references
    public void AddOrderItem(OrderItem item)
    {
        // Business logic for adding an item
        _orderItems.Add(item);
    }
    // Encapsulated method to remove an item
    0 references
    public void RemoveOrderItem(OrderItemId itemId)
    {
        // Business logic for removing an item
        var item = _orderItems.FirstOrDefault(i => i.Id == itemId)
        if (item != null)
        {
            _orderItems.Remove(item);
        }
    }
```

Rich-domain model for Entities and ValueObjects

- **Entities > Create Static Method**
- **ValueObjects > Of Static Method**

Domain Event in DDD

- **Domain Events** represent something that **happened** in the **past** and the other parts of the **same service boundary** same domain need to react to these changes.
- **Domain Event** is a **business event** that occurs within the **domain model**. It often represents a side effect of a domain operation.
- **Achieve consistency** between **aggregates** in the same domain.
- When an order is placed, an OrderPlaced event might be triggered.
- Trigger **side effects** or **notify other parts** of the system about changes within the domain.

How to Use Domain Events in DDD?

- **Encapsulate** the event details and **dispatch** them to interested parties.
- **Communicate** changes **within the domain** to external handlers which may perform actions based on these events.

Eshop Domain Events in Ordering Module

https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/domain-events-design-implementation

Domain vs Integration Events

Domain Events:

- Published and consumed within a single domain. Strictly within the boundary of the microservice/domain context.
- Indicate something that has happened within the aggregate.
- In-process and synchronously, sent using an in-memory message bus.
- **Example:** OrderPlacedEvent.

Integration Events:

- Used to communicate state changes or events between contexts or microservices.
- Overall system's reaction to certain domain events.
- Asynchronously, sent with a message broker over a queue.
- **Example:** After handling OrderPlacedEvent, an OrderPlacedIntegrationEvent might be published to a message broker like RabbitMQ, then consumes by other microservices.

Eshop Catalog Domain and Integration Events

Eshop Domain and Integration Events - OLD

Data Management of Modular Monolith: Setup Postgres as a Main Database

DB Schema separation of Modules in Modular Monoliths apps

Backing Services

Docker-compose Run Backing Services and Databases

Add PostgreSQL DB as a Main DB of EShop app into Docker-compose file

Deployment Strategy of Eshop Modular Monolith App

Mehmet Ozkaya

Big Picture – Course Project

Develop Catalog Domain w/ Product Domain Entity Models

- Data Management of Modular Monoliths
- DB Schema separation of Modules in Modular Monoliths apps
- Backing Services
- Docker-compose Run Backing Services and Databases
- Add PostgreSQL DB as a Main DB of EShop app into Docker-compose file
- Deployment Strategy of Eshop Modular Monolith App
- Setup PostgreSQL DB using Docker-compose file for Multi-container docker environment
- Run Docker-Compose on Visual Studio to setup PostgreSQL DB on Docker

Why Is Data Isolation Important ?

Maintainability

- Isolating data per module makes the system easier to maintain and evolve.

Scalability

- Different parts of the system can scale independently.

Security

- Sensitive data can be isolated and protected more effectively.

Conflict Reduction

- Reduces the risk of data conflicts between modules, ensuring data integrity.

Maintainability
Scalability
Security
Conflict Reduction

4 Data Isolation Strategies

1. Separate table
2. Separate schema
3. Separate database
4. Different Persistence

1. Separate table

Advantages

- Simple to implement and manage.
- Easy to query across modules.

Disadvantages

- Limited isolation; schema changes in one module can affect others.
- Hard to enforce boundaries, leading to potential data leakage.

2. Separate schema

Advantages

- Each module has its own schema within the same database.
- Good balance between isolation and manageability.
- Logical separation of data, making it clear which data belongs to which module.
- Easy to enforce boundaries and access controls at the schema level.

Disadvantages

- More complex than separate tables, but still manageable.
- Shared database resources can become a bottleneck.

3. Separate database

Advantages

- Each module uses its own database.
- Excellent isolation; changes in one database do not affect others.
- Independent scaling and maintenance for each database.

Disadvantages

- Increased complexity in management and backup.
- Higher operational costs due to multiple database instances.
- Cross-module queries become more complex and may require inter-database communication mechanisms.

4. Different persistence mechanisms

Advantages

- Each module can use different types of storage technologies (e.g., SQL, NoSQL, file storage).
- Modules can choose the best storage technology for their specific needs.
- High degree of isolation and flexibility.

Disadvantages

- Significant increase in complexity and operational overhead.
- Difficult to manage and maintain consistency across different storage technologies.

Why Is DB Schema Separation Important ?

- In a **modular monolith architecture**, **maintaining clear boundaries** between **different modules** is **essential**.
- **Database schema separation** is a **practice** that helps achieve this **by isolating the data** of each **module** within **its own schema**.

Best Practice: Separate Schema

- For our **EShop Modular Monoliths** application, we will follow the "**Separate Schema**" approach.

Logical Data Isolation

- Each module's data is stored in a separate schema, making it clear which data belongs to which module.
- maintain the integrity and consistency of the data.
- Provides a good balance between isolation and ease of management.
- Ensures logical separation of data, which simplifies maintenance and security.

Best Practice: Separate Schema

Ease of Management

- Easier to manage compared to multiple databases or different persistence mechanisms.
- Schema separation reduces the complexity of managing the database.
- Changes in one module's schema do not impact other modules, making it easier to evolve and maintain the application.
- Simplifies backup, restore, and migration processes.

Best Practice: Separate Schema

Performance

- Reduces the complexity of cross-module queries.
- Shared database resources can be optimized more effectively.

Security

- Allows for fine-grained access control at the schema level.
- Simplifies auditing and compliance requirements.

Implement Separate Schema in EShop Modular Monoliths

Multi-context EF DB Context for DB Separate Schema

CatalogDbContext

```
public class CatalogDbContext : DbContext
  {
    public CatalogDbContext(DbContextOptions<CatalogDbContext> options) : base(options) { }
    public DbSet<Product> Products { get; set; }
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
      modelBuilder.HasDefaultSchema("catalog");
      modelBuilder.Entity<Product>(entity =>
      {
        entity.HasKey(e => e.Id);
        entity.Property(e => e.Name).IsRequired();
        entity.Property(e => e.Price).HasColumnType("decimal(18,2)");
      });
      base.OnModelCreating(modelBuilder);
    }
  }
}
```

What is Backing Services for Cloud-Native Architectures

- **Backing services** are the **external components** that applications 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 application and can be **swapped** or **replaced without changing** the application's cor logic.
- **Decoupled from the application** themselves, promoting flexibility, scalability, and easier maintenance.

Backing Services for Cloud-Native Architectures

Big Picture – Course Project

EShop Modular Monolith Deployment Strategy

https://learn.microsoft.com/en-us/dotnet/architecture/microservices/

EShop Microservices Deployment Strategy

1. Local Development with Docker Compose for Backing Services

- Focus on local microservices development.
- Use Docker Compose to spin up backing services like databases, which are exposed to our local environment.
- Test microservices locally while connecting to containerized backing services.

2. Full Docker Environment Orchestration

- Orchestrating all microservices, including their dependencies, within a Docker Compose environment.
- Simulates a more production-like environment, allowing end-to-end testing.

3. Future Deployment to Kubernetes and Cloud

- Orchestrating all Deploy microservices to Kubernetes, including local Minikube and cloud-based EKS (Elastic Kubernetes Service)

kubernetes

Setup PostgreSQL DB using Docker-compose file

https://hub.docker.com/_/postgres

Develop Shared and Catalog Data-Infrastructure with EF Core Code First and PostgreSQL

EF Core 8 Features, Code First Approach, Migrations

Install EF Core Nuget Packages in Shared.Infrastructure

Develop EF Core DBContext Object for Storing Entities, Create and apply EF Core Migrations

Auto Migrate an Seed EF Core Entities

Dispatch Domain Events w/ EF Core SaveChangesInterceptor

Mehmet Ozkaya

Big Picture – Course Project

Catalog Module Internal Architecture

Develop Shared and Catalog Data-Infrastructure with EF Core Code First and PostgreSQL

- EF Core 8 Features, Code First Approach, Migrations and PostgreSQL
- Install EF Core Nuget Packages in Shared Infrastructure
- Develop EF Core DBContext Object for Storing Entities in Catalog Infrastructure
- EF Core PostgreSQL Connection String in appsettings.json file
- Create and apply EF Core Migrations for CatalogDbContext
- Auto Migrate EF Core Entities to PostgreSQL DB when Application Startup
- Seed EShopDb Product Data When Application Startup Just After UseMigration
- EF Core Interceptors: SaveChangesInterceptor for Auditing Entities
- Develop SaveChangesInterceptor to Auditing Entities for Product Entities
- Dispatch Domain Events w/ EF Core SaveChangesInterceptor
- Develop DispatchDomainEventsInterceptor to Trigger Domain Event Handlers

4. Underlying Data Structures of Catalog Module

- **Catalog Module** will be stored in a **PostgreSQL** relational database.
- Use a single **EShop** relational database, specifically **PostgreSQL**.
- Create a **Catalog schema** within the **PostgreSQL database** for **data isolation**.
- Use **Entity Framework Core's Code-First approach** for **database migrations** and **entity configurations**.
- **Configure** our **entities** and their **relationships** using **EF Core**.

4. Project Folder Structure of Catalog Module

- The project is organized into **Data, Feature Folder-Products** and **Models**.
- **Features** like **CreateProduct** and **GetProduct** have dedicated handlers and endpoint definitions.
- **Feature** folder will be **Products**.
- **Data folder** and **Context objects** manages database interactions.

Way of Development: Shared -> Modules -> API

Way of Development: Shared -> Modules -> API

Development Path

Module Development Path

3. Libraries Nuget Packages of Catalog Module

- **MediatR for CQRS:** MediatR library simplifies the implementation of the CQRS pattern.
- **Carter for API Endpoints:** Routing and handling HTTP requests, easier to define API endpoints with clean and concise code.
- **EF Core for PostgreSQL Interaction:** EF Core provides a robust ORM for interacting with PostgreSQL, enabling us to manage database operations with ease.
- **Mapster for Object Mapping:** Mapster is a fast, configurable object mapper that simplifies the task of mapping objects.
- **FluentValidation for Input Validation:** For building strongly-typed validation rules, ensure inputs are correct before processed.

Entity Framework Core

- **Entity Framework (EF) Core** is a lightweight, extensible, open source and cross-platform data access technology.
- **EF Core** can serve as an **object-relational mapper** (O/RM)
- Enables **.NET developers** to work with a **database** using **.NET objects.**
- **Eliminates** the need for most of the **data-access code** that typically needs to be written.
- EF Core **supports many database engines**.

Entity Framework Core

Entity Framework Core NuGet Packages for DB Providers

- **EF Core** supports **different database systems** through the use of "**database providers**".
- **Each system** has its **own database provider**, which is shipped as NuGet package.
- Applications should **install** one or more of these **provider package**

Database system	Package
SQL Server and SQL Azure	Microsoft.EntityFrameworkCore.SqlServer
SQLite	Microsoft.EntityFrameworkCore.Sqlite
Azure Cosmos DB	Microsoft.EntityFrameworkCore.Cosmos
PostgreSQL	Npgsql.EntityFrameworkCore.PostgreSQL *
MySQL	Pomelo.EntityFrameworkCore.MySql *
EF Core in-memory database**	Microsoft.EntityFrameworkCore.InMemory

https://learn.microsoft.com/en-us/ef/core/what-is-new/nuget-packages

EF Core Tools

- Tooling for EF Core migrations and reverse engineering (scaffolding) from an existing database has 2 way:

1- .NET Core CLI:

- **dotnet-ef** and **Microsoft.EntityFrameworkCore.Design** for cross-platform command line tooling
  - **dotnet tool install --global dotnet-ef**
  - **dotnet add package Microsoft.EntityFrameworkCore.Design**

2- Package Manager Console (PMC) in Visual Studio:

- **Microsoft.EntityFrameworkCore.Tools** for PowerShell tooling that works in the Visual Studio **Package Manager Console**
  - **Install-Package Microsoft.EntityFrameworkCore.Tools**

EF Core Developments Steps for Catalog Module

1- Install EF Core Nuget Packages:

- Microsoft.EntityFrameworkCore.Tools for PowerShell tooling that works in the Visual Studio Package Manager Console
  - Install-Package Microsoft.EntityFrameworkCore
  - Install-Package Microsoft.EntityFrameworkCore.Tools
  - Install-Package Microsoft.EntityFrameworkCore.PostgreSQL

2- Develop DBContext Object:

- Adding Entity Objects into Ef Core DBContext object

```
public class CatalogDbContext : DbContext
{
    0 references
    public CatalogDbContext(DbContextOptions<CatalogDbContext> options)
        : base(options) { }

    11 references
    public DbSet<Product> Products => Set<Product>();

    0 references
    protected override void OnModelCreating(ModelBuilder builder)
    {
        builder.HasDefaultSchema("catalog");
        builder.ApplyConfigurationsFromAssembly(Assembly.GetExecutingAssembly()
        base.OnModelCreating(builder);
    }
```

CatalogDbContext

```
public class CatalogDbContext : DbContext
  {
    public CatalogDbContext(DbContextOptions<CatalogDbContext> options) : base(options) { }
    public DbSet<Product> Products { get; set; }
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
      modelBuilder.HasDefaultSchema("catalog");
      modelBuilder.Entity<Product>(entity =>
      {
        entity.HasKey(e => e.Id);
        entity.Property(e => e.Name).IsRequired();
        entity.Property(e => e.Price).HasColumnType("decimal(18,2)");
      });
      base.OnModelCreating(modelBuilder);
    }
  }
}
```

EF Core Developments Steps for Catalog Module - continue

3- Configure Domain Entities to EF Core Entities for PostgreSQL:

- Using Ef Core ModelBuilder for mapping DDD Entities to EF Core Entities

```
public class ProductConfiguration : IEntityTypeConfiguration<Product>
{
    0 references
    public void Configure(EntityTypeBuilder<Product> builder)
    {
        builder.HasKey(p => p.Id);

        builder.Property(p => p.Name).HasMaxLength(50).IsRequired();

        builder.Property(p => p.Category).IsRequired();

        builder.Property(p => p.Description).HasMaxLength(200);

        builder.Property(p => p.ImageFile).HasMaxLength(100);

        builder.Property(p => p.Price).IsRequired();
```

Entity Framework Core

4- Asp.net Core Setup & Configurations:

- appsettings.json Database connection string

```
"ConnectionStrings": {
  "Database": "Server=localhost;Port=5432;Database=EShopDb;User Id=postgres;Password=postgres;
```

Setup PostgreSQL DB using Docker-compose file

https://hub.docker.com/_/postgres

EF Core Developments Steps for Catalog Module - continue

5- Register EF Core DBContext to Asp.Net:

- Register into CatalogModule.cs

```
services.AddDbContext<CatalogDbContext>((sp, options) =>
{
    options.AddInterceptors(sp.GetServices<ISaveChangesInterceptor>());
    options.UseNpgsql(connectionString);
});
```

6- Create Migrations and create Database w/ Commands:

- Install-Package Microsoft.EntityFrameworkCore.Tools
- Add-Migration InitialCreate
- Update-Database

Entity Framework Core

Wiring Up Dependencies Between Modules

EF Core Developments Steps for Catalog Module - continue

7- Create Database w/ Migrate method:

- Api-Program.cs to automatically apply migrations

```
public static IApplicationBuilder UseCatalogModule(this IApplicationBuilder app)
{
    // Configure the HTTP request pipeline.

    // 1. Use Api Endpoint services

    // 2. Use Application Use Case services

    // 3. Use Data - Infrastructure services
    app.UseMigration<CatalogDbContext>();

    return app;
```

8- Seed Database:

- Create Seed/Extentions.cs

```
public static IApplicationBuilder UseMigration<TContext>(this IApplicationBuilder
    where TContext : DbContext
{
    MigrateDatabaseAsync<TContext>(app.ApplicationServices).GetAwaiter().GetResult

    SeedDataAsync(app.ApplicationServices).GetAwaiter().GetResult();
```

Entity Framework Core

Way of Development: Shared -> Modules -> API

Development Path

Module Development Path

EF Core Developments Steps for Catalog Module - continue

9- Dispatch DomainEvents w/ EF Core Interceptor:

- Use EF Core SaveChangesInterceptor

```
internal class DispatchDomainEventsInterceptor(IMediator mediator) : SaveChangesIntercept
{
    0 references
    public override InterceptionResult<int> SavingChanges(DbContextEventData eventData, I
    {
        DispatchDomainEvents(eventData.Context).GetAwaiter().GetResult();

        return base.SavingChanges(eventData, result);
```

10- Test all from local w/ containerize PostgreSQL in Docker:

- Docker PostgreSQL Setup - Backing Services

```
if (!await dbContext.Products.AnyAsync())
{
    await dbContext.Products.AddRangeAsync(InitialData.Products);
    await dbContext.SaveChangesAsync();
}
```

Entity Framework Core

What is EF Core Migrations ?

- **Entity Framework Core Migrations** are a powerful feature for managing **database schema changes** in an application's lifecycle.
- Enable a developer to **evolve** a **database schema** in a controlled and **consistent way**, in line with the domain model changes.
- **Data models change** as features get implemented: **new entities** or properties are **added** and **removed**, and database schemas need to be changed accordingly.
- **Migrations** provide a way to **incrementally update** the **database** schema so that it stays in sync with the application's data model.
- Useful in **team environments** and **continuous deployment** scenarios where database schema **changes** are **frequent** and need to be applied systematically.

Entity Framework Core

How EF Core Migrations works ?

1- Model Change Occurs:

- When a data model change occurs, developers use EF Core tools to create a corresponding migration that describes the necessary database updates.
- EF Core compares the current model against a snapshot of the previous model to find differences and generates migration source files.

2- Migration classes generated:

- Once a new migration has been generated, it can be applied to a database in various ways.
- EF Core records all applied migrations in a special history table, allowing it to know which migrations have been applied and which haven't.

Entity Framework Core

EF Core Code-First Approach

- **Database creation** and **schema updates** are driven by the entity classes and **DbContext configuration** in the application.
- First, we should **develop Entity classes** and **DbContexts** after that run **EF Migration Commands**.

Entity Framework Core

EF Migration Commands to Create the Database

1- .NET Core CLI:

- **dotnet-ef** and **Microsoft.EntityFrameworkCore.Design** for cross-platform command line tooling
  - **dotnet tool install --global dotnet-ef**
  - **dotnet add package Microsoft.EntityFrameworkCore.Design**
  - **dotnet ef migrations add InitialCreate**
  - **dotnet ef database update**

2- Package Manager Console (PMC) in Visual Studio:

- **Microsoft.EntityFrameworkCore.Tools** for PowerShell tooling that works in the Visual Studio **Package Manager Console**
  - **Install-Package Microsoft.EntityFrameworkCore.Tools**
  - **Add-Migration InitialCreate**
  - **Update-Database**

Entity Framework Core

EF Migration Add-Migration Command Parameters

Option	Description
-Name	The name of the migration you want to create. While the `-Name` parameter is there, it's often simpler to directly specify the migration name as the first parameter. For example, `Add-Migration AddingEFExtensions` will create a migration named **AddingEFExtensions**.
-OutputDir	The output directory relative to the project where **migration files** will be created. For example, `Add-Migration AddingEFExtensions -OutputDir MyMigrationDirectoryName` will create the migration file in the **MyMigrationDirectoryName** directory.
-Namespace	The namespace to use for the generated classes in the migration file. If omitted, the following namespace `[ProjectNamespace].[OutputDirectoryName]` will be used. For example, `Add-Migration AddingEFExtensions -Namespace Z.MyMigrationNamespace` will add classes in the **Z.MyMigrationNamespace** namespace.
-Context	The context class to use to generate the migration file, it can be either the `name` or the `fullname` including namespaces. For example, `Add-Migration AddingEFExtensions -Context Z.MyContextName` will generate a migration for the **Z.MyContextName** context.
-Project	The project containing the context used to generate the migration file. If omitted, the default project in the Package Manager Console is used. For example, `Add-Migration AddingEFExtensions -Project Z.MyProjectName` will generate a migration for the context in the **Z.MyProjectName** project.
-StartupProject	The project with the configurations and settings. If omitted, the startup project of your solution is used. For example, `Add-Migration AddingEFExtensions -StartupProject Z.MyStartupProjectName` will make a migration using the configurations and settings of the **Z.My StartupProjectName** project.

Special Consideration for Different Assemblies: Use –OutputDir Parameter

https://www.learnentityframeworkcore.com/migrations/add-migration#what-are-the-parameters-for-the-add-migration-command-in-ef-core

Catalog Module Internal Architecture

EShop Modular Monolith Deployment Strategy

https://learn.microsoft.com/en-us/dotnet/architecture/microservices/

EF Core Interceptors: SaveChangesInterceptor for Auditing Entities

- **Interceptors** in EF Core enable the interception, modification, or suppression of **EF Core operations**.
- This includes **low-level database operations** such as executing a command, as well as **higher-level operations**, such as calls to SaveChanges.

SaveChanges Interception:

- The **SaveChanges** and **SaveChangesAsync** interception points are used to execute **custom logic when saving changes** to the database.
- These interception points are defined by the **ISaveChangesInterceptor** interface.
- **EF Core** provides a **SaveChangesInterceptor** base class with no-op (no operation) methods as a convenience.

Entity Framework Core

Use Case: SaveChanges interception for Auditing

- Interception of **SaveChanges** can be used to create an independent **audit record** of changes.
- This is useful for **maintaining a history** of who changed an entity and when.
- **Before saving** changes to the database, you can **iterate** through the **changed entities** in the **DbContext** and **log** or **store** the **audit information**, like timestamps or user identifiers.

Entity Framework Core

Registering Interceptors

How to Register Interceptors ?

- Interceptors are registered using **AddInterceptors** when configuring a **DbContext** instance.
- Common approach is usually done in the **OnConfiguring method** of the **DbContext**.
- **Example Code:** Registering a SaveChangesInterceptor

```
public class ExampleContext : BlogsContext
{
    0 references
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        => optionsBuilder.AddInterceptors(new TaggedQueryCommandInterceptor());
}
```

Entity Framework Core

Implementing a Custom SaveChangesInterceptor

- Implement a custom interceptor by extending **SaveChangesInterceptor** or implementing **ISaveChangesInterceptor**.
- **Override** methods like **SavingChanges** and **SavedChanges** to execute custom logic.
- **Example Code:** Implementing Auditing in an Interceptor

```
public class MySaveChangesInterceptor : SaveChangesInterceptor
{
    0 references
    public override InterceptionResult<int> SavingChanges(DbContextEventData eventDa
    {
        var context = eventData.Context;
        foreach (var entry in context.ChangeTracker.Entries().Where(e => e.State ==
        {
            // Example: Set audit fields based on entity state
            if (entry.Entity is AuditableEntity entity)
            {
                if (entry.State == EntityState.Added)
                {
                    entity.CreatedAt = DateTime.UtcNow;
                    entity.CreatedBy = // get current user
                }
```

Entity Framework Core

Domain Event in DDD

- **Domain Events** represent something that **happened** in the **past** and the other parts of the **same service boundary** same domain need to react to these changes.
- **Domain Event** is a **business event** that occurs within the **domain model**. It often represents a side effect of a domain operation.
- **Achieve consistency** between **aggregates** in the same domain.
- When an order is placed, an OrderPlaced event might be triggered.
- Trigger **side effects** or **notify other parts** of the system about changes within the domain.

How to Use Domain Events in DDD?

- **Encapsulate** the event details and **dispatch** them to interested parties.
- **Communicate** changes **within the domain** to external handlers which may perform actions based on these events.

Eshop Domain Events in Ordering Microservices

https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/domain-events-design-implementation

Dispatch Domain Events w/ EF Core SaveChangesInterceptor

Steps dispatch Events:

- Create a SaveChangesInterceptor
- Identify and Dispatch Domain Events
- public class DomainEventDispatcherInterceptor : SaveChangesInterceptor
- Registering the Interceptor into DependencyInjection class

Best Practice to dispatch Events:
EF Core SaveChangesInterceptor

https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/domain-events-design-implementation

Eshop Catalog Domain and Integration Events

Best Practice to dispatch Events:
EF Core SaveChangesInterceptor

Steps dispatch Events:

- Create a SaveChangesInterceptor
- Identify and Dispatch Domain Events
- public class DomainEventDispatcherInterceptor : SaveChangesInterceptor
- Registering the Interceptor into DependencyInjection class

Application Use Case development with CQRS and MediatR (Abstraction

Create Abstraction Develop Shared.CQRS

CQRS Design Pattern

Logical and Physical Implementation of CQRS Pattern

CQRS Design Pattern With MediatR Library

Develop Vertical Slice Feature Folder with CQRS and MediatR

Mehmet Ozkaya

Big Picture – Course Project

Catalog Module Internal Architecture

Application Use Case development with CQRS and MediatR

- Vertical Slice versus Clean Architecture
- CQRS Design Pattern
- Logical and Physical Implementation of CQRS Pattern
- CQRS Design Pattern With MediatR Library (Logical CQRS Implementation)
- Develop Vertical Slice Feature Folder with CQRS and MediatR
- Develop Feature Handler class with CQRS and MediatR
- **FOCUS: Create Abstraction on MediatR for CQRS - Command and Query separation**
- Implement CQRS Abstractions ICommand into Vertical Slice Feature Handler class

Way of Development: Shared -> Modules -> API

Development Path

Module Development Path

CQRS - Command Query Responsibility Segregation

- **CQRS design pattern** in order to avoid **complex queries** to get rid of **inefficient joins**.
- **Separates read** and **write operations** with separating databases.
- **Commands:** changing the state of data into application.
- **Queries:** handling complex join operations and returning 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.
- **CQRS** offers to **separates read** and **write data** that provide to maximize query performance and scalability.

CQRS – Read and Write Operations

- **Monolithic** has **single database** is both working for **complex join queries**, and also perform **CRUD operations**.
- When **application** goes **more complex**, this **query** and **CRUD** operations will become **un-manageable situation**.
- Application required some **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 logics**, will cause to **lock database** operations.
- **Reading** and **writing database** has different approaches, define different strategy.
- «**Separation of concerns**» principles: separate 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.

CQRS – Read and Write Operations - 2

- If our **application** is **mostly reading use cases** and not writing so much, it is **read-incentive application**.
- **Read** and **write operations** are **asymmetrical** and has very different performance and scale requirements.
- To **improve query performance**, the read operation perform **queries** from a **highly denormalized materialized views** to **avoid** expensive repetitive **table joins** and **table locks**.
- **Write operation** which is **command operation**, can perform commands into separate **fully normalized relational database**.
- Supporting **ACID transactions** and **strong data consistency**.
- Commands: **task-based operations** like "add item into shopping cart" or "checkout order".
- **Commands** can be handle with message broker systems that provide to process commands in async way.
- **Queries:** never modify the database, always return the JSON data with DTO objects.

CQRS – Synchronization with Read and Write DB

- **Keep sync** both **read** and **write databases**.
- **Publishes an event t**hat subscribe from **Read Database** and **update** the **read table** accordingly.
- **Synchronization** handles with async communication using message brokers.
- This creates **Eventual Consistency principle**.
- The **Read** database **eventually synchronizes** from the **Write** database.
- **Some lag** in the process due to **async communication** with message brokers that applies publish/subscribe pattern.
- Welcome to **Eventual consistency.**

Benefits of CQRS

- **Scalability**
  When we separate Read and Write databases, we can also scale these separate databases independently. Read databases follows denormalized data to perform complex join queries.
- If application is read-incentive application, we can scale read database more then write database.
- **Query Performance**
  The read database includes denormalized data that reduce to comlex and long-running join queries. Complex business logic goes into the write database. Improves application performance for all aspects.
- **Maintability and Flexibility**
  Flexibility of system that is better evolve over time and not affected to update commands and schema changes by separating read and write concerns into different databases.
- Better implemented if we physically separate the read and write databases.

Drawbacks of CQRS

- **Complexity**
  CQRS makes your system more complex design.
  Strategically choose where we use and how we can separate read and write database.
- **Eventual Consistency**
  The read data may be stay old and not-updated for a particular time. So the client could see old data even write database updated, it will take some time to update read data due to publish/subscribe mechanism.
- We should **embrace** the **Eventual Consistency** when using **CQRS**, if your application **required strong consistency** than **CQRS is not good to apply**.

Best Practices for CQRS

- **Best practices** to **separate read** and **write database** with 2 database **physically**.
- **Read-intensive** that means **reading more than writing**, can define **custom data schema** to **optimized for queries**.
- **Materialized View Pattern** is good example to implement reading databases.
- **Avoid complex joins** and mappings with **pre-defined** fine-grained **data** for **query operations**.
- **Use different database** for reading and writing database types.
- Using **No-SQL** document database **for reading** and using **Relational** database **for CRUD** operations.

Logical and Physical Implementation of CQRS

- **Logical Implementation of CQRS:** Splitting Operations, Not Databases. Separate 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:** Separate Databases. Splitting the read and write operations not just at the code level but also physically using separate 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 a 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 the distinction between commands and queries clearer, you can define two custom interfaces: **ICommand** and **IQuery**.
  - public interface ICommand<TResult> : IRequest<TResult> { }
  - public interface IQuery<TResult> : IRequest<TResult> { }

Command and Command Handler in CQRS

- **Commands:** Change the state of the system. encapsulates the data required for a specific action.
- **Command Handlers:** Each command has a corresponding handler. The handler interacts with the domain model and persists changes.
- Executes business logic and updates the state through repositories.

MediatR and ASP.NET Core Minimal Apis

- **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**.

- Controller receives a request, it **creates** a **command** or **query** object and sends it to **MediatR**. MediatR then **dispatches** this object to the **appropriate handler**.

https://learn.microsoft.com/en-us/dotnet/architecture/microservices/

MediatR and ASP.NET Core Minimal Apis

- **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. **(REPR pattern)**

- **Minimal Api** receives a request, it **creates** a **command** or **query** object and sends it to **MediatR**. MediatR then **dispatches** this object to the **appropriate handler**.

CQRS and MediatR Request LifeCycle

https://learn.microsoft.com/en-us/dotnet/architecture/microservices

Mediator Pattern and Pipeline Behaviours

- **Mediator pattern** useful 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.

EShop Modules Pipeline Behaviours

- **LogBehavior:** A behavior that logs details about the handling of a request.
- **ValidatorBehavior:** A behavior that validates incoming requests before they reach the handler.

- Every request processed by MediatR in the application is logged and validated consistently.

- Using MediatR with the mediator pattern in ASP.NET Core applications provides a structured and clean way to handle complex request processing.

Catalog Module Internal Architecture

4. Project Folder Structure of Catalog Module

- The project is organized into **Data, Feature Folder-Products** and **Models**.
- **Features** like **CreateProduct** and **GetProduct** have dedicated handlers and endpoint definitions.
- **Feature** folder will be **Products**.
- **Data folder** and **Context objects** manages database interactions.

CQRS and MediatR Request LifeCycle

Create Abstraction on MediatR for CQRS

Eshop Modules uses Shared Library

Catalog Application Use Case Development with CQRS and MediatR

Develop Products CreateProductCommand and Handler

Develop Products Create/Update/Delete ProductCommandHandler using CatalogDbContext in Catalog Module

Develop Product Queries with CQRS and MediatR in Catalog Application Layer

Mehmet Ozkaya

Big Picture – Course Project

Catalog Module Internal Architecture

Catalog Application Use Case Development with CQRS and MediatR

- Develop Products CreateProductCommand and Handler
- Develop Products Create/Update/Delete ProductCommandHandler using CatalogDbContext in Catalog Module
- Develop Product Queries with CQRS and MediatR in Catalog Application Layer
- Develop GetProductByCategory Queries with CQRS and MediatR
- Develop GetProductById Queries with CQRS and MediatR
- Develop Domain Event Handlers with MediatR INotificationHandler

Way of Development: Shared -> Modules -> API

Development Path

Module Development Path

2. Application Use Cases of Catalog Module

- Main use case are **Listing Products** and **Categories**, able to **search products**.

CRUD Operations:

- Listing Products and Categories
- Get Product with product Id
- Get Products with category
- Create new Product
- Update Product
- Delete Product

- *Along with this design our APIs according to REST perspective.

1. Application Architecture Style of Catalog Module

Vertical Slice Architecture

- Organizes our code into feature folders, each feature encapsulated in a single .cs file

2. Patterns & Principles of Catalog Module

- **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.
- **ORM Pattern:** Object-Relational Mapping abstracts database interactions, work with database objects using high-level codes.

3. Libraries Nuget Packages of Catalog Module

- **MediatR for CQRS:** MediatR library simplifies the implementation of the CQRS pattern.
- **Carter for API Endpoints:** Routing and handling HTTP requests, easier to define API endpoints with clean and concise code.
- **EF Core for PostgreSQL Interaction:** EF Core provides a robust ORM for interacting with PostgreSQL, enabling us to manage database operations with ease.
- **Mapster for Object Mapping:** Mapster is a fast, configurable object mapper that simplifies the task of mapping objects.
- **FluentValidation for Input Validation:** For building strongly-typed validation rules, ensure inputs are correct before processed.

4. Project Folder Structure of Catalog Module

- The project is organized into **Data, Feature Folder-Products** and **Models**.
- **Features** like **CreateProduct** and **GetProduct** have dedicated handlers and endpoint definitions.
- **Feature** folder will be **Products**.
- **Data folder** and **Context objects** manages database interactions.

Command and Command Handler in CQRS

- **Commands:** Change the state of the system. encapsulates the data required for a specific action.
- **Command Handlers:** Each command has a corresponding handler. The handler interacts with the domain model and persists changes.
- Executes business logic and updates the state through repositories.

CQRS and MediatR Request LifeCycle

Eshop Modules uses Shared Library

Logical and Physical Implementation of CQRS

- **Logical Implementation of CQRS:** Splitting Operations, Not Databases. Separate 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:** Separate Databases. Splitting the read and write operations not just at the code level but also physically using separate databases.
- Introduces data consistency and synchronization problems.

Command and Command Handler in CQRS

- **Commands:** Change the state of the system. encapsulates the data required for a specific action.
- **Command Handlers:** Each command has a corresponding handler. The handler interacts with the domain model and persists changes.
- Executes business logic and updates the state through repositories.

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 a 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 the distinction between commands and queries clearer, you can define two custom interfaces: **ICommand** and **IQuery**.
  - public interface ICommand<TResult> : IRequest<TResult> { }
  - public interface IQuery<TResult> : IRequest<TResult> { }

MediatR and ASP.NET Core Minimal Apis

- **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. **(REPR pattern)**

- **Minimal Api** receives a request, it **creates** a **command** or **query** object and sends it to **MediatR**. MediatR then **dispatches** this object to the **appropriate handler**.

CQRS and MediatR Request LifeCycle

https://learn.microsoft.com/en-us/dotnet/architecture/microservices

Dispatch Domain Events w/ EF Core SaveChangesInterceptor

Steps dispatch Events:

- Create a SaveChangesInterceptor
- Identify and Dispatch Domain Events
- public class DomainEventDispatcherInterceptor : SaveChangesInterceptor
- Registering the Interceptor into DependencyInjection class

Best Practice to dispatch Events:
EF Core SaveChangesInterceptor

https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/domain-events-design-implementation

Eshop Catalog Domain and Integration Events

Best Practice to dispatch Events:
EF Core SaveChangesInterceptor

Steps dispatch Events:

- Create a SaveChangesInterceptor
- Identify and Dispatch Domain Events
- public class DomainEventDispatcherInterceptor : SaveChangesInterceptor
- Registering the Interceptor into DependencyInjection class

Catalog API Endpoints Exposing Minimal Apis with Carter and REPR Pattern

Develop Create/Update/Delete Product POST Endpoint with Carter implements ICarterModule for Minimal Apis

Developing GetProducts Endpoint for REST APIs in Catalog Module

Register Carter libraries into Asp.Net Dependency Injection Service and Request Pipelines

Mehmet Ozkaya

Big Picture – Course Project

Catalog Module Internal Architecture

Develop Catalog API Endpoints Exposing Minimal Api Endpoints with Carter and REPR Pattern

- Develop Create/Update/Delete Product POST Endpoint with Carter implements ICarterModule for Minimal Apis
- Developing GetProducts Endpoint for REST APIs in Catalog Module
- Developing GetProductById Endpoint for REST APIs in Catalog Module
- Developing GetProductByCategory Endpoint for REST APIs in Catalog Module
- Register Carter libraries into Asp.Net Dependency Injection Service and Request Pipelines
- Create Postman Collection for EShopModules
- Testing Catalog Endpoints with CQRS and VSA

Way of Development: Shared -> Modules -> API

Development Path

Module Development Path

REPR Design Pattern

- Developing **API Endpoints** implementing a simple **Request EndPoint Response** flow.
- **REPR** stands for **Request, Endpoint, Response** – a pattern simplifying the development of **REST API endpoints**.
- **REPR Design Pattern** defines web API endpoints as havi components:
- **Request -** Data structure the endpoint expects.
- **Endpoint -** Logic the endpoint performs given a request.
- **Response -** Data structure the endpoint returns.
- Simplified concepts to enable the development REST API endpoints and enforcing the **Single Responsibility Principle** for **endpoints**.
- **Developers** need to **care** about are the **Request** and **Response**.

CQRS and MediatR Request LifeCycle

https://learn.microsoft.com/en-us/dotnet/architecture/microservices

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 APIs.
- **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.
- **Goto Github page:**
- https://github.com/CarterCommunity/Carter

Eshop Modules uses Shared Library

3. Rest API Endpoints of Catalog Module

Method	Request URI	Use Cases
GET	**/products**	List all products
GET	**/products/{id}**	Fetch a specific product
GET	**/products/category**	Get products by category
POST	**/products**	Create a new product
PUT	**/products/{id}**	Update a product
DELETE	**/products/{id}**	Remove a product

2. Patterns & Principles of Catalog Module

Presentation Api Layer Patterns

- Minimal APIs
- REPR Design Pattern
- Asp.net DI, Routing with latest Asp.net 8 features

SECTION FLOW

- Domain > Infrastructure > Application > Presentation

3. Libraries Nuget Packages of Catalog Module

- **MediatR for CQRS:** MediatR library simplifies the implementation of the CQRS pattern.
- **Carter for API Endpoints:** Routing and handling HTTP requests, easier to define API endpoints with clean and concise code.
- **EF Core for PostgreSQL Interaction:** EF Core provides a robust ORM for interacting with PostgreSQL, enabling us to manage database operations with ease.
- **Mapster for Object Mapping:** Mapster is a fast, configurable object mapper that simplifies the task of mapping objects.
- **FluentValidation for Input Validation:** For building strongly-typed validation rules, ensure inputs are correct before processed.

4. Project Folder Structure of Catalog Module

- The project is organized into **Data, Feature Folder-Products** and **Models**.
- **Features** like **CreateProduct** and **GetProduct** have dedicated handlers and endpoint definitions.
- **Feature** folder will be **Products**.
- **Data folder** and **Context objects** manages database interactions.

3. Rest API Endpoints of Catalog Module

Method	Request URI	Use Cases
GET	**/products**	List all products
GET	**/products/{id}**	Fetch a specific product
GET	**/products/category**	Get products by category
POST	**/products**	Create a new product
PUT	**/products**	Update a product
DELETE	**/products/{id}**	Remove a product

CQRS and MediatR Request LifeCycle

EShop Modular Monolith Deployment Strategy

https://learn.microsoft.com/en-us/dotnet/architecture/microservices/

Wiring Up Dependencies Between Modules

Eshop Modules uses Shared Library

Big Picture – Course Project

Develop Cross-cutting Concerns for Catalog and Each Module in EShop Modules

MediatR Pipeline Behaviours and Fluent Validation Library

Apply Validation with AbstractValidator using Fluent Validation Library

Validation Pipeline Behaviour with MediatR using Fluent Validation

Global Handling Exceptions in ASP.NET Core with app.UseExceptionHandler

Logging Behavior in MediatR Pipeline for Cross-cutting concerns

Mehmet Ozkaya

Big Picture – Course Project

Develop Cross-cutting Concerns for Catalog and Each Module in EShop Modules

- MediatR Pipeline Behaviours
- Apply Validation with AbstractValidator using Fluent Validation
- Validation Pipeline Behaviour with MediatR using Fluent Validation Library
- Global Handling Exceptions in ASP.NET Core with app.UseExceptionHandler
- Logging Behavior in MediatR Pipeline for Cross-cutting concerns
- Develop GetProducts Query apply Pagination using CQRS and MediatR
- Structured Logging Using Serilog in .NET 8.0 Minimal APIs
- Serilog WriteTo Seq for Structured Logging in .NET

Eshop Modules uses Shared Library

MediatR Pipeline Behaviours

- **MediatR** powerful features 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**.

How MediatR Pipeline Behaviours Works

```
public class LoggingBehavior<TRequest, TResponse> (ILogger<LoggingBehavior<TRequest,
    : IPipelineBehavior<TRequest, TResponse>
    where TRequest : notnull, IRequest<TResponse>
    where TResponse : notnull
{
    0 references
    public async Task<TResponse> Handle(TRequest request, RequestHandlerDelegate<TRe
    {
        logger.LogInformation("[START] Handle request={Request} - Response={Response
            typeof(TRequest).Name, typeof(TResponse).Name, request);

        var timer = new Stopwatch();
        timer.Start();

        var response = await next();
```

MediatR Pipeline Behaviours 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 **separate classes**, using a fluent interface to specify conditions that each property of your models must satisfy.

```
public class CreateProductCommandValidator : AbstractValidator<CreateProductCommand>
{
    0 references
    public CreateProductCommandValidator()
    {
        RuleFor(x => x.Name).NotEmpty().WithMessage("Name is required");
        RuleFor(x => x.Category).NotEmpty().WithMessage("Category is required");
        RuleFor(x => x.ImageFile).NotEmpty().WithMessage("ImageFile is required");
        RuleFor(x => x.Price).GreaterThan(0).WithMessage("Price must be greater than 0")
    }
}
```

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

Eshop Modules uses Shared Library

Structured Logging Using Serilog in .NET Minimal APIs

- **Structured logging** allows us to **capture detailed** and **structured log information**, making it easier to **query** and **analyze logs**.
- **Serilog** is a **diagnostic logging library** for **.NET applications** that provides powerful and **flexible logging** capabilities.
- It allows developers to **create structured logs**, which can be very helpful for **debugging** and **monitoring applications**.
- **Goto GitHub Page:**
- https://github.com/serilog/serilog

Serilog WriteTo Seq for Structured Logging in .NET

- **Seq** is a **centralized logging platform** that enables **efficient log management**, **analysis**, and **visualization**.
- It allows you to **store logs** in a **structured format**, making it easier to **filter** and **search** for **specific log entries**.
- **Seq** is particularly useful for **monitoring** and **debugging applications**.

Pagination with EF Core

- **EF Core** supports pagination through the Skip and Take LINQ methods.
- Allow us to **retrieve only a subset of data** corresponding to a specific page.
- **Skip Method:** Skips the first n elements of a sequence.
- **Take Method:** Returns a specified number of contiguous elements from the start of a sequence.
- **Pagination** through a combination of the **Skip** and **Take** methods, which are part of **LINQ**.
- Dividing the **data** into "**pages**" and retrieving only a specific page.

EF Core

Pagination

Develop Basket Module 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

CQRS implementation using MediatR library.

EF Core connection with PostgreSQL

Repository Pattern Implementation over EF Core and Redis Cache

Mehmet Ozkaya

Big Picture – Course Project

Basket Module Internal Architecture

Upcomming Sections for Basket

mehmetozkaya

Basket Module with Vertical Slice Architecture and CQRS

- Vertical Slice Architecture implementation with Feature folders
- CQRS implementation using MediatR library
- EF Core connection with PostgreSQL
- Minimal APIs for building fast HTTP APIs-fully functioning REST endpoints and top-level Program.cs
- Repository Pattern Implementation over EF Core and Redis Cache

Way of Development: Shared -> Modules -> API

Development Path

Module Development Path

Domain Analysis of Basket Module

1. Domain Models
2. Application Use Cases
3. Rest API Endpoints
4. Underlying Data Structures

1. Domain Models of Basket Module

- Primary domain model is '**ShoppingCart**', '**ShoppingCartItem**', and '**Basket Checkout**'

```
public class ShoppingCart
{
    5 references
    public string UserName { get;
    2 references
    public List<ShoppingCartItem>
    1 reference
    public decimal TotalPrice =>

    0 references
    public ShoppingCart(string us
    {
        UserName = userName;
    }
```

- Consider Domain event for **BasketCheckout**, leading to integration events.
- Also handle **ProductPriceChanged** Integration event published from Catalog.

2. Application Use Cases of Basket Module

CRUD Basket Operations:

- Get ShoppingCart with Items
- Create and Update ShoppingCart with Items that includes;
  - Add Item into ShoppingCart
  - Remove Item into ShoppingCart
  - Update ShoppingCartItem into ShoppingCart (increase Quantity or Price)
- Delete ShoppingCart with Items

Sync Basket Operations:

- When AddItem into Basket: sync get product price from Catalog module

Async Basket Operations:

- Checkout Basket and Publish event to RabbitMQ Message Broker
- Handle Product price changed integration event in basket; update existing basket with new product item prices

3. Rest API Endpoints of Basket Module

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/{userName}/items**	Add a new item to the basket
POST	**/basket/{userName}/items /{itemId}**	Remove an item from the basket
POST	**/basket/checkout**	Checkout basket

4. Underlying Data Structures of Basket Module

- **Basket Module** has **2 Datastore: PostgreSQL** and **Redis**
- Create a **Basket schema** within the **PostgreSQL database** for **data isolation**.
- Use **Entity Framework Core's Code-First approach** for **database migrations** and **entity configurations**.
- **Configure** our **entities** and their **relationships** using **EF Core**.
- We will add **Redis** upcoming sections.

Basket Module Internal Architecture

Upcomming Sections for Basket

mehmetozkaya

Technical Analysis of Basket Module

1. Application Architecture Style
2. Patterns & Principles
3. Libraries Nuget Packages
4. Project Folder Structure

1. Application Architecture Style of Basket Module

Vertical Slice Architecture

- Organizes our code into feature folders, each feature encapsulated in a single .cs file

2. Patterns & Principles of Basket Module

- **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 divi operations into commands (write) and queries (read).
- **Mediator Pattern:** Facilitates object interaction through a 'mediat 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.

3. Libraries Nuget Packages of Basket Module

- **MediatR for CQRS:** MediatR library simplifies the implementation of the CQRS pattern.
- **Carter for API Endpoints:** Routing and handling HTTP requests, easier to define API endpoints with clean and concise code.
- **EF Core for PostgreSQL Interaction:** EF Core provides a robust ORM for interacting with PostgreSQL, enabling us to manage database operations with ease.
- **Mapster for Object Mapping:** Mapster is a fast, configurable object mapper that simplifies the task of mapping objects.
- **FluentValidation for Input Validation:** For building strongly-typed validation rules, ensure inputs are correct before processed.
- **Next Sections:**Scrutor, gRPC, StackExchange.Redis, MassTransit

4. Project Folder Structure of Basket Module

- The project is organized into **Data, Feature Folder-Basket** and **Models**.
- **Features** like **GetBasket** and **CreateBasket** have dedicated handlers and endpoint definitions.
- **Feature** folder will be **Basket**.
- **Data folder** and **Repository objects** manages database interactions.

Way of Development: Shared -> Modules -> API

Development Path

Module Development Path

Way of Development: Shared -> Modules -> API

Domain Models of Basket Module

- Primary domain model is '**ShoppingCart**', '**ShoppingCartItem**', and '**Basket Checkout**'

```
public class ShoppingCart
{
    5 references
    public string UserName { get; set; } = default!;
    2 references
    public List<ShoppingCartItem> Items { get; set; } = new();
    1 reference
    public decimal TotalPrice => Items.Sum(x => x.Price * x.Quantity)

    0 references
    public ShoppingCart(string userName)
    {
        UserName = userName;
    }
```

- Consider Domain event for **BasketCheckout**, leading to integration events.
- Also handle **ProductPriceChanged** Integration event published from Catalog.

Anemic-Domain Model Entity

- **Entity** have **little** or **no business logic** (domain logic).
- Essentially **data structures** with **getters** and **setters**.
- But the **business rules** and **behaviors** are typically **implemented outside the entity**, often in service layers.

```
public class Order
{
    0 references
    public List<OrderItem> OrderItems { get; set; }
    // Other properties...
}
```

- **Order class** is **anemic** because it **only contains data** and lacks any domain logic or behaviors.

Rich-Domain Model Entity

- **Entities encapsulate** both **data** and **behavior**.
- This model enriches entities with methods that **embody business rules** and **domain logic**.

```
public class Order : Aggregate<Guid>
{
    1 reference
    private readonly List<OrderItem> _orderItems = new();
    0 references
    public IReadOnlyList<OrderItem> OrderItems => _orderItems.AsReadOnly();
    0 references
    public void AddOrderItem(OrderItem item)
    {
        // Domain logic to add an item
    }
    0 references
    public void RemoveOrderItem(OrderItemId itemId)
    {
        // Domain logic to remove an item
    }
    // Other properties...
}
```

- **Order** is a **rich domain model** as it includes methods **AddOrderItem** and **RemoveOrderItem** which **encapsulate the business logic** for manipulating the order items.

Anemic-domain vs Rich-domain Model Entities

Anemic Domain Model:

- Business logic is scattered across the application, often in services.
- Entities are simple data containers.
- Lead to issues with maintainability and understanding the business logic.

Rich Domain Model:

- Encapsulates business logic within the entities.
- More closely represents the real-world business domain.
- Can be more complex to design but leads to a more maintainable and cohesive model.

Order Example with Rich Domain Model

```
public class Order : Aggregate<OrderId>
{
    3 references
    private readonly List<OrderItem> _orderItems = new();

    // Encapsulated method to add an item
    0 references
    public void AddOrderItem(OrderItem item)
    {
        // Business logic for adding an item
        _orderItems.Add(item);
    }
    // Encapsulated method to remove an item
    0 references
    public void RemoveOrderItem(OrderItemId itemId)
    {
        // Business logic for removing an item
        var item = _orderItems.FirstOrDefault(i => i.Id == itemId)
        if (item != null)
        {
            _orderItems.Remove(item);
        }
    }
```

Develop Basket Data-Infrastructure with EF Core Code First and PostgreSQ

EF Core 8 Features, Code First Approach, Migrations and PostgreSQL Connection

Develop EF Core DBContext Object for Storing Entities

Develop Order Entity Configurations w/ ModelBuilder

Auto Migrate and Seed EF Core Entities

Raise & Dispatch Domain Events with EF Core and MediatR

Mehmet Ozkaya

Big Picture – Course Project

Basket Module Internal Architecture

Upcomming Sections for Basket

mehmetozkaya

Basket Data-Infrastructure with EF Core Code First and PostgreSQL

- EF Core 8 Features, Code First Approach, Migrations and PostgreSQL Connection
- Develop EF Core DBContext Object for Storing Entities
- Mapping DDD Objects to EF Entities
- Develop ShoppingCart Entity Configurations w/ ModelBuilder
- Auto Migrate and Seed EF Core Entities
- Raise & Dispatch Domain Events with EF Core and MediatR

4. Project Folder Structure of Basket Module

- The project is organized into **Data, Feature Folder-Basket** and **Models**.
- **Features** like **GetBasket** and **CreateBasket** have dedicated handlers and endpoint definitions.
- **Feature** folder will be **Basket**.
- **Data folder** and **Repository objects** manages database interactions.

Implement Separate Schema in EShop Modular Monoliths

Multi-context EF DB Context for DB Separate Schema

Way of Development: Shared -> Modules -> API

Development Path

Module Development Path

3. Libraries Nuget Packages of Basket Module

- **MediatR for CQRS:** MediatR library simplifies the implementation of the CQRS pattern.
- **Carter for API Endpoints:** Routing and handling HTTP requests, easier to define API endpoints with clean and concise code.
- **EF Core for PostgreSQL Interaction:** EF Core provides a robust ORM for interacting with PostgreSQL, enabling us to manage database operations with ease.
- **Mapster for Object Mapping:** Mapster is a fast, configurable object mapper that simplifies the task of mapping objects.
- **FluentValidation for Input Validation:** For building strongly-typed validation rules, ensure inputs are correct before processed.
- **Next Sections:**Scrutor, gRPC, StackExchange.Redis, MassTransit

EF Core Tools

- Tooling for EF Core migrations and reverse engineering (scaffolding) from an existing database has 2 way:

1- .NET Core CLI:

- **dotnet-ef** and **Microsoft.EntityFrameworkCore.Design** for cross-platform command line tooling
  - **dotnet tool install --global dotnet-ef**
  - **dotnet add package Microsoft.EntityFrameworkCore.Design**

2- Package Manager Console (PMC) in Visual Studio:

- **Microsoft.EntityFrameworkCore.Tools** for PowerShell tooling that works in the Visual Studio **Package Manager Console**
  - **Install-Package Microsoft.EntityFrameworkCore.Tools**

Eshop Modules uses Shared Library

EF Core Developments Steps for Basket Module

1- Install EF Core Nuget Packages:

- Microsoft.EntityFrameworkCore.Tools for PowerShell tooling that works in the Visual Studio Package Manager Console
  - Install-Package Microsoft.EntityFrameworkCore
  - Install-Package Microsoft.EntityFrameworkCore.Tools
  - Install-Package Microsoft.EntityFrameworkCore.PostgreSQL

2- Develop DBContext Object:

- Adding Entity Objects into Ef Core DBContext object

```
public class CatalogDbContext : DbContext
{
    0 references
    public CatalogDbContext(DbContextOptions<CatalogDbContext> options)
        : base(options) { }

    11 references
    public DbSet<Product> Products => Set<Product>();

    0 references
    protected override void OnModelCreating(ModelBuilder builder)
    {
        builder.HasDefaultSchema("catalog");
        builder.ApplyConfigurationsFromAssembly(Assembly.GetExecutingAssembly()
        base.OnModelCreating(builder);
    }
```

BasketDbContext

```
public class BasketDbContext : DbContext
  {
    public BasketDbContext(DbContextOptions< BasketDbContext> options) : base(options) { }
    public DbSet<Product> Products { get; set; }
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
      modelBuilder.HasDefaultSchema("basket");
      modelBuilder.Entity<ShoppingCart>(entity =>
      {
        entity.HasKey(e => e.Id);
        entity.Property(e => e.Name).IsRequired();
        entity.Property(e => e.Price).HasColumnType("decimal(18,2)");
      });
      base.OnModelCreating(modelBuilder);
    }
  }
}
```

EF Core Developments Steps for Basket Module - continue

3- Configure Domain Entities to EF Core Entities for PostgreSQL:

- Using Ef Core ModelBuilder for mapping DDD Entities to EF Core Entities

```
public class ProductConfiguration : IEntityTypeConfiguration<Product>
{
    0 references
    public void Configure(EntityTypeBuilder<Product> builder)
    {
        builder.HasKey(p => p.Id);

        builder.Property(p => p.Name).HasMaxLength(50).IsRequired();

        builder.Property(p => p.Category).IsRequired();

        builder.Property(p => p.Description).HasMaxLength(200);

        builder.Property(p => p.ImageFile).HasMaxLength(100);

        builder.Property(p => p.Price).IsRequired();
```

Entity Framework Core

4- Asp.net Core Setup & Configurations:

- appsettings.json Database connection string

```
"ConnectionStrings": {
  "Database": "Server=localhost;Port=5432;Database=EShopDb;User Id=postgres;Password=postgres;
```

Setup PostgreSQL DB using Docker-compose file

https://hub.docker.com/_/postgres

EF Core Developments Steps for Basket Module - continue

5- Register EF Core DBContext to Asp.Net:

- Register into CatalogModule.cs

```
services.AddDbContext<CatalogDbContext>((sp, options) =>
{
    options.AddInterceptors(sp.GetServices<ISaveChangesInterceptor>());
    options.UseNpgsql(connectionString);
});
```

6- Create Migrations and create Database w/ Commands:

- Install-Package Microsoft.EntityFrameworkCore.Tools
- Add-Migration InitialCreate
- Update-Database

Entity Framework Core

Wiring Up Dependencies Between Modules

EF Core Developments Steps for Basket Module - continue

7- Create Database w/ Migrate method:

- Api-Program.cs to automatically apply migrations

```
public static IApplicationBuilder UseCatalogModule(this IApplicationBuilder app)
{
    // Configure the HTTP request pipeline.

    // 1. Use Api Endpoint services

    // 2. Use Application Use Case services

    // 3. Use Data - Infrastructure services
    app.UseMigration<CatalogDbContext>();

    return app;
```

8- Seed Database:

- Create Seed/Extentions.cs

```
public static IApplicationBuilder UseMigration<TContext>(this IApplicationBuilder
    where TContext : DbContext
{
    MigrateDatabaseAsync<TContext>(app.ApplicationServices).GetAwaiter().GetResult

    SeedDataAsync(app.ApplicationServices).GetAwaiter().GetResult();
```

Entity Framework Core

Way of Development: Shared -> Modules -> API

Development Path

Module Development Path

EF Core Developments Steps for Basket Module - continue

9- Dispatch DomainEvents w/ EF Core Interceptor:

- Use EF Core SaveChangesInterceptor

```
internal class DispatchDomainEventsInterceptor(IMediator mediator) : SaveChangesIntercept
{
    0 references
    public override InterceptionResult<int> SavingChanges(DbContextEventData eventData, I
    {
        DispatchDomainEvents(eventData.Context).GetAwaiter().GetResult();

        return base.SavingChanges(eventData, result);
```

10- Test all from local w/ containerize PostgreSQL in Docker:

- Docker PostgreSQL Setup - Backing Services

```
if (!await dbContext.Products.AnyAsync())
{
    await dbContext.Products.AddRangeAsync(InitialData.Products);
    await dbContext.SaveChangesAsync();
}
```

Entity Framework Core

EF Core 8 Relations w/ Has/With Pattern

- **Relationships** with other entities **one-to-one, one-to-many,** or **many-to-many**.
- EF Core handles these relationships using the **Has/With pattern**.
- **Relationships** can be configured using **HasMany/WithOne** methods for one-to-many and **HasOne/WithMany** for many-to-one relationships.

One-to-Many Relationships:

- **Order** and **OrderItem** have a **one-to-many relationship** (an order can have multiple items) configured using EF Core Fluent API as follows:

```
builder.Entity<Order>()
        .HasMany(o => o.OrderItems)
        .WithOne()
        .HasForeignKey(oi => oi.OrderId);
```

- Each **OrderItem** references an **Order** through a **foreign key OrderId**.

ValueObject Mapping with ComplexType and ComplexProperty

- EF Core 8, "**Complex Types**" are introduced to better support **value objects** in **DDD**.
- **Complex type** is an object that does not have a primary key and is used to represent a set of properties in an entity.

Entity Framework Core

Example of Complex Types (Value Objects):

- **Address** can be a **complex type** representing the shipping and billing addresses for an order. And Configuring **Complex Types** in **OnModelCreating**:

```
[ComplexType]
0 references
public class Address
{
    0 references
    public required string Line1 { get; set; }
    0 references
    public string? Line2 { get; set; }
    0 references
    public required string City { get; set; }
    0 references
    public required string Country { get; set; }
    0 references
    public required string PostCode { get; set; }
}
```

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Order>(b =>
    {
        b.ComplexProperty(e => e.BillingAddress);
        b.ComplexProperty(e => e.ShippingAddress);
    });
}
```

Setup PostgreSQL DB using Docker-compose file

https://learn.microsoft.com/en-us/dotnet/architecture/microservices/

EShop Modular Monolith Deployment Strategy

https://learn.microsoft.com/en-us/dotnet/architecture/microservices/

EShop Microservices Deployment Strategy

1. Local Development with Docker Compose for Backing Services

- Focus on local microservices development.
- Use Docker Compose to spin up backing services like databases, which are exposed to our local environment.
- Test microservices locally while connecting to containerized backing services.

2. Full Docker Environment Orchestration

- Orchestrating all microservices, including their dependencies, within a Docker Compose environment.
- Simulates a more production-like environment, allowing end-to-end testing.

3. Future Deployment to Kubernetes and Cloud

- Orchestrating all Deploy microservices to Kubernetes, including local Minikube and cloud-based EKS (Elastic Kubernetes Service)

kubernetes

Basket Application Use Case Development with CQRS and MediatR

Develop Command and Query class with CQRS and MediatR

Develop Basket Create/DeleteBasketCommand and Handler using BasketDbContext in Basket Module

Develop Basket Queries with CQRS and MediatR in Catalog Application Layer

Develop Basket AddItemIntoBasketHandler with CQRS and Media

Mehmet Ozkaya

Big Picture – Course Project

Basket Module Internal Architecture

EShop Basket Module

Internal "Basket" Module Architecture

Client

API

New Feature

Vertical Slice

UI

Application

Domain

Infrastructure

CQRS

Vertical Slice Architecture
+ DDD + CQRS

EShop DB

PostgreSQL

Upcomming
Sections for Basket

gRPC

RabbitMQ

Message Sender

Message Receivers

Seq

Entity Framework Core

PostgreSQL

ASP.NET Core

.NET 8

mehmetozkaya

Basket Data-Infrastructure with EF Core Code First and PostgreSQL

- Develop Command and Query class with CQRS and MediatR
- Develop Basket CreateBasketCommand and Handler using BasketDbContext in Basket Module
- Develop DeleteBasketCommand and Handler using BasketDbContext in Basket Module
- Develop Basket Queries with CQRS and MediatR in Basket Module
- Develop Basket AddItemIntoBasketHandler with CQRS and MediatR
- Develop Basket RemoveItemFromBasketHandler with CQRS and MediatR

Way of Development: Shared -> Modules -> API

Development Path

Module Development Path

3. Libraries Nuget Packages of Basket Module

- **MediatR for CQRS:** MediatR library simplifies the implementation of the CQRS pattern.
- **Carter for API Endpoints:** Routing and handling HTTP requests, easier to define API endpoints with clean and concise code.
- **EF Core for PostgreSQL Interaction:** EF Core provides a robust ORM for interacting with PostgreSQL, enabling us to manage database operations with ease.
- **Mapster for Object Mapping:** Mapster is a fast, configurable object mapper that simplifies the task of mapping objects.
- **FluentValidation for Input Validation:** For building strongly-typed validation rules, ensure inputs are correct before processed.
- **Next Sections:**Scrutor, gRPC, StackExchange.Redis, MassTransit

4. Project Folder Structure of Basket Module

- The project is organized into **Data, Feature Folder-Basket** and **Models**.
- **Features** like **GetBasket** and **CreateBasket** have dedicated handlers and endpoint definitions.
- **Feature** folder will be **Basket**.
- **Data folder** and **Repository objects** manages database interactions.

Command and Command Handler in CQRS

- **Commands:** Change the state of the system. encapsulates the data required for a specific action.
- **Command Handlers:** Each command has a corresponding handler. The handler interacts with the domain model and persists changes.
- Executes business logic and updates the state through repositories.

CQRS and MediatR Request LifeCycle

https://learn.microsoft.com/en-us/dotnet/architecture/microservices

MediatR and ASP.NET Core Minimal Apis

- **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**.

- Controller receives a request, it **creates** a **command** or **query** object and sends it to **MediatR**. MediatR then **dispatches** this object to the **appropriate handler**.

https://learn.microsoft.com/en-us/dotnet/architecture/microservices/

MediatR and ASP.NET Core Minimal Apis

- **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. **(REPR pattern)**

- **Minimal Api** receives a request, it **creates** a **command** or **query** object and sends it to **MediatR**. MediatR then **dispatches** this object to the **appropriate handler**.

Eshop Modules uses Shared Library

Dispatch Domain Events w/ EF Core SaveChangesInterceptor

Steps dispatch Events:

- Create a SaveChangesInterceptor
- Identify and Dispatch Domain Events
- public class DomainEventDispatcherInterceptor : SaveChangesInterceptor
- Registering the Interceptor into DependencyInjection class

Best Practice to dispatch Events:
EF Core SaveChangesInterceptor

https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/domain-events-design-implementation

Eshop Catalog Domain and Integration Events

Best Practice to dispatch Events:
EF Core SaveChangesInterceptor

Steps dispatch Events:

- Create a SaveChangesInterceptor
- Identify and Dispatch Domain Events
- public class DomainEventDispatcherInterceptor : SaveChangesInterceptor
- Registering the Interceptor into DependencyInjection class

Basket API Endpoints Exposing Minimal Api Endpoints with Carter and REPR Pattern

Develop Get/Create/Delete Basket POST Endpoint with Carter implements ICarterModule

Develop Add/Remove ItemIntoBasket Endpoint with Carter implements ICarterModule

Register MediatR and Carter libraries into Asp.Net Dependency Injection Service and Request Pipelines

Mehmet Ozkaya

Big Picture – Course Project

Basket Module Internal Architecture

Upcomming Sections for Basket

mehmetozkaya

Basket API Endpoints Exposing Minimal Api Endpoints with Carter and REPR Pattern

- Develop CreateBasket POST Endpoint with Carter implements ICarterModule
- Develop DeleteBasket Endpoint with Carter implements ICarterModule
- Develop GetBasket Endpoint with Carter implements ICarterModule
- Develop AddItemIntoBasket Endpoint with Carter implements ICarterModule
- Develop RemoveItemFromBasket Endpoint with Carter implements ICarterModule
- Register MediatR and Carter libraries into Asp.Net Dependency Injection Service and Request Pipelines

Way of Development: Shared -> Modules -> API

Development Path

Module Development Path

2. Application Use Cases of Basket Module

CRUD Basket Operations:

- Get ShoppingCart with Items
- Create and Update ShoppingCart with Items that includes;
  - Add Item into ShoppingCart
  - Remove Item into ShoppingCart
  - Update ShoppingCartItem into ShoppingCart (increase Quantity or Price)
- Delete ShoppingCart with Items

Sync Basket Operations:

- When AddItem into Basket: sync get product price from Catalog module

Async Basket Operations:

- Checkout Basket and Publish event to RabbitMQ Message Broker
- Handle Product price changed integration event in basket; update existing basket with new product item prices

3. Rest API Endpoints of Basket Module

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/{userName}/items**	Add a new item to the basket
DELETE	**/basket/{userName}/items /{itemId}**	Remove an item from the basket
POST	**/basket/checkout**	Checkout basket

Command and Command Handler in CQRS

- **Commands:** Change the state of the system. encapsulates the data required for a specific action.
- **Command Handlers:** Each command has a corresponding handler. The handler interacts with the domain model and persists changes.
- Executes business logic and updates the state through repositories.

REPR Design Pattern

- Developing **API Endpoints** implementing a simple **Request EndPoint Response** flow.
- **REPR** stands for **Request, Endpoint, Response** – a pattern simplifying the development of **REST API endpoints**.
- **REPR Design Pattern** defines web API endpoints as havi components:
- **Request -** Data structure the endpoint expects.
- **Endpoint -** Logic the endpoint performs given a request.
- **Response -** Data structure the endpoint returns.
- Simplified concepts to enable the development REST API endpoints and enforcing the **Single Responsibility Principle** for **endpoints**.
- **Developers** need to **care** about are the **Request** and **Response**.

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 APIs.
- **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.
- **Goto Github page:**
- https://github.com/CarterCommunity/Carter

2. Patterns & Principles of Basket Module

Presentation Api Layer Patterns

- Minimal APIs
- REPR Design Pattern
- Asp.net DI, Routing with latest Asp.net 8 features

SECTION FLOW

- Domain > Infrastructure > Application > Presentation

3. Libraries Nuget Packages of Basket Module

- **MediatR for CQRS:** MediatR library simplifies the implementation of the CQRS pattern.
- **Carter for API Endpoints:** Routing and handling HTTP requests, easier to define API endpoints with clean and concise code.
- **EF Core for PostgreSQL Interaction:** EF Core provides a robust ORM for interacting with PostgreSQL, enabling us to manage database operations with ease.
- **Mapster for Object Mapping:** Mapster is a fast, configurable object mapper that simplifies the task of mapping objects.
- **FluentValidation for Input Validation:** For building strongly-typed validation rules, ensure inputs are correct before processed.

CQRS and MediatR Request LifeCycle

https://learn.microsoft.com/en-us/dotnet/architecture/microservices

4. Project Folder Structure of Basket Module

- The project is organized into **Data, Feature Folder-Products** and **Models**.
- **Features** like **GetBasket** and **CreateBasket** have dedicated handlers and endpoint definitions.
- **Feature** folder will be **Basket**.
- **Data folder** and **Repository objects** manages database interactions.

Wiring Up Dependencies Between Modules

Eshop Modules uses Shared Library

Distributed Caching with Redis in Basket Module

Implement Repository Pattern and Develop BasketRepository class

Implement Proxy Pattern, Decorator Pattern and Scrutor Library

Implement Cache-aside Pattern / Cache invalidation with Redis

Implement the Decorator Pattern Developing Cached BasketRepository with Scrutor Library

Setup Redis as a Distributed Cache using Docker-compose

Mehmet Ozkaya

Big Picture – Course Project

Basket Module Internal Architecture

Develop Distributed Caching with Redis in Basket Module

- Implement Repository Pattern and Develop BasketRepository class
- Implement Proxy Pattern, Decorator Pattern and Scrutor Library
- Implement Cache-aside Pattern / Cache invalidation with Redis
- Implement the Decorator Pattern Developing Cached BasketRepository with Scrutor Library
- Setup Redis as a Distributed Cache using Docker-compose file for Multi-container docker environment
- Integrate Redis into our Basket Module to cache user cart data.
- Boost performance by reducing the load on our database and speeding up data retrieval.

Why Choose Redis for Distributed Caching

- **Redis** is an **advanced key-value store** known for its 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.
- Its support for **various data structures** makes it versatile for different use cases.
- **Redis** is an **excellent choice for modular architecture** primarily due to its inherent distributed characteristics.
- Enabling services to **access shared data quickly** and reducing the load on databases.

Way of Development: Shared -> Modules -> API

Development Path

Module Development Path

Basket Module Repository Class Diagram

- **Patterns:** Repository, Decorator and Cache-aside Pattern

Basket Module Repository Class Diagram

redis

StackExchange.Redis
Scrutor

- **Patterns:** Repository, Decorator and Cache-aside Pattern

Basket Module Internal Architecture

CQRS and MediatR Request LifeCycle

https://learn.microsoft.com/en-us/dotnet/architecture/microservices

3. Rest API Endpoints of Basket Module

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/{userName}/items**	Add a new item to the basket
DELETE	**/basket/{userName}/items /{itemId}**	Remove an item from the basket
POST	**/basket/checkout**	Checkout basket

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 applications 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 for Microservices - 2

- **Cache-aside pattern** can **improve performance** of a **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.

Process:

- When a service needs to access data, it first checks to see if the data is in the cache.
- If the data is in the cache, the service retrieves it from the cache and returns it to the caller.
- If the data is not in the cache, the service retrieves it from the database or other data store, stores it in the cache, and then returns it to the caller.

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 its structure. It involves a set of decorator classes that are used to extend the functionality of the original class without changing its code.
- Useful for adding functionalities to objects at runtime.
- For example, enhancing a window object with additional features like borders, scrollbars dynamically.

How Can We Use Decorator Pattern?

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.

Developing Basket Module w/ Redis

Setup Redis using Docker-compose file

https://learn.microsoft.com/en-us/dotnet/architecture/microservices/

EShop Modular Monolith Deployment Strategy

https://learn.microsoft.com/en-us/dotnet/architecture/microservices/

Custom JsonConverter When Reading to Redis Cache

- Develop **Custom JsonConverter** for ShoppingCart and Items When **Reading** to **Redis Cache**.
- Handle **deserialization** and **serialization** of our domain models when **reading** from and **writing** to **Redis cache**.
- Necessary due to the **rich domain model** of DDD (Domain-Driven Design) which uses **private setters** and **internal constructors.**

Steps:

- Step 1: Modifying CachedBasketRepository
- Step 2: Developing Custom JsonConverters: ShoppingCartConverter, ShoppingCartItemConverter

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 & 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).

```
builder.Services.AddScoped<IBasketRepository, BasketRepository>();
builder.Services.Decorate<IBasketRepository, CachedBasketRepository>();
```

Eshop Modules uses Shared Library

CQRS

Entity Framework Core

StackExchange.Redis
Scrutor

Sync Communications between Modules with In-process Method Calls (Public APIs)

How Modular Monolithic Architecture Handle Communications Between Modules - Basket Module Use Cases for Sync

AddItemIntoBasketHandler Class ShoppingCart.AddItem Price and ProductName

Sync Communication Between Modules Using Contracts CL

Create Catalog.Contracts Class Library Move Query or Commands

Mehmet Ozkaya

Big Picture – Course Project

Sync Communications with In-process Method Calls

Sync Communications between Modules with In-process Method Calls (Public APIs)

- How Modular Monolithic Architecture Handle Communications Between Modules - Basket Module Use Cases for Sync
- AddItemIntoBasketHandler Class ShoppingCart.AddItem Price and ProductName
- Sync Communication Between Modules Using Contracts Class Libraries
- 1- Create Catalog.Contracts Class Library
- 2- Move exposing MediatR method Query or Commands into Catalog.Contracts Class
- Spliting CQRS Contract related common operations into Shared.Contracts Class library
- 3- Modify Catalog Module getting reference from Catalog.Contracts Class Library
- 4- Basket Get reference from Catalog.Contracts Class Library
- 6- Get Product entity values into AddItemIntoBasketHandler and set Price and ProductName information

Compare of Monolith, Microservices, Modular Monolith

	Architectures	Communications	Data Management	Transaction Management
Monolith	Traditional, single, unified application	Same process (In-process Method Calls)	Single Database	Single Transaction Scope
Microservices	Distributed architecture independent services	Inter-Service Communication over the network	Polyglot Persistence	Distributed Transactions, Eventual Consistency
Modular Monolith	Well-defined, loosely coupled modules	Same process (In-process Method Calls) – Public API	Shared or Coordinated Database – DB Schemas	Single Transaction Scope

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 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** its 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**.
- **Asynchronous** communication also divided by 2:
  - **one-to-one(queue)**
  - **one-to-many (topic)**

How Modular Monolithic Architecture Handle Communications Between Modules

- Different **modules** often need to **interact** to **fulfill business requirements**.
- Various methods for **inter-module communication**, including synchronous (**sync**) and asynchronous (**async**) methods.

- **Synchronous communication** involves **direct method calls** between modules.
- When **one module calls** a method in **another module**, it **waits** (or blocks) until the called method **completes** its **execution** and **returns** a **response**.
- **Easy** to **implement** within a **single process**.

In-process Method Calls (Public APIs)

- **In-process method calls** are a common way to **achieve synchronous communication** in a modular monolithic architecture.
- **Each module exposes** a set of **public APIs** (**methods**) that **other modules** can **call**.
- These **APIs** are typically **defined** in **interfaces**, ensuring a **clear contract** for **interaction**.
- **Benefits:** Performance, Simplicity, Ease of Debugging, Strong Typing.

Handling Dependencies and Decoupling in Public APIs

Basket Module Synchronous Use Case: Add an Item into Basket - Communicate with Catalog Module

- **Add Item into SC:**
- Checking the product's price from the Catalog module.
- Adding the item to the cart if it's available.

Dependency Inversion Principles (DIP)

- Provide to **broke dependency** of classes by **inverting dependencies**, inject dependent classes via constructor.
- **Upper-level** modules or classes and **lower-level** classes **must not be dependent** on modules.
- **Lower-level** modules **must be dependent** on **higher-level** modules (interfaces of modules).

Introduction to Contracts

- **Instead of** modules **directly depending on each other**, they should **communicate** via **Contracts** class libraries.
- **Contracts** libraries contain **shared abstractions** (like DTOs, Queries, and Commands) that **facilitate communication between modules**.
- Allows modules to **remain independent** and **loosely coupled**, **similar to** how **microservices** interact.

Implementing Sync Communication Using Contracts

- Avoid direct references, Instead, we **expose methods** via **lightweight Contracts** class libraries, following to the **DIP-dependency inversion principle**.
- **Move Query** and **Command** record types into separate "**Contracts**" class libraries.
- Modules will **reference these Contracts** libraries to **communicate synchronously** via **MediatR**.

Steps to Implement Sync Communication Using Contracts

- 1- Create Catalog.Contracts Class Library
- 2- Move exposing MediatR method Query or Commands into Catalog.Contracts Class Library -same folder structure with feature uc folders
- 3- Modify Catalog Module getting reference from Catalog.Contracts Class Library
- 4- Basket Get reference from Catalog.Contracts Class Library
- 5- Basket use Query or Command record types, to sync communicate between Basket-Catalog to trigger GetProductByIdHandler class sending to GetProductByIdQuery object into MediatR Sender
- 6- Get Product entity values into AddItemIntoBasketHandler and set Price and ProductName information when adding item into Shopping Cart
- 7- Explain Migrating Microservices - make Contracts to exposed http calls

Basket Module Synchronous Use Case: Add an Item into Basket - Communicate with Catalog Module

- **Add Item into SC:**
- Checking the product's price from the Catalog module.
- Adding the item to the cart if it's available.

Module Project References for Sync Call

Handling Dependencies and Decoupling in Public APIs

CQRS and MediatR Request LifeCycle

https://learn.microsoft.com/en-us/dotnet/architecture/microservices

Create the Shared.Contracts Class Library

- Separate CQRS and MediatR operations into Shared.Contracts for each Module.Contracts

Module Project References for Sync Call

Migrate to Microservices: EShop Modules to Microservices

Current State: Modular Monolithic w/ Contracts

- **Add Item into SC:**
- Checking the product's price from the Catalog module.
- Adding the item to the cart if it's available.

Evolving Sync Communication to Microservices HTTP Calls

- Expose Public APIs
- Service Discovery
- API Gateway

Async Communications between Modules w/ RabbitMQ & MassTransit

Developing Async Communication between Catalog and Basket Modules

Create Shared.Messaging Class library

Develop ProductPriceChangedIntegration Event

ProductPriceChanged Integration Event: How Domain Event leads t
Integration Event

Mehmet Ozkaya

Big Picture – Course Project

Async Communications w/ RabbitMQ & MassTransit

Async Communications w/ RabbitMQ & MassTransit

- Developing Async Communication between Catalog and Basket Modules
- Examine Catalog Module Domain and Integration Events
- Create Shared.Messaging Class library, Develop ProductPriceChangedIntegration Event
- Develop MassTransit Extention Methods for abstraction over RabbitMQ Message-Broker
- ProductPriceChanged Integration Event: How Domain Event leads to Integration Event
- Catalog Publish UpdatePrice integration event
- Basket Subscribe and consume ProductPriceChanged integration event
- Using RabbitMQ Publish/Subscribe Topic Exchange Model
- Shifting In-memory bus to RabbitMQ Message broker in Masstransit

Pub/Sub RabbitMQ Architecture

https://learn.microsoft.com/en-us/dotnet/architecture/microservices

Microservices Async Communication w/ RabbitMQ & MassTransit

Why Use Asynchronous Communication?

- **Decoupling:** Asynchronous communication decouples the sender and receiver, allowing them to operate independently.
- **Scalability:** By not waiting for responses, the system can handle more requests and scale effectively.
- **Resilience:** If a module fails, messages can be retried or processed later, improving fault tolerance.
- **Responsiveness:** Users receive immediate feedback, even if the operation takes time to complete.

Problem: Chain of request and highly coupled dependent modules

Async Communication in Modular Monoliths

- **Event-Driven Architecture:** Modules communicate by publishing and subscribing to events. This decouples the modules and allows for asynchronous processing.
- **Message Brokers:** Tools like RabbitMQ, Kafka, or in-memory message buses facilitate message passing between modules.

Implementing Asynchronous Communication - 1

Step 1: Domain events leads to Integration Events:

- **Domain Events:** Events that indicate something significant has happened within a module, like "OrderPlaced" or "UserRegistered". These events are typically internal.
- Use ProductPriceChanged Domain event that raise from Catalog module in Domain Entities.
- **Integration Events:** Events intended to notify other modules about changes or actions, such as "ProductPriceUpdated" or "OrderShipped".
- We will handle domain event and publish it to as a integration event.

Implementing Asynchronous Communication - 2

Step 2: Using In-Memory Service Bus that can shift to Message Broker

- **MassTransit:** A popular .NET library for message-based communication, supporting various message brokers like RabbitMQ and Kafka.
- **In-Memory Service Bus:** For simplicity and development purposes, an in-memory message bus can be used to simulate async communication.

Fan-Out & Message Filtering with Publish/Subscribe Pattern

- **Fan-out** is a messaging pattern**; ‘fanned out’** to **multiple destination** in parallel.
- Each of destinations can **work** and **process** this messages **in parallel.**
- **Publisher/subscriber model** to **define a topic** which is **logical access point** to enabling message communication with asynchronously.
- Publisher **sends the message to the topic**, message is immediately **fanned out** to all **subscribers** of this topic.
- Each service can operate and **scale independently** and individually that **completely decoupled** and **asnycronously**.
- The publisher and the subscribers **don’t need to know** who is publishing / consuming this message that is broadcasting.
- Deliver the same message to **multiple receivers** is to use the **Fanout Publish/Subscribe Messaging Pattern**.

Publish/Subscribe Messaging Pattern

- **Publish/subscribe messaging** is a form of **asynchronous** service-to-service communication.
- Any message **published** to a **topic** is **immediately received** by all of the **subscribers** to the **topic**.
- Enable **event-driven architectures**, and **decouple applications** to increase performance, reliability and scalability.
- Applications are **decoupled** into **smaller**, **independent building blocks** that are easier to develop, deploy and maintain.
- Publish/Subscribe (Pub/Sub) messaging provides **instant event notifications** for these distributed applications.
- A **message topic** provides a **lightweight mechanism** to broadcast **asynchronous event notifications**.
- All components that subscribe to the topic receive every message, unless a **message filtering policy** is set by the subscriber.

Publish/Subscribe Messaging Pattern-2

Domain Event in DDD

- **Domain Events** represent something that **happened** in the **past** and the other parts of the **same service boundary** same domain need to react to these changes.
- **Domain Event** is a **business event** that occurs within the **domain model**. It often represents a side effect of a domain operation.
- **Achieve consistency** between **aggregates** in the same domain.
- When an order is placed, an OrderPlaced event might be triggered.
- Trigger **side effects** or **notify other parts** of the system about changes within the domain.

How to Use Domain Events in DDD?

- **Encapsulate** the event details and **dispatch** them to interested parties.
- **Communicate** changes **within the domain** to external handlers which may perform actions based on these events.

Eshop Domain Events in Ordering Module

https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/domain-events-design-implementation

Domain vs Integration Events

Domain Events:

- Published and consumed within a single domain. Strictly within the boundary of the microservice/domain context.
- Indicate something that has happened within the aggregate.
- In-process and synchronously, sent using an in-memory message bus.
- **Example:** OrderPlacedEvent.

Integration Events:

- Used to communicate state changes or events between contexts or microservices.
- Overall system's reaction to certain domain events.
- Asynchronously, sent with a message broker over a queue.
- **Example:** After handling OrderPlacedEvent, an OrderPlacedIntegrationEvent might be published to a message broker like RabbitMQ, then consumes by other microservices.

Eshop Catalog Domain and Integration Events

Domain Analysis of Async Communication between Catalog and Basket Modules

1. Event Models
2. Application Use Cases
3. Rest API Endpoints
4. Underlying Data Structures

1. Event Models of ProductPriceChangedEvent

- Primary event model is '**ProductPriceChangedEvent**'

- Consider Domain event for ProductPriceChanged, leading to integration events.

2. Application Use Cases between Catalog & Basket

Async Cases for Catalog:

- **Update Product Price:** publish domain event that lead to an integration event that contains the updated product information.

Async Cases for Basket:

- **Basket Update Price:** Consume ProductPriceChangedEvent integration event from RabbitMQ using MassTransit.
  - Get existing basket with changed productId
  - Set ProductName and Price from incoming integration event
  - Update all basket shopping carts according to changed product information

3. Rest API Endpoints of Catalog Module

Method	Request URI	Use Cases
PUT	**/products**	Update Product Price

RabbitMQ AMQP Protocol

- AMQP: Consume ProductPriceChangedEvent Event from RabbitMQ using MassTransit

4. Underlying Data Structures of Async Communications

For Async Communicatin Between Modules has 2 Backing Services:

1. **MassTransit In-memory Bus**
2. **RabbitMQ Distributed Message Broker**

Async Communications w/ RabbitMQ & MassTransit

Technical Analysis of Async Communication between Catalog and Basket Modules

1. Application Architecture Style
2. Patterns & Principles
3. Libraries Nuget Packages
4. Project Folder Structure

1. Application Architecture Style of Async Communication

Event-driven Architecture

- Focuses on events as the primary source of communication between different modules.
- Enhances decoupling, scalability, and responsiveness.

2. Patterns & Principles of Async Communication

- **Publish/Subscribe Pattern**
- A core pattern enabling event-driven architecture.
- One service (publisher) sends messages/events without knowing the receivers (subscribers). Subscribers listen and react to these events.

Class Library Architecture of Async Communicatio

Shared

- Contains common classes and logic shared across services.

Shared.Messaging

- Dedicated class library for async communication over RabbitMQ, containing common integration events and MassTransit configurations.

3. Libraries Nuget Packages of Async Communicati

- **MassTransit:** An abstraction over message brokers for .NET, simplifying the implementation of asynchronous messaging.
- **MassTransit.RabbitMQ:** Specific integration for RabbitMQ with MassTransit.

RabbitMQ

.NET Core

docker

4. Project Folder Structure of Async Communicatio

- Shared.Messaging
- Catalog Module – ProductPriceChangedEventHandler
- Basket Module – ProductPriceChangedIntegrationEventHandler

EShop Modular Monolith Deployment Strategy

https://learn.microsoft.com/en-us/dotnet/architecture/microservices/

Async Communications w/ RabbitMQ & MassTransit

Steps of Developing Async Communication between Catalog and Basket Modules

1. Examine Catalog Module Domain and Integration Events
2. Create Shared.Messaging Class library
3. Catalog Publish UpdatePrice integration event
4. Basket Subscribe and consume ProductPriceChanged integration event

EShop Modules Domain and Integration Events - 1

1- ProductPriceChanged Domain Events:

- When the product price is updated, the Catalog will publish the ProductPriceChanged domain event, triggers internal operations.

2- ProductPriceChanged Integration Event:

- ProductPriceChanged domain event will lead to an integration event that contains the updated product information.
- This updated information needs to be propagated to the Basket module to update the prices of items in the shopping carts.

Class Library Architecture of Async Communicatio

- **Catalog** -- referenced **Shared** and **Shared.Messaging**
- **Basket** -- referenced **Shared** and **Shared.Messaging**
- **Identity**
- **Ordering**-- referenced **Shared** and **Shared.Messaging**

Module Project References for Async Call

Steps of Developing Async Communication between Catalog and Basket Modules

Step 5: Test in-memory MassTransit

Step 6: Outbox Pattern for Reliable Messaging

Step 7: RabbitMQ as a Backing Services

Async Communications w/ RabbitMQ & MassTransit

Class Library Architecture of Async Communicatio

- **Catalog** -- referenced **Shared** and **Shared.Messaging**
- **Basket** -- referenced **Shared** and **Shared.Messaging**
- **Identity**
- **Ordering**-- referenced **Shared** and **Shared.Messaging**

Module Project References for Async Call

Register MassTransit in Program.cs Common Services

Eshop Modules uses Shared Libraries

Domain vs Integration Events

Domain Events:

- Published and consumed within a single domain. Strictly within the boundary of the microservice/domain context.
- Indicate something that has happened within the aggregate.
- In-process and synchronously, sent using an in-memory message bus.
- **Example:** OrderPlacedEvent.

Integration Events:

- Used to communicate state changes or events between contexts or microservices.
- Overall system's reaction to certain domain events.
- Asynchronously, sent with a message broker over a queue.
- **Example:** After handling OrderPlacedEvent, an OrderPlacedIntegrationEvent might be published to a message broker like RabbitMQ, then consumes by other microservices.

Eshop Catalog Domain and Integration Events

EShop Modules Domain and Integration Events - 1

1- ProductPriceChanged Domain Events:

- When the product price is updated, the Catalog will publish the ProductPriceChanged domain event, triggers internal operations.

2- ProductPriceChanged Integration Event:

- ProductPriceChanged domain event will lead to an integration event that contains the updated product information.
- This updated information needs to be propagated to the Basket module to update the prices of items in the shopping carts.

EShop Modules Domain and Integration Events - 2

3- BasketCheckout Integration Events:

- BasketCheckout is an integration event that published from Basket to RabbitMQ.
- Subscribe-Consume by Ordering from RabbitMQ using MassTransit to perform async event-driven.

4- OrderCreated Domain Event:

- When consumes BasketCheckout Integration Event into Handler class, it will create a new Order into Ordering.
- When a new Order is created, it will raise an OrderCreated domain event.

5- OrderCreated Integration Event for Order Fullfilment:

- Once the OrderCreated domain event is handled, an OrderCreatedIntegrationEvent is published for order fullfilment and implement SAGA Pattern.

Eshop All Domain and Integration Events

What is RabbitMQ ?

- **RabbitMQ** is a **message broker software** that implements the **Advanced Message Queuing Protocol (AMQP).**
- It allows applications to communicate with each other by **sending** and **receiving messages** through **queues**.
- **RabbitMQ** is a **message queuing system** that transmit a message received from any source to another source.
- Similar ones can be listed as **Apache Kafka, Msmq, Microsoft Azure Service Bus, Kestrel, ActiveMQ**.
- All **transactions** can be **listed in a queue** until the source to be transmitted gets up.
- It allows to **send** and **receive** messages **asynchronously**.
- **RabbitMQ's** support for **multiple operating systems** and open source code is one of the most preferred reasons.
- Main Components of RabbitMQ: **Producer**, **Queue**, **Consumer**, **Message**, **Exchange**, **Binding** and **FIFO**.

RabbitMQ Components: Producer, Queue, Consumer, Message, Exchange, Binding

https://www.cloudamqp.com/img/blog/exchanges-topic-fanout-direct.png

RabbitMQ Queue Properties

- **Queue Name**
  The name of the queue we have defined.
- **Durable**
  Determines the lifetime of the queue. If we want persistence, we have to set it true.
- **Exclusive**
  Contains information whether the queue will be used with other connections.
- **AutoDelete**
  Contains information about deletion of the queue with the data sent to the queue passes to the consumer side.

https://www.rabbitmq.com/tutorials/amqp-concepts.html

RabbitMQ Exchange Types

- **RabbitMQ** provides **four types** of **exchanges**: Direct, Fanout, Topic, and Headers.
- **Direct Exchange**
  Uses of a single queue is being addressed. The most appropriate queue is reached with the relevant direct exchange.
- **Topic Exchange**
  Messages are sent to different queues according to their subject. Incoming message is classified and sent to the related queue.
- Variation of the Publish / Subscribe pattern. Topic Exchange should be used to determine what kind of message they want to receive.

https://www.rabbitmq.com/tutorials/amqp-concepts.html

RabbitMQ Exchange Types-2

- **Fanout Exchange**
  Used in situations where the message should be sent to more than one queue. Especially applied in Broadcasting systems.
- Mainly used for games for global announcements.
- **Headers Exchange**
  Guided by the features added to the header of the message. Routing-Key used in other models is not used.
- Transmits to the correct queue with a few features and descriptions in message headers.
- The attributes on the header and the attributes on the queue must match each other's values.

https://www.rabbitmq.com/tutorials/amqp-concepts.html

Async Communications w/ RabbitMQ & MassTransit

Setup RabbitMQ using Docker-compose file

https://learn.microsoft.com/en-us/dotnet/architecture/microservices/

Class Library Architecture of Async Communicatio

- **Catalog** -- referenced **Shared** and **Shared.Messaging**
- **Basket** -- referenced **Shared** and **Shared.Messaging**
- **Identity**
- **Ordering**-- referenced **Shared** and **Shared.Messaging**

Module Project References for Async Call

Develop User Identity Module with Keycloak Authentication

Keycloak Identity and Access Management

OAuth2 + OpenID Connect Flows with Keycloak

Keycloak as a Backing Services of Modular Monolith Modules

Setup Keycloak into Docker-compose file for Identity Provider

Setup Keycloak User Identity Database as a PostgreSQL DB identit

Mehmet Ozkaya

Big Picture – Course Project

Identity Module Internal Architecture

Develop User Identity Module with Keycloak Authentication

- Keycloak Identity and Access Management
- OAuth2 + OpenID Connect Flows with Keycloak
- Keycloak as a Backing Services of Modular Monolith Modules
- Setup Keycloak into Docker-compose file for Identity Provider as a Backing Service
- Setup Keycloak User Identity Database as a PostgreSQL DB identity schema
- Create Realm, User and Client for OpenID Connect with Keycloak Identity Provider
- Secure EShop Modules with Keycloak OpenId Connect in AspNet
- JwtBearer token for OpenID Connect with Keycloak Identity
- Get Current User from Token with ClaimsPrincipal in Aspnet Authentication
- Secure Basket Endpoints with OpenID Connect using Keycloak

Keycloak Identity and Access Management

- **Open-source Identity** and **Access Management** solution aimed at modern applications and services.
- Provides **comprehensive support** for **authentication** and **authorization**, offering features such as **Single Sign-On (SSO), Identity Brokering**, and **Social Login**.
- **Simplifies** the **process** of **securing apps** and **services** with **minimal coding** effort.
- **Single Sign-On (SSO):** Users can log in once and access multiple app.
- **Identity Brokering and Social Login:** Integrate with external identity providers like Google, Facebook, etc.
- **User Federation:** Connect to existing user databases.
- **Centralized Management:** Manage all user identities and permissions in one place.
- **Standard Protocols:** Supports OAuth 2.0, OpenID Connect, SAML 2.0.

Why Use Keycloak ?

- Significantly **reduce the complexity** of **managing user identities** and **access controls** in your apps.
- **Security:** Robust security features that adhere to industry standards.
- **Scalability:** Capable of handling millions of users and requests.
- **Flexibility:** Easy to integrate with various applications and services.
- **User Experience:** Enhances user experience with SSO and Social Login.

Main Advantages of using Keycloak

Single Sign-On (SSO)

- Allows users to authenticate once and gain access to multiple applications without needing to log in again.

Standard Protocols

- OAuth 2.0, OpenID Connect, and SAML 2.0 that provide secure and standardized authentication and authorization mechanisms.

Centralized Management for Admins and Users

- Enables administrators to manage users, roles, and permissions from a single console. Ensures consistent policy enforcement across all apps.

Identity Brokering and Social Login

- Allows users to authenticate using external identity providers such as Facebook, GitHub, Google, Instagram, Twitter, Paypal.

Main Advantages of using Keycloak - 2

Built-in Admin Portal

- Provides a user-friendly interface for managing realms, clients, roles, and users.

Integration with SQL Databases

- Supports integration with various SQL databases including MySQL, Microsoft SQL, MariaDB, and PostgreSQL.

High Performance

- Capable of handling millions of users and transactions efficiently. This ensures that your apps remain responsive and scalable.

Password Policies

- Enforce strong security practices by requiring users to adhere to specific password rules, such as minimum length, complexity, and expiration policies.

Best Practices when using Keycloak

Use a Docker Compose File

- Use a Docker Compose File to manage your Keycloak deployment.
- Simplifies the setup and management of Keycloak and its dependencies, ensure consistent and reproducible environments.

Use Kubernetes Nodes

- Protect your Identity Server and have a fallback mechanism.
- Kubernetes provides robust orchestration capabilities, ensuring high availability, scalability, and disaster recovery for your Keycloak deployment.

Use PostgreSQL

- Use PostgreSQL as the backend database for Keycloak.
- PostgreSQL is a powerful, open-source relational database.

Use Integrated Middleware from .NET SDK

- Integrated middleware from .NET SDK for secure and seamless integration with your .NET applications.
- JwtBearer or OpenIdConnect Middlewares to handle authentication and authorization.

```yaml
version: '3.4'

services:
  keycloak:
    image: jboss/keycloak:latest
    container_name: keycloak
    environment:
      - KEYCLOAK_USER=admin
      - KEYCLOAK_PASSWORD=admin
      - DB_VENDOR=postgres
      - DB_ADDR=postgres
      - DB_DATABASE=keycloak
      - DB_USER=keycloak
      - DB_PASSWORD=password
    ports:
      - "8080:8080"
    restart: always

  postgres:
    image: postgres:latest
    container_name: postgres
    environment:
      - POSTGRES_DB=keycloak
      - POSTGRES_USER=keycloak
      - POSTGRES_PASSWORD=password
    ports:
      - "5432:5432"
    restart: always
```

OAuth2 + OpenID Connect

OAuth2

- Authorization framework that allows apps to obtain limited access to user accounts on an HTTP service.
- Delegating user authentication to the service that hosts the user account and authorizing third-party apps to access the user account.
- **Key Concepts of OAuth2:** Resource Owner, Client, Authorization Server, Resource Server.

OpenID Connect (OIDC)

- Identity layer built on top of OAuth2.
- Clients to verify the identity of the end-user based on the authentication performed by an authorization server and to obtain basic profile information about the end-user.
- **ID Token:** A JSON Web Token (JWT) that contains user profile info.
- **UserInfo Endpoint:** An endpoint that returns additional profile information about the user.

OAuth2 + OpenID Connect Flows

Authorization Code Flow

- 1. The client directs the user to the Keycloak login page.
- 2. The user authenticates and consents.
- 3. Keycloak redirects the user back to the client with an authorization code.
- 4. The client exchanges the authorization code for an access token and an ID token.

Implicit Flow

- SPAs where tokens are returned directly to the client. However, it's less secure and generally not recommended for new applications.

Client Credentials Flow

- Used for machine-to-machine (M2M) communication. The client authenticates directly with Keycloak using its client ID and secret to obtain an access token.

OAuth2 + OpenID Connect Flows - 2

Resource Owner Password Credentials Flow

- The client to collect the user's credentials and exchange them for an access token.
- This flow is suitable only for highly trusted clients and is generally not recommended.

OpenID Connect Flow

- Extends the OAuth2 Authorization Code Flow to include an ID token, which contains user profile information.
- This flow is ideal for applications that need to verify the user's identity and obtain basic profile information.

Keycloak: OAuth2 + OpenID Connect Compliant Provider

Keycloak is an open-source identity and access management solution that provides support for OAuth2 and OpenID Connect.

Keycloak.AuthServices.Authentication Library

- Simplifies the integration of Keycloak with ASP.NET Core applications.
- It bridges the gap between Keycloak's configuration and the ASP.NET authentication model.

Benefits of Keycloak.AuthServices.Authentication Library

- **Ease of Use:** Simplifies the setup and integration process.
- **Compatibility:** Ensures compatibility with ASP.NET Core's authentication configuration.
- **Convenience:** Provides a convenience library to speed up the integration process.

Big Picture – Course Project

Setup Keycloak as a Backing Service using Docker-compose file

https://learn.microsoft.com/en-us/dotnet/architecture/microservices/

Migrate to Microservices: EShop Modules to Microservices

EShop Modular Monoliths Extracted Identity

Setup Keycloak as a Backing Service using Docker-compose file

https://learn.microsoft.com/en-us/dotnet/architecture/microservices/

Data Isolation with Separated Identity Schema

Create Realm, User and Client for OpenID Connect w/ Keycloak Identity Provider

https://learn.microsoft.com/en-us/dotnet/architecture/microservices/

Keycloak: OAuth2 + OpenID Connect Compliant Provider

Keycloak is an open-source identity and access management solution that provides support for OAuth2 and OpenID Connect.

Keycloak.AuthServices.Authentication Library

- Simplifies the integration of Keycloak with ASP.NET Core applications.
- It bridges the gap between Keycloak's configuration and the ASP.NET authentication model.

Benefits of Keycloak.AuthServices.Authentication Library

- **Ease of Use:** Simplifies the setup and integration process.
- **Compatibility:** Ensures compatibility with ASP.NET Core's authentication configuration.
- **Convenience:** Provides a convenience library to speed up the integration process.

Secure EShop Modules with Keycloak OpenId Connect

https://www.cncf.io/blog/2023/05/17/securing-cloud-native-microservices-with-role-based-access-control-using-keycloak/

Migrate to Microservices: EShop Modules to Microservices

Big Picture – Course Project

Ordering Module with Vertical Slice Architecture and CQRS

Vertical Slice Architecture implementation with Feature folders

CQRS implementation using MediatR library.

EF Core connection with PostgreSQL

Minimal APIs for building fast HTTP APIs-fully functioning REST endpoints and top-level Program.cs

Mehmet Ozkaya

Big Picture – Course Project

Ordering Module Internal Architecture

Ordering Module with Vertical Slice Architecture and CQRS

- Vertical Slice Architecture implementation with Feature folders
- CQRS implementation using MediatR library.
- EF Core connection with PostgreSQL
- Minimal APIs for building fast HTTP APIs-fully functioning REST endpoints and top-level Program.cs

After this section we will develop

- Outbox Pattern For Reliable Messaging w/ BasketCheckout Use Case between Basket and Ordering Modules

Way of Development: Shared -> Modules -> API

Development Path

Module Development Path

Ordering Module Layers

- **Domain**
  - Tactical DDD implementations, including entities, value objects, and aggregates.
  - Anemic-domain model vs Rich-domain model Entities in DDD
  - Domain Events & Integration Events
- **Infrastructure**
  - Entity Framework Core 8 Code First Approach, Migrations and PostgreSQL Connection
  - EF Core 8 Relations and DDD ValueObject Mapping with ComplexType and ComplexProperty
  - Auto Migrate and Seed EF Core Entities to PostgreSQL DB when Application Startup
  - EF Core Interceptors: SaveChangesInterceptor
- **Application**
  - CQRS with MediatR library to implement the Command and Command Handlers
  - MediatR Pipeline Behaviours (Validation, Logging)
  - Domain Event Handlers with MediatR INotificationHandler
- **Presentation**
  - Using Carter for Minimal API endpoint definition
  - Cross-Cutting Concerns Custom Exception Handler in Ordering Module

Domain Analysis of Ordering Module

1. Domain Models
2. Application Use Cases
3. Rest API Endpoints
4. Underlying Data Structures

1. Domain Models of Ordering Module

- Primary domain model is '**Order**' and '**OrderItem**'

```
public class Order : Aggregate<OrderId>
{
    private readonly List<OrderItem>
    6 references
    public IReadOnlyList<OrderItem>

    5 references
    public CustomerId CustomerId { g
    9 references
    public OrderName OrderName { get
    10 references
    public Address ShippingAddress {
    10 references
    public Address BillingAddress {
    8 references
    public Payment Payment { get; pr
    4 references
    public OrderStatus Status { get;
```

- Consider integration event for order created action.

2. Application Use Cases of Ordering Module

CRUD Ordering Operations:

- Get Order with Items (can filter by Name and Customer)
- Create a new Order
- Update an existing Order
- Delete Order
- Add and Remove Item from Order

Asynchronous Ordering Operations:

- **Basket Checkout:** Consume Basket Checkout event from RabbitMQ using MassTransit
- **Order Fulfilment:** Perform order fulfilment operations (billing, shipment, notification..)
- Raise OrderCreated Domain Event that leads to integration event

3. Rest API Endpoints of Ordering Module

Method	Request URI	Use Cases
GET	**/orders**	Get Orders w/ Pagination
GET	**/orders/{orderName}**	Get Orders by OrderName
GET	**/orders/customer/{customerId}**	Get Orders by Customer
POST	**/orders**	Create a new Order
PUT	**/orders**	Update an existing Order
DELETE	**/order/{id}**	Delete Order with Id

- **AMQP:** Consume BasketCheckout Event from RabbitMQ using MassTransit

4. Underlying Data Structures of Ordering Module

- **Ordering Module** has **1 Datastore: PostgreSQL**
- Create a **Ordering schema** within the **PostgreSQL database** for **data isolation**.
- Use **Entity Framework Core's Code-First approach** for **database migrations** and **entity configurations**.
- **Configure** our **entities** and their **relationships** using **EF Core**.

Big Picture – Course Project

Ordering Module Internal Architecture

Technical Analysis of Ordering Module

1. Application Architecture Style
2. Patterns & Principles
3. Libraries Nuget Packages
4. Project Folder Structure

1. Application Architecture Style of Ordering Module

Vertical Slice Architecture

- Organizes our code into feature folders, each feature encapsulated in a single .cs file

2. Patterns & Principles of Ordering Module

- Design a **DDD-oriented modules**, **DDD Tactical Patterns**
- **CQRS Pattern:** Command Query Responsibility Segregation divides operations into commands (write) and queries (read).
- **Mediator Pattern:** Facilitates object interaction through a 'mediat 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.

3. Libraries Nuget Packages of Ordering Module

- **MediatR for CQRS:** MediatR library simplifies the implementation of the CQRS pattern.
- **Carter for API Endpoints:** Routing and handling HTTP requests, easier to define API endpoints with clean and concise code.
- **EF Core for PostgreSQL Interaction:** EF Core provides a robust ORM for interacting with PostgreSQL, enabling us to manage database operations with ease.
- **Mapster for Object Mapping:** Mapster is a fast, configurable object mapper that simplifies the task of mapping objects.
- **FluentValidation for Input Validation:** For building strongly-typed validation rules, ensure inputs are correct before processed.

4. Project Folder Structure of Ordering Module

- The project is organized into **Data, Feature Folder-Orders** and **Models**.
- **Features** like **GetOrder** and **CreateOrder** have dedicated handlers and endpoint definitions.
- **Feature** folder will be **Orders**.
- **Data folder** and **Repository objects** manages database interactions.

3. Rest API Endpoints of Ordering Module

Method	Request URI	Use Cases
GET	**/orders**	Get Orders w/ Pagination
GET	**/orders/{orderName}**	Get Orders by OrderName
GET	**/orders/customer/{customerId}**	Get Orders by Customer
POST	**/orders**	Create a new Order
PUT	**/orders**	Update an existing Order
DELETE	**/order/{id}**	Delete Order with Id

- **AMQP:** Consume BasketCheckout Event from RabbitMQ using MassTransit

Data Isolation with Separated Identity Schema

Way of Development: Shared -> Modules -> API

Development Path

Module Development Path

Eshop All Domain and Integration Events

Eshop Catalog Domain and Integration Events

Outbox Pattern For Reliable Modulith Messaging w/ BasketCheckout Use Case

Implement the Outbox Pattern to ensure reliable messaging

Async use case: BasketCheckout

Pub/Sub RabbitMQ Architecture

Stored in an outbox table before being sent to the message broker

Mehmet Ozkaya

Big Picture – Course Project

Outbox Pattern For Reliable Messaging

Outbox Pattern For Reliable Messaging w/ Basket & Order

- Messages are stored in a persistent store before being sent to the message broker.
- This guarantees that messages are not lost even if the system crashes.
- 1- BasketCheckout command will comes from the client app
- 2- Basket module perform their operations in a single db transaction
  - Removing basket in redis database
  - In same transaction, Save event into Outbox table
- 3- After that, another process read the Outbox table and publish BasketCheckout integration event to the RabbitMQ with using MassTransit
- 4- Ordering module will consume this event and Create an order in PostgreSQL database - ordering schema.

Domain Analysis of Async Communication between Basket and Ordering Modules

1. Event Models
2. Application Use Cases
3. Rest API Endpoints
4. Underlying Data Structures

1. Event Models of BasketCheckout Event

- Primary domain model is '**BasketCheckout**'

```
public class BasketCheckout
{
    public string UserName { get; set; }
    public decimal TotalPrice { get; set; }

    // BillingAddress
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string EmailAddress { get; set; }
    public string AddressLine { get; set; }
    public string Country { get; set; }
    public string State { get; set; }
    public string ZipCode { get; set; }

    // Payment
    public string CardName { get; set; }
    public string CardNumber { get; set; }
    public string Expiration { get; set; }
    public string CVV { get; set; }
    public int PaymentMethod { get; set; }
}
```

- Consider Domain event for basket checkout, leading to integration events.

2. Application Use Cases of Async Communication

Async Basket Operations:

- Checkout Basket and Publish event to RabbitMQ Message Broker

Asynchronous Ordering Operations:

- **Basket Checkout:** Consume Basket Checkout event from RabbitMQ using MassTransit
- **Order Fulfilment:** Perform order fulfilment operations (billing, shipment, notification..)
- Raise OrderCreated Domain Event that leads to integration event

Eshop All Domain and Integration Events

3. Rest API Endpoints of Basket Module

Method	Request URI	Use Cases
POST	**/basket/checkout**	Checkout basket

Ordering AMQP Protocol

- AMQP: Consume BasketCheckout Event from RabbitMQ using MassTransit

4. Underlying Data Structures of Async Communications

For Async Communicatin Between modules has 1 Backing Services:

1. RabbitMQ Distributed Message Broker

- **RabbitMQ** allows for robust, efficient, and **scalable asynchronous communication**. Facilitates **pub/sub messaging** between **Basket** and **Ordering modules.**
- Message Broker for both Basket and Ordering modules connect for pub/sub messaging is RabbitMQ.

Outbox Pattern For Reliable Messaging

Class Library Architecture of Async Communicatio

Shared

- Contains common classes and logic shared across services.

Shared.Messaging

- Dedicated class library for async communication over RabbitMQ, containing common integration events and MassTransit configurations.

3. Libraries Nuget Packages of Async Communicati

- **MassTransit:** An abstraction over message brokers for .NET, simplifying the implementation of asynchronous messaging.
- **MassTransit.RabbitMQ:** Specific integration for RabbitMQ with MassTransit.

RabbitMQ

.NET Core

docker

4. Project Folder Structure of Async Communicatio

- Shared.Messaging
- Basket Module – Add New feature CheckoutBasket use case
- Ordering Module – BasketCheckoutIntegrationEventHandler

Outbox Pattern For Reliable Messaging

Problem: Dual Write Problem

- When application **needs to change data** in **two different systems**, i.e. a database and a message queue, if one of the writes fails, it can result in inconsistent data.
- Happens when you use a **local transaction** with **each of the external systems** operations.
- I.e. app needs to **persist data** in the **database** and **send** a **message** to **Kafka** for notifying other systems.
- **If one of these two operations fails**, the **data** will be **inconsistent** data and these two systems becomes inconsistent.

Problems

- Data loss or corruption.
- Difficult to resolve without proper error handling and recovery mechanisms.
- Dual writes can be hard to detect and fix.

Example of Dual-write problem: Create Order

1- Change data in Order database with creating new record
2- Send an order_created event to the EventBus like Apache Kafka.

Run Transactions

- 1, 2 or
- 2, 1

- Both becomes dual writes

How to avoid dual write problems in microservices ?

- **Monolith** applications use the **2 phase commit** protocol.
- It **splits the commit** process of the transaction into 2 steps and ensures the **ACID principles** for all systems.
- **Can't use 2-phase commit** transactions when building **microservices**.
- These transactions **require locks** and **don't scale well**.
- Need all systems to be **up and running** at the same time.

Solutions

- Transactional Outbox Pattern
- CDC - Change Data Capture

Best Practice

- Use Red Hat **Apache Kafka** and **CDC** using **Debezium** in event-driven applications.
- Use New databases like **CockroachDB** which has built-in **Change Data Capture** feature.

Transactional Outbox Pattern

- The idea is to have an “**Outbox**” **table** in the microservice’s database. It provides to publish events reliably.
- **Dual write problem** happens when application needs to change data in two different systems.
- Instead of sending the data to two separate locations, **send a single transaction** that will **store two separate copies** of the data on the database.
- **One copy** is **stored** in the relevant **database table**, and the **other copy** is **stored** in an **outbox table** that will publish to event bus.
- When API **publishes event messages**, it doesn’t directly send them, Instead, the **messages** are **persisted** in a **database table**.
- After that, **a job publish events** to message broker system in predefined time intervals.
- **Events** are **not written directly** to a event bus, it is written to a table in the "**Outbox**" role of the service.

Transactional Outbox Pattern - 2

- Transaction **performed before the event** and the **event written** to the **outbox table** are part of the **same transaction.**
- When a **new order is added** to the system, the process of adding the order and writing the **Order_Created event** to the **Outbox table** is done **in the same transaction** to ensure the event is saved to the database.
- If one of the **process is fail**, this will **rollback the whole operations** with following ACID principles.
- The second step is to **receive these events** written to the **Outbox table** by an **independent service** and **write** them to the **Event bus**. Another service **listen** and **polls the Outbox table** records and publish events.

Transactional Outbox Pattern in Microservices

- **Microservice** provides an **outbox table** within **its database**. Outbox table will **include** all the **events**.
- There will be a **CDC (change data capture) plugin** that **reads** the **commit log** of the **outbox table** and **publish the events** to the relevant queue.
- It provides that **messages** are **reliably delivered** from a **microservice** to **another microservice** even if the transaction that triggered the message fails.
- It involves **storing** the **message** in a **local "Outbox" table** within the microservice, that message sent to the consumer after the transaction is committed.
- **Outbox pattern** can be used to ensure that **messages** are **delivered consistently**, even if the microservice that sent the message is **unavailable** or **experiencing errors**.
- Useful for **communicating important information** or updates between services.

Customer Places an Order Use Case
Transactional Outbox pattern in Microservices

Why Use Outbox Pattern ?

- When **working with critical data** that **need to consistent** and **need to accurate** to catch all requests.
- When the **database update** and **sending of the message** should be **atomic** to make sure **data consistency**.
- For example the **order sale transactions**, because they are about **financial business**. Thus, the calculations **must be correct 100%.**
- To access this **accuracy**, must be sure that our system is not **losing any event messages**.
- **The Outbox Pattern** should be applied this kind of cases.

Steps of Develop Outbox Pattern

- 1. Create the Outbox Entity
- 2. Modify the DbContext to include the Outbox Entity
- 3. Update the CheckoutBasketHandler to save messages to the Outbox table
- 4. Create a background service to process Outbox messages and publish them to RabbitMQ

Outbox Pattern For Reliable Messaging

Steps of Ordering Consume BasketCheckout

- 1. Project Reference to Shared.Messaging in Ordering Module
- 2. Register MassTransit RabbitMQ packages for Ordering Module
- 3. Develop Basket Checkout Event Consumer integration event handler class
- 4. BONUS- Publish OrderCreated Integration event for order fullfilment

Eshop All Domain and Integration Events

Outbox Pattern For Reliable Messaging

Checkout Basket Use Case

Basket Module:

- Create new basket with POST /basket endpoint.
- Checkout basket send POST /basket/checkout endpoint
- OutboxProcessor process outbox messages and publish basketcheckout event

RabbitMQ Dashboard:

- Basketcheckout integration event will publish from Basket module
- Basketcheckout integration event will consume from Ordering module

Ordering Module:

- BasketCheckoutIntegrationEventHandler.cs consume BasketCheckout event and send mediatR createOrderCommand

Containerize and Orchestrate EShop Modular App in Docker-Compose

Deployment strategy

Containerize and Orchestrate Api project

Full docker compose run

Test all apis on postman

Develop Client Apps

Mehmet Ozkaya

Big Picture – Course Project

EShop Modular Monolith Deployment Strategy

https://learn.microsoft.com/en-us/dotnet/architecture/microservices/

Big Picture – Course Project

Catalog Module Internal Architecture

Async Communications w/ RabbitMQ & MassTransit

Basket Module Internal Architecture

Sync Communications with In-process Method Calls

Identity Module Internal Architecture

Ordering Module Internal Architecture

Outbox Pattern For Reliable Messaging

Client Apps - Consume Api Module in EShop Modules

Modern Frameworks:
- **React**
- **Angular**
- **Blazor**

Migrating to Microservices: EShop Modules to Microservices w/ Strangler Fig Pattern

Decompose EShop Modules to Microservices with DDD Bounded Contexts

Prioritize modules-services for migration

Extract a service from a monolith, Manage a monolithic database

Design and implement interservice communication

Data consistency, Distributed transactions

Mehmet Ozkaya

Big Picture – Course Project

Migrate to Microservices: EShop Modules to Microservices

EShop Modular Monoliths Extracted Identity

Evolving Sync Communication to Microservices HTTP Calls

- Expose Public APIs
- Service Discovery
- API Gateway

Challenges of Migration

Complexity

- Splitting a monolithic application into smaller services can introduce significant complexity: architecture, deployment..

Data Management

- Ensuring data consistency across services is a major concern, particularly when dealing with distributed transactions.

Interservice Communication

- Designing efficient communication between services is critical to maintain performance and reliability.

Deployment and Monitoring

- Managing multiple services requires robust deployment strategies and comprehensive monitoring solutions.

Team Coordination

- Necessitates changes in team structure and collaboration processes.

Key Considerations of Migration

Business Priorities

- Identify the most critical parts of the application that need to be migrated first based on business priorities and potential impact.

Domain-Driven Design (DDD)

- Use DDD principles to identify bounded contexts and define service boundaries.

Incremental Approach

- Adopt an incremental migration strategy to minimize risks and allow for continuous improvement.

Technology Stack

- Ensure that the technology stack for microservices supports scalability, resilience, and maintainability.

Cultural Shift

- Encourage a cultural shift towards DevOps practices, continuous delivery, and automation.

Strangler Fig Pattern for Refactoring From Monolith

- **Strategic approach** to **refactoring monolithic** apps into microservices.
- It emphasizes **incremental migration**, ensuring **minimal disruption** to the existing system while **gradually** introducing microservices.
- Transforming a "**strangler**" (**monolith**) into a "**fig**" (**microservices**
- By **incrementally migrating** parts of the app, we **reduce the risk** of **system downtime** and **user disruption**.
- The existing **monolithic** app **continues to function** while individual modules are **extracted** as **microservices**.
- Allows for a **scalable** and **flexible transition**, **accommodating changes** and **improvements** throughout the process.
- **Gradual migration** allows for thorough **testing** and **validation** of each microservice, ensuring stability and reliability.

Considerations for Successful Migration

Incremental Approach

- Adopt an incremental migration strategy to minimize risks and allow for continuous improvement.

Team Coordination

- Ensure effective communication and collaboration among team members. Clearly define roles and responsibilities to streamline the migration process.

Cultural Shift

- Encourage a cultural shift towards DevOps practices, continuous delivery, and automation. Foster a mindset of continuous learning and adaptation.

Documentation

- Maintain comprehensive documentation includes API documentation, architecture diagrams, and deployment strategies.

Steps of Refactoring Monolith into Microservices

- **Step 1:** Decompose EShop Modules to Microservices with DDD Bounded Contexts
- **Step 2:** Prioritize Modules-Services for Migration
- **Step 3:** Extract a Service from a Monolith
- **Step 4:** Manage a Monolithic Database
- **Step 5:** Design and Implement Interservice Communication
- **Step 6:** Async Communication Between Services
- **Step 7:** Distributed Transactions and Data Consistency
- **Step 8:** Deployment of Microservices after Monolithic

Migrate to Microservices: EShop Modules to Microservices

Step 1: Decompose EShop Modules to Microservices with DDD Bounded Contexts

Identify Bounded Contexts

- Analyze the existing monolithic application to identify bounded contexts within the domain.
- For the EShop application, modules such as Catalog, Basket, Ordering, and Identity can be considered as bounded contexts.

Define Service Boundaries

- Define clear boundaries for each microservice based on the identified bounded contexts.
- Each service should encapsulate a specific business capability and be responsible for its own data.

Identifying EShop Modules

Step 2: Prioritize Modules-Services for Migration

Assess Module Complexity

- Evaluate the complexity of each module and its dependencies.
- Prioritize the migration of less complex and highly decoupled modules first to gain confidence and experience.

Business Impact

- Prioritize modules that have a significant impact on the business and user experience.
- This approach ensures that the most valuable parts of the application are migrated early.

Big Picture – Course Project

Identity Module Internal Architecture

Step 3: Extract a Service from a Monolith

Isolate Code and Data

- Extract the code and data related to the chosen module from the monolith.
- Ensure that the extracted service can function independently.

Create APIs

- Develop APIs for the extracted service to interact with other services and the monolith during the transition phase.
- Ensure backward compatibility to avoid disruptions.

EShop Modular Monoliths Extracted Identity

Step 4: Manage a Monolithic Database

Database Decomposition

- Gradually decompose the monolithic database into smaller, service-specific databases.
- Implement strategies like database replication, shared databases with schema separation, or eventual consistency to manage data

Data Synchronization

- Implement data synchronization mechanisms to keep the data consistent across services during the transition period.
- Use change data capture (CDC) or event sourcing for real-time data updates.

EShop Modular Monoliths Extracted Identity

Separated PostgreSQL Database

Step 5: Design and Implement Interservice Communication

Synchronous Communication

- Use RESTful APIs or gRPC for synchronous communication between services.
- Design APIs with clear contracts and versioning to ensure compatibility.

Asynchronous Communication

- Implement message brokers like RabbitMQ or Kafka for asynchronous communication.
- Use events and message queues to decouple services and improve resilience.

EShop Modular Monoliths Extracted Identity

Sync and Async Communication between Eshop App and Identity Service

Current State: Modular Monolithic w/ Contracts

- **Add Item into SC:**
- Checking the product's price from the Catalog module.
- Adding the item to the cart if it's available.

Evolving Sync Communication to Microservices HTTP Calls

- Expose Public APIs
- Service Discovery
- API Gateway

Step 6: Async Communication Between Services

Event-Driven Architecture

- Design services to publish and subscribe to events using an event-driven architecture.
- This approach enhances decoupling and allows services to react to changes asynchronously.

Eventual Consistency

- Embrace eventual consistency for data synchronization across services.
- Design services to handle data inconsistencies gracefully and ensure eventual resolution.

Async Communications w/ RabbitMQ & MassTransit

Microservices Async Communication w/ RabbitMQ & MassTransit

Step 7: Distributed Transactions and Data Consistency

Saga Pattern

- Implement the Saga pattern to manage distributed transactions across multiple services.
- Design compensating transactions to handle failures and ensure data consistency.

Outbox Pattern

- Use the Outbox pattern to reliably publish events as part of the transaction that updates the database.
- Ensure that events are published only if the transaction commits successfully.

Outbox Pattern For Reliable Messaging

Step 8: Deployment of Microservices after Monolithic

Containerization

- Containerize each microservice using Docker to ensure consistent deployment environments.
- Use Kubernetes for orchestration and scaling of containers.

Continuous Integration and Continuous Deployment (CI/CD):

- Implement CI/CD pipelines to automate the build, test, and deployment processes.
- Ensure that each service can be deployed independently and rolled back if necessary.

Monitoring and Logging

- Implement comprehensive monitoring and logging solutions to track the performance and health of each service.
- Use tools like Prometheus, Grafana, ELK Stack, or Jaeger for observability.

EShop Modular Monolith Deployment Strategy

https://learn.microsoft.com/en-us/dotnet/architecture/microservices/

Migrate to Microservices: EShop Modules to Microservices

.NET Microservices: DDD, CQRS, Vertical/Clean Architecture

https://github.com/aspnetrun/run-aspnetcore-microservices

Thanks

Thank you so much for being with me on this journey.

Reviews and feedback is really encourage to me for pushing forward to create new courses like this.

Mehmet Ozkaya