BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

DEVELOPING MODULAR INFORMATION SYSTEM WITH DOMAIN-DRIVEN DESIGN

VÝVOJ MODULÁRNÍHO INFORMAČNÍHO SYSTÉMU POMOCÍ DOMÉNOVĚ ŘÍZENÉHO NÁVRHU

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTOR **Bc. MICHAEL ŠKRÁŠEK**
AUTOR PRÁCE

SUPERVISOR **doc. Ing. RADEK BURGET, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2024

Master's Thesis Assignment

153935

Institut:	Department of Information Systems (DIFS)
Student:	**Škrášek Michael, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Application Development
Title:	**Developing Modular Information System with Domain-Driven Design**
Category:	Information Systems
Academic year:	2023/24

Assignment:

1. Study monolithic information system architectures with a focus on modularity (Modular Monolith), study DDD (Domain-Driven Design) and its connection to information system architecture.
2. In coordination with the supervisor, select two architectures, less and more modular, and design a solution for a chosen demonstration application.
3. Implement the server side of the application in both architectures, implement the domain as a rich domain according to DDD.
4. Implement the front end of the application. During development, consider cloud deployment, unit and integration testing, server-side or client-side caching, and other extensions in agreement with the supervisor.
5. Evaluate the results achieved.

Literature:

- Martin, R. C.: Clean Architecture: A Craftsman's Guide to Software Structure and Design, Pearson, 2017
- Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley Professional, 2003

Requirements for the semestral defence:
Items 1 and 2.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Burget Radek, doc. Ing., Ph.D.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	17.5.2024
Approval date:	30.10.2023

Abstract

This thesis deals with monolithic architectures and Domain-Driven Design (DDD) and its combination in the development of modular information systems. It provides comprehensive overview of Domain-Driven Design principles and various monolithic architectures, including Clean Architecture and Modular Monolith architecture. It then demonstrates the use of these patterns and architectures on a demonstration application. The thesis offers insights into the development of information systems using popular approaches such as CQRS, Clean Architecture, Domain-Driven Design, Modular Monolith architecture, and more.

Abstrakt

Tato diplomová práce se zabývá monolitickými architekturami a doménově řízeným návrhem (angl. Domain-Driven Design) a jejich kombinací při vývoji modulárních informačních systémů. Poskytuje ucelený přehled principů doménově řízeného návrhu a různých monolitických architektur, včetně čisté architektury a modulární monolitické architektury. Poté demonstruje použití těchto vzorů a architektur na ukázkové aplikaci. Práce nabízí pohled na vývoj informačních systémů s využitím populárních přístupů, jako jsou CQRS, čistá architecktura, doménově řízený přístup, modulární monolit a další.

Keywords

Domain-Driven Design, Modular Monolith, Majestic Monolith, Monolith, Microservices, Clean Architecture, CQRS, Modular Information System, Information System, Rich Domain Model, Strategy Design, Domain, Sub-domain, Bounded Context, Tactical Design, Aggregates, Entities, Value Objects, Domain Events, Integration Event, Domain Services, Transactional Outbox Pattern, Outbox, Inbox, Information System Architecture, Distributed Monolith, .NET, dotnet, Blazor, ASP.NET Core, Entity Framework Core, Postgres, MassTransit, TeamUp

Klíčová slova

doménově řízený návrh, modulární monolit, majestátní monolit, monolit, mikroslužby, čistá architecture, CQRS, modulární informační systém, informační systém, bohatý doménový model, strategický návrh, doména, poddoména, ohraničený kontext, taktický návrh, entity, agregáty, hodnotové objekty, doménové události, integrační událost, doménové služby, transakční outbox vzor, outbox, inbox, architektura informačního systému, distribuovaný monolit, .NET, dotnet, Blazor, ASP.NET Core, Entity Framework Core, MassTransit, Postgres, TeamUp

Reference

ŠKRÁŠEK, Michael. *Developing Modular Information System with Domain-Driven Design.* Brno, 2024. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Ing. Radek Burget, Ph.D.

Rozšířený abstrakt

V oblasti architektur informačních systémů je již delší dobu velmi populární architektura mikroslužeb, a i kvůli své popularitě je často využívána v situacích, kdy její použití přináší více problémů a složitostí než užitku. Mnoho autorů zabývajících se softwarovým inženýrstvím zastává názor, že nový systém by se nikdy neměl začínat s architekturou mikroslužeb, ale vždy jako monolit (tzv. *Monolith First*).

V poslední době nabývá na popularitě architektura, která se snaží kombinovat výhody obou přístupů, monolitů i mikroslužeb, a představuje tak střední cestu pro vývoj informačních systémů – modulární monolit (angl. **Modular Monolith**). Díky svým modulárním vlastnostem je tato architektura často prezentována jako svatý grál architektur pro rozsáhlé informační systémy s možností snadné konverze na architekturu mikroslužeb.

Dalším populárním přístupem pro vývoj informačních systémů je doménově řízený návrh (angl. **Domain-Driven Design**), zaměřující se na návrh rozsáhlých informačních systémů s komplexními doménami, systematickou dekompozici systému na subsystémy a aplikaci návrhových vzorů při návrhu jejich domén.

Tato diplomová práce se zaobírá studiem monolitických architektur informačních systémů s důrazem na modularitu a propojení s principy doménově řízeného návrhu. Práce porovnává zvolenou méně modulární architekturu, čistou architekturu (angl. **Clean Architecture**), a modulární monolit; implementuje demonstrační aplikaci v obou architekturách s použitím principů doménově řízeného návrhu a dalších používaných návrhových vzorů.

Cílem této práce je přiblížit výhody a nevýhody vybraných architektur, problémy, které je třeba řešit při návrhu a implementaci, problémy pramenící z technologických či jiných důvodů a jak tyto architektury obstávají či selhávají ve srovnání s architekturou mikroslužeb.

Praktická demonstrace vývoje je předvedena na aplikaci pro správu týmu, která navzdory své nevelké doméně obsahuje poměrně složitou logiku s podrobnými autorizačními pravidly a speciální funkcí pro pozvání registrovaných i neregistrovaných uživatelů do týmu. Veškerá implementace je provedena v jazyce C# na platformě **.NET**.

Část práce popisující modelování domény této demonstrační aplikace pomocí doménově řízeného návrhu poukazuje na problematické volby při modelování a dekompozici domény, např. modelování kontextů systému (angl. **Bounded Context**), modelování agregátů, přiřazování a vlastnictví entit apod.

Z částí zabývajících se návrhem a implementací jednotlivých architektur lze vypozorovat další komplikované problémy vyskytující se při vývoji, zejména v modulárním monolitu, jako třeba prosazování konzistence pomocí doménových a integračních událostí (angl. **Domain Events** & **Integration Events**), udržování spolehlivosti systému pomocí návrhového vzoru *outbox*, identifikace modulů, vypořádání se s eventuální konzistencí atd.

Práce také uvádí implementaci frontendové aplikace jako důkaz reálné použitelnosti obou navržených a implementovaných backendových řešení.

Předposlední kapitola vyhodnocuje architektury na základě shromážděných zkušeností z uplatnění a implementace obou architektur a doménově řízeného návrhu.

Kromě zmíněných architektur, vzorů a postupů se práce zabývá také běžnými tématy v kontextu vývoje informačních systémů, např. testováním, cachováním apod.

Developing Modular Information System with Domain-Driven Design

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of doc. Ing. Radek Burget, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Michael Škrášek
May 15, 2024

Acknowledgements

I want to thank my supervisor, doc. Ing. Radek Burget, Ph.D., for his help in determining the thesis assignment and valuable feedback on its content.

Contents

Chapter 1

Introduction

"You shouldn't start a new project with microservices, even if you're sure your application will be big enough to make it worthwhile."

Martin Fowler [10]

This cautionary guidance underlines the importance of deliberate consideration in choosing the right architectural approach for a given project.

The goal of this thesis is to emphasize the ability of a monolithic architecture for rapid development and its abilities to be on par with microservices in the context of developer productivity, application maintainability and scalability, and more.

This thesis presents existing architectures, focusing on the developer productivity and maintainability aspects, with an emphasis on **Clean Architecture** and **Modular Monolith**, with the Modular Monolith as an ideal architecture for migrating to microservices.

Moreover, this thesis highlights how **Domain-Driven Design** influences and improves the development of information systems and how essential it is for developing complex systems with a complex domain.

This thesis comprises of four chapters, with the 1st chapter being the introduction.

The 2nd chapter explores basic information system architectures, followed by a detailed description of the **Domain-Driven Design** approach for modeling domains of information systems (section 2.2). Next, the chapter focuses on monolithic architectures in section 2.3, including **Clean Architecture** and **Modular Monolith** architecture. The chapter concludes with a list of design patterns commonly used in the information system design.

The 3th chapter introduces the demonstration application (section 3.1); goes through the development phases of an information system, that is, application analysis, modeling domain using the domain-driven design (section 3.3); developing clean architecture and modular monolith backends (sections 3.4 and 3.5); and includes a small section about frontend implementation (section 3.6), concluding with a section 3.7 covering interesting parts and problems of the development process.

The 4th chapter based of gathered experience summarizes selected monolithic architectures and briefly compares them in terms of modularity, complexity, maintainability, and so on.

The thesis concludes with the 5th chapter, containing final thoughts on the subject, thesis assessment, and potential further extensions.

Chapter 2

Architectures and approaches to information system design

This chapter provides a comprehensive overview of key concepts and approaches in the information systems design leaning towards designing modular monoliths.

The chapter begins with an overview of information system architectures (section 2.1), including monolith and microservices as architecture designs (covering their differences, benefits and drawbacks), continuing with potential danger of designing system as a distributed monolith, concluding with brief stop at domain model types.

This is followed by a deep dive into **Domain-Driven Design** principles (section 2.2); Strategy Design and Tactical Design and their patterns, which are widely used in development of both monolithic architectures and microservices.

The chapter than in section 2.3 explores and describes various monolithic architectures from less to more modular:

- Layered Architecture
- **Clean Architecture**
- Vertical Slice Architecture
- ending with **Modular Monolith** Architecture

The chapter concludes with covering essential design patterns (section 2.4) that are often used in the information systems design and are heavily utilized in the development process covered in the next chapter, that is: Dependency Injection, CQRS, Unit Of Work and the Transactional Outbox Pattern.

2.1 Information system architecture

In the context of information system we can talk about multiple types of architectures:

- Architecture as a way of structuring code and establishing rules that help with developing and maintaining an information system as it grows. (more see 2.3)
- Architecture describing the way information system runs as an application and what parts comprise it (servers, databases, CDN, distributed cache ...). There are 3 basic ways for running the business logic of an information system:
  - Monolith – run entire application as one unified unit on one or more servers with a load balancer, each server contains the entire app. (see 2.1.1)
  - Microservices – break down the application into individual services and run them separately on many servers. Each server holds only one microservice. (see 2.1.2)
  - Serverless – approach where the application is composed of functions that are run in the cloud on demand (allowing scale to zero).

2.1.1 Monolith

Traditional monolithic architecture refers to an application design where the entire information system is build and run as single unit. In practice, an information system is considered monolithic when all the business logic runs as one large monolithic unit, even if the UI is an application separated from the server. Although the application is made of many components (services, modules, controllers, ...), all the code resides in a single repository and the application is built and deployed as one single piece.

Benefits

- Development – at the beginning of a project, it is easy to develop a monolith and over the years several architectures have been introduced to help with development and maintenance as the application grows (see section 2.3).
- Performance – interactions between application modules and services take place in a single unit on the same machine, which means that all calls are direct function calls with low overhead.
- Testing – it is easy to perform integration tests or even end-to-end tests, because the developer always has access to all parts of the application.
- Deployment – monolithic applications are easy to deploy because the application is deployed as a single unit.

Drawbacks

- Scaling flexibility – since the application runs as a single unit, it has to be scaled as a single unit (more demanding modules cannot be scaled horizontally/vertically).

- Reliability – DoS, high demand or a single bug can potentially crash the entire application.
- Maintenance – as the code base grows, the application becomes harder to maintain (slower updates/fixes . . .) and each new developer takes increasingly longer to grasp how the system works.
- Deployment – even after a small code change (i.e. a bug fix), the entire application has to be build (long build time) and deployed (the entire app might be unavailable during deployment process) again.

Some of these drawbacks can be addressed to some extent by applying certain architectures (see 2.3), design patterns (section 2.4) and design approaches (see section 2.2). For example, introducing a load balancer would allow the application to handle higher demands through horizontal scaling, which would also make the system more resilient.

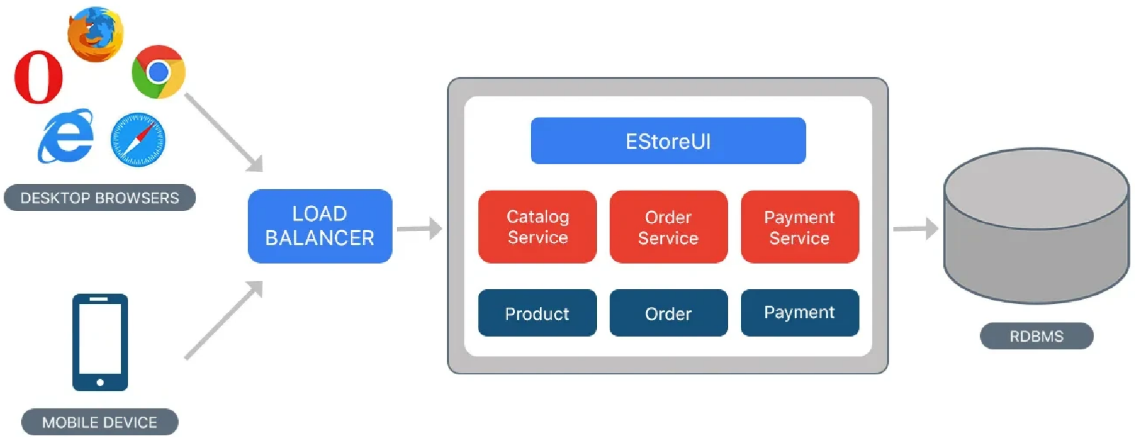


Figure 2.1: Example of monolithic architecture, taken from [22].

2.1.2 Microservices

Kirsten Westende [27] described microservices architecture as an approach in which a large application is built as a suite of smaller services that are deployed independently.

Typically, each microservice has its own repository and a team of developers working on it. This enables a large company to be split into many small teams (single-digit teams) with agile development techniques.

Each microservice runs in its own process and communication with other services using protocols such as classic REST (HTTP/HTTP), gRPC, AMQP, and so on. These microservices are not (should not be) accessible directly by the client, but through an **API gateway** (reverse proxy) that routes traffic to the desired services. Services that are not stateless need to have a custom database and changes to the same object need to be propagated across the distributed system.

Benefits

- Productivity – developers can understand small microservices much quicker and teams can take advantage of agile development, smaller services and smaller codebase also mean less stress on the developers' machines (faster IDE, fewer docker container dependencies, etc.).
- Scaling flexibility – more demanded services can be scaled independently.
- Deployment – new features and bug fixes do not affect deployment (and runtime) of other services, also faster build times and independent deployments allow for multiple deployments a day.
- Technology flexibility – each service can be developed using a different technology stack, and use cases can be solved by languages that are better suited to the problem (i.e. python for AI use cases and so on) . . . large technological disparities could also be viewed as a disadvantage.
- Reliability – isolation from other parts of the system makes it more difficult for the entire system to crash due to bugs, DoS attacks etc.

Drawbacks

- Debugging and Testing – it is hard to test the integration of a microservice with other microservices and to catch bugs that originate from these scenarios. For integration testing, the entire test environment with all microservices needs to be set up.
- Distributed systems problems and complexity – with microservices arise problems connected to distributed systems, such as understanding the system, dealing with data spread across several microservices, correctly handling data consistency through eventual consistency and so on. Complex topology may have the effect of not knowing which service is dependent on which (dependency hell), and thus be prone to cascading failures.
- Infrastructure complexity – it could be challenging to manage the infrastructure of a microservice architecture, which must have a secure internal network and operate multiple databases, servers, containers (or even cluster), resulting in higher infrastructure costs as opposed to a monolithic architecture.
- Performance – network calls between microservices bring high latency, not to mention the possibility of losing communication packets, leading to even higher latency.
- Reliability – the system might be prone to **cascading failures** – failure of a single microservice leading to a chain reaction of failures resulting in bringing down a large part or even the entire system.

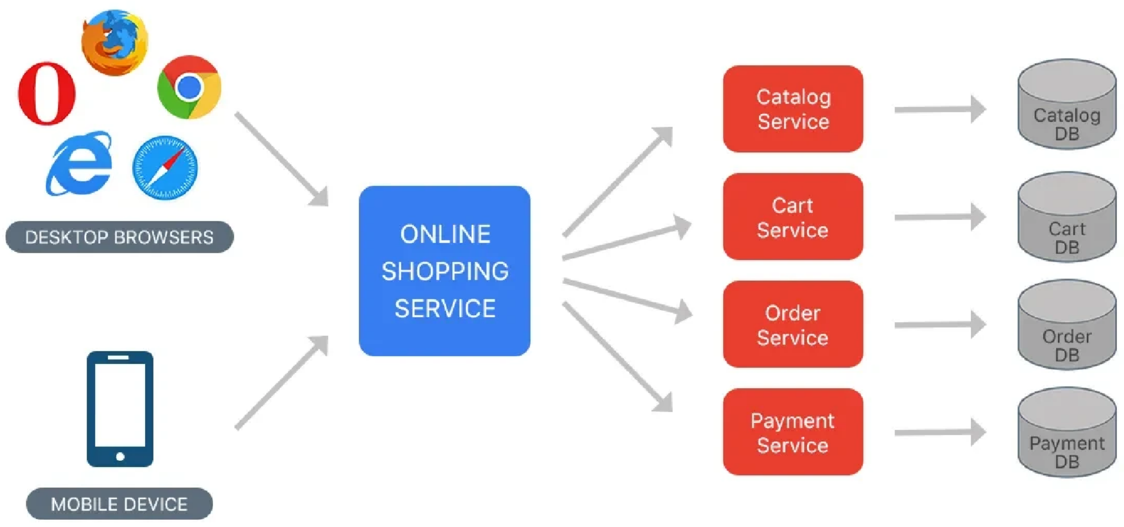


Figure 2.2: Example of microservices architecture, taken from [22].

Microservices architecture is necessary for large companies (in the context of the number of developers) to enable hundreds of people to work on the system, but it is easy to misuse this architecture and design it poorly (see the next section 2.3).

2.1.3 Distributed monolith anti-pattern

A distributed monolith is an architecture which has distributed components, but these components (services) are tightly coupled with mutual calls all over the place (also known as **big ball of mud** anti-pattern), and therefore this monolith has the drawbacks of both the monolithic and microservices architecture.

A distributed monolith usually results from a poorly designed microservices solution. That is why Martin Fowler [10] recommends starting the development of the system as a monolith first, even if using microservices would be worthwhile due to the scale of the application (aka **monolith-first strategy**).

A distributed monolith can also be the result of poorly decomposed monolith, where the original monolith was tightly coupled, and the distribution made it worse... In this case a special monolithic architecture could be used that specializes for a system that would later need to migrate to microservices or gradually peel off services (see section 2.3.4 about **Modular Monolith**).

A pragmatic approach should be used when choosing a system architecture, considering all the advantages and disadvantages for a given problem space.

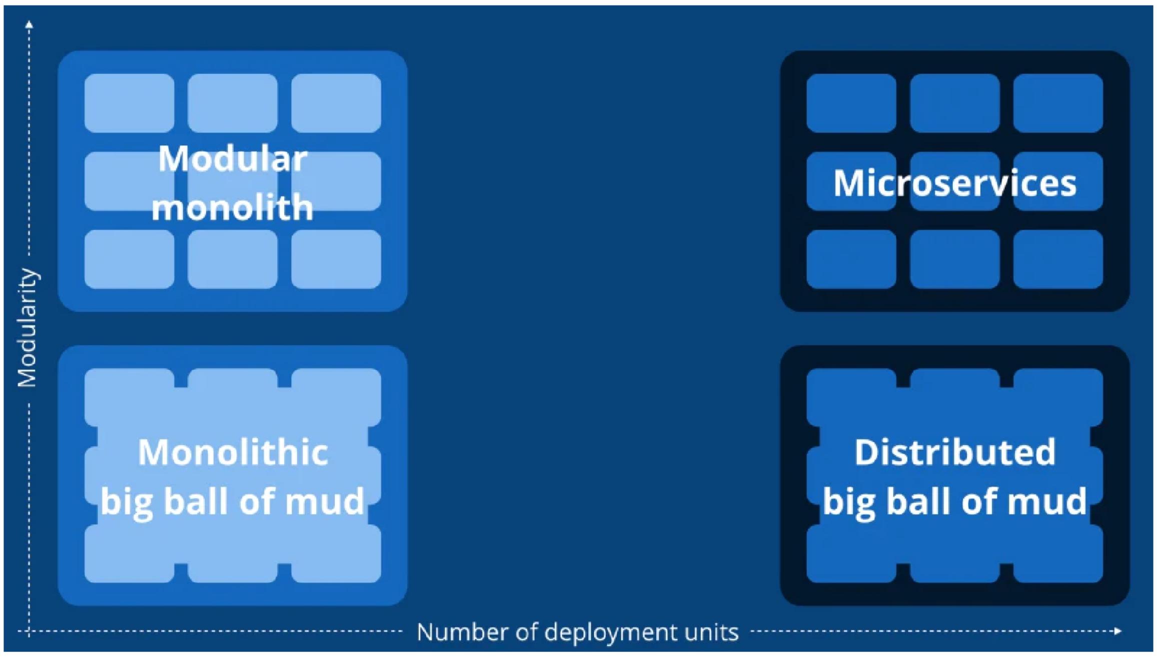

Modular monolith
Microservices
Monolithic big ball of mud
Distributed big ball of mud
Modularity
Number of deployment units

Figure 2.3: Diagram placing the types of architectures on the scale of modularity (y-axis) and distributability (x-axis). Taken from [27].

2.1.4 Domain Model

Domain model describes selected aspects of an area of knowledge (aka the domain). It outlines key entities and objects, their relationships and rules within this problem area, providing a shared understanding for developers.

Anemic domain model

The anemic domain model is a model where domain objects contain close to none business logic, this comes from the idea of separating data and logic. In this scenario, the consumer service is responsible for using the object in various business use-cases and the service is responsible for handling the anemic objects correctly (validations, calculations, business rules, . . .). Typically, an anemic object is a mirror of an entity stored in the database.

According to Martin Fowler [7], anemic domain model is an **anti-pattern** that contradicts with idea of objects-oriented design; which is to combine data and behaviour together.

Rich domain model

Rich domain model is a contrary to the anemic domain model, hence the object associated logic is within the object itself, including validation and other business rules. Rich domain model brings cohesion and encapsulation and it can be reached by Domain-Driven Design (see 2.2) or simply by object-oriented modeling.

2.2 Domain-Driven Design

Domain-Driven Design (hereinafter DDD) is an approach to software development that emphasizes understanding and modeling the business domain model. It focuses on collaboration between developers and **domain experts** to capture and implement a complex domain with complex business logic.

It was first introduced by Eric Evans in his book from 2003 [5] where he described DDD through a catalog of patterns (described in sections 2.2.1 and 2.2.2). Eric Evans also deemed as required to develop **ubiquitous language** for shared understanding of the domain and its concepts between developers and domain experts.

2.2.1 Strategy Design

The Strategy Design refers to the process of defining and modeling the high-level structure and organization of the modeled system. It is the first step in DDD developing process of large domains and it emphasizes identifying and dividing domain into the **core domains**, **subdomains** and **bounded contexts**. By dividing the domain into these smaller pieces, we can easily build models for each piece, as opposed to large enterprise-wide domain where, according to Eric Evans [5], building one uniform model is not feasible or cost-effective.

Subdomains

The entire domain of the business organization is composed of smaller and more manageable subdomains. There are 3 basic subdomains that need to be identified:

- **Core Domain** – refers to the central and most critical part of a business domain and the most important part for the success of the organization (where the business gets its money or application gets its users, i.e. product catalog and order processing for an e-commerce application etc.).

  There can be more than one core subdomain and the significance of the core subdomain should come from the business side (not the user side) . . . For example Facebook gets a lot of revenue from advertising, making it one of the core subdomains of Facebook.

- **Supporting Subdomain** – models some aspect of the business that is essential, yet not core. The business creates a supporting subdomain because it is somewhat specialized (i.e. feedback and reviews in the context of an e-commerce application).

- **Generic Subdomain** – captures nothing special to the business, but is required for the overall solution. Generic subdomains are common and standardized across the industry, solutions that could be applied with little adjustments already exist (i.e. payment processing, notifications etc.).

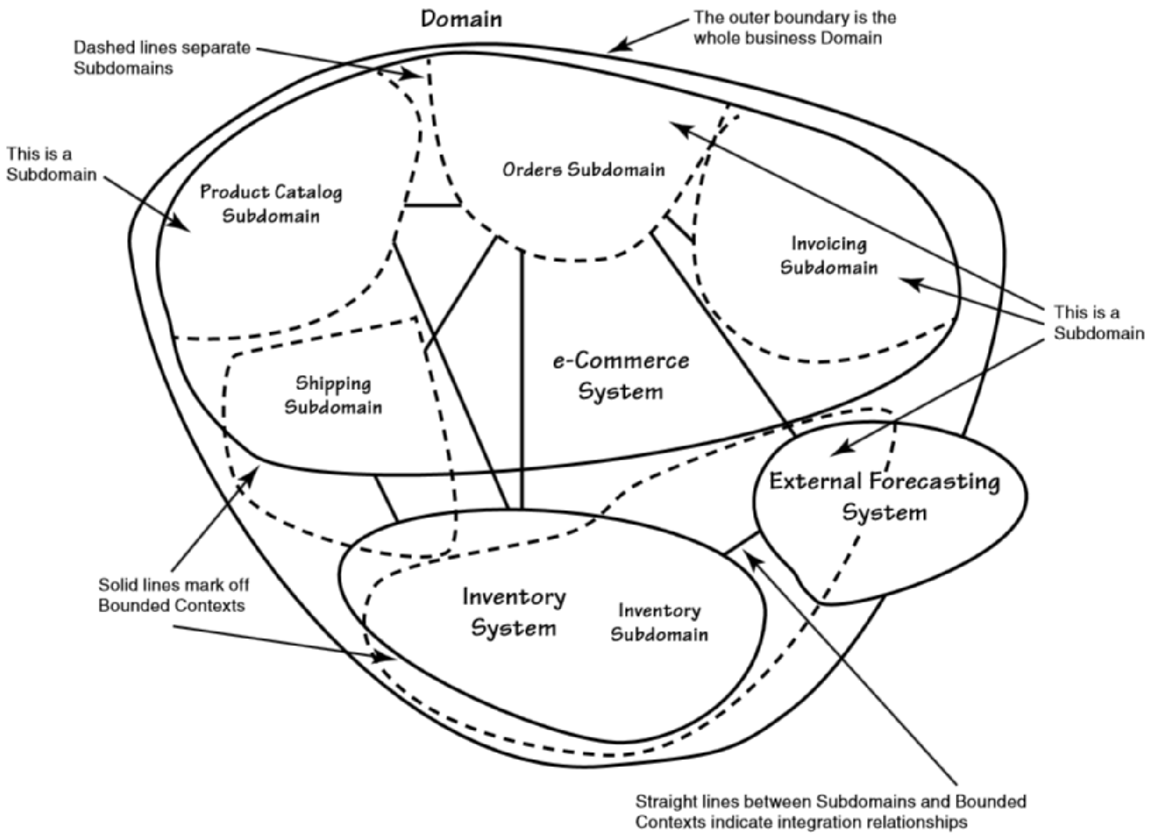

Domain
The outer boundary is the whole business Domain
Dashed lines separate Subdomains
This is a Subdomain
Product Catalog Subdomain
Orders Subdomain
Invoicing Subdomain
This is a Subdomain
e-Commerce System
Shipping Subdomain
External Forecasting System
Inventory System
Inventory Subdomain
Solid lines mark off Bounded Contexts
Straight lines between Subdomains and Bounded Contexts indicate integration relationships

Figure 2.4: Example of a domain with subdomains and bounded contexts, taken from [25].

Bounded Contexts

Bounded context is a key concept in DDD and a central pattern in the strategy design. It helps with assigning explicit meaning to the terms used in the model for a certain scope, preventing ambiguity and misunderstandings – common objects may have slightly or completely different meanings in different contexts (see figure 2.5); bounded contexts border and isolate these contexts.

To help with duplication when it comes to the same objects in multiple bounded contexts, Eric Evans [5] introduced the concept of **shared kernels**, where a shared kernel can contain shared objects from two bounded contexts on which the development teams of both models agree. Vernon Vaughn [25] cautions and specifies that the shared kernel should be small, and shared objects have special status and shouldn't be changed without consulting the other team.

A single bounded context does not necessarily fall within the scope of a single subdomain, but can span across multiple subdomains – as was illustrated in figure 2.4. From the subdomains covered by the bounded context, it can be easier to determine which domain experts are needed to collaborate with a given development team.

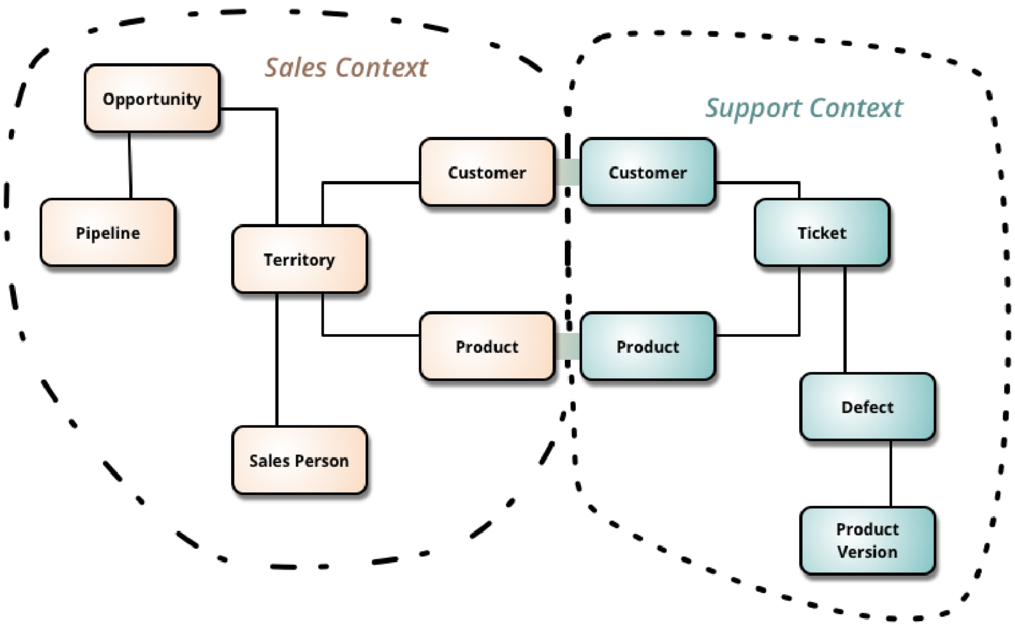


Figure 2.5: Example of bounded contexts – `Customer` and `Product` are objects that are both in `Sales Context` and `Support Context`, but could (and probably will) have a different meaning, taken from [9].

Anti-corruption Layer

On his blog, Nader Medhat [15] defines anti-corruption layer as a pattern for isolating a bounded context (or a subsystem) from an external system that uses a different or incompatible language, architecture, or data structure.

An anti-corruption layer is typically implemented as a set of service, interfaces and adapters that are placed between systems and translate external data/behavior into compatible form with the local system, and vice versa.

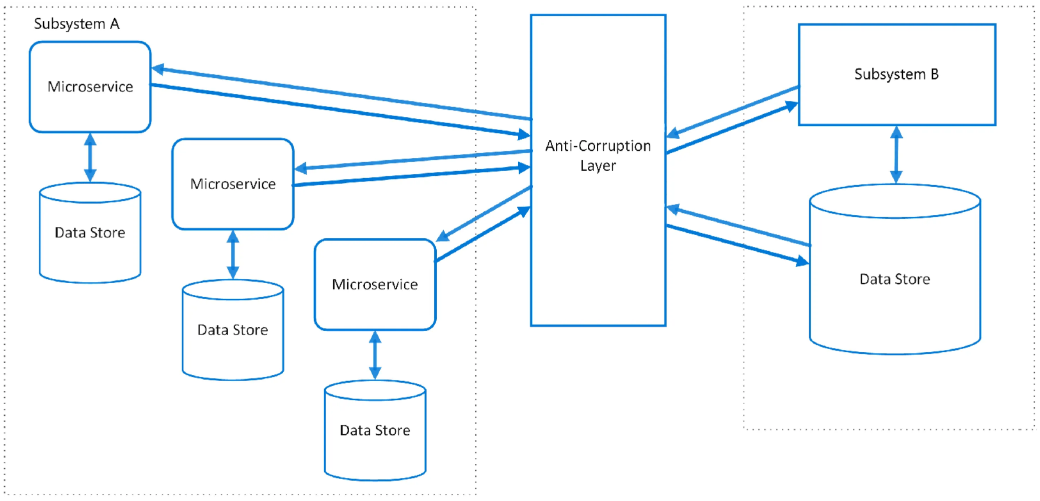


Figure 2.6: Example of an anti-corruption layer, taken from [15].

2.2.2 Tactical Design

The tactical design is a set of design patterns used in the construction of the domain of a single bounded context. These patterns are the fundamental building blocks of a domain model and they help to avoid designing the domain as a **big ball of mud**[1] (figure 2.7).

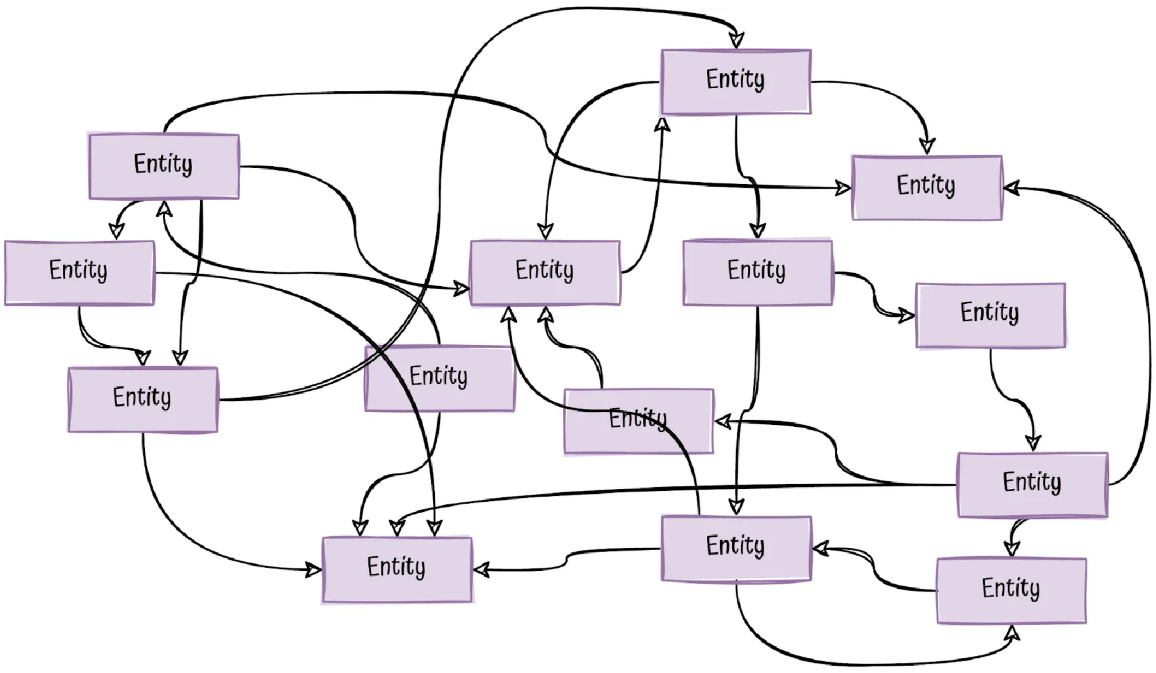


Figure 2.7: Complex graph of entities connected to each other, a software anti-pattern also known as big ball of mud. Taken from [14].

Entities & Value Objects

According to Eric Evans [5], an entity is an object primarily defined by its identity (Id). If non-identifying attribute of an entity changes (i.e. a product gets cheaper), it is still the same entity, therefore comparing two Entities should be done by comparing their Ids.

Unlike entities, value objects have no identity, but they describe some characteristic (i.e. Color). If any of the attributes of a value object change, the entire value object describes this characteristic as a new fact (i.e. red attribute of RGB Color changes), therefore the value object should be **immutable** and the comparison should be done by comparing all attributes of the value object.

[1] https://en.wikipedia.org/wiki/Anti-pattern#Big_ball_of_mud

Aggregates

To quote Eric Evans [5], *"an aggregate is a cluster of associated objects that we treat as a unit for the purpose of data changes"*. Each aggregate has a its **aggregate root** (single entity) which serves as an interface for modifying objects that are within the aggregate boundary. The root entity is the only object inside the aggregate that can be referenced by objects outside the aggregate boundary.

The aggregate root is the only object that can modify objects inside the aggregate boundary so that the aggregate root can enforce **invariants**, which Vaughn Vernon [25] describes as business rules that must always be consistent and cannot be broken, therefore:

- models cannot be created if any invariant is broken – ensured by creator (factory method or factory, alternatively constructor);
- models cannot be modified if any invariant is broken – ensured by modifying data using public methods of the aggregate root.

With this in mind, it is necessary to always retrieve the entire aggregate from the database, otherwise the aggregate root may incorrectly evaluate the business rules. This implies that the aggregate should be as small as possible to minimize large data transfers.

A typical example is an order and its line items. Although line items are separated objects, together with an order they would be treated as a single aggregate, with the order serving as the aggregate root for data changes.

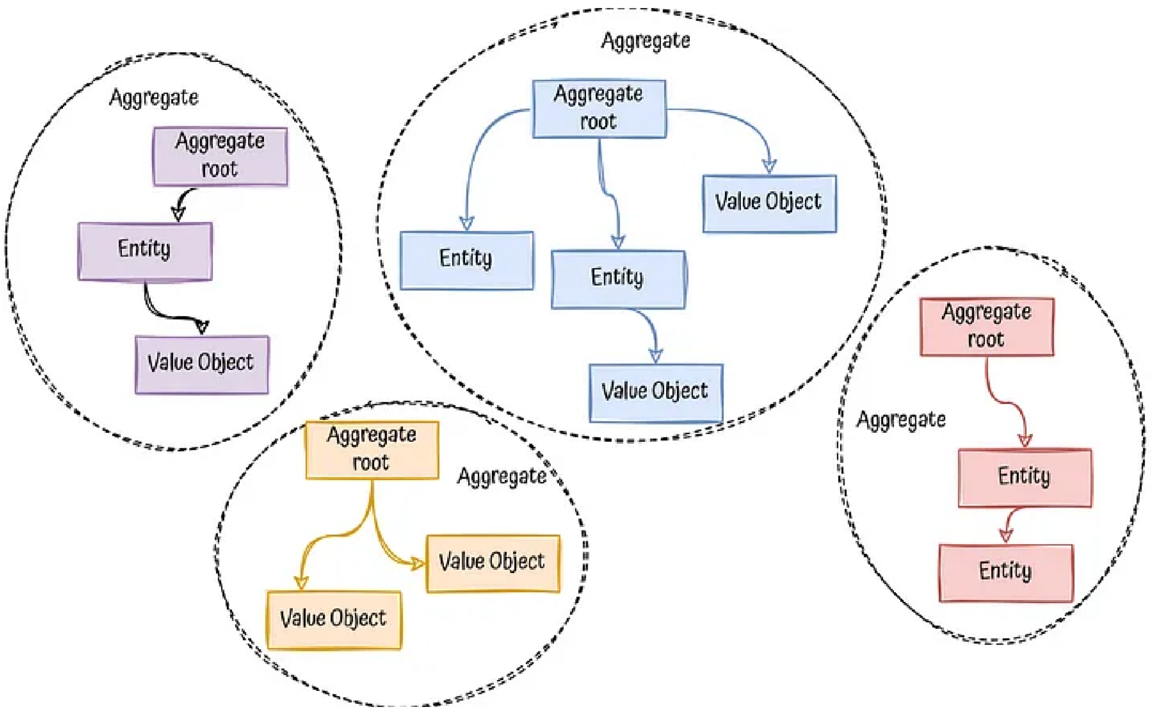


Figure 2.8: Domain modeled using DDD aggregates. Taken from [14].

Domain Events

A domain event is something that happened in the domain that other parts of the domain need to be aware of. An event is a domain object that holds necessary data for event consumers to properly react – i.e. `UserCreatedDomainEvent` should hold user identifier (id/email/...) and possibly other needed data (date, nationality, etc.).

Using domain events provides a loosely coupled and a highly cohesive model, which is crucial for designing modular systems.

The main use case for domain events is to propagate changes between aggregates, where publishing, as defined in Microsoft's book on microservices [4], can be done via 2 options:

A) Right **before** committing data to the database (this makes a single transaction including side effects from the event, possibly spanning across multiple aggregates).

B) Right **after** committing data to the database (this makes multiple transactions and a necessity to handle eventual consistency and compensatory actions for failures).

Many DDD authors, including Eric Evans [5] and Vaughn Vernon [24], advocate rule that one transaction = one aggregate, thus publishing should be done via option B.

Microsoft's book on microservices [4] explains the idea of option B by claiming that the number of database locks will be substantial in large-scale applications with high scalability needs and atomic changes are often not needed by the business.

However, Jimmy Bogard [2] argues that it is justifiable to span a single transaction across multiple aggregates if those aggregates are related to the side effects for the same original command, thus taking advantage of Option A and strong consistency.

Another use case for a domain event is to apply the operation side effects (i.e. sending email, pushing notifications, etc.). These side effects do not access the database but call other services, so publishing needs to be asynchronous outside of the database transaction with a little bit of tweaking so that the side effects do not apply if the transaction fails, this can be done using an **outbox pattern** (see 2.4.4).

Domain events can also be used to propagate committed transaction changes to different bounded contexts – modules/microservices (eventual consistency), in which case the domain event is called an **integration event** (as defined in Microsoft's book on microservices [4]). Publishing integration events should always be asynchronous, hence it is usually solved using an outbox pattern and an event bus/message-broker. This is often used with **Event Sourcing**[2] and **Event-Driven Architecture**.

Domain Services

Vernon Vaughn [25] characterized a domain service as a stateless operation that fulfills a domain-specific task. The domain service provides means to handle operations, that naturally does not belong to any of the entities or value objects. The interface of the domain service is defined in terms of other elements of the domain model (the contracts are part of the domain model).

[2] https://martinfowler.com/eaaDev/EventSourcing.html

An example of a domain service could be a `DiscountCalculator` service, in the context of an e-commence application. The order aggregate root would handle aggregate specific business rules (adding line items, calculating total price, etc.), but could not handle discount calculations, because it requires additional data that are not available in the order aggregate – user aggregate details, such as account type (standard/premium), student discounts (ISIC), etc. Calculating discount has business logic that requires access to several aggregates, making it a good candidate for a domain service.

As described in the section 2.1.4, handling all business logic in domain service would result in an anemic domain model, which is considered an anti-pattern, and DDD aims for a rich domain model (see 2.1.4), therefore it is necessary to consider each use of the domain service.

Repositories

The repository pattern is a widely used design pattern outside of DDD. The repository provides an interface for acquiring references to already existing domain objects persisted in the database and storing new domain objects. Repositories decouple the domain from the persistence technology (one or more database strategies and data sources, etc.). Another benefit mentioned by Eric Evans [5], is the possibility to use dummy in-memory collections for easy testing at the beginning of a project.

Ideally, the domain model should contain only repository interfaces with the contracts specified in the domain. Repository implementations should reside elsewhere, for example in the infrastructure or persistence layer.

Factories

The factory pattern, belonging to the Gang of Four's Creational design patterns, offers encapsulation for the creation of the domain objects. Eric Evans [5] recommends using factory when the creation of an object, or an entire aggregate, becomes complicated or reveals too much of the internal structure. Factory can enforce invartiants and use services (repositories, domain services, infrastructure services, ...), injected via DI (see 2.4.1), to resolve more complex business rules.

Alternatively, if creating a domain object does not require additional services, it is preferable to use the **factory method** design pattern, which is a static method on the objects themselves. This provides means for enforcing invariants and returning a null reference if an invariant is broken.

Constructors of targeted objects should be accessible only from factories and factory methods, so that the object creation cannot be done incorrectly:

- When using factory methods, objects should have private constructors.
- Factories are harder to solve since programming languages do not provide ways to restrict method access only to one class. Maximum language capability should be used to restrict the access (internal to the assembly etc.) and additional means could be added, such as custom code analyzers or tests for incorrect object creation.

2.3 Monolithic architectures

There are many monolithic architectures, Robert C. Martin mentions in his blog [20] that all architectures[3] have the same objective, which is the separation of concerns and production of systems that are:

- Independent of frameworks.
- Testable business rules without UI, Database, Server and other external elements.
- Independent of UI – frontend can be replaced with a console application without changing the business logic.
- Independent of Database – Database could be swapped for other database providers or even database types (i.e. SQL to NoSQL).
- Independent of any external agency – business logic does not know anything about the outside world.

This is accomplished by dividing the code into layers/modules, specifying what each layer should contain and solve, and defining strict dependency rules between them.

2.3.1 Layered Architecture

The layered or N-layered architecture splits the code into 3 layers (or more), where each layer can only depend on the lower layers. Most of monolithic architectures are based on layered architecture and are in some form layered architecture, just with more strict rules/rearranged layers/slightly different component placement etc.

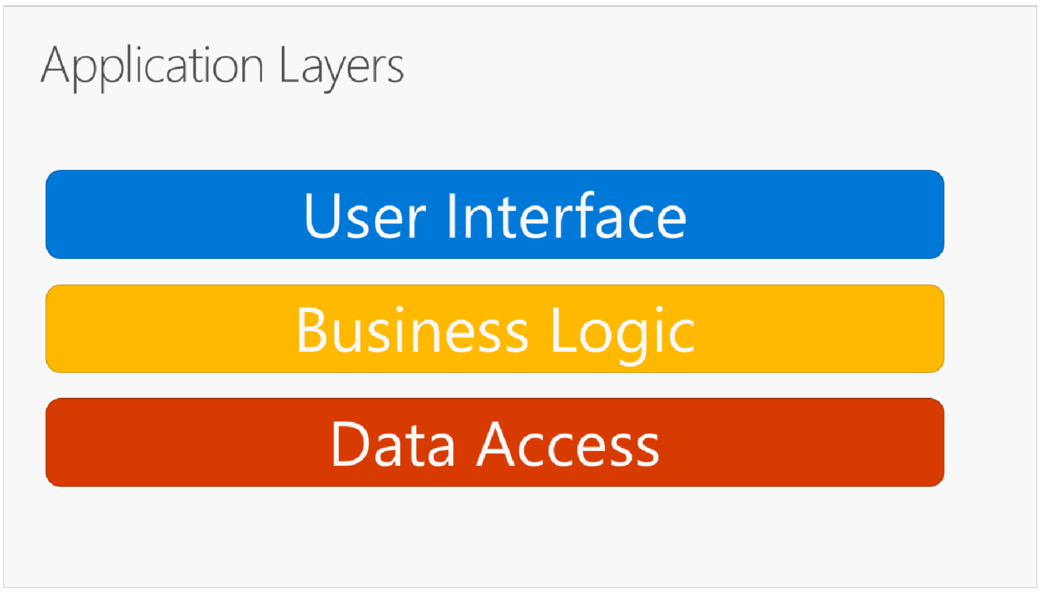


Figure 2.9: Architecture diagram of the tradition 3-layered architecture, taken from [23].

[3]He was referring to the 6 architecture examples that he had listed, but it should generally apply to all.

Typical layered architecture decomposes system into layers as follows:

- Presentation Layer – presents the system to the client in form of a UI or an API (Rest, gRPC, GraphQL, . . .).
- Business Layer (BL) or Business Logic Layer (BLL) – encompasses application business logic (services, facades, potentially mappers, DTOs and so on).
- Data Access Layer (DAL) or Data Layer (DL) – contains data specific system parts such as entities, repositories, migrations . . . and connection to the database.

There could be additional layers to extract some componests for better applicability and reusability, for example layer with API contracts (DTO requests and responses) or layer for common objects and interfaces (i.e. `IDateTimeProvider`, a result pattern and so on).

In my experience, the layered architecture often results in an anemic domain model (see 2.1.4) as it encourages to have domain objects as anemic data objects in DAL and business logic in services in BL, and is therefore not a good candidate for DDD.

2.3.2 Clean Architecture

Clean Architecture, defined by Robert C. Martin [21], is an architecture emerging from Onion Architecture (and some others) as it splits the system into concentric circles (layers).

As with the layered architecture, Clean Architecture has a dependency rule pointing inwards (see figures 2.10 and 2.11), meaning the inner layers cannot depend on the outer layers.

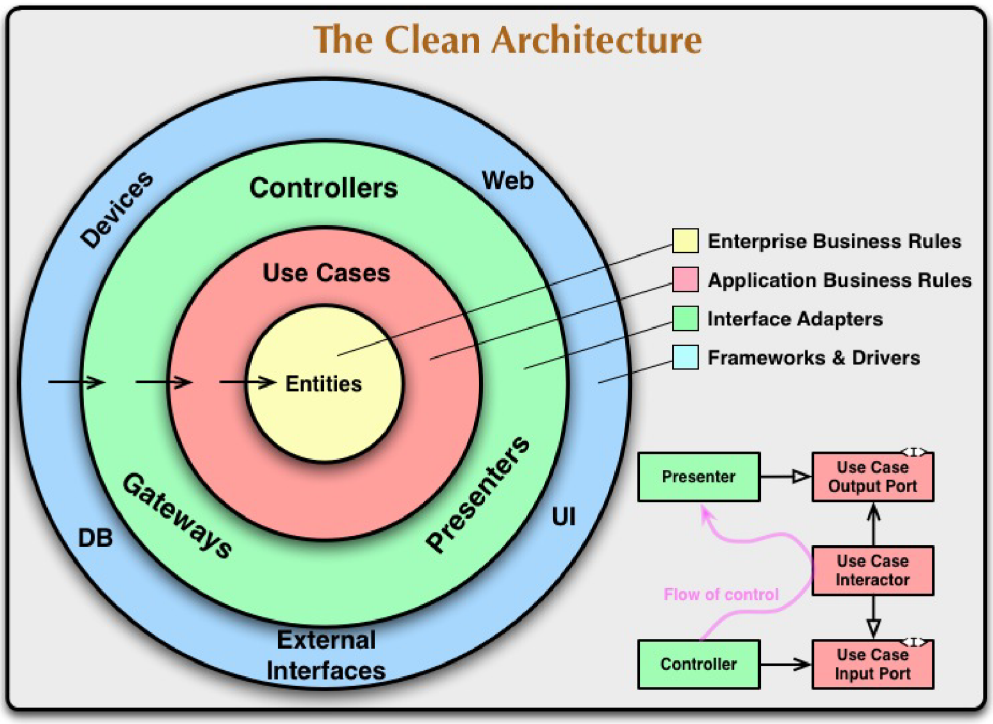


Figure 2.10: Architecture diagram of the Clean Architecture by Robert C. Martin [21].

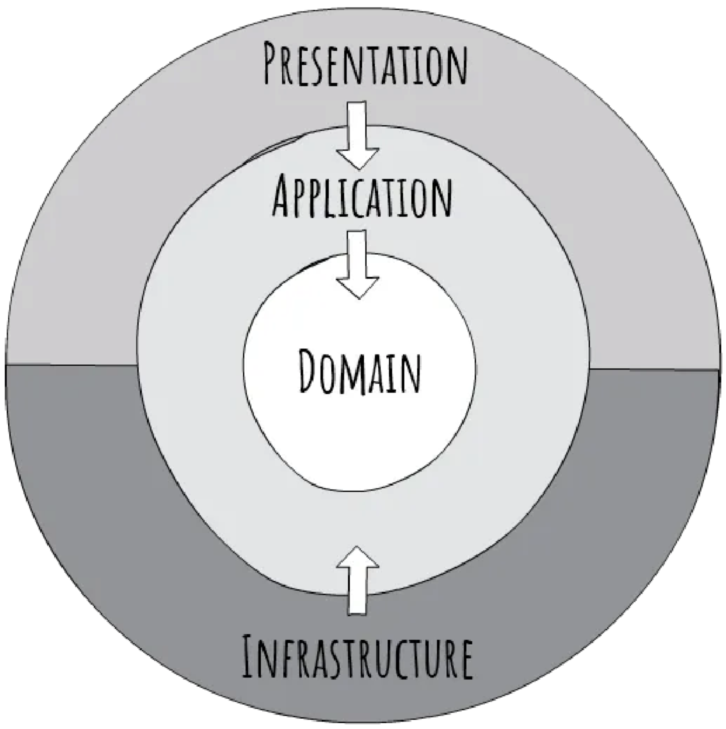


Figure 2.11: Another view at the Clean Architecture, taken from [1].

A more popular view at Clean Architecture, illustrated in figure 2.11, is as follows:

- Presentation Layer – is an application entry point, contains components necessary for presentation to the client, it can be API endpoints (REST, gRPC etc.), MVC architecture (Controllers and Views) or components of more advanced frameworks enabling both SSR (Server-side rendering) and CSR (Client-side rendering).
- Application Layer – includes the handling of application use-cases, that is, services or queries/commands and their handlers if used with CQRS pattern (see 2.4.2).
- Domain Layer – is the core of the system, holds all the business logic of the domain and emphasizes rich domain design (see 2.1.4).
- Infrastructure Layer – is often divided into infrastructure and persistence, consists of infrastructure-specific code such as integration with database, message broker, mail sender, 3rd party APIs and so on.

2.3.3 Vertical Slice Architecture

Layered architectures organize the system around technical layers and the cohesion between layers is low, leading to changes in many layers as features are added. The vertical slice architecture introduced by Jimmy Bogard [3] addresses this problem by organizing code around vertical slices, where each slice represent an application feature (use case).

The key for vertical slice approach is to minimize coupling between slices, and maximize coupling inside a slice. Therefore, new features should only add code without changing the shared code and worrying about side effects. The vertical slice architecture naturally comes with CQRS (see 2.4.2), as it already separates the code into features where the feature is a Command or a Query.

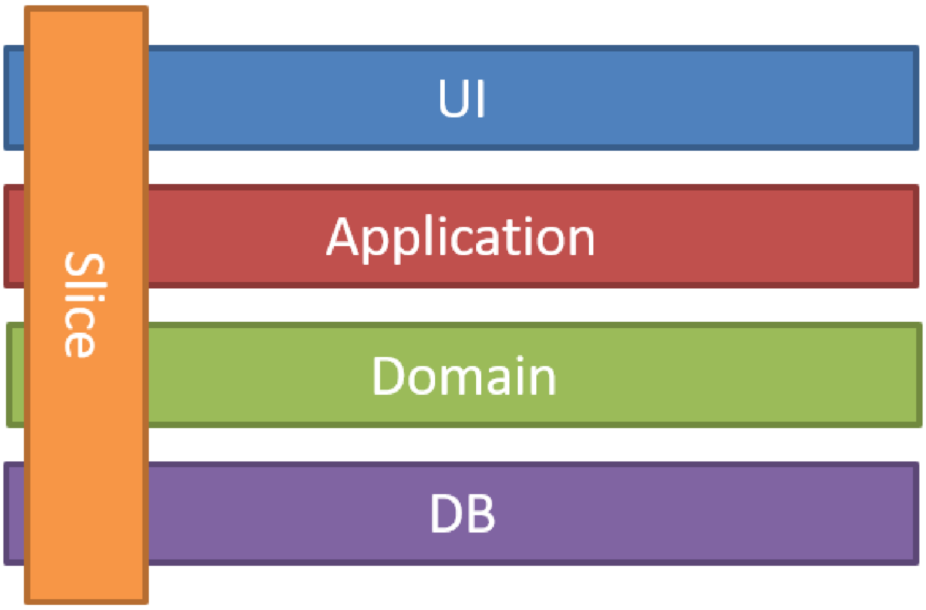


Figure 2.12: Architecture diagram of a vertical slice architecture, taken from [3].

An example of an application structure using the vertical slice architecture with CQRS:

```
ProjectManager/
├── Entities/
├── Features/ ....................................... directory with vertical slices
│   ├── Projects/
│   │   ├── GetProject/
│   │   │   ├── GetProjectEndpoint.cs
│   │   │   ├── GetProjectQuery.cs
│   │   │   ├── GetProjectQueryHandler.cs
│   │   │   └── ...
│   │   └── CreateProject/
│   │       ├── CreateProjectEndpoint.cs
│   │       ├── CreateProjectCommand.cs
│   │       └── ...
│   └── ...
├── Middlewares/
├── Program.cs
└── ...
```

Noteworthy from the example is the **vertical approach to naming** and structuring directories, which allows developers easier/faster understanding of the code as it outlines what the application does and what features it has.

The main downside mentioned by Jimmy Bogard [3] is that this approach does assume that the team understands code smells and refactoring.

2.3.4 Modular Monolith

The modular monolith (sometimes called majestic monolith) is a monolith designed with an emphasis on interchangeable, independent (and potentially reusable) modules, whereby the architecture seeks to gain benefits from both monolithic and microservices architectures.

By Kirsten Westeinde's definition [27], *"A modular monolith is a system where all of the code powers a single application and there are strictly enforced boundaries between different domains."* The **strict boundaries** between the modules are the essence of the modular monolith architecture.

The modular monolith is composed of **loosely coupled modules**, where modules represent cohesive sets of functionalities and are independent of each other (similar to microservices). Modules may mirror sub-domains/bounded contexts of the DDD. Example of module may be payments module, shipping module, booking module etc.

This is very similar to microservices and therefore it is easy to transition from a modular monolith to a microservice architecture. Main benefit is the ability to gradually transition and extract modules one by one, f.e. if a module faces high demand, it is easy to peel off this module into standalone microservices and scale it separately.

Separating a system into modules brings more microservice-like way of developing and maintaining the system while retaining the ease of testing and debugging.

It should be noted that when the system starts to be decomposed into microservice, the system looses the characteristics of easy debugging and integration testing.

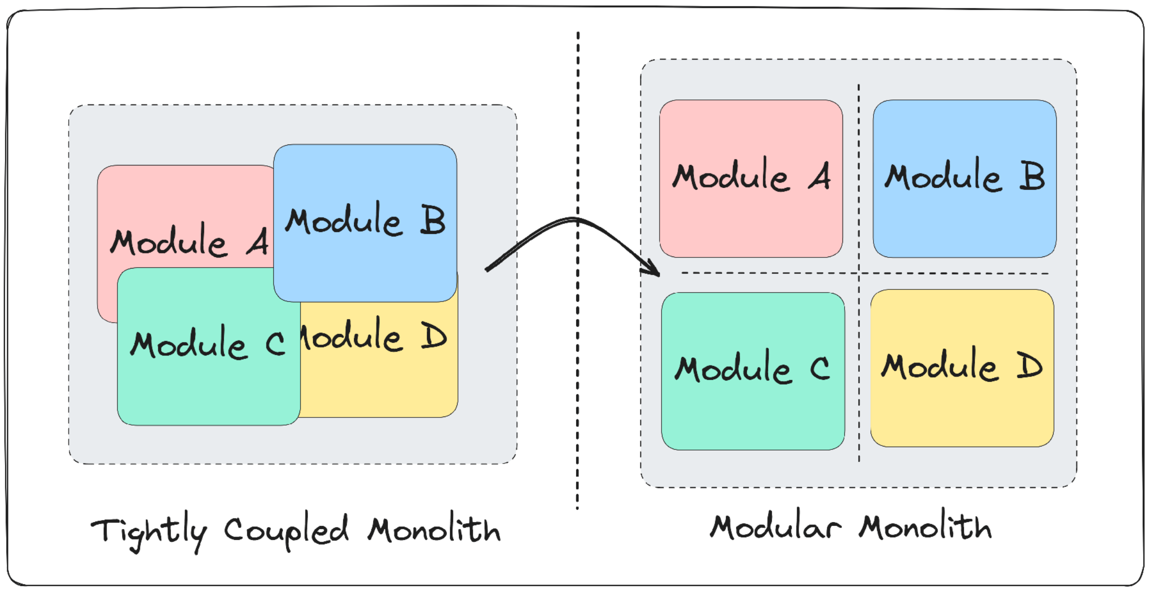


Figure 2.13: Diagram showing decoupling of a monolith into modules, taken from [12].

Communication between modules

When separating modules, it is necessary to define how the modules should communicate, Milan Jovanović in his blog post [12] specified 2 types of communication patterns between modules – **Method Calls** (synchronous) and **Messaging** (asynchronous).

The first and simpler approach is to communicate by calling methods of the public API (interfaces) exposed by each module. The modules are then dependent on other modules' APIs at compile-time and DI will provide their implementation at run-time. The advantages of of this approach are speed (fast in-memory calls) and easy implementation. The disadvantage is the **strong coupling** between modules.

The second approach is to introduce an indirection layer in the form of a **message broker**. The modules communicate with each other by sending messages in the fire-and-forget fashion and subscribing to the relevant messages in their context. Modules know nothing about other modules and depend only on **message contracts**. This solves the coupling problem, but has some drawbacks, such as increased complexity.

Data isolation

Milan Jovanović [13] mentioned that the modular monolith has strict rules for data integrity:

- Each module can only access its own tables.
- No sharing of tables or objects between modules.
- Joins are only allowed between tables of the same module.

The modules should be self-contained and each module should handle its own data. Other modules can access this data using the module's public API, this data isolation could be handled on 4 levels of separation:

- Separate Table – basically zero separation, tables for each module live inside one database. It may be hard to determine which tables belong to which module.
- Separate Schema – each module has its own database schema containing all the modules tables.
- Separate Database – each module has its own database in which resides all of the module's data. This level of separation forces developers to solve any coupling problems, resulting in a microservices-style of separation and providing an easy way to extract modules.
- Different Database Type – each module may have different database types (relational, document, graph etc.), this level of separation is the same as in a microservices architecture and it might be forced by business needs.

Real world example

Roberta Arcoverde, Director of Engineering at **Stack Overflow**, in the podcast interview with Scott Hanselman [11] reveled aspects of the stack overflow architecture.

One of 9 stack overflow servers handles 6000 request per second and whole system handles roughly 2 billion page requests per month. The system is designed for low latency and rendering a page takes about 12 milliseconds.

Surprisingly, despite this high demand and need to scale, stack overflow does not use microservices (and they neither run in the cloud), but uses a modular monolith architecture. Roberta Arcoverde mentioned in the podcast that the question of migrating to microservices has been brought up many times, but they have never come up with the pragmatic benefits because they do not straggle with the typical monolithic disadvantages:

- Roberta Arcoverde mentioned good **time to merge** metric in the context of maintainability and developer productivity.
- High demand is not a problem with the monolithic approaches to scaling and high degree of cashing on the database side.
- 9 servers have sufficient reliability for the system.
- Quality DevOps provides easy to deploy system several times per day.

2.4 Design patterns

Design patterns play a crucial role in architecting information systems as they provide proven solutions to common design problems increasing developer productivity, system maintainability, extensibility and eventually system performance.

2.4.1 Dependency Injection

Dependency Injection (DI) is a technique commonly used in object-oriented programming to achieve Inversion of Control (IoC). DI introduces a container with registered classes and dependencies are resolved from this container. There are 2 types of dependencies injection:

- Property Injection – the class specifies properties and the DI framework injects instances of these dependencies from DI container after instance creation.
- Constructor Injection – the class has all dependencies passed through the constructor and the DI framework resolves these dependencies when creating an instance.

The DI container creates dependencies respectfully to their registered lifetime:

- Singleton – the DI container returns the same instance each time a dependency is being resolved. The instance lives for the application's lifetime.
- Transient – the DI container returns a new instance each time a dependency is being resolved. The instance lives for the client instance's lifetime.
- Scoped – for each scope created in the application (i.e. session scope, http request scope, etc.), the DI container returns a new instance. The instance lives for the scope's lifetime.

2.4.2 CQRS

CQRS stands for Command Query Responsibility Segregation. The idea of CQRS is to separate the responsibility for handling command input (changing the system's state) from the responsibility for handling query input (reading the system's state).

Martin Fowler stated [8] that *"For many problems, particularly in more complicated domains, having the same model for commands and queries leads to a more complex model that does neither well."*

Separating the model introduces better code readability and the ability to use different technologies for each model:

The command model can take advantage of the ORM framework to ensure retrieval of all necessary data for correct validation of business rules and tracking changes.

On the other hand, since query model does not have to check for business rules except for authorization, it can make do with a **micro ORM**[4] framework or even sending raw SQL queries to improve performance.

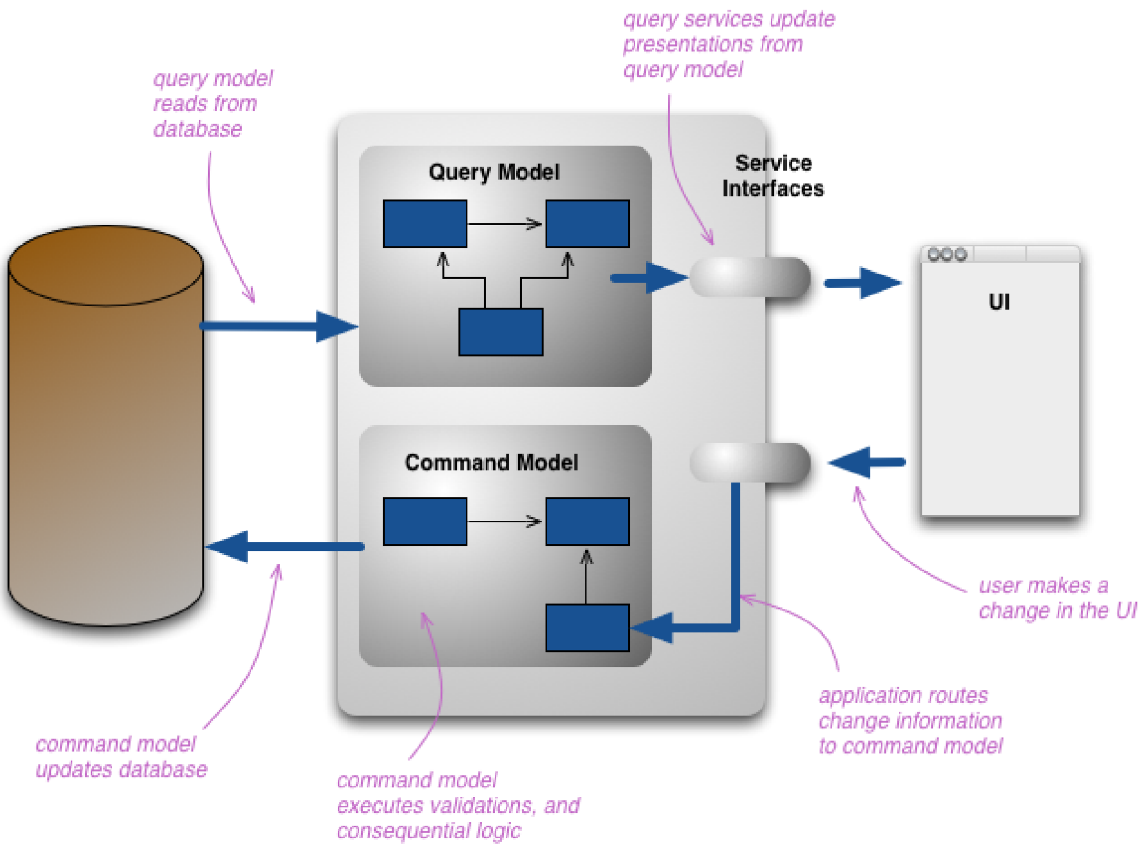

query services update presentations from query model
query model reads from database
Query Model
Service Interfaces
UI
Command Model
user makes a change in the UI
application routes change information to command model
command model updates database
command model executes validations, and consequential logic

Figure 2.14: System split into query and command model following CQRS, taken from [8].

To take this even further, CQRS can also separate the system at the database level (see figure 2.15), where the command model has its own database for writing data and the query model has its own database that is optimized for reading (NoSQL database). All data

[4]https://gunnarpeipman.com/micro-orm/

changes (commands) then need to be propagated from the write database to the read database (i.e. by integration events and message queue).

Typically, after an update command, the command model accesses its storage (also called the single source of truth) to ensure that the business rules are met, and then proceeds with the update. It is popular to use an **event sourcing** as the storage type, but it can also be used with a traditional relational database. The command model than publishes these changes to the message broker.

The query model is subscribed to data changing events from the message queue and updates its own database (i.e. MongoDB) as they arrive. To really improve performance and reduce read latency, it is necessary to make use of data redundancy in the NoSQL database to fit the query specifications, which likely implies complex updates to the NoSQL database.

This system leverages to maximum the ability of CQRS to use different technologies for each model, but comes with different sort of problems. The main downside of this approach is the more complex system with eventual consistency. Not every system is ideal for this approach and it may be beneficial to use simple CQRS and introduce **Cache-Aside pattern**[5] to reduce latency.

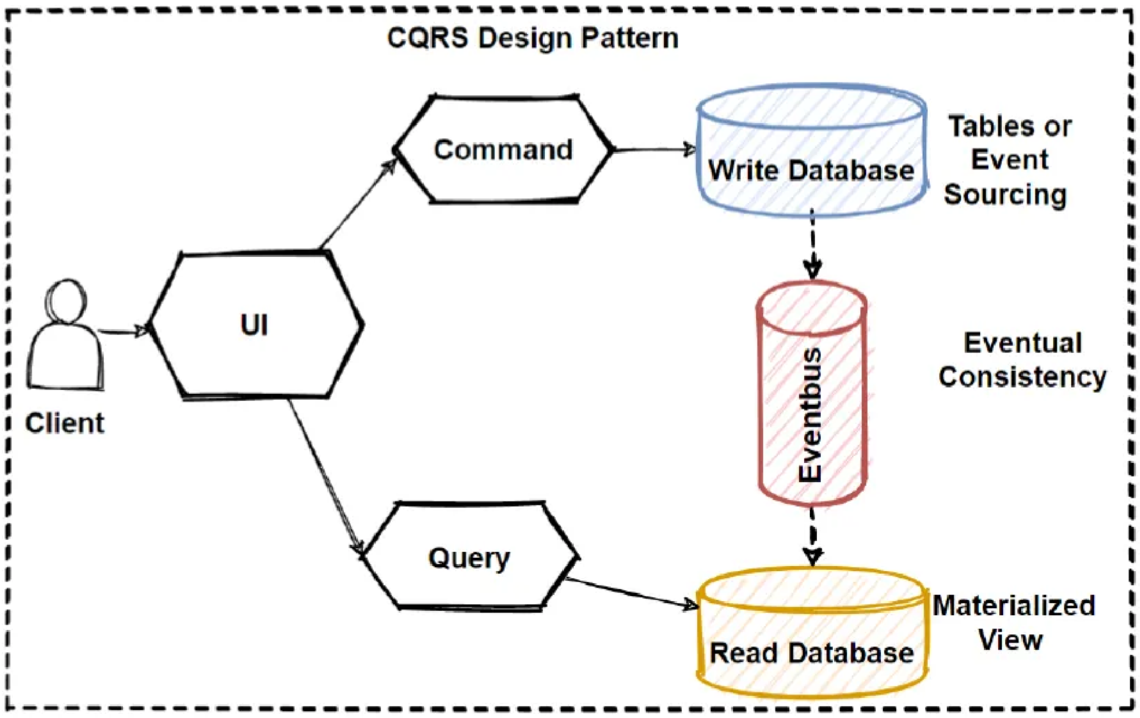


Figure 2.15: CQRS system modeled with split database tier and eventual consistency between databases, taken from [18].

[5]https://learn.microsoft.com/en-us/azure/architecture/patterns/cache-aside

2.4.3 Unit of Work

According to the book Patterns of Enterprise Application Architecture [6] the Unit of Work:

„Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.“

Essentially, instead of multiple SQL update requests, the unit of work merges all changes made to domain objects into a single transaction, thereby increasing performance, simplifying error handling and reducing the potential scope of unwanted side effects and bugs.

The unit of work is usually implemented into the ORM frameworks, but it might be beneficial to abstract the ORM away from the domain and application layers (Clean Architecture principles, see 2.3.2). The abstraction can be in the form of repositories (for registering domain changes) and a custom unit of work interface/façade around the ORM for committing transactions.

2.4.4 Transactional Outbox Pattern

The outbox pattern is a way to reliably publish asynchronous messages (e.g. send email), and is commonly used when publishing integration events or domain events in case of domain with eventual consistency (for example with CQRS, see 2.4.2).

Reliability problem

The reliability problem in the context of publishing asynchronous messages can be demonstrated by a simple example of the requirement to send an email after a bank balance changed with concrete amount.

Sending emails must be asynchronous as the systems ability to send emails might be limited and the server response cannot be stalled, furthermore:

- The email must be send only if the balance change is successful, that is, the transaction with saving the new balance completed successfully. Hence sending the email must be done outside of the application transaction.
- Email sending must be tracked so that if the server/application crashes right after a successful transaction and before the email is sent, after recovery the system can determine whether the email was sent or not and act accordingly.

Outbox

The solution to the reliability problem is to introduce an additional table into the database called outbox, to which the application stores records (outbox messages) indicating a desire to publish a message. The saving of outbox messages takes place within the application transaction, meaning that if other parts of the business logic (or event the outbox message storing itself) fail, and therefore transaction fails, the outbox message will not get persisted (and published).

The outbox message records contain information necessary for correct publishing, such as the message body (for example, a serialized integration event), timestamp, or whether the message has already been published.

The final part of the solution is a periodic job (e.g. every 10 seconds) responsible for retrieving unpublished messages from outbox and trying to publish them. If publishing was successful, the outbox message is marked as published (or deleted right away). Failing to publish results in a retry in the next job period.

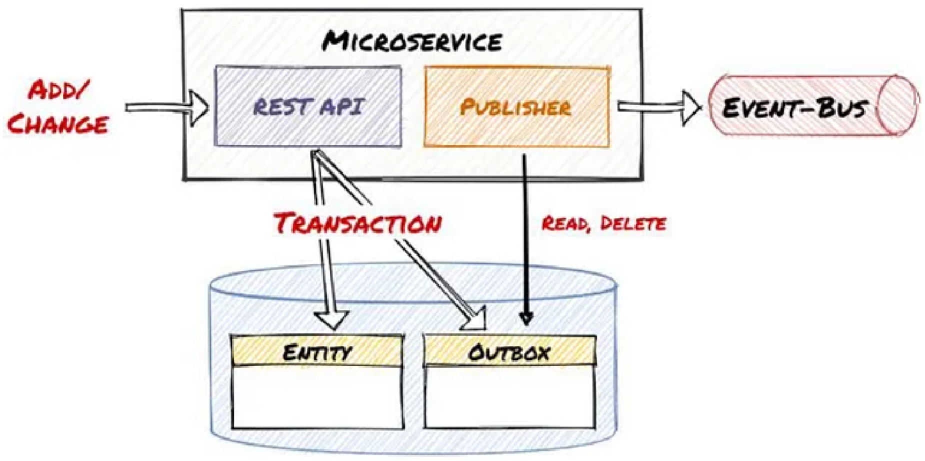


Figure 2.16: A diagram showcasing necessary parts and steps when utilizing the transactional outbox pattern, taken from [19].

Likewise, to reliably consume messages/events from message-broker/event bus, an inverse approach with **inbox** can be used.

The inbox and outbox patterns are commonly implemented in the communication libraries and frameworks (such as MassTransit[6], NServiceBus[7], etc.), which then only need to be properly configured to access the database.

[6]https://masstransit.io/
[7]https://particular.net/nservicebus

Chapter 3

Developing information system

This chapter delves into the developing process of a modular information system. It opens with a description of a demonstration application (section 3.1).

Next, in section 3.2 the chapter covers the basic application analysis, including its use cases and analysis for the usage of CQRS and selected technologies.

It then continues with section 3.3 about modeling the domain according to the domain-driven design, including the application of strategy design (subdomains and bounded contexts) and tactical design (modeling aggregates).

This is followed by sections 3.4 and 3.5 covering the development of the Clean Architecture and Modular Monolith solutions, respectively, and their implementation details.

Section 3.6 deals with implementing a frontend application that effectively uses the backend solution regardless of the backend architecture variant.

The chapter concludes with section 3.7, describing interesting parts of the development of modular information system solutions, such as enforcing architecture rules, and so on.

3.1 Demonstration application

Team and event management application, with working title **TeamUp**, primarily focused on managing small sports teams.[1]

In the application, the user can be part of a team as a regular team member, team coordinator or a team owner. In order to invite new users to the team or remove them, a member must be at least a team coordinator. There can be only one team owner, and a user can become one by creating a team or by being the target of ownership change. The team coordinator (as well as the team owner) creates team events, to which all team members respond whether they can attend or not. Only the team owner can assign roles.

Details

- Inviting is done via user's email regardless of the user's existence.

[1]Similar to application `https://tymuj.cz/`.

- Users can have a different nickname for each team they are member of.
- There must be business rules for maximum owned teams and maximum team capacity.
- Events can be distinguished by the event type.
- Members have several reply options to the team event – member will not attend, member might attend, member will attend with delay, member will attend on time.
- Members can add a message to the reply to specify the reason behind their response.

3.2 Application analysis

The application use case can be summarized using the use case diagram (see figure 3.1).

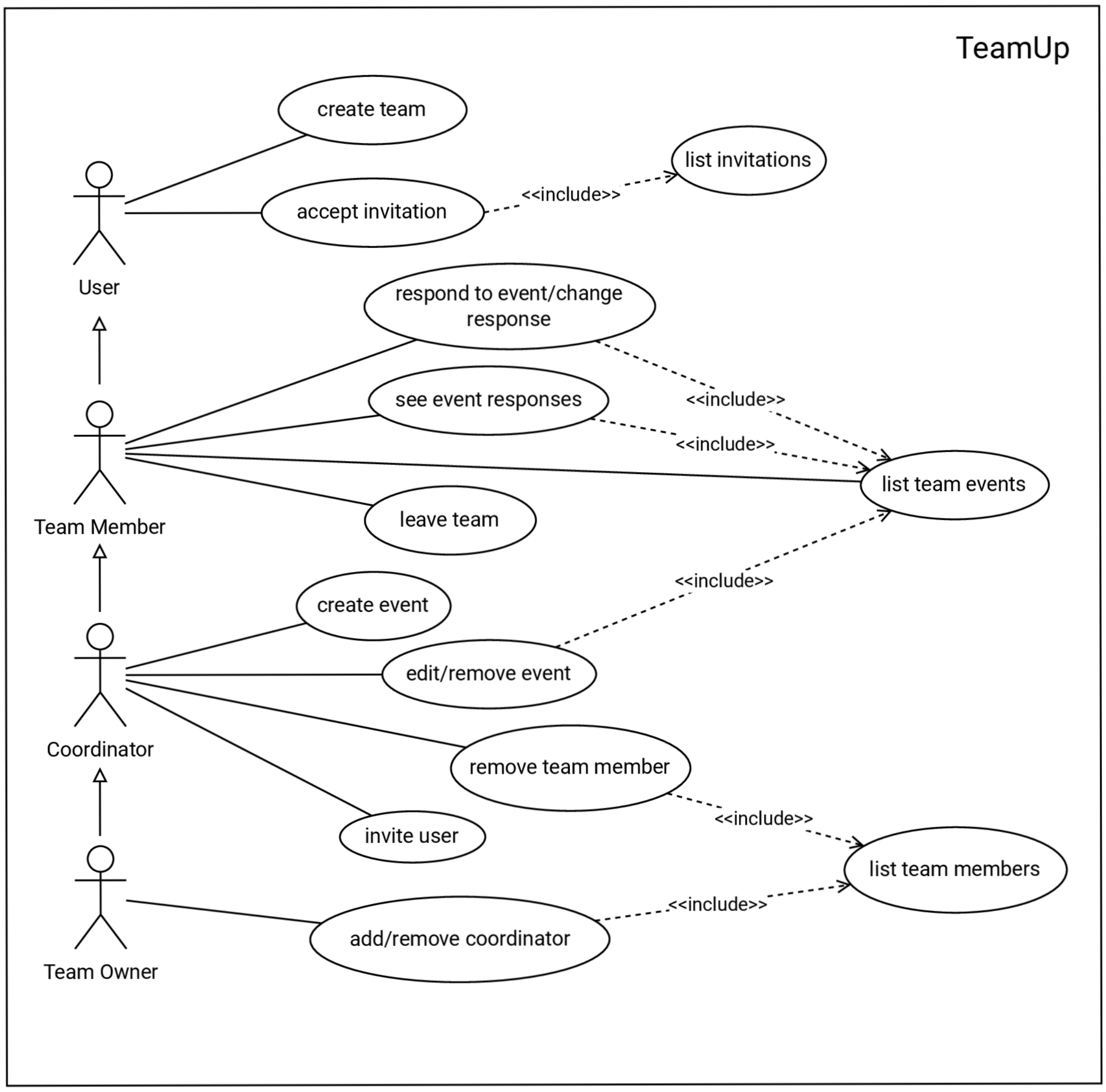


Figure 3.1: Use case diagram of the demonstration application.

The use case diagram can serve as a basis for identifying basic commands and queries in the context of CQRS:

Commands: create/update/remove team, accept invitation, create response, update response, create/update/remove event type, create/(update)/remove event, invite user/remove invitation, remove team member, assign role.

Updating an event may not be a desirable feature because when changing, for example, the start time of an event, members may need to change their responses, leading to a hard-to-solve business problem of how to react in such situation (clear all responses/leave all responses/sophisticated solution).

Queries: get team (including team members and event types), get events (including number of responses of each type), get event (including all responses), get user's invitations, get team invitations, get user's teams.

Based on the way the application works, it can be assumed that most of the demands will target the queries **get events** and **get event** and the **create response** command, as these are the application's daily use cases. There may be other frequently called queries, such as **get team**, but these change relatively infrequently and may be cached.

3.2.1 General technologies

- Platform: **dotnet** platform with the **C#** language is considered an elite platform for enterprise-wide information systems. In addition to quality documentation, Microsoft provides a variety of free eBooks[2] for building systems on the dotnet platform.
- Backend RESTful API: **ASP.NET Core** is an open source cross-platform framework maintained by Microsoft for building web applications and services.
- Database: **PostgreSQL**[3] (postgres) is a free and open-source relational database management system, and in my experience it spins up instance in docker much faster then the SQL Server (traditional choice for dotnet), resulting in faster integration and E2E testing, and thus making it my choice.
- Data access: **Entity Framework Core** (EF Core) is widely used ORM by Microsoft with support for code-first modeling workflow and database migrations.
- Background service scheduling: **Quartz.NET**[4] is a popular open-source job scheduling system targeting dotnet.

3.3 Modeling domain

When using DDD, the first step is to apply strategy design, which is mainly identifying subdomains and bounded contexts. Additionally, denoting shared kernels for multi-bounded context systems, and so on.

[2]https://dotnet.microsoft.com/en-us/learn/dotnet/architecture-guides
[3]https://www.postgresql.org/
[4]https://www.quartz-scheduler.net/

The next step is to apply tactical design, which consists of modeling the domain for a given bounded context and identifying aggregates and value objects.

Further modeling is done in the code using patterns such as domain events, domain services, etc. (described in section 2.2.2), in conjunction with continuous refactoring.

3.3.1 Identifying subdomains and bounded contexts

The core business logic of the system is team management, which involves managing team members and their roles + managing team events and responding to events. This could be viewed as one big team management subdomain, but I am leaning towards splitting it into 2 core subdomains – **Teams** subdomain and **Events** subdomain (see 2.2.1), which decouples parts of the system and better visualizes the integration relationships between individual subdomains.

Inviting users to the team is not the core of the application, as no user would start using the application because of this functionality, indicating it as a supporting subdomain.

The application has to provide basic user account functions (registration, login, etc.), this is a common problem and the system does not need some application specific features, based on that this area appears as a generic subdomain – in the diagram 3.2 called **User Access** subdomain.

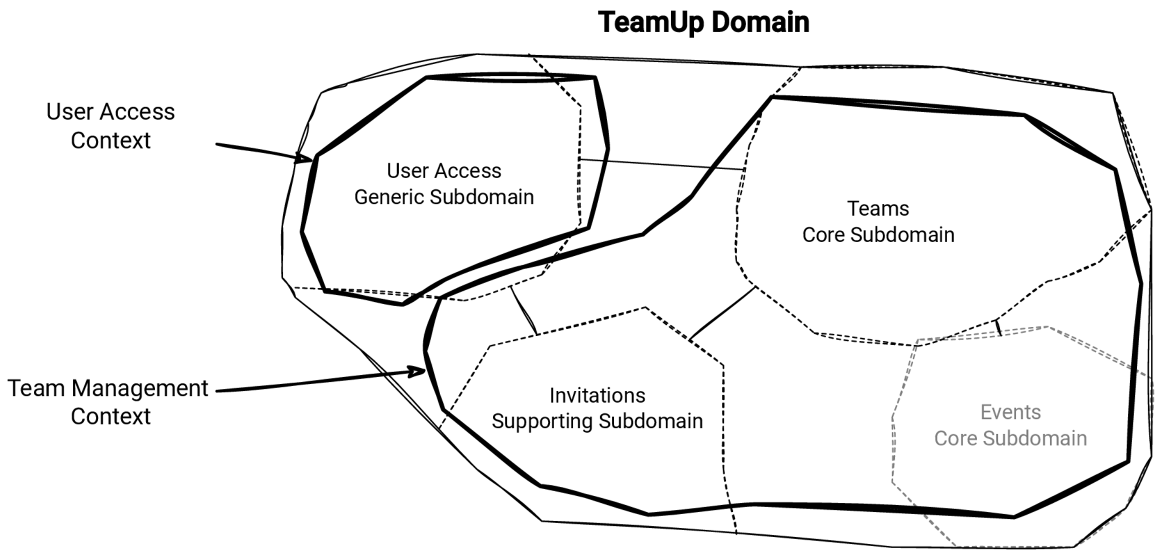


Figure 3.2: System domain decomposed to subdomains and bounded contexts.

User access has its own bounded context as it should be managed by an existing solution[5] and will not need any insight into the core/supporting domains of the application.

The rest of the subdomains can be modelled as a single bounded context as the domains have the same view at the system entities (the team is the same from both the invitation

[5]This would be true for a real-world application, however, the purpose of this thesis is not the integration of an identity server, so the solutions will implement a simple JWT authentication only with basic functions required by the business (register, login, delete and activate account).

and the event perspective). There is the overlap with user access subdomain as the team management context has to have some reference to users (i.e. in form of an id) and occasionally access users properties to implement application use cases (inviting via email and so on).

3.3.2 Domain model

Typically each bounded context would have its own domain model, however, since the user access context has only one entity from which most of the properties are used in the team management context, it would be counterproductive to treat them as separate contexts (may be different for modular monolith architecture as it separates system into modules).

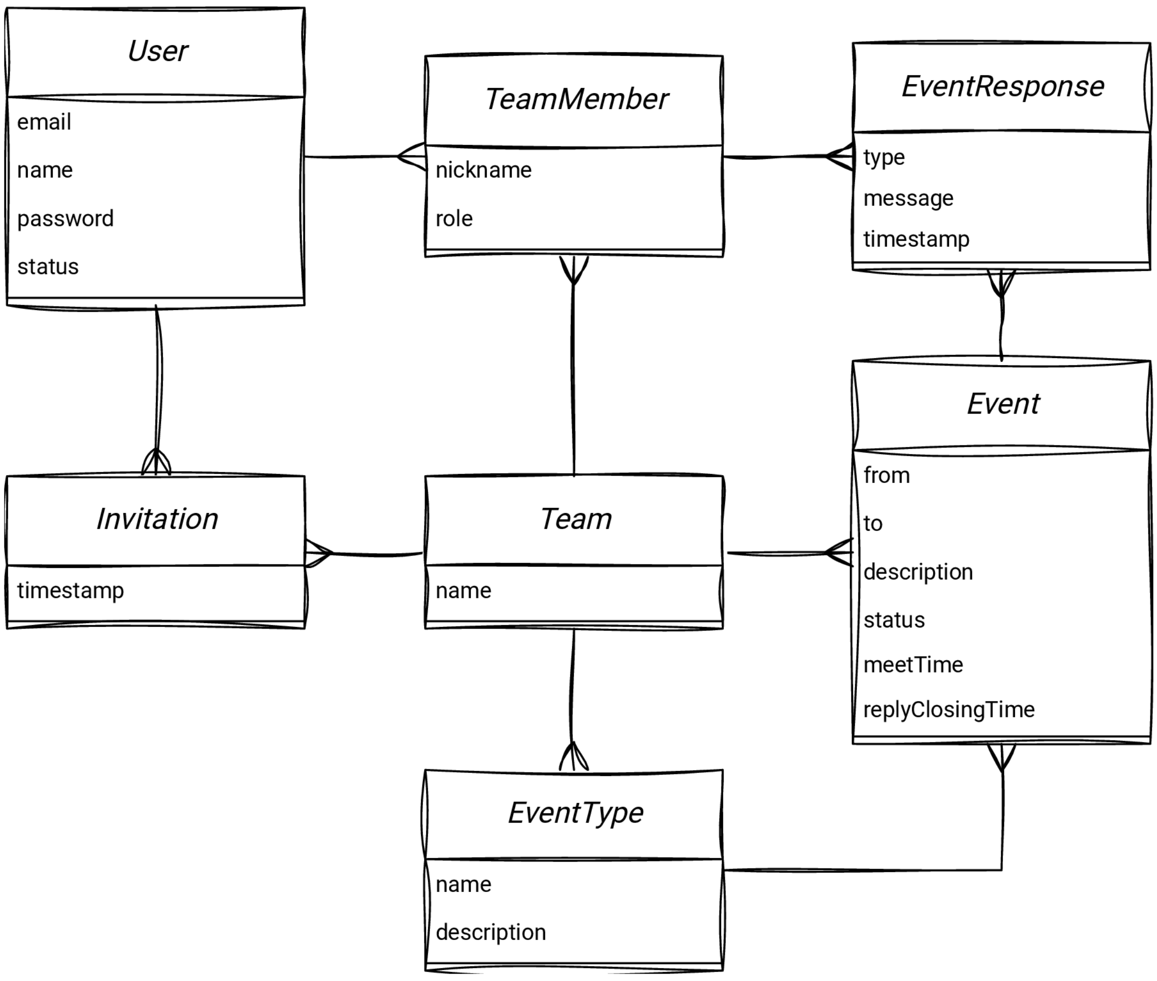


Figure 3.3: The initial domain model of the modeled system, where identifiers are PK for all entities and all relationships are 1:N, the model serves as the starting point and does not specify data types, visibility or methods implementing the business logic. It is mainly intended to showcase relationships and properties required by the business.

3.3.3 Modeling aggregates

A common approach to modeling aggregates is to start with an initial domain and mark all entities as the aggregate roots (see figure 3.4), then gradually try to merge the aggregates together or mark them as a final aggregate.

Before modeling aggregates, it is necessary to identify the value objects, for instance, the `EventResponse` containing the response type and message can be extracted into an `EventReply` value object to improve object manipulation and enforce business rules.

Theoretically, the `EventType` could be a value object as the team already contains event references and it is defined only by name and description, however, this would cause problems when updating the event type name since value objects are immutable (see 2.2.2).

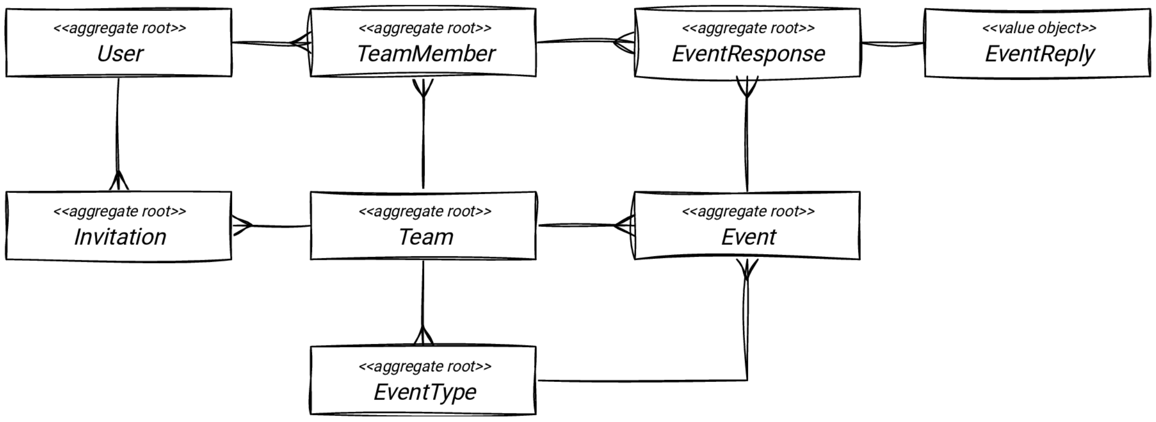


Figure 3.4: Diagram representing initial point for modeling aggregates with each entity serving as an aggregate root. Diagram contains unresolved relationships as DDD states that aggregates cannot reference each other directly.

Merging `EventResponse` into the **event aggregate** makes sense as the response is directly associated with the event and the user must interact with the event to create/update the response. The Event serves as an interface for event response manipulation and takes advantage of the `EventReply` value object to set the response directly from its interface.

Similarly, the `TeamMember` is an ideal candidate for merging into the **team aggregate** since the `Team` has all the necessary data for validating actions affecting team members (assigning roles, removing members, and so on).

The `User` has no interaction with other entities and thus is a good candidate for aggregate root. To some extent, it might make sense to manipulate invitations through the `User`, however, the same applies to the invitation-team relationship. Moreover, in the strategy design (figure 3.2), the invitations are modeled as a separate subdomain, therefore it is logical to leave `Invitation` as its own aggregate and handle invitations through a domain service.

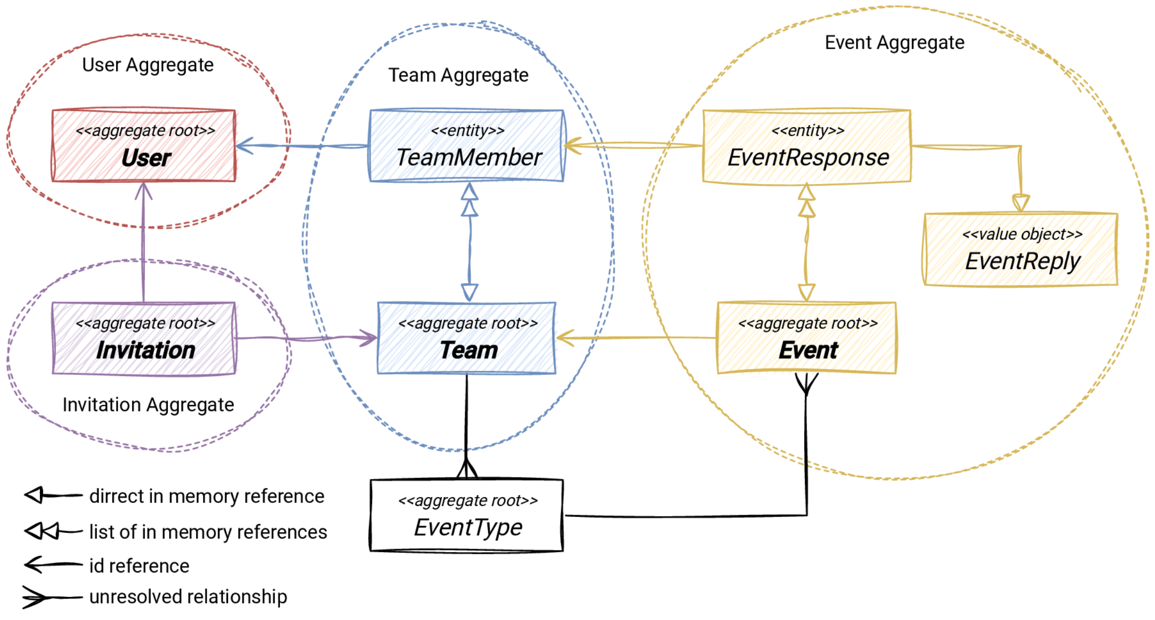


Figure 3.5: Domain model with partially modeled aggregates.

The `EventType` is tightly associated with the `Event`, however, all the manipulation needs to go trough the **team aggregate** so that the team aggregate root can authorize changes based on the initiator role. Furthermore, `EventType` references multiple events, so merging it into the event aggregate does not really make sense.

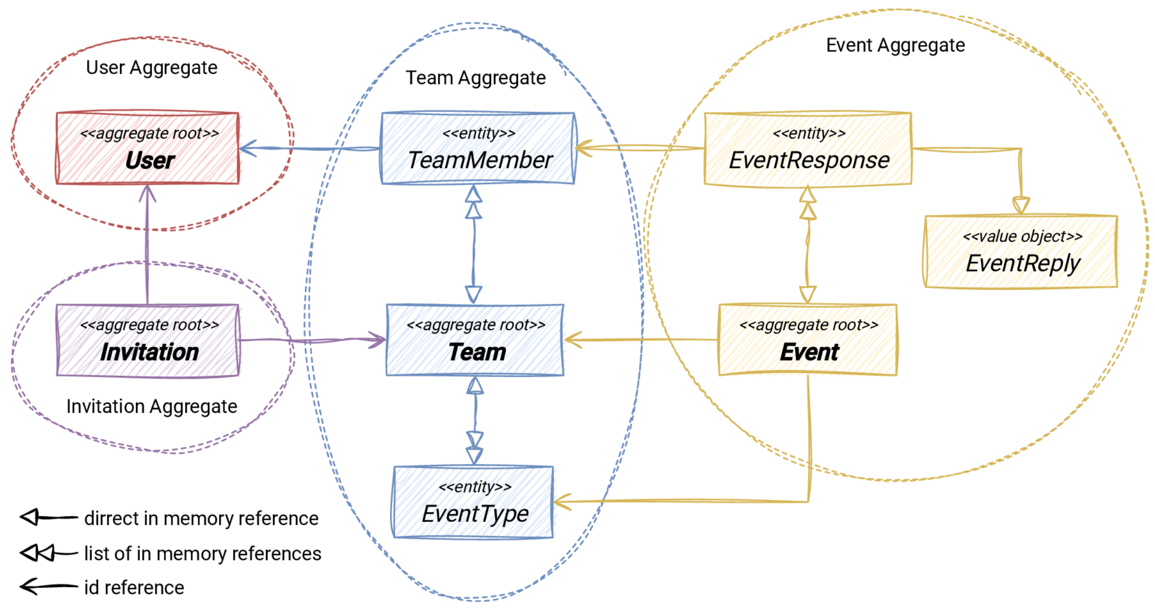


Figure 3.6: Diagram of final domain model designed with DDD. The diagram shows 3 apparent references from the event aggregate to the team aggregate. Although this might indicate candidates for merging, the resulting aggregate would be far too large. Furthermore, the domain services will resolve any logic requiring both aggregates.

3.4 Clean Architecture solution

The entire Clean Architecture solution is available online as a repository on GitHub:

https://github.com/skrasekmichael/CleanArchitecture

3.4.1 System Design

The system takes advantage of the CQRS with a single database:

- Query Part – applies simple authorization validations in handlers and communicates with the database through a query context to enable more performant querying.
- Command Part – utilizes Domain-Driven Design and its patterns to enforce complex business rules and validations (factories, aggregates and domain services) and ensures system consistency (domain events).

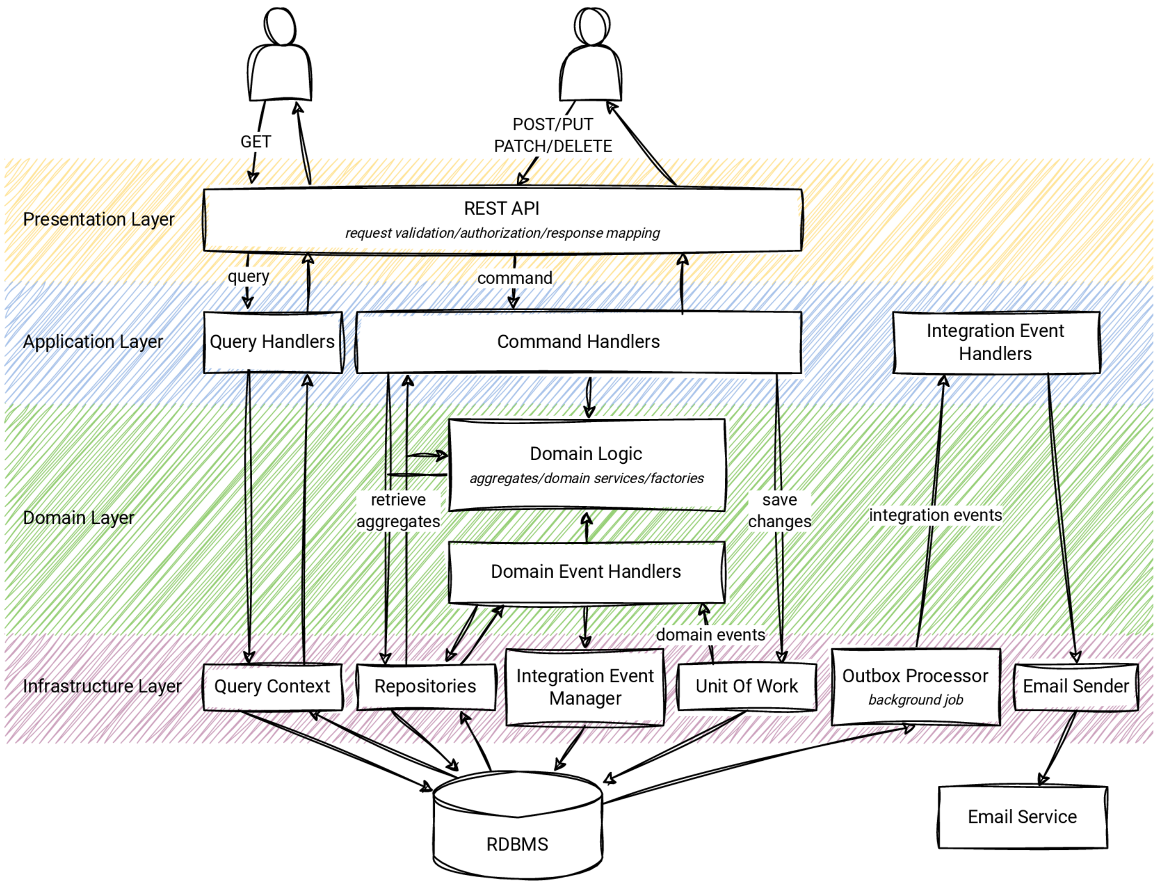


Figure 3.7: Diagram showcasing the flow through the Clean Architecture solution and some of its components. The infrastructure layer may look like innermost layer, however, the application and domain layers do not depend on the infrastructure layer at compile time, but rather use interfaces whose implementation is provided by the infrastructure layer via dependency injection at runtime. The diagram also illustrates the apparent complexity difference in the implementation of the query part (2 components) and the command part.

3.4.2 Implementing solution

Project structure:

```
TeamUp/ ........................................... Clean Architecture repository
├── src/
│   ├── TeamUp.Common/ ................... common types, abstractions and extensions
│   ├── TeamUp.Contracts/ ........................... API contracts and validators
│   ├── TeamUp.Domain/ ............................ domain model according to DDD
│   ├── TeamUp.Application/ .............. commands and queries and their handlers
│   ├── TeamUp.Infrastructure/ ............. EF Core configurations, migrations etc.
│   └── TeamUp.Api/ ................................... ASP.NET Core RESTful API
├── tests/
├── TeamUp.sln
└── ...
```

The `src` directory contains individual layers regarding the Clean Architecture with 2 additional layers (`Common` and `Contracts`).

The `tests` directory contains all aplication tests, including integration/E2E tests (see section 3.7.4), architecture tests (section 3.7.5), and so on.

3.4.3 Common Layer

The common layer contains common abstractions such as `IDateTimeProvider`, common types (e.g. exceptions/errors) and extension methods.

The purpose of this layer, similarly to so called *interface layer*, is to provide the necessary abstractions that can be used in other layers. Implementations of these abstractions reside in other layers (mainly in the infrastructure layer).

The common layer is not dependent on any other layer, and the only dependency is on the RailwayResult package (see section 3.7.2 about the result pattern), providing contracts and extension methods for the entire system.

3.4.4 Contract Layer

The contract layer contains all the contracts required for calling the API, including responses, enums, strongly typed ids, requests and their validators. The layer also contains constants used for validating and preparing API requests.

The contract layer could be reused by other presentation layers (MVC backend, SPA, mobile app, 3rd party services, etc.) as a shared dependency without the need to redefining types.

The **FluentValidation**[6] package is used to validate requests by creating a validation class for each request and creating rules for request properties using the fluent API. The main advantage of the FluentValidation library is the ease of validation for more complex rules as showcased in figure 3.9.

[6]https://github.com/FluentValidation/FluentValidation

```
public sealed record CreateEventRequest : IRequestBody
{
    public required EventTypeId EventTypeId { get; init; }

    [DataType(DataType.DateTime)]
    public required DateTime FromUtc { get; init; }

    [DataType(DataType.DateTime)]
    public required DateTime ToUtc { get; init; }

    [DataType(DataType.Text)]
    public required string Description { get; init; }

    [DataType(DataType.Time)]
    public required TimeSpan MeetTime { get; init; }

    [DataType(DataType.Time)]
    public required TimeSpan ReplyClosingTimeBeforeMeetTime { get; init; }
}
```

Figure 3.8: Code sample of request (body) for creating an event. Properties are annotated with meta informations that could be used by templating engines from other presentation layers. Note that the `EventTypeId` property is defined as a strongly typed id, same as all other idefntifiers in the solution.

```
public sealed class Validator : AbstractValidator<CreateEventRequest>
{
    public Validator(IDateTimeProvider dateTimeProvider)
    {
        RuleFor(x => x.EventTypeId).NotEmpty();

        RuleFor(x => x.FromUtc)
            .NotEmpty()
            .GreaterThan(dateTimeProvider.UtcNow)
            .WithMessage("Cannot create event in past.");

        RuleFor(x => x.ToUtc)
            .NotEmpty()
            .Must((model, to) => model.FromUtc < to)
            .WithMessage("Event cannot end before it starts.");

        RuleFor(x => x.Description)
            .NotEmpty()
            .MaximumLength(EventConstants.EVENT_DESCRIPTION_MAX_SIZE);

        RuleFor(x => x.MeetTime).GreaterThan(TimeSpan.Zero);

        RuleFor(x => x.ReplyClosingTimeBeforeMeetTime).GreaterThan(TimeSpan.Zero);
    }
}
```

Figure 3.9: Code sample of a validator for `CreateEventRequest` (figure 3.8) with more complex rules, working with `DateTimeProvider` or the rule of the `ToUtc` property dependent on the `FromUtc` property, which would be complicated to implement using the default annotation-based validation build in ASP.NET Core.

3.4.5 Domain Layer

The domain layer contains domain-specific logic, that is:

- Aggregates – the **aggregate root entity** and additional nested entities and value objects for each aggregate.
- Aggregate specifics (grouped together with aggregates) – factories, **domain events**, integration events and a **repository** interface that is used in other services (factories, event handlers, domain services, and command handlers). Additional possible aggregate specifics may include logic (extension methods) over enum types and static classes with defined business rules and errors.
- **Event Handlers** to synchronize and enforce a consistent state across aggregates (preferred method) and potentially to publish integration events.
- **Domain Services** with logic involving multiple aggregates and the necessity of data from other aggregates for immediate business rules validation.
- Abstractions for domain objects, i.e. for entities, domain events, event handlers, ...

Detailed structure:

```
TeamUp.Domain/ ................................................... domain layer
├── Abstractions/ ........... domain objects abstractions (aggregate root, entity ...)
├── Aggregates/
│   ├── Invitations/ ...................... domain objects of the invitation aggregate
│   ├── Users/ ................................ domain objects of the user aggregate
│   ├── Events/
│   │   ├── DomainEvents/ .................... events published by the event aggregate
│   │   ├── Event.cs ............................................... aggregate root
│   │   ├── EventResponse.cs ....................................... nested entity
│   │   ├── EventReply.cs .......................................... value object
│   │   ├── EventStatusExtensions.cs ............ added logic for EventStatus enum
│   │   ├── EventErrors.cs ..................... errors returned by the event aggregate
│   │   └── IEventRepository.cs ......... repository definition for working with events
│   └── Teams/
│       ├── DomainEvents/ ..................... events published by the team aggregate
│       ├── Team.cs ................................................ aggregate root
│       ├── TeamMember.cs .......................................... nested entity
│       ├── EventType.cs ........................................... nested entity
│       ├── TeamRoleExtensions.cs .................. added logic for TeamRole enum
│       ├── TeamErrors.cs ...................... errors returned by the tean aggregate
│       ├── TeamRules.cs ........................ business rules of the team aggregate
│       └── ITeamRepository.cs .......... repository definition for working with teams
├── DomainServices/ .............................................. domain services
├── EventHandlers/ ........................................ domain event handlers
├── ServiceCollectionExtensions.cs ............ methods for registering layer to DI
└── TeamUp.Domain.csproj ........................................... assembly file
```

Domain Events

Domain events are meant to propagate changes to other parts of the domain (mainly aggregates) and therefore enforce necessary actions to bring the domain into consistent state.

Some of the logic performed by the domain event handlers could be solved by database triggers, however, that would split up the business logic and pile up additional complexity.

Traditionally, domain events are not published/handled immediately, but rather gathered and handled at the end of a transaction (see more in section 3.4.7).

```
public async Task Handle(UserDeletedDomainEvent domainEvent, CancellationToken ct)
{
    var teams = await _teamRepository.GetTeamsByUserIdAsync(domainEvent.User.Id, ct);

    foreach (var team in teams)
    {
        team.GetTeamMemberByUserId(domainEvent.User.Id)
            .Ensure(TeamRules.MemberCanChangeOwnership)
            .Tap(initiator =>
            {
                if (team.Members.Count == 1)
                {
                    //remove team if user that is being removed is the only member
                    _teamRepository.RemoveTeam(team);
                }
                else
                {
                    //change ownership when removing user that is owner of the team
                    var newOwner = team.GetHighestNonOwnerTeamMember()!;
                    initiator.UpdateRole(TeamRole.Admin);
                    newOwner.UpdateRole(TeamRole.Owner);
                    team.AddDomainEvent(new OwnerChangedEvent(initiator, newOwner));
                }
            });

        //db will cascade delete member, number of members has to be updated manually
        team.DecreaseNumberOfMembers();
    }

    _userRepository.RemoveUser(domainEvent.User);
}
```

Figure 3.10: Code sample of an event handler handling the *user deleted domain event*.

The event handler in figure 3.10 solves consistency problems resulting from the user deletion:

- Deleting a team if the user is the only member (implicitly owner).
- Change ownership to another member if the user is the team owner (and raise an *ownership changed domain event*). This consistency problem could also be solved by failing to delete the user if the user is an owner of a team, and the user would have to manually change ownerships before deletion.
- Decreasing number of members in a team (required for race conditions, see 3.7.3).

Deletion of team member records linked to the user is handled automatically by the RDBMS and db model with the `ON DELETE CASCADE` configuration.

3.4.6 Application Layer

The application layer contains implementation of application use cases, that is:

- Implementation of commands and command handlers, which consists primarily of calling the domain logic of aggregates, factories and domain services, and subsequently persisting the changes via the unit of work.
- Implementation of queries and query handlers, which consists of calling the query context façade around the EF Core context that builds and sends SQL queries, and subsequent validating of authorization rules (restricting what can be read by whom).

The sending requests (queries/commands) and processing them (handlers) is implemented using the **MediatR**[7] package, an in-process messaging mediator by Jimmy Bogard.

Additionally, the application layer contains helper services/interfaces and integration event handlers for performing asynchronous business actions (sending emails).

```
public async Task<Result<TeamResponse>> Handle(GetTeamQuery query, CancellationToken ct)
{
    var team = await _appQueryContext.Teams
        .Where(team => team.Id == query.TeamId)
        .Select(team => new TeamResponse
        {
            Name = team.Name,
            Members = team.Members
                .Select(member => new TeamMemberResponse
                {
                    Id = member.Id,
                    UserId = member.UserId,
                    Nickname = member.Nickname,
                    Role = member.Role
                })
                .ToList()
                .AsReadOnly()
        })
        .FirstOrDefaultAsync(ct);

    return team
        .EnsureNotNull(TeamErrors.TeamNotFound)
        .Ensure(team =>
            team.Members.Any(member => member.UserId == query.InitiatorId),
            TeamErrors.NotMemberOfTeam);
}
```

Figure 3.11: Code sample of a query handler of `GetTeamQuery` utilizing EF Core, where the EF Core builds SQL queries from given LINQ statements (and caches them for later reuse). After retrieving data from the database, the handler checks whether the team exists and whether the user is a member of the team. Note that the query statement uses `Select` to map database result directly to desired response type (DTO).

[7]https://github.com/jbogard/MediatR

3.4.7 Infrastructure Layer

The purpose of the infrastructure layer is to shield the application core from dependence on third-party libraries and external services, specifically by:

- implementing core services (service abstractions from inner layers) including:
  - `DateTimeProvider` – it is considered a best practice to use a service to access date time as it provides an easy way to test logic involving date times.
  - `UnitOfWork` – since repositories and EF Core keep track of all changes to domain data, the unit of work exposes only the *save changes* method to persist changes in a single transaction. Note that the method also ensures correct dispatching of all domain events raised within the transaction (see below).
  - `QueryContext` – essential for *query handlers* because it enables to access database (no modification) by exposing `IQueryable` collections of the requested data (used to build LINQ queries that EF Core translates into SQL) with a *no tracking* configuration, meaning EF Core will not track any changes to these objects.
    ...
- ensuring data persistence, that is, integrating database:
  - Configuration of the domain objects (mapping the domain object definitions to database tables) using the EF Core.
  - Implementation of repositories – integration to concrete database (postgres) using a specific technology (EF Core).
    ...
- integrating additional services, i.e. emailing service.
- implementing and configuring cross-cutting concerns, such as API security, dispatching integration events, and so on.

Structure:

```
TeamUp.Infrastructure/
├── Core/ .......................................... implementation of core services
├── Extensions/ .................... extension classes used in the infrastructure layer
├── Options/ ......................... option classes loaded from appsettings.json
├── Persistence/
│   ├── Domain/ ................. entity configurations and repository implementations
│   ├── Migrations/ ................................ EF Core db schema migrations
│   ├── ApplicationDbContext.cs ........................ EF Core database context
│   └── DesignTimeDbContextFactory.cs ............. factory for creating migrations
├── Processing/ ........................ processing services – events, messages, etc.
├── Security/ ......... security concerns – password service, JWT configuration, etc.
├── AssemblyInfo.cs
├── ServiceCollectionExtensions.cs ...... methods for registering the layer into DI
└── TeamUp.Infrastructure.csproj ................................. assembly file
```

Publishing Domain Events

As previously mentioned, when a domain event is raised, it is collected from publishing at the end of the transaction. The collection is done by adding the event to the event list, and each entity has its own event list, hence, the domain events are bound to the entities that raised them, and the domain is not polluted with injection of service for raising/gathering domain events.

The *unit of work* executes the actual publishing of the domain events as calling the *save changes* method marks the end of the application transaction, another way may be by using EF Core interceptors to intercept *save changes* and then proceed to publish events.

As mentioned in the segment about domain events (see section 2.2.2 about DDD tactical design), they can be published within a transaction (single transaction for multiple aggregates) or asynchronously after committing one, making the system eventual consistent with multiple transactions when handling domain events.

I opted for **strong consistency** within a **single transaction**. In this application, eventual consistency would introduce unnecessary complexity and a single transaction involving multiple aggregates does not seem to cause any problems for this low scale application with few rather limited number of aggregates.

Problems with intensive locking when committing large transaction spanning multiple aggregates may be addressed by better domain modelling or by breaking the bounded context into modules and introducing eventual consistency between modules as in the Modular Monolith architecture, thereby reducing the transaction size.

```
public async Task DispatchDomainEventsAsync(CancellationToken ct = default)
{
    List<IHasDomainEvent> entities;

    while ((entities = GetEntitiesWithUnpublishedDomainEvents()).Count != 0)
    {
        //get all unpublished domain events
        var domainEvents = entities.SelectMany(entity => entity.DomainEvents).ToList();

        //clear all domain events
        entities.ForEach(entity => entity.ClearDomainEvents());

        //publish all domain events
        foreach (var domainEvent in domainEvents)
        {
            await _publisher.Publish(domainEvent, ct);
        }
    }
}
```

Figure 3.12: Code sample for publishing domain events. The method for gathering entities with unpublished domain events uses the EF Core database context and `ChangeTracker` to gather all changed objects that implement `IHasDomainEvent` interface and whose event list is not empty. Note that the loop repeats until all domain events are published as event handling may produce additional events that were not initially gathered. This could lead to a potential infinite loop, but hopefully it would be detected by the integration tests.

Outbox & Publishing Integration Events

To reliably handle asynchronous side effects – integration events – a transactional outbox pattern has to be used (see section 2.4.4 about outbox pattern).

This means that the side effect producer has to raise an integration event from a domain event handler (or possibly from a domain service) using the `IntegrationEventManager`, the idea being that since side effects affecting the database are processed inside domain event handlers, asynchronous side effects should be raised from them as well.

Since integration events are serialized into outbox messages, they do not contain object references, but only direct values or references via identifiers (uuid/guid).

```
private async Task DispatchEventAsync(OutboxMessage msg, CancellationToken ct = default)
{
    var integrationEventType = msg.Type.ResolveType();
    if (integrationEventType is null)
    {
        _logger.LogCritical("Failed to identify outbox message type {msg}.", msg);
        msg.Error = "Type not found.";
        return;
    }

    var integrationEvent = JsonSerializer.Deserialize(msg.Data, integrationEventType);
    if (integrationEvent is null)
    {
        _logger.LogCritical("Failed to deserialize outbox message {msg}.", msg);
        msg.Error = "Failed to deserialize.";
        return;
    }

    try
    {
        await _publisher.Publish(integrationEvent, ct);
        msg.ProcessedUtc = _dateTimeProvider.UtcNow;
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Failed to publish from outbox {msg}.", msg);
        msg.Error = "Failed to publish event.";
    }
}
```

Figure 3.13: Code sample for dispatching a integration event from an outbox message. By updating the `ProcessedUtc` property, the outbox message is marked as published. If an error occurs during the consumption of outbox message or while publishing an event, the step in which the error occurred is logged and stored in the db.

Publishing integration event workflow goes as follows

1. Integration event producer serializes event and persists it into the outbox within transaction caused by a command.
2. Every M seconds, the outbox processor takes N unconsumed messages, deserializes them and publishes them; successfully consumed messages are marked as processed in the outbox, failed ones will be re-consumed in the next iteration.

3.4.8 Presentation Layer

The presentation layer contains application entry point with building and configuring the server using the ASP.NET Core framework:

- Configuration of server components (authentication, API versioning, etc.) and registration of services into DI container via extension methods from individual layers.
- Application of required middlewares, for example for routing, authentication, authorization, CORS, and so forth.
- Mappings of RESTful endpoints that call application logic via commands and queries. Endpoints are decorated for automatic generation of OpenAPI[8] specification.
- Addition of development-specific parts (development-only) such as swagger (for demo testing API), logging middlewares, automatic database migrating, etc.

```
public void MapEndpoints(RouteGroupBuilder group)
{
    group.MapDelete("/{teamId:guid}/members/{teamMemberId:guid}", RemoveTeamMemberAsync)
        .Produces(StatusCodes.Status200OK)
        .ProducesProblem(StatusCodes.Status400BadRequest)
        .ProducesProblem(StatusCodes.Status401Unauthorized)
        .ProducesProblem(StatusCodes.Status403Forbidden)
        .ProducesProblem(StatusCodes.Status404NotFound)
        .WithName(nameof(RemoveTeamMemberEndpoint))
        .MapToApiVersion(1);
}

private async Task<IResult> RemoveTeamMemberAsync(
    [FromRoute] Guid teamId,
    [FromRoute] Guid teamMemberId,
    [FromServices] ISender sender,
    HttpContext httpContext,
    CancellationToken ct)
{
    var command = new RemoveTeamMemberCommand(
        httpContext.GetCurrentUserId(),
        TeamId.FromGuid(teamId),
        TeamMemberId.FromGuid(teamMemberId)
    );
    var result = await sender.Send(command, ct);
    return result.ToResponse(TypedResults.Ok);
}
```

Figure 3.14: Code example of an endpoint for removing a team member from the team. The `MapEndpoints` method utilizes the dotnet functionality (**Minimal API**[9]) to map endpoints, and additional methods for automatic generating of the OpenAPI specification. The `RemoveTeamMemberAsync` method is an actual endpoint handler that creates a command/query from the http context (request parameters, headers, etc.), calls the application logic via the command/query, and maps the result to the correct http response.

[8] https://swagger.io/resources/open-api/
[9] https://learn.microsoft.com/en-us/aspnet/core/fundamentals/minimal-apis

3.4.9 Invite unregistered user workflow example

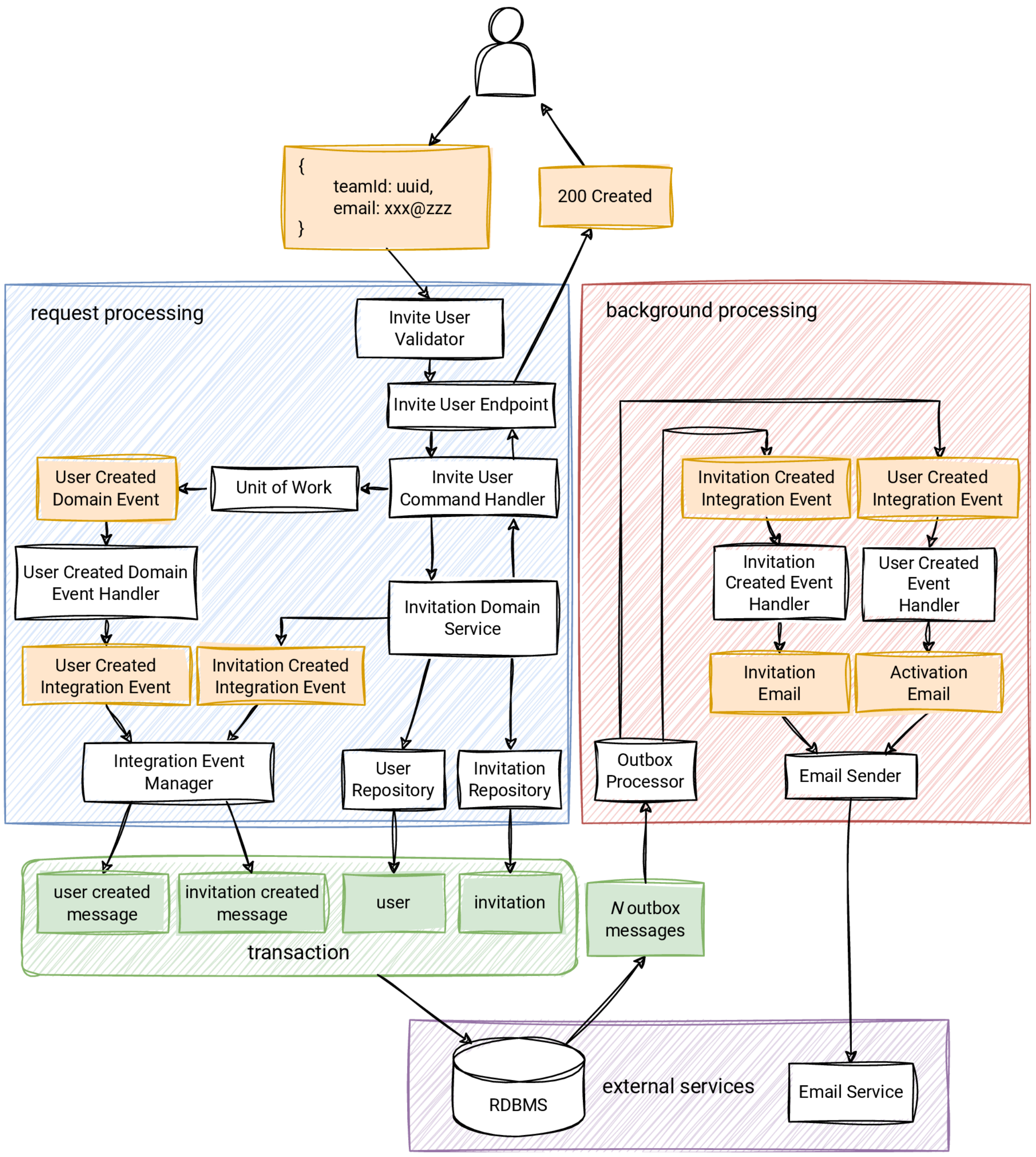

{
teamId: uuid,
email: xxx@zzz
}
200 Created
request processing
Invite User Validator
Invite User Endpoint
Invite User Command Handler
Unit of Work
User Created Domain Event
User Created Domain Event Handler
Invitation Domain Service
User Created Integration Event
Invitation Created Integration Event
Integration Event Manager
User Repository
Invitation Repository
user created message
invitation created message
user
invitation
transaction
background processing
Invitation Created Integration Event
User Created Integration Event
Invitation Created Event Handler
User Created Event Handler
Invitation Email
Activation Email
Outbox Processor
Email Sender
N outbox messages
RDBMS
external services
Email Service

Figure 3.15: Diagram showing the workflow of inviting a user that is not registered to a team, assuming all data is valid. Diagram outlines key components that the workflow goes through and the events that are published along the way. Note that the transaction is committed by the *unit of work* and before committing, the *user created domain event* is collected and published. This diagram skips the API authorization process.

3.5 Modular Monolith solution

The entire Modular Monolith solution is available online as a repository on GitHub:

https://github.com/skrasekmichael/ModularMonolith

3.5.1 Identifying modules

Appropriate decomposition of the system into modules is crucial for system maintainability and scalability, each module should be independent of others and focus on its own functionality.

To some extent, module decomposition mirrors the DDD strategy design (bounded context = module), although even a bounded context can be divided into multiple modules if it is advantageous (smaller modules = smaller transactions).

The core modules can be supplemented by supporting modules that cover highly distinct or specific functionality from other modules, such as infrastructure integration (SMS gateway, etc.), data processing (pdf, image, video, etc.), and so on.

Modules

- **User Access Module** – mirrors the functionality for the business needs of the User Access bounded context.
- **Team Management Module** – mirrors functionality and business needs of the Team Management bounded context. It might be beneficial to split this module into more sub-modules (for instance, a separate module for invitations), however, it should be easy to decompose it in later development iterations.
- **Notifications Module** – encapsulates the functionality for notifying users (currently only by email) via special services/technology.

  The existence of a separate module allows for easy addition of notification methods (e.g. sms) and in case of transition to microservices, each microservice does not need to have its own notification infrastructure, but can use this internal service for pushing notifications instead.
- Bootstrapper – entry point to the application bootstrapping all modules, necessary services and configurations to a single executable (ASP.NET Core app). When migrating to microservices, this module would be turned into an API gateway/reverse proxy.

3.5.2 System Design

The system is designed for easy transition to microservices in the context of data modeling, isolation and communication patterns incorporating eventual consistency.

Each module that provides data manipulation functionality integrates the CQRS pattern, with the queries being lightweight database calls and commands leveraging DDD for complex business logic (similarly to the Clean Architecture solution design in section 3.4.1).

Communication

Communication between modules is primarily **asynchronous**, facilitated over a **message broker** (see figure 3.16). This approach ensures loose coupling between modules. Each module exposes contracts, such as integration events, for external communication. Internally, modules communicate synchronously using in-process calls.

To support more complex interactions, the system also registers request-response paths over the message-broker, enabling synchronous (blocking) command and query invocations between modules. This allows to call commands or queries of other modules when necessary. Furthermore, the input validation is moved to the command/query handlers to validate incoming communications not only from external sources but also from other modules, ensuring robustness and consistency across the system.

To ensure reliable communication and consistency, each module has its own storage for outbox and inbox (see section 2.4.4 describing transactional outbox pattern), even if a module does not have to persist state to execute its logic. Implementing this demand resulted in some problems (see more in segment about multiple outboxes in section 3.7.1).

Data Isolation

Data isolation is achieved via a **separate schema** per module in a **single database** (see figure 3.16). Since the system does not contain cross-schema transactions, there is no data coupling between modules, providing sufficient isolation akin to microservices.

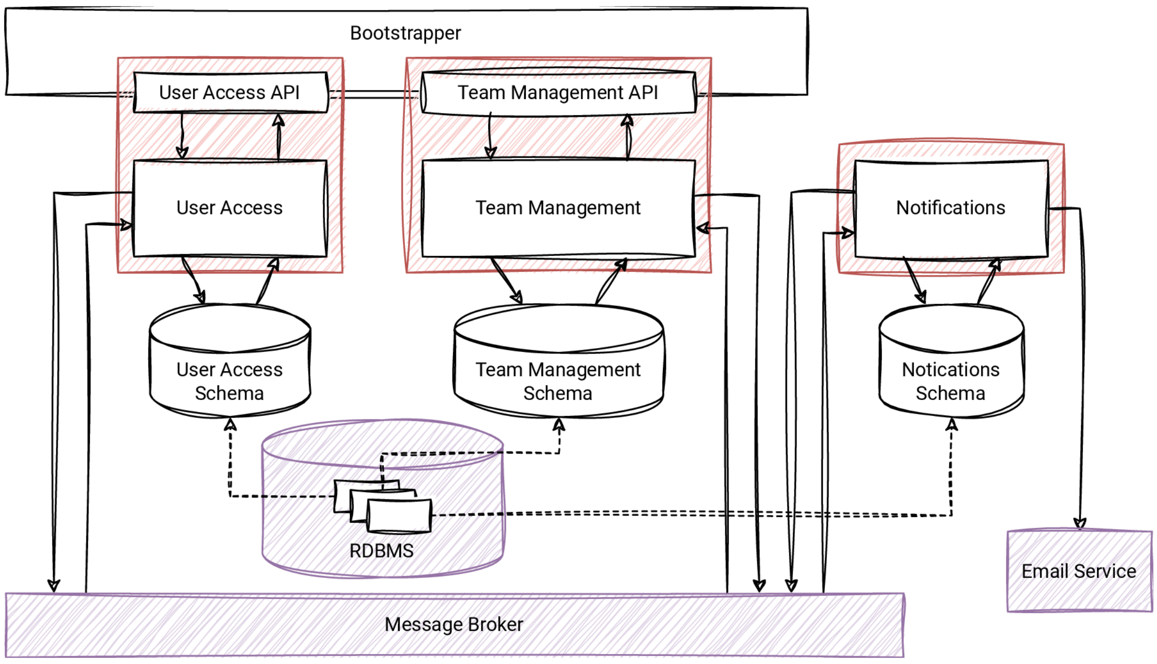


Figure 3.16: Diagram showing data isolation, communication between modules (red) and integration with infrastructure services (purple). Note that each module also has background processing services, which are omitted from this diagram for simplicity.

3.5.3 Implementing solution

- Message Broker: **RabbitMQ** as a reliable messaging and streaming broker without cloud vendor lock-in. Popular non-cloud specific alternatives are Apache Kafka, ActiveMQ or Redis.
- Communication framework: **MassTransit** is popular open-source distributed application framework for .NET, providing a consistent abstraction on top of the supported message transports (RabbitMQ, Amazon SQS, Azure Service Bus, etc.). Popular alternatives to MassTransit are NServiceBus or Wolverine.

Project structure:

```
TeamUp/ ........................................... Modular Monolith repository
├── src/
│   ├── Common/ .............................. shared types and building blocks
│   ├── Modules/
│   │   ├── UserAccess/ ............................. user access module layers
│   │   ├── TeamManagement/ .................... team management module layers
│   │   └── Notifications/ ....................... notifications module layers
│   └── TeamUp.Bootstrapper/ ........................... entry point assembly
├── tests/
├── TeamUp.sln
└── ...
```

Modules Separation

Each module is organized with a clean architecture-like structure, composed of required layers and optional layers tailored to its functionality:

- **Contracts Layer** containing all types and abstractions necessary for communication with a given module (commands and queries, response types, integration events, etc.).
- **Application Layer** with the logic of the module
- **Infrastructure Layer** containing the module's abstraction and services for configuring, registering, and using the module.
- Optional **Endpoint Layer** if module exposes REST API endpoints.
- Optional **Domain Layer** for modules that work with domain objects applying the rich domain model and DDD principles.

Each layer type within a module utilizes building blocks and common types, shared across assemblies for a consistent and reusable layer implementation = a common assembly for each layer type (see figure 3.17).

The **Bootstrapper** assembly then links all the modules and assemblies together (see figure 3.18), establishing the application's architecture, initiating runtime environment, and providing application entry point as ASP.NET Core app.

The higher number of assemblies creates a vast amount of interconnecting dependencies, which can lead to linking incorrect dependencies. To avoid dependency hell, each layer must follow strict rules on which assemblies it can depend on (shown in figure 3.17); for enforcing these rules, see section 3.7.5 on architecture testing.

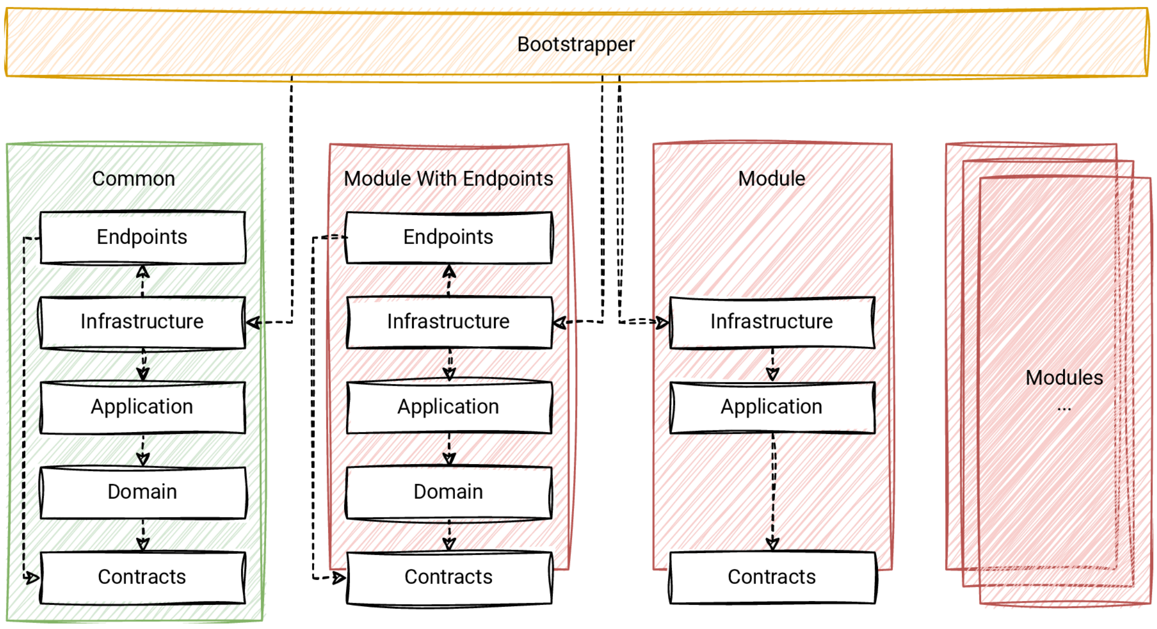


Figure 3.17: Diagram indicating the separation of modules and assemblies and the dependencies between them. Each module can depend on contracts of any module. Note that each layer in a module always depends on the corresponding common layer assembly; these dependencies are excluded from the diagram to make it more readable (dependencies are implied by the same row position).

```
...

var modules = builder.Services.AddModules(modulesBuilder =>
{
    modulesBuilder
        .AddModule<UserAccessModule>()
        .AddModule<TeamManagementModule>()
        .AddModule<NotificationsModule>();
});

var app = builder.Build();

...

app.MapEndpoints(modules);

...
```

Figure 3.18: Code from the bootstrapper assembly demonstrating the registration of modules (configurations, service implementations, etc.) and endpoint mapping using an automated process built around module abstractions (see below).

Module Abstraction

Each module (logical boundary) defines a module (class) in its infrastructure layer, where a module can inherit one of 2 base class implementations:

- `ModuleWithEndpoints<ModuleId, ModuleDbContext, ModuleEndpointGroup>` for a general module exposing REST API endpoints.
- `Module<ModuleId, ModuleDbContext>` for internal module.

The module then only needs to implement `ConfigureServices` method for registration of module specific services (and some layer information about assemblies) and based on those, the base class automatically registers/configures all required services, that is:

- command/query and event handlers
- MassTransit consumers for commands, queries and integration events
- essential services such as db context, unit of work, background jobs . . .

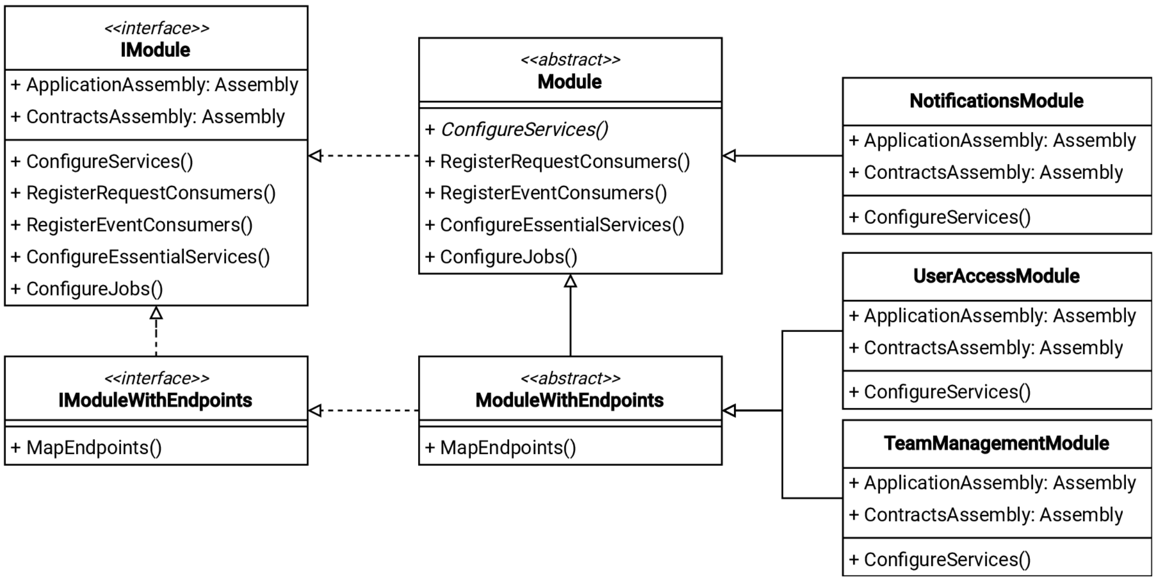

<<interface>>
IModule
+ ApplicationAssembly: Assembly
+ ContractsAssembly: Assembly
+ ConfigureServices()
+ RegisterRequestConsumers()
+ RegisterEventConsumers()
+ ConfigureEssentialServices()
+ ConfigureJobs()
<<abstract>>
Module
+ ConfigureServices()
+ RegisterRequestConsumers()
+ RegisterEventConsumers()
+ ConfigureEssentialServices()
+ ConfigureJobs()
NotificationsModule
+ ApplicationAssembly: Assembly
+ ContractsAssembly: Assembly
+ ConfigureServices()
UserAccessModule
+ ApplicationAssembly: Assembly
+ ContractsAssembly: Assembly
+ ConfigureServices()
<<interface>>
IModuleWithEndpoints
+ MapEndpoints()
<<abstract>>
ModuleWithEndpoints
+ MapEndpoints()
TeamManagementModule
+ ApplicationAssembly: Assembly
+ ContractsAssembly: Assembly
+ ConfigureServices()

Figure 3.19: A class diagram representing the module abstraction, required assembly properties serve to automatically register commands, queries, and event handlers. The diagram intentionally excludes parameters, data types, and some other properties and methods on order to better outline the abstraction design. The diagram also does not contain generics (f.e. notifications module inheriting `Module<NotificationsModuleId, NotificationsDbContext>`) that is needed by the `Module` base class to properly register essesntial services.

Some services (called *shared services* for the purpose of this thesis) need a different implementation per module, but with exactly the same logic. So the service can, for example, do the same but with a different database context (e.g. unit of work). The challenge is to provide the correct service implementations to the respective modules.

Since the DI provider integrated in frameworks on the .NET platform does not support modular dependency injection[10], it is necessary to work around this so that each module is injected with the corresponding *shared service* implementation:

- One solution is to register *shared services* as a **keyed service** and retrieve service implementation from DI container by type and key.

  e.g. `ctor([FromKeyedServices("userAccessModule")] IUnitOfWork instance)`

- The more convenient approach is to take advantage of generics and mark service implementations with class representing **module id** (in contracts layer) which is passed to the module base class and leveraged to automatically register *shared services* with it.

  e.g. `ctor(IUnitOfWork<UserAccessModuleId> instance)`

Inbox and Integration Events

While publishing and processing domain events works the same as in the Clean Architecture solution (see 3.4.7), the modular monolith design adds to the complexity of integration events as the integration event, in addition to the asynchronous side effects (e.g. sending an email), also represents state changes that need to be propagated over the message bus to multiple modules, therefore introducing necessity for an outbox and an inbox per module.

Furthermore, implementing multiple outboxes and inboxes with MassTransit poses some issues (see section 3.7.1 about encountered problems).

Integration event lifetime:

1. Integration event is stored as an outbox message in database within a single transaction (covering all request changes) using the `IntegrationEventPublisher` (inside the domain event handler/domain service).
2. In the background processing iteration, the outbox consumer of a given module takes X outbox messages, converts them back to integration events, and publishes them to the RabbitMQ bus using the MassTransit API.

   If the publishing is successful, the message is removed from the outbox.
3. The MassTransit consumer registered to a given integration event receives a message with the event and stores this event as an inbox message.
4. In the background processing iteration, the inbox consumer of a given module takes Y inbox messages, deserializes them into integration events, and invokes concrete integration event handlers for a given module.

   If the handling is successful, the message is removed from the inbox.

One of the difficulties in event-driven systems is the possibility of receiving the same message multiple times, so the handlers should be idempotent if possible; the module can also maintain a buffer of recent events to detect duplicates (not used in the solution).

[10]Microsoft recommends using 3rd party DI providers, such as **Autofac** (https://autofac.org/).

Another problem originates from the possibility of handling events out of order, especially when duplicates may occur as well. The basic idea is to keep failing until the data required for correct handling arrives (some handlers may require more granular error handling).

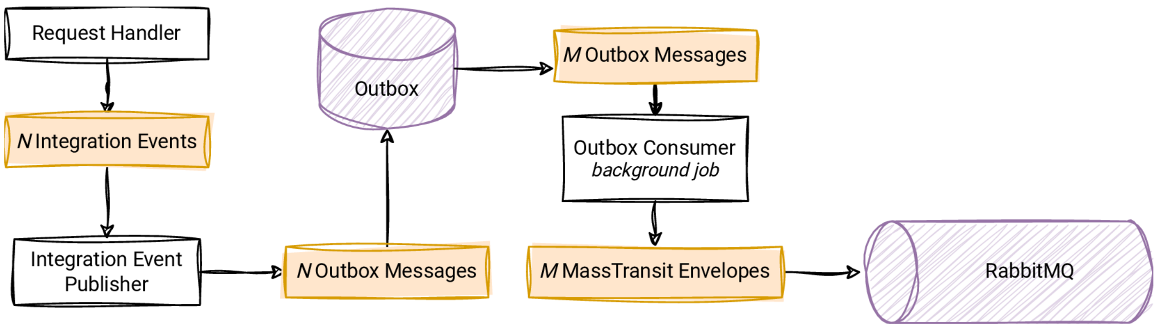


Figure 3.20: Detailed diagram of a pipeline for reliable asynchronous publishing of integration events to the event bus. This entire pipeline is executed within a single module.

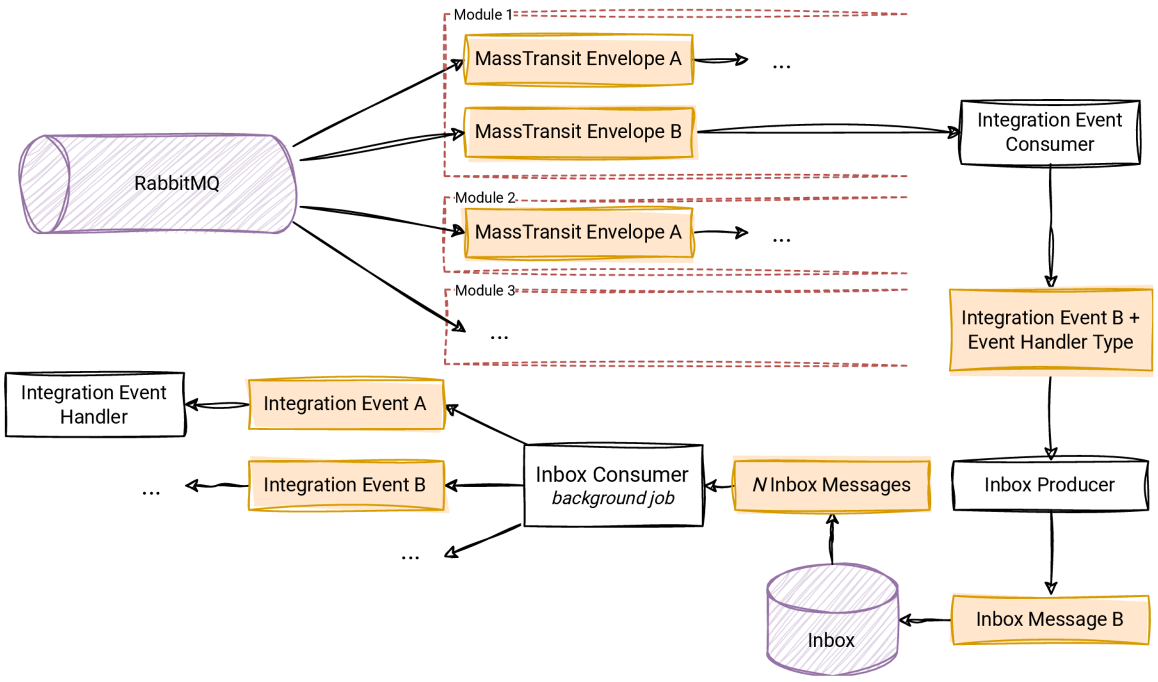


Figure 3.21: Detailed diagram of a pipeline for reliable asynchronous consumption of integration events from the event bus. As illustrated, a single event can be send to multiple modules. In case a module has more consumers for a single event, each will produce its own inbox message with a dedicated event handler.

Note that event handlers must not implement more than 1 atomic functionality (side effect/transaction). This is because the execution of one functionality can be successful and if the event handling is repeated due to the failure of others, the successful one would be repeated as well (i.e. sending 1 email multiple times). The solution does register a separate inbox message per integration event consumption for flawless multi-handling of a single event within a module (as illustrated in figure 3.21).

3.5.4 Invite unregistered user workflow example

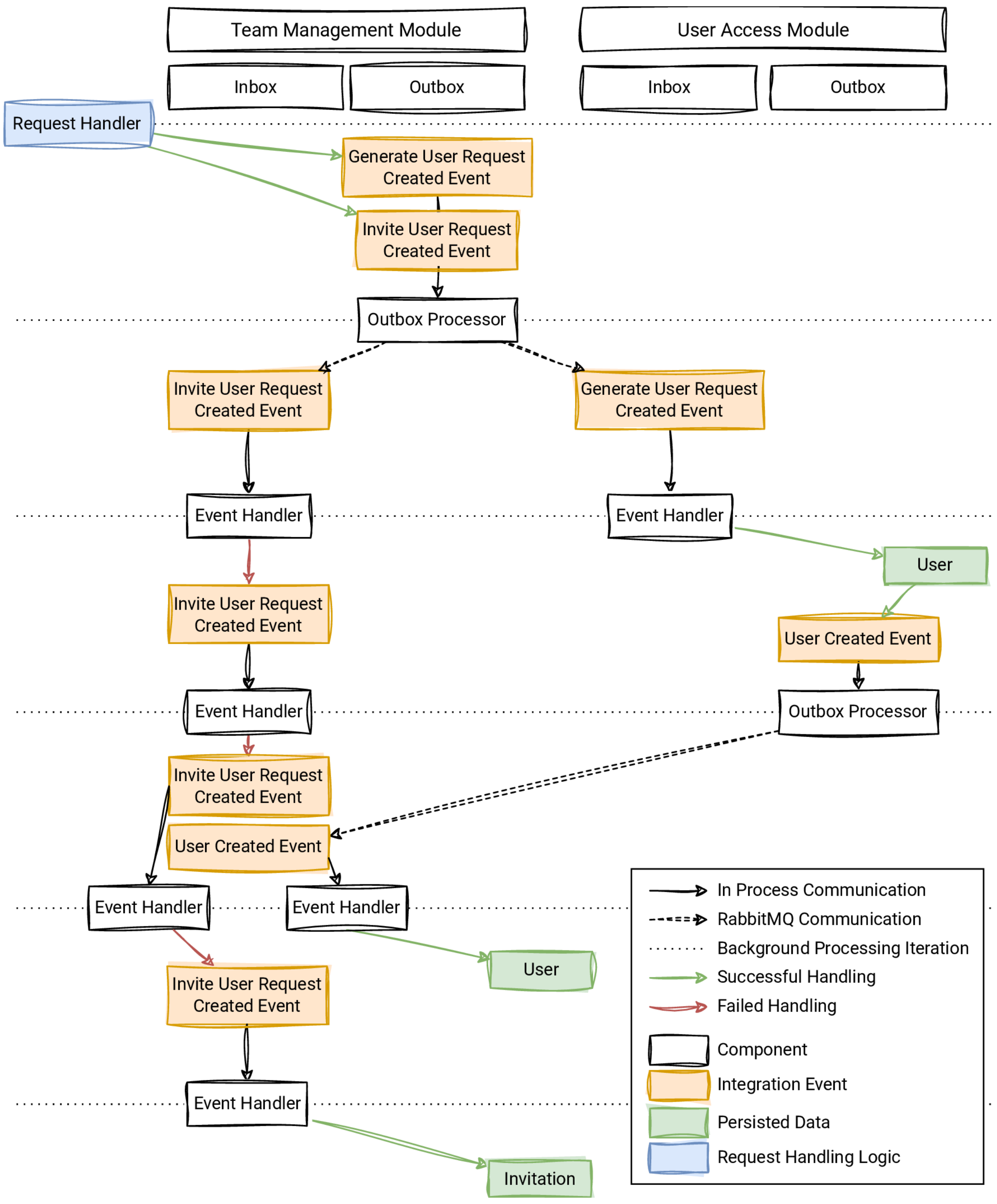


Figure 3.22: Simplified diagram illustrating the communication within and between modules to create a user invitation. Note that the handling of the *invite user request created event* failed several times until the module eventually obtained the necessary data and arrived to a consistent state for creating the invitation. In contrast to the workflow from the Clean Architecture solution (figure 3.15), the system cannot immediately return `201 Created` because the system is eventual consistent, but rather returns `202 Accepted` indicating that the API request has been successfully processed. The diagram also omits integration events sent to the notification module caused by the user and invitation creation.

3.6 Frontend app

The entire frontend solution is available online as a repository on GitHub:

https://github.com/skrasekmichael/ModularInformationSystemFrontend

3.6.1 Implementation

The frontend application is implemented using the **Blazor**[11] fronted framework with a separate hosting server from the backend and with `InteractiveAuto` render mode (see below).

Since the Blazor application logic can run both on the server and in the client browser, it is necessary to keep the authentication/authorization state accessible from both modes, that is, by storing the JWT token in a cookie.

Application UI is built using the **Fluent UI**[12] components library for Blazor by Microsoft, utilizing the fluent design system and fluent icons.

To notify and propagate data and UI changes between components across the application, a **mediator pattern** with a runtime message subscription capabilities is used; the MVVM Community Toolkit library implements such a mediator called **messenger**[13] (application uses only the `WeakReferenceMessenger`).

Project structure:

```
TeamUp/ .................................................. frontend repository
├── src/
│   ├── TeamUp/ .................................... Blazor hosting server assembly
│   │   ├── Components/ .................................. server side components
│   │   ├── wwwroot/ ................................................ static files
│   │   └── ...
│   ├── TeamUp.Client ...................................... Blazor client assembly
│   │   ├── AnonymousPages/ ......................... pages accessible without login
│   │   ├── Components/ ................................................ components
│   │   ├── Layout/ ......................................... application layouts
│   │   ├── Pages/ ................................... pages requiring logged-in user
│   │   └── ...
│   ├── TeamUp.Contracts ........................................... API contracts
│   └── TeamUp.DAL .......................................... data access components
│       ├── Api/ ........................................................ API client
│       ├── Cache/ ................................................ caching services
│       ├── Messages/ ........................................... mediator messages
│       ├── Services/ ......................................... data access services
│       └── ...
├── TeamUp.sln
└── ...
```

[11] https://learn.microsoft.com/en-us/aspnet/core/blazor/
[12] https://github.com/microsoft/fluentui-blazor
[13] https://learn.microsoft.com/en-us/dotnet/communitytoolkit/mvvm/messenger

Blazor

ASP.NET Core **Blazor** is a .NET frontend web framework for building full-stack interactive web applications and as such is the .NET flagship for frontend development.

Blazor provides a single model for both server side and client side interactivity written in C# with component-based architecture and several render modes:

- **Static SSR** where server renders non-interactive HTML content. This is also used for fast pre-rendering of content when using interactive modes.
- **Interactive SSR** (also used in *Blazor Server*) where client has open SignalR connection (Web Socket) to the server and interactive handling (button click, . . .) is executed on the server side together with re-rendering of affected components.
- **Interactive CSR** (also used in *Blazor WASM*) relies on .NET runtime build with Web Assembly (WASM) that is downloaded with the application, all code is then executed in the browser. This provides full access to functionality of the browser and interop with Java Script (DOM, local storage, etc.).

The main advantage of Blazor is the ability to combine interactive SSR and CSR together:

Starting with .NET 8.0, the first time web application is accessed, the interactivity starts on the server, and once the .NET WASM runtime and application are downloaded, the application switches to the client-side interactivity, eliminating the main drawback of Blazor WASM – the long download time. This render mode is called `InteractiveAuto`, marking that system automatically decides which render mode to use.

The framework also allows to specify the rendering mode per component, but it is not allowed to have an application with interactive SSR and CSR at the same time.

API Client

There are several key aspects to the API client integration with the backend:

- **CORS** – server CORS settings allow API calls from the Blazor application.
- **Token Injection** – the API client automatically injects JWT tokens into API calls when user is logged in.
- **Error Handling** – since the backend API returns a problem detail response (or validation problem details) in case of errors, the API client can easily parse the responses and act accordingly. If the API returns an unauthorized response code, the API client logs the user out since the unauthorized code indicates token expiration.
- API calls vary based on interactive modes:
  - SSR: Calls originate from the frontend server to the backend server API.
  - CSR: Calls are made directly from the browser to the backend server API.

Caching

Client-side caching can eliminate many network calls and therefore has the potential to improve performance more than the server-side caching, in addition to using user resources. However, the client-side caching introduces bigger chance for a user to work on invalid data, since updates from other users cannot invalidate the client-side cache,

The **cache-aside pattern** is implemented using the browser's key-value local storage (see figure 3.23), working as follows:

1. UI requests data (API contracts) via a DAL service.
2. DAL service calls to the *Cache Façade* with the following parameters: cache key, API fallback endpoint, and cache lifetime. DAL services have methods for all GET endpoints allowing to specify the cache lifetime per contract/endpoint.
3. The *Cache Façade* data lookup:
   - On cache hit – validates cache records lifetime expiration (cache miss on expired) and returns cached data.
   - On cache miss – fetches data from the fallback API endpoint, caches them and returns them to the DAL service.

One problem arises when a user logs out (intentionally or by token expiration) and the data remains in the local storage. Hence, if another user logs in, the application displays incorrect data and the user has read access to someone else's data until the token expires. This is resolved in the application by also caching the *cache owner*, and clearing the cache if the logged-in user differs from the *cache owner*. This allows reuse of the cached data when re-logging in, however, the data is still accessible before logging in and the application provides a *clear cache* button for multi-user-single-browser situations (presumably unusual).

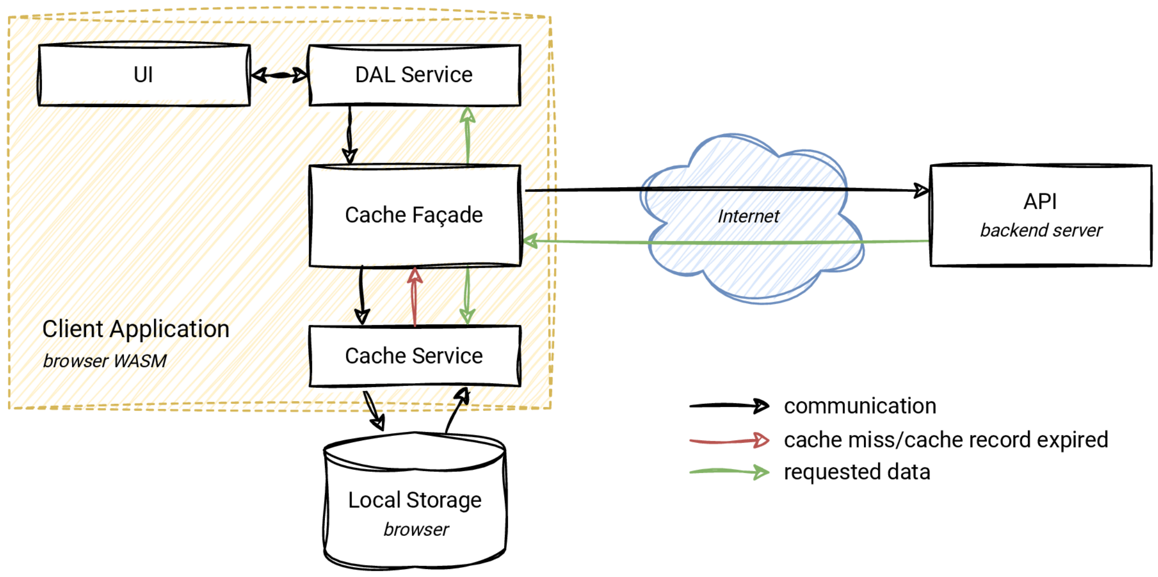


Figure 3.23: A diagram showing the implementation of the cache-aside pattern with a browner local storage serving as a cache storage and an API call as a fallback in case of a cache miss. Note that the data is cached after each GET request.

Client-side caching in the Blazor applications is challenging due to the possibility of running the code both on the client-side and server-side, introducing problem to uniformly access a cache. That is why the diagram in figure 3.23 has additional *Cache Service* layer around the local storage to provide different caching service on server-side and client-side.

Server-side rendering modes have following properties:

- In static SSR mode, the application does not have access to the browser resources and the only caching option is to use server-side caching solution or some sort of HTTP cache (both are undesirable due to incompatibility with the client-side cache).
- In interactive SSR mode, the application has access to Java Script interop after a page rendering, resulting in initially calling the API and later in cache hit/miss.

Possible solutions:

- One solution is to not use cache on the server, and always call API to retrieve data.
- Another solution migth be to implement different caching mechanism on the server (i.e. in-memory cache) and lose all cached data when switching to CSR. In such case, it may be preferable to implement a robust caching mechanism on the backend server and always call the API to retrieve data in the Blazor application.
- The solution used in the application is to not use the cache when the local storage is unavailable (static SSR, early interactive SSR), but otherwise to use it even though cached data is transferred from the client to the server, processed there, and transferred back to the client, because this approach maintains a single consistent cache storage and the application can use these cached entries when switching to CSR mode.

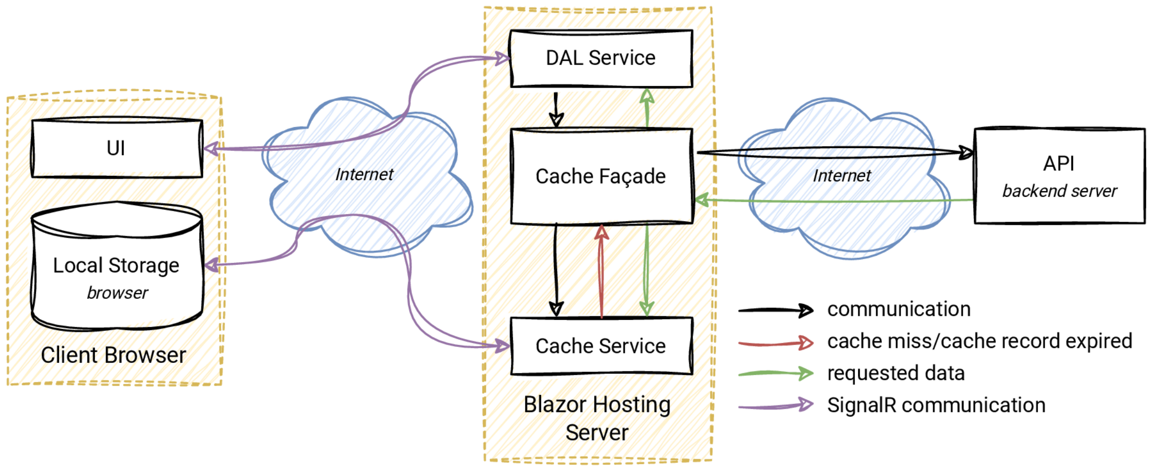


Figure 3.24: A diagram showing the implementation of the cache-aside pattern when the application is running in the interactive server mode. Note that the Blazor hosting server can be positioned either close to the server (e.g. on the same machine) or distributed at the cloud edge (close to the client), reducing communication penalty either to the client or to the API, respectively.

3.7 Interesting parts

This section covers topics and parts of the solutions and development process that are not the primary focus of this thesis, but may still be of interest to the reader.

3.7.1 Encountered problems

While developing backend solutions, I encountered problems regarding technologies and design choices that were challenging or interesting to solve.

Inbox Starvation

The inbox (outbox) starvation is a term coined by me[14] to refer to a problem where the inbox contains a sufficient number of integration events that continuously fail to be consumed, and the entire consumption gets stalled because the background service keeps processing the same failing events to preserve the order of events.

This can be addressed by continually increasing the delay between the consumption of failed events, by deleting/skipping events that have failed multiple times, or by a combination of both; the solution heavily depends on the concrete logic of integration event handlers.

Some integration event handlers may even count on repeated failures due to eventual consistency (waiting for data that will be eventually available, see for example workflow diagram in figure 3.22), this leads to a reduction of effectively consumed events per iteration or even to temporary inbox starvation until repeatedly failing events are successfully handled.

In the Modular Monolith solution, the outbox starvation is not a problem because the inability to consume outbox messages indicates that the event bus is down (low probability, the application should not work in this case) or that there is a bug.

The Clean Architecture solution has the same problem with outbox starvation, because the outbox and inbox are merged together. Since the system uses integration events only for sending emails, failing to send should only delay the processing of other emails, but there is a possibility of a tailored attack targeting the inbox that could lead to starvation. Moreover, the inbox starvation problem should be taken into account when adding more asynchronous side effects to the system.

Both solutions implements measures of tracking the number of times a message failed to be processed and continuously postponing the next processing of a given message. This also potentially changes timings and duration of certain workflows that are based on waiting in the inbox for an eventually consistent state (for example workflow diagram in figure 3.22 might fail fewer times but take longer to complete.).

Note that designing strategies (postponing timings, growth rate, etc.) for handling repeated message processing failures is a complex task and bad error handling strategy might crash the entire system. Continuous monitoring and possible strategy adjustments are a good start to address such problem. In addition, it would be possible to split the inbox/outbox messages into groups with different error handling strategies or even specify a strategy per integration event handler as a more granular approach.

[14]The analogy is that the system is starving for new events.

Multiple outboxes with MassTransit

MassTransit is an excellent framework for building distributed systems (e.g. microservices) with the ability to configure an outbox pattern, however, it is limited to one database context and one outbox per application, which leads to limited usability in a modular monolith applications.

Explored solutions:

- Different communication framework, for example NServiceBus, might solve the problem, but a brief study of the documentation have not confirmed this ability (it is not the most requested feature) and migrating to/learning a new framework is expansive.
- A single outbox for the entire modular monolith in a separate database schema and transactions covering the module schema and the outbox schema is a possible solution, but it moves away from the modular ("microservice-like") approach and may introduce unintuitive issues (odd behaviour during tests).
- Mimicking microservices in the context of dependency injection (looking like a single executable) using a modular DI provider (e.g. Autofac) so that each module registers its own services required by MassTransit, including the database context for the outbox, etc. This solution does not imply success, as there might still be issues with registration of background services, plus switching DI providers is both knowledge and resource intensive and the new provider may be incompatible with other libraries.
- Solution used in this thesis: custom implementation of inbox and outbox, and integrating them with MassTransit as a transport layer, that includes:
  - Definition/configuration of outbox and inbox messages, including db migration.
  - Implementation of background services for consuming the inbox and outbox (2 background services per module).
  - Wiring MassTransit consumers to respective module inbox.

  This allows complete control over how the system should work, inducing inbox starvation, but introduces extra work and complexity. Custom solution might also be full of bugs, inefficiencies, and bad design choices.

Implementing the outbox pattern for request-response communication (commands/queries) is unnecessary because the failure is propagated back to the request producer, which is:

A) Inside the integration event handler (the main use case), when the integration event handler fails and the inbox pattern for integration events kicks in.

B) Inside the request processing logic, in which case the request fails and the error message is delivered back to the API client (the user) who resolves the error accordingly.

Alternatively, MassTransit offers the option to use In-Memory Outbox, which supposedly ensures eventual consistency in the event of a database or a message transfer failure.[15]

[15]see `https://masstransit.io/documentation/patterns/in-memory-outbox`

Value Objects in Entity Framework Core

The Ef Core supports multiple ways of persisting and dealing with value types:

- The EF Core has the ability to provide custom **value conversions** to serialize types into a single column, which is most probably not a solution for a value object, because the system loses the ability to effectively use object properties in queries.

  This option might be useful for single-valued objects (i.e. strongly typed id) or for types where querying over nested properties does not make sense or will not be used.

  For example, the query capability over the `R` part of the `RGB` value seems pointless, however, the ability to select only records where `money` value is in certain `currency` (nested property of the `money` value object) may be desirable.

- Since version 2.0, the EF Core provides the **Owned Entity Type** capability, as described in the EF Cores documentation [16]; entities use implicit keys, meaning that a shadow id is created so that the EF Core is able to keep track of them, which contradicts the idea of a value object.

- As of version 8.0, the EF Core introduces the **Complex Type** option. An official blog post about the version [26] highlights 2 main advantages over owned types:
  - Complex types are not identified or tracked by key value.
  - Complex types can be both **.NET value types** (structs) and reference types.

The solutions use value converters for strongly typed ids and for password type (composed from *seed* and *hash* values) without problems.

Unfortunately, the use of a complex type (`EventReply` composed from *reply type* and *message*) caused a problem when using `GROUP BY` over the *reply type* property:

The EF Core was throwing an exception that it was unable to build an SQL query and wanted to fetch all data from the database and perform grouping in-memory.

Possible solutions:

- Solution used in this thesis: removing the value object from the entity property and using it only as contract in methods with manual mappings to entities properties.

- Another solution could be to use EF Core's ability to execute RAW SQL queries and build the SQL query manually, since the database has the actual properties available in separate columns for available grouping.

- Probably the best solution would be to embrace the CQRS pattern and use a more lightweight approach for querying the data, like a direct SQL client or preferably a micro ORM framework, which would most likely lead to better performance.

Although attempts to reproduce this error have been made, they have not been succesful, and using `GROUP BY` in similar situations resulted in the correct and expected behaviour.

3.7.2 Result Pattern and Railway Oriented Programming

The entire application solution leverages the result pattern for error handling, and most of the business logic takes advantage of railway-oriented programming to provide cleaner and easier-to-understand code.

To avoid code duplicity between the clean architecture solution and the modular monolith solution, as well as the possibility of future usage, a decision to publish the result pattern implementation and railway-oriented extension methods as separate nuget packages[16] has been made.

Result Pattern

The result pattern, result object or operation result, is a way to control error handling in an application without the use of exceptions, as the exception handling (try-catch blocks) branches the code, thereby reducing code clarity and obscuring the underlying business logic. Another drawback of using exceptions might be the performance, as throwing an exception is an expensive operation. On the other hand, the ability to trace the exact location of the thrown exceptions in code (stack trace) brings superior debugging experience.

The result object is a **discriminated union** of **success state** and **failure state**, where the operation (method) returns a success result if no error occurred, and a failure result if some sort of error/exception case occurred (validation error, IO error and so on). The result object also carries the returned operation value in case of a success state, or error details (error code, message) in case of a failure state.

In the context of web applications providing REST APIs, the result object can be mapped to the corresponding http response – Ok/Created/Accepted/... for a successful result, and in the case of a failed result, the error can be mapped to the **problem details**[17] with appropriate details including status code (BadRequest/NotFound/Conflict/...) and other.

Predefined error types for result pattern:

- `AuthenticationError` – authentication failure (invalid credentials).
- `AuthorizationError` – forbidden to perform operation/forbidden access.
- `ValidationError` – invalid input request parameter.
- `NotFoundError` – domain object not found.
- `ConflictError` – existing conflicting domain object, concurrency conflict, etc.
- `DomainError` – broken business rule that does not fit into other errors types.
- `InternalError` – unexpected behaviour (uncaught exception, bug, etc.).

Using the implicit operators feature in C-Sharp, the boilerplate for creating the result object can be removed, and therefore the error types and returning values implicitly create a new result object (see example in figure 3.25).

[16] https://github.com/skrasekmichael/RailwayResult
[17] https://datatracker.ietf.org/doc/html/rfc7807

```
public Result SetMemberRole(UserId initiatorId, TeamMemberId memberId, TeamRole newRole)
{
    if (newRole.IsOwner())
        return Errors.CannotHaveMultipleTeamOwners;

    var initiatorResult = GetTeamMemberByUserId(initiatorId);
    if (initiatorResult.IsFailure)
        return initiatorResult.Error;

    if (!initiatorResult.Value.Role.CanUpdateTeamRoles())
        return Errors.UnauthorizedToUpdateTeamRoles;

    var memberResult = GetTeamMember(memberId);
    if (memberResult.IsFailure)
        return memberResult.Error;

    if (memberResult.Value.Role.IsOwner())
        return Errors.CannotChangeTeamOwnersRole;

    memberResult.Value.UpdateRole(newRole);
    return Result.Success;
}
```

Figure 3.25: Code example of using the result pattern in the `SetMemberRole` method of the team aggregate, where `Errors` is a static class containing predefined errors.

Railway Oriented Programming

Railway-oriented programming (ROP) is an approach to error handling in functional programming. The name comes from an analogy presented by Scott Wlaschin at the NDC London 2014 conference [28], which presents the flow through a program as a railway with 2 tracks, a success track and a failure track, and for each operation/method the state can switch to failure track if an error occurs (see figure 3.26).

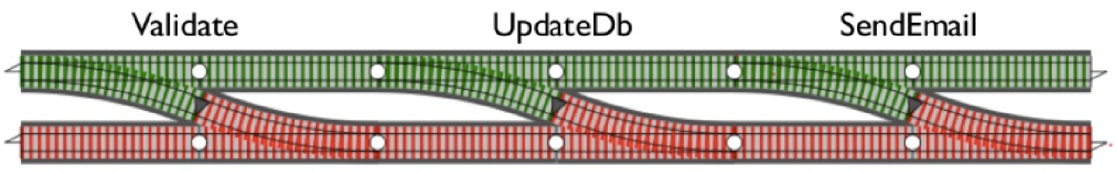


Figure 3.26: An example of a program flow where each function can throw an error and switch the state to the failure track, thus not invoke the following functions. This example it taken from the presentation slides from the [28] blog post.

This is a natural extension of the result pattern as the result object representa a state on the railway (either success or error) and the methods returning result pattern act as switches on the program railway.

However, in classical programming using the result pattern, the result object is only returned and not taken as an input in the methods, and it is undesirable for each function to take the result object as a parameter and have all the original parameters as a nested value of the result object. The solution is to introduce functional extension methods on the result object that always check the state of the result object and then apply the logic wrapped in the passed parameter.

Main extension methods:

- `.Ensure(predicate, error)` – validates whether the value meets the condition on the success track, or returns the `error` (switches to failure track).
- `.Then(mappingFunction)` – maps the value of the result object on the success track to a new value (also called .Map). If the mapping function returns a failure result, it returns its error (switches to failure track).
- `.Tap(function)` – invokes a function (side effect) on the success track but does not change the type of the result object. If the mapping function returns a failure result, it returns its error (switches to failure track).
- `.And(mappingFunction)` – generates another value using the mapping function on the success track and encapsulates it with the initial value inside the result object.

To fully utilize ROP in the .NET, the asynchronous variants of the methods need to be implemented to perform non-blocking operations.

```
public Result SetMemberRole(UserId initiatorId, TeamMemberId memberId, TeamRole newRole)
{
    return newRole
        .Ensure(Rules.RoleIsNotOwner, Errors.CannotHaveMultipleTeamOwners)
        .Then(_ => GetTeamMemberByUserId(initiatorId))
        .Ensure(Rules.MemberCanUpdateTeamRoles)
        .Then(_ => GetTeamMember(memberId))
        .Ensure(Rules.MemberIsNotTeamOwner, Errors.CannotChangeTeamOwnersRole)
        .Tap(teamMember => teamMember.UpdateRole(newRole))
        .ToResult();
}
```

Figure 3.27: Code example of the same `SetMemberRole` method in the team aggregate as in figure 3.25, but utilizing railway-oriented programming. `Rules` is a static class with predefined rules, that is, predicates and predicates with preassigned appropriate errors. The `.ToResult()` method drops the wrapped value and returns plain result object. This is a copy of the exact code used in the solution source code.

Even though the ROP results in shorter and (hopefully) cleaner code, there are a few drawbacks mentioned by Scott Wlaschin [29].

The main disadvantage of railway-oriented programming is **performance**, since when using ROP the code loses the ability to return early from a function (aka fail fast), thus introducing the necessity to go through the entire railway statement even if an error occurs in the first method.

Furthermore each method of railway statement has an overhead of at least 1 track check and an allocation of data (parameters and possible new result), and even more for asynchronous variants as the methods bootstrap a state machine and allocate more data.

3.7.3 Database Concurrency Conflicts

The EF Core does not support multiple parallel operations running on the same database context, so in ASP.NET Core applications, contexts are scoped per http request, meaning that there may be 2 (or more) concurrent modifying requests to the same record without the contexts knowing and addressing this conflict.

If a concurrent request modifies the same record at the time of checking the business rules (after fetching data from the database but before committing changes), the initial request transaction will conclude with possibly incorrectly evaluated business rules or update the record to a new value calculated from the initially fetched data, resulting in data loss and leaving the database in an inconsistent state. This phenomenon is also known as a **race condition**.

An example of data loss when updating account points:

```
req A: get points -> pointsA = 100
req B: get points -> pointsB = 100
req B: set points -> points = pointsB - 50 = 50 -> commit 50
req A: set points -> points = pointsA + 50 = 150 -> commit 150
```
`->` as a result, request B loses its effect.

Inconsistent state example when reserving tickets:

```
req A: get number of available spots -> availableA = 3
req B: get number of available spots -> availableB = 3
req A: reserve 2 tickets -> availableA - 2 >= 0 -> commit reservationA
req B: reserve 3 tickets -> availableB - 3 >= 0 -> commit reservationB
```
`->` resulting in reservation of 2 extra spots over the capacity.

One way to solve this is to lock the database record for the duration of a business transaction (from fetching data till committing the transaction, requires holding an open connection), this is also known as **pessimistic locking**. This affects performance as locking is not a cheap operation, and other transactions affecting same record must wait until the exclusive lock is released, which can lead to a potential **deadlock**. Another downside of pessimistic locking is that it is not supported by EF Core out of the box.

Optimistic Locking

The other solution is to use optimistic locking, which does not use any locks, and will fail to update if the data has changed since it was loaded. The idea is to have an additional property (aka a concurrency token) on the record indicating the record's version, based on which the system detects that the data has been changed since fetching.

This approach is directly supported by EF Core and the concurrency token implementation varies across databases [17]:

- SQL Server – uses implicit hidden auto-updating column `rowversion` containing timestamp of last modification (data type is `byte[]`).
- PostgreSQL – uses implicit hidden auto-updating column `xmin` containing the ID of the latest updating transaction (data type is `uint`, see figure 3.28).

The EF Core then always fetches the concurrency token when retrieving data of record with the concurrency token. Although this results in overhead, it is still more efficient than pessimistic locking, assuming a low probability of concurrency conflict.

```
protected override void ConfigureEntity(EntityTypeBuilder<TeamMember> teamMemberBuilder)
{
    ...

    teamMemberBuilder
        .Property(teamMember => teamMember.Nickname)
        .IsRequired()
        .HasMaxLength(255);

    teamMemberBuilder
        .Property<uint>("RowVersion")
        .IsRowVersion();
}
```

Figure 3.28: Code sample configuring EF Core (utilizing `IEntityTypeConfiguration` interface) to use a property of the `TeamMember` entity as a concurrency token. Note that the code does not explicitly reference the property, but rather uses a shadow property, meaning the property exists in database as a column but does not exist in the domain object itself, and thus does not pollute the domain with properties with no business value.

The concurrency token most probably does not need to be in every entity because the race condition may not always cause a problem, with regards to the *TeamUp* application:

- `User` entity needs a concurrency token as the entity has a *NumberOfOwnedTeams* column used to solve the enforcing of the business rule (max number of owned teams) in a race condition when concurrently creating teams by the same user. The conflict with concurrent creation of users with the same email is solved by a unique constrain.
- `Team` entity requires a concurrency token since this entity, similarly to the `User` entity, has a column (*NumberOfMembers*) that is used to solve the enforcing of the business rule (max number of members) in a race condition when concurrently adding new members to the same team.
- `TeamMember` entity requires a concurrency token because concurrent requests to change a team ownership may leave the team with multiple owners.
- `Event` entity has to have a concurrency token as concurrent changes to the event may leave the record in an invalid state – f.e., changes of start and end dates.
- `EventType` and `EventResponse` entities do not require a concurrency token since concurrent modification does not leave the system in an inconsistent state and data loss is irrelevant. The race condition resulting from concurrent creation of `EventResponse` is solved by a unique constraint on the composite index (`eventId` and `teamMemberId`).
- `Invitation` entity does not have mutable columns and concurrent creating of invitations is solved by a unique constraint on the composite index (`teamId` and `userId`).

Note that these solutions describe the clean architecture solution, the modular monolith solution is mostly the same, some entities are just split/copied into multiple db schemas.

3.7.4 Integration Testing

In integration and end-to-end (E2E) testing, there is the question of whether to test with real service instances or in-memory mocks/versions, especially for testing database integration.

The advantage of mocking service dependencies via in-memory alternatives is performance as tests run really fast and can be run in parallel.

The disadvantage is that some services may not have in-memory versions (EF Core does provide option to use db context around in-memory database), moreover, the in-memory mocks-ups may differ from the real implementation (i.e. database transactions …) and therefore change the way the system works in testing and production.

For E2E testing, the application is tested with real instances of database and bus (for modular monolith solution) and the mail server is mocked with in memory inbox. The testing is implemented using **xUnit**[18] as a testing framework and following libraries:

- **TestContainers**[19] for running docker containers of real services, such as postgres or RabbitMQ, and using them for E2E testing of the application.
- **FluentAssertions**[20] for providing a more natural way of specifying expected outputs.
- **Respawn**[21] by Jimmy Bogard for resetting database between tests.
- **Bogus**[22] for deterministic generation of fake testing data.
- `Microsoft.AspNetCore.Mvc.Testing` package providing ASP.NET Core application factory configurable for testing purposes and running it in-memory.

Using the xUnit's *Collection Fixtures*, all E2E tests are run against the same services: the backend, the database and the service bus. These services are initialized at the start and xUnit then runs all tests sequentially:

0. Init: reset database into initial state using the **Respawn** library, reset additional properties used for testing (inbox, counters, datetime provider and so on).
1. **Arrange**: generate testing data using the **Bogus** library and custom extension methods on top of it, persist data into the database or use them as payload, authenticate.
2. **Act**: send HTTP request against the tested endpoint of the app running in memory.
3. **Assert**:
    - Test correct HTTP response code and error response.
    - Test correct response payload when requesting query using the **FluentAssertuions** library and the *object graph comparison* feature.
    - Test correct database state when requesting Command.

[18]https://xunit.net/
[19]https://testcontainers.com/
[20]https://fluentassertions.com/
[21]https://github.com/jbogard/Respawn
[22]https://github.com/bchavez/Bogus

```
[Theory]
[InlineData(TeamRole.Member)]
[InlineData(TeamRole.Coordinator)]
[InlineData(TeamRole.Owner)]
public async Task GetTeam_AsTeamMember_Should_ReturnTeam(TeamRole initiatorRole)
{
    //arrange
    var initiatorUser = UserGenerators.User.Generate();
    var members = UserGenerators.User.Generate(19);
    var team = TeamGenerators.Team
        .WithMembers(initiatorUser, initiatorRole, members)
        .Generate();

    await UseDbContextAsync(dbContext =>
    {
        dbContext.Users.AddRange(members.With(initiatorUser));
        dbContext.Teams.Add(team);
        return dbContext.SaveChangesAsync();
    });

    Authenticate(initiatorUser);

    //act
    var response = await Client.GetAsync(GetUrl(team.Id));

    //assert
    response.Should().Be200Ok();

    var teamResponse = await response.ReadFromJsonAsync<TeamResponse>();
    team.Should().BeEquivalentTo(teamResponse);
}
```

Figure 3.29: Code sample from the Clean Architecture solution of E2E test against a GET endpoint. Note that the test is written using the AAA pattern.

Testing Eventual Consistency

The system consists of components that respond to a backend request with a delay and only after the request is completed, thus introducing eventual consistency (e.g. sending emails, but mainly propagating data between modules in the modular monolith). Testing eventual consistency might be challenging since waiting for a fixed amount of time may be inefficient and lead to nondeterministic test results.

In this case, the solution was to introduce callback invocations to desired components in testing environment and wait for these callbacks after a request (with timeout to avoid livelock), where desired components are:

- Outbox processor (background job) in the Clean Architecture solution, because after the outbox processing completes, the integration event raised by the request is handled or the handling has failed (test failed and caught a bug).
- Inbox consumer (called by background job) in the Modular Monolith solution with separate callback for each integration event handler invoked when successfully handled. In testing, it is necessary to specify for which (or multiple) integration event handler the test needs to wait (see figure 3.30).

The implementation of the callback mechanic uses the C-sharps `TaskCompletionSource<T>` feature[23], which enables the ability to wait for a result of a task (waiting for a callback) and set a result of that task from different/concurrent thread (invoking the callback).

```
...

//clean architecture
await WaitForIntegrationEvents();

...

//modular monolith
await WaitForIntegrationEventHandler<TeamManagement.UserCreatedEventHandler>();
await WaitForIntegrationEventHandler<Notifications.EmailCreatedEventHandler>();

...
```

Figure 3.30: Rough code example of the methods used in the solutions for waiting for the completion of integration event handlers. Note that in the modular monolith, each module may have a handler for a given event, so a distinction by module needs to be used.

Testing Race Conditions

To test that the measures against concurrency conflicts and race conditions (mentioned in section 3.7.3) work, it is necessary to be able to force a conflicting situation.

Using the previously mentioned callback mechanic, it is possible to adjust the `SaveChanges` method of the unit of work pattern that is used to persist changes and introduce *before commit* and *can commit* callbacks to enforce a conflicting situation (see figure 3.31).

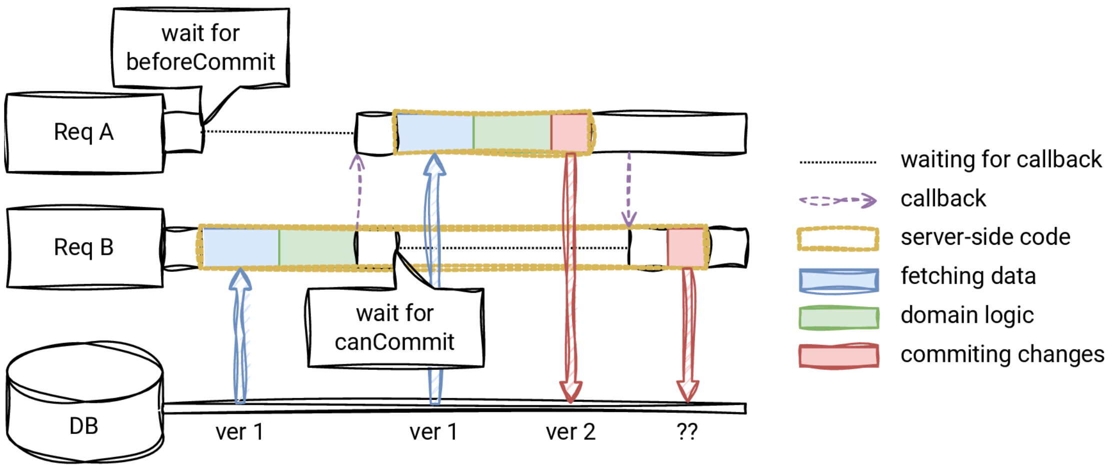


Figure 3.31: Diagram showing the enforcement of a conflicting situation between 2 requests by delaying committing changes of request B till request A is initiated and fully processed.

[23]https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskcompletionsource-1

3.7.5 Architecture Testing

As the project grows, it is essential to ensure that all developers follow the architecture rules and DDD principles to take advantage of the architecture design and to avoid introducing bugs and unwanted coupling that would lead to the big ball of mud.

.NET compiler – **Roslyn**[24] – provides a powerful API surface for building code analysis tools and as such can be used to develop analyzers that flag violations of architecture rules and guidelines at compile time and subsequently suggest fixes.

However powerful and versatile analyzers can be, their development is not cheap (in terms of knowledge) as a developer may need to understand static code analysis, compiler theory and framework for building the analyzers – something that a traditional backend developer does not.

An alternative approach taken in the development of *TeamUp* is to create tests using a specialized library, such as **NetArchTest**[25], with a basic knowledge of reflection (known by a generic backend developer) to test rules like:

- dependency rules – dependency of types in assembly/namespace on types from another assembly/namespace (i.e. clean architecture dependency tests, see figure 3.32).
- type naming conventions – e.g. a domain event must have a *DomainEvent* suffix.
- type properties – types have to be abstract/static/sealed/read-only etc.
- type existence – for instance, class representing `Command` contain a nested validator.
- DI rules – service can be injected only into certain classes (assuming injection via constructors, i.e. query context can be injected only into query handlers).
  ...

```
[Theory]
[InlineData(Application)]
[InlineData(Infrastructure)]
[InlineData(Presentation)]
public void Domain_Should_NotHaveDependencyOn(string dependency)
{
    var result = Types.InAssembly(DomainAssembly)
        .Should()
        .NotHaveDependencyOn(dependency)
        .GetResult();

    result.IsSuccessful.Should().BeTrue();
}
```

Figure 3.32: Code sample of a dependency test from the Clean Architecture solution testing whether the Domain layer depends on any types from the Application, Presentation, or Infrastructure layers. Noteworthy is that the Modular Monolith solution has more complex and more important dependency tests (see repository in section 3.5), as it has to follow more complex dependency rules to avoid the big ball of mud.

[24] `https://github.com/dotnet/roslyn`
[25] `https://github.com/BenMorris/NetArchTest`

Chapter 4

Conclusion on selected architectures and their comparison

This chapter summarizes my final thoughts on selected architectures – Clean Architecture (section 4.1) and Modular Monolith (section 4.2) – after developing a demo application in both architectures while applying domain-driven design principles.

4.1 Clean Architecture

The clean architecture is more modular than the traditional layered architectures, especially when integrating with external systems (databases, emailing services, etc.), providing easier exchangeability of such systems. Compare to more modular architectures, it may be difficult to implement larger systems.

The architecture does not really invite the application of DDD strategy design since it is difficult to implement/separate bounded contexts and in the demo application it seemed counterproductive to do so. However, the DDD tactical design is easily applicable and introduces positive effects on the system (rich domain, clear boundaries between entities, etc.).

From the way the clean architecture is interconnected, it is not the most suitable architecture for migrating to microservices.

The main advantage of clean architecture is the benefit of rapid development in early development stages while retaining decent maintainability as the system grows. Adding features to the system does change multiple layers, which limits development in larger/multiple teams. It is therefore not suitable for larger systems.

In my opinion, the clean architecture together with DDD provides a good solution for mid-level information systems with basic modularity or for implementing modules (in a modular monolith) or services (in a microservice architecture).

Given the scale of the demonstration application, its business complexity and technical requirements, a clean architecture with DDD tactical design and CQRS seems fitting.

4.2 Modular Monolith

The modular monolith architecture excels in modularity (as the name suggests) and can match microservices in its ability to implement large-scale information systems.

The separation into modules works well with the DDD strategy design, as a module can represent a bounded context and the strict rules between modules mimic the nature of bounded contexts. A module can then be implemented in whatever manner, but implementing modules using a clean architecture and DDD tactical design has proven itself.

Because a system is decomposed into modules similarly to microservice, the architecture can leverage its benefits for developer productivity in large/multiple teams and large information systems. The similarities with microservice architecture imply an easy (relatively speaking) transition to microservices, although it depends on the level of modularity and isolation chosen to implement a modular monolith.

Modular monolith introduces multiple complexities (as encountered in developing of demo application – communication problems, eventual consistency complexity, etc.) and is still vulnerable to poor design choices, resulting in distributed monolith.

I am of the opinion that the modular monolith is the ideal starting architecture for large enterprise-wide systems, however, it is uncommon to start building enterprise system out of nothing and it is hard to know whether a system becomes large-enough. Introducing this architecture for a smaller project (similar to microservices) brings unnecessary complexity.

The development of the demo application had not produced problems that a modular monolith is designed to solve, and the use of the modular monolith seemed excessive. Although some changes appeared cleaner/easier in the modular monolith (e.g. adding background services for cleaning invalid data), such occurrences were minimal and to properly see the benefits of modular monolith, either further experiments on the same application would need to be made, or a large-scale application in large teams would have to be developed (which is not coverable in a thesis).

Due to complexities in modular monolith, a more convenient approach might be to start a project with a modular monolith architecture with a lower lever of modularity (in-process communication, etc., or even use a clean architecture) and later migrate to fully modular monolith with microservices-like aspects.

Chapter 5

Conclusion

The aim of this thesis was to study and explore architectures for building modular monolithic information systems with domain-driven design and practically demonstrate two of the architectures on a chosen application.

The personal goal was to level-up my software engineering skills by studying currently popular architectures and accumulating additional knowledge on building information systems along the way, hence the scope of my own assignment.

In the context of meeting assignment objectives, I have:

- extensively studied the domain-driven design principles and explored several architectures for building modular information systems (1th objective);
- selected less and more modular architectures for deeper and practical exploration: Clean Architecture and Modular Monolith (2th objective);
- implemented demonstration application in both architectures and designed the domain using the DDD (3th objective), while continuously implementing end-to-end tests to validate correct functionality (4th objective extension);
- implemented frontend application to verify practical API usability of the backend solutions (4th objective) and introduced client-side cache (4th objective extension);
- summarized gathered experience on the selected architectures from the development process and stated my opinions in the context of application maintainability, extensibility, etc. in the penultimate chapter (5th objective).

In addition, the thesis contains a chapter covering parts of the development process that are outside the main scope of the thesis, but still may be interesting to the potential reader.

To explore and analyse more of the architectures' capabilities, additional extensions/experiments can be performed on the implemented solutions, such as:

- Changing the core domain to support new functionalities.
- Adding features and new technologies (e.g. real-time notifications using web sockets) or changing supporting infrastructure technology (database, event-bus provider, etc.).
- Transitioning implemented solutions to microservices architecture.

Bibliography

[1] Baba, D. *Using Clean Architecture to ensure Separation of Concerns* [online]. 19. october 2022 [cit. 2024-01-21]. Available at: https://medium.com/@dorinbaba/using-clean-architecture-to-ensure-separation-of-concerns-c4a9b7d8f0c1.

[2] Bogard, J. *A better domain events pattern* [online]. 13. may 2014 [cit. 2023-01-05]. Available at: https://lostechies.com/jimmybogard/2014/05/13/a-better-domain-events-pattern/.

[3] Bogard, J. *Vertical Slice Architecture* [online]. 19. april 2018 [cit. 2023-12-4]. Available at: https://www.jimmybogard.com/vertical-slice-architecture/.

[4] Cesar de la Torre, Wagner, B. and Rousos, M. *.NET Microservices. Architecture for Containerized .NET Applications* [ebook]. 7th ed. Microsoft Developer Division, 2023. Available at: https://raw.githubusercontent.com/dotnet-architecture/eBooks/main/archives/microservices/NET-Microservices-Architecture-for-Containerized-NET-Applications-7.0.pdf.

[5] Evans, E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. 1st ed. Addison Wesley, august 2003. ISBN 0-321-12521-5.

[6] Fowler, M. Unit of Work. In: Fowler, M., Rice, D., Foemmel, M., Hieatt, E., Mee, R. et al., ed. *Patterns of Enterprise Application Architecture*. 1st ed. Addison Wesley, November 2002, chap. 11. Object-Relational Behavioral Patterns, p. 166–174. ISBN 0-321-12742-0.

[7] Fowler, M. *AnemicDomainModel* [online]. 25. november 2003 [cit. 2023-12-11]. Available at: https://www.martinfowler.com/bliki/AnemicDomainModel.html.

[8] Fowler, M. *CQRS* [online]. 14. july 2011 [cit. 2024-01-20]. Available at: https://martinfowler.com/bliki/CQRS.html.

[9] Fowler, M. *BoundedContext* [online]. 15. january 2015 [cit. 2023-12-10]. Available at: https://martinfowler.com/bliki/BoundedContext.html.

[10] Fowler, M. *MonolithFirst* [online]. 3. june 2015 [cit. 2023-12-11]. Available at: https://martinfowler.com/bliki/MonolithFirst.html.

[11] Hanselman, S. and Arcoverde, R. *Engineering Stack Overflow with Roberta Arcoverde* [podcast]. Scott Hanselman, 30. june 2022 [cit. 2024-01-24]. Available at: https://hanselminutes.com/847/engineering-stack-overflow-with-roberta-arcoverde.

[12] Jovanović, M. *Modular Monolith Communication Patterns* [online]. 5. august 2023 [cit. 2024-01-23]. Available at: https://www.milanjovanovic.tech/blog/modular-monolith-communication-patterns.

[13] Jovanović, M. *Modular Monolith Data Isolation* [online]. 9. december 2023 [cit. 2024-01-23]. Available at: https://www.milanjovanovic.tech/blog/modular-monolith-data-isolation.

[14] Llousas, A. *Designing DDD aggregates* [online]. 26. october 2022 [cit. 2024-01-27]. Available at: https://medium.com/@albert.llousas/designing-ddd-aggregates-db633f1caf88.

[15] Medhat, N. *Why Strategic Design Matters in Domain-Driven Design* [online]. 9. may 2023 [cit. 2024-01-27]. Available at: https://towardsdev.com/why-strategic-design-matters-in-domain-driven-design-5ea56b0ee219.

[16] Microsoft Development Team. *Entity Framework Core Documentation: Owned Entity Types* [online]. Microsoft Corporation, may 2023 [cit. 2024-05-02]. Available at: https://www.npgsql.org/efcore/modeling/concurrency.html.

[17] Npgsql Development Team. *Npgsql Entity Framework Core Provider Documentation: Concurrency Tokens* [online]. Npgsql, 2023 [cit. 2024-03-25]. Available at: https://www.npgsql.org/efcore/modeling/concurrency.html.

[18] Ozkaya, M. *CQRS Design Pattern in Microservices Architectures* [online]. 8. september 2021 [cit. 2024-01-20]. Available at: https://medium.com/design-microservices-architecture-with-patterns/cqrs-design-pattern-in-microservices-architectures-5d41e359768c.

[19] Ozkaya, M. *Outbox Pattern for Microservices Architectures* [online]. 8. september 2021 [cit. 2024-01-20]. Available at: https://medium.com/design-microservices-architecture-with-patterns/outbox-pattern-for-microservices-architectures-1b8648dfaa27.

[20] Robert C. Martin. *The Clean Architecture* [online]. 13. august 2012 [cit. 2023-12-12]. Available at: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html.

[21] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. 1st ed. Pearson, september 2017. ISBN 978-0-13-449416-6.

[22] Siraj ul Haq. *Introduction to Monolithic Architecture and MicroServices Architecture* [online]. 2. may 2018 [cit. 2023-11-26]. Available at: https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63.

[23] Smith, S. "ardalis". *Architecting Modern Web Applications with ASP.NET Core and Microsoft Azure* [ebook]. 7th ed. Microsoft Developer Division, 2023. Available at: https://raw.githubusercontent.com/dotnet-architecture/eBooks/main/archives/architecting-modern-web-apps-azure/Architecting-Modern-Web-Applications-with-ASP.NET-Core-and-Azure-v7.0.pdf.

[24] Vernon, V. *Effective Aggregate Design Part II: Making Aggregates Work Together* [online]. 1. october 2011 [cit. 2023-01-05]. Available at: https://www.dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_2.pdf.

[25] Vernon, V. *Implementing Domain-Driven Design.* 1st ed. Addison Wesley, february 2013. ISBN 978-0-321-83457-7.

[26] Vickers, A. *EF Core 8 RC1: Complex types as value objects* [online]. 12. september 2023 [cit. 2024-05-02]. Available at: https://devblogs.microsoft.com/dotnet/announcing-ef8-rc1/.

[27] Westeinde, K. *Deconstructing the Monolith: Designing Software that Maximizes Developer Productivity* [online]. 21. february 2019 [cit. 2024-01-26]. Available at: https://shopify.engineering/deconstructing-monolith-designing-software-maximizes-developer-productivity.

[28] Wlaschin, S. *Railway Oriented Programming* [online]. 14. march 2014 [cit. 2024-03-21]. Available at: https://fsharpforfunandprofit.com/rop/.

[29] Wlaschin, S. *Against Railway-Oriented Programming* [online]. 20. december 2019 [cit. 2024-03-21]. Available at: https://fsharpforfunandprofit.com/posts/against-railway-oriented-programming/.

Appendix A

Contents of the included storage media

```
/
├── Figures/ ........................ created vector figures and original pictures taken
├── Sources/
│   ├── CleanArchitecture/ ...................... clean architecture solution sources
│   ├── ModularMonolith/ ........................ modular monolith solution sources
│   └── Frontend/ ........................................ frontend solution sources
├── Thesis/ ............................... all thing necessary for building thesis pdf
├── thesis.pdf ........................................ this thesis in online format
├── README.md .............................. basic information about the repository
└── .gitmodules
```

Appendix B

Application Screenshots

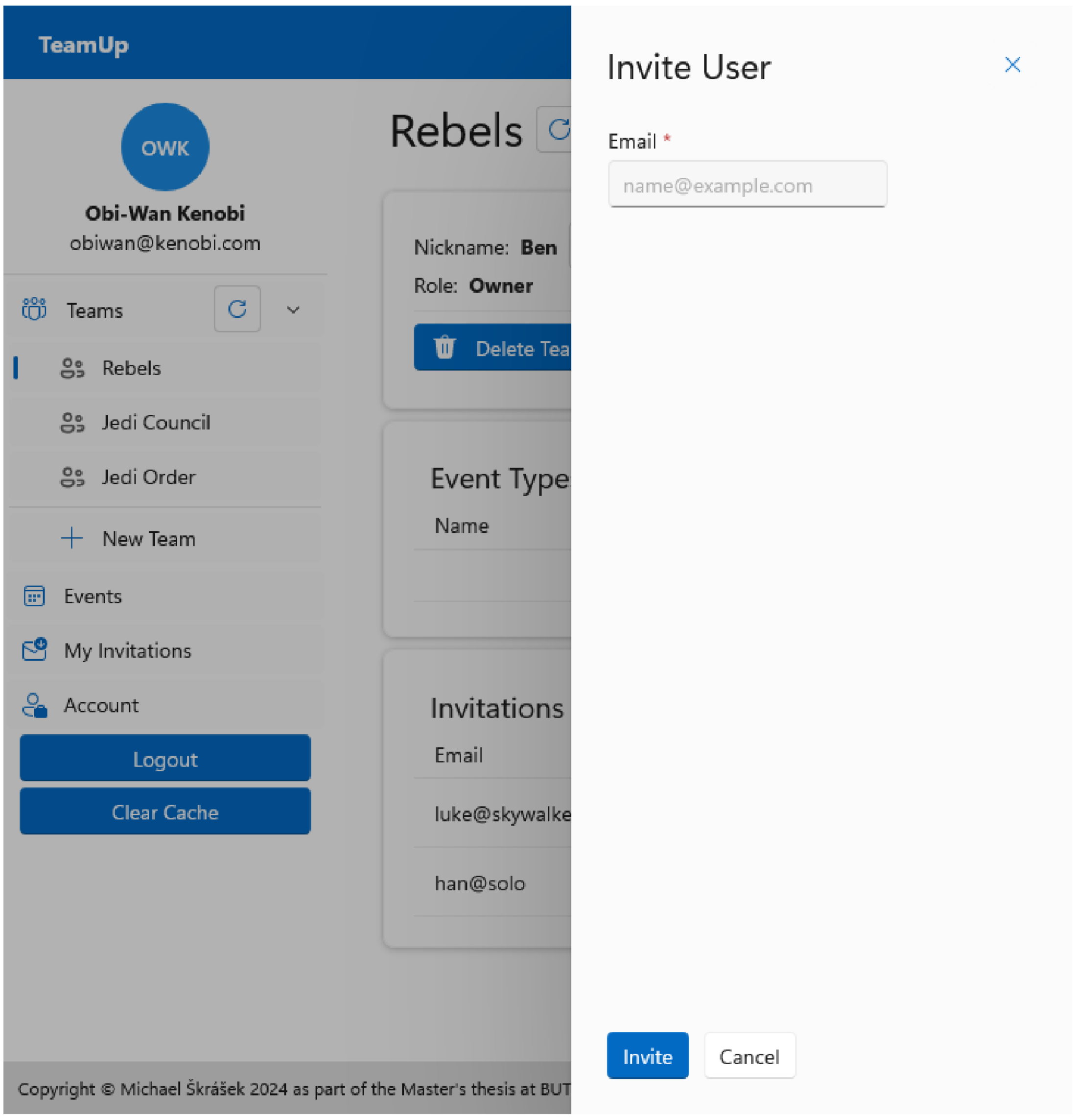
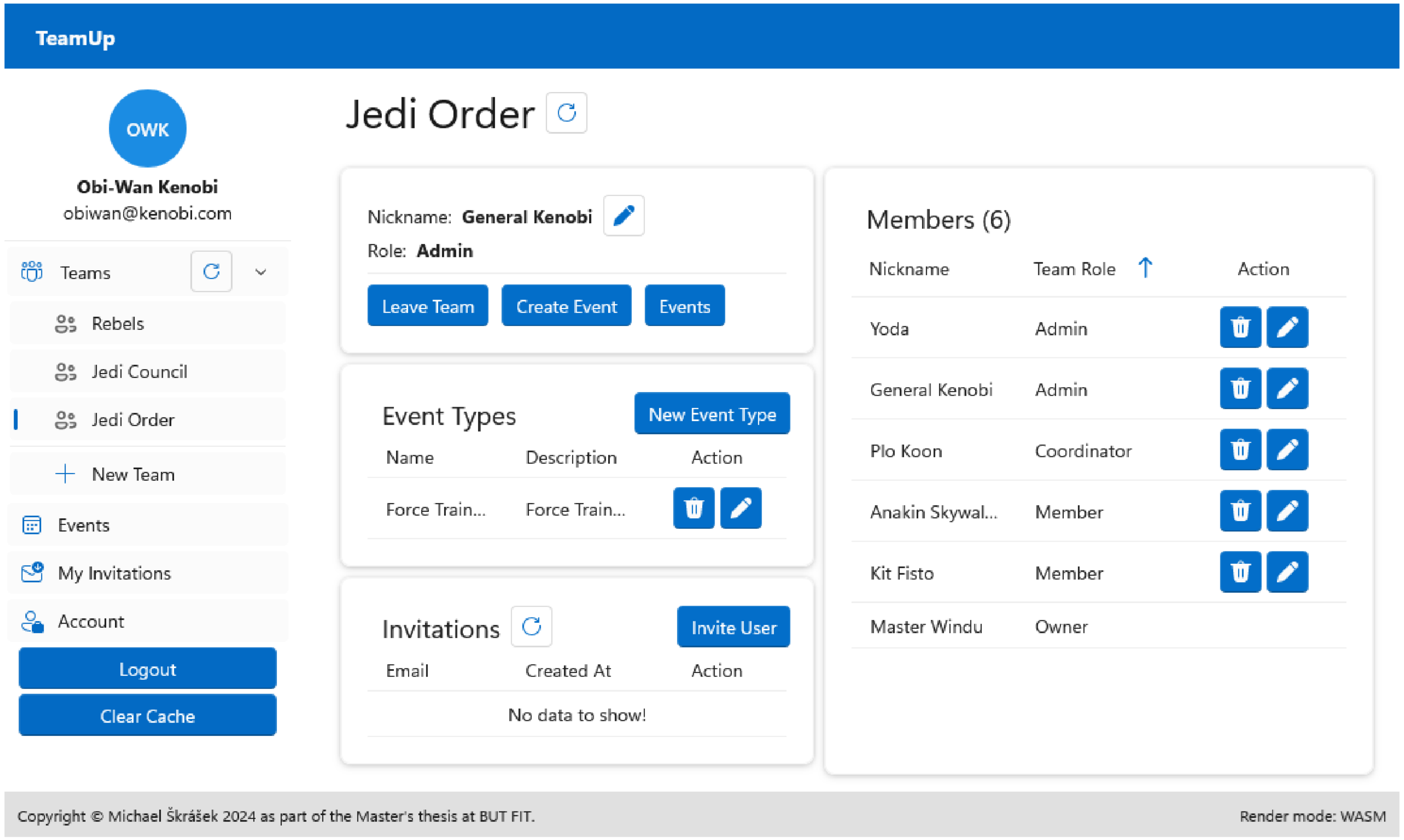


Figure B.1: Screenshot of the invite user prompt panel.



Obi-Wan Kenobi
obiwan@kenobi.com

Jedi Order

Nickname: **General Kenobi**
Role: **Admin**

Event Types

Name	Description	Action
Force Train...	Force Train...	

Invitations

Email	Created At	Action
No data to show!		

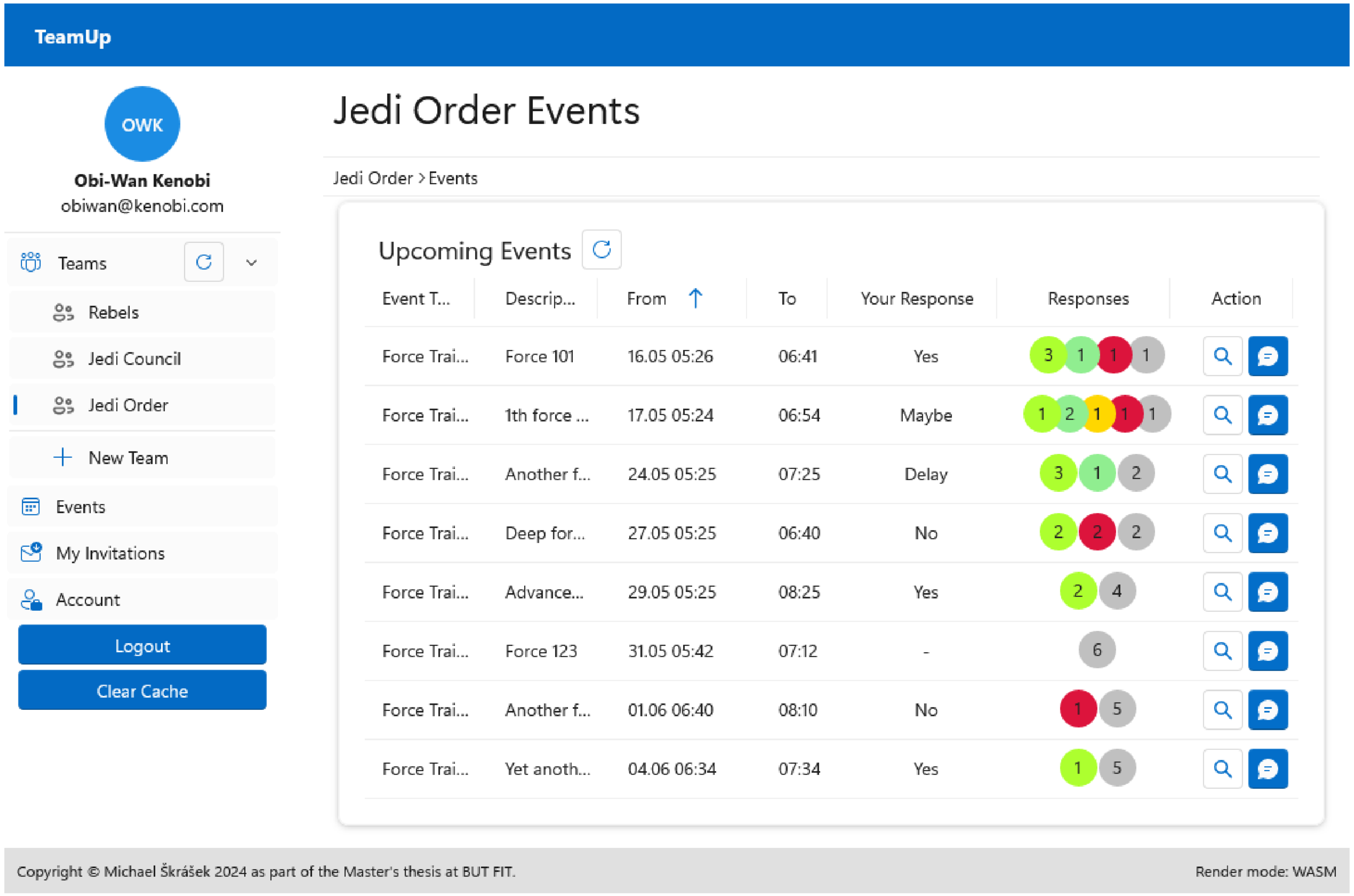
Members (6)

Nickname	Team Role	Action
Yoda	Admin	
General Kenobi	Admin	
Plo Koon	Coordinator	
Anakin Skywal...	Member	
Kit Fisto	Member	
Master Windu	Owner	

Copyright © Michael Škrášek 2024 as part of the Master's thesis at BUT FIT.

Render mode: WASM

Figure B.2: Screenshot of the team management page.



Obi-Wan Kenobi
obiwan@kenobi.com

Jedi Order Events

Jedi Order > Events

Upcoming Events

Event T...	Descrip...	From ↑	To	Your Response	Responses	Action
Force Trai...	Force 101	16.05 05:26	06:41	Yes	3 1 1 1	🔍 💬
Force Trai...	1th force ...	17.05 05:24	06:54	Maybe	1 2 1 1 1	🔍 💬
Force Trai...	Another f...	24.05 05:25	07:25	Delay	3 1 2	🔍 💬
Force Trai...	Deep for...	27.05 05:25	06:40	No	2 2 2	🔍 💬
Force Trai...	Advance...	29.05 05:25	08:25	Yes	2 4	🔍 💬
Force Trai...	Force 123	31.05 05:42	07:12	-	6	🔍 💬
Force Trai...	Another f...	01.06 06:40	08:10	No	1 5	🔍 💬
Force Trai...	Yet anoth...	04.06 06:34	07:34	Yes	1 5	🔍 💬

Copyright © Michael Škrášek 2024 as part of the Master's thesis at BUT FIT.

Render mode: WASM

Figure B.3: Screenshot of the team events page.