FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Migration of Atlantis inventory system's PHP backend to C#, including architectural considerations
Student:	Bc. Duc Minh Pham
Supervisor:	Ing. Jiří Hunka
Study program:	Informatics
Branch / specialization:	Web Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2024/2025

Instructions

The objective of the project, in collaboration with colleague Bc. Max Hejda, is to migrate the existing functional backend of the Atlantis warehouse system, which is in PHP, to C# using the .NET framework. Due to the extensive scope and to ensure isolation of the task, the focus of this work is on the architecture, technical setup of the project and the main technological concepts aimed at a smooth and seamless transition between the mentioned technologies, which brings a significant amount of challenges. A secondary goal involves a part of the actual implementation.

1. Analyze the backend solution, including previous works on the Atlantis warehouse system. Also analyze the technologies currently used, including the capabilities of the C# language (.NET). Further, analyze the possibilities and consequences of migrating between languages in such a large project.
2. Based on analysis, together with Bc. Max Hejda, design a suitable backend solution and ideal migration procedures.
3. Properly consult the proposals with the instructor and the developers of the Atlantis warehouse system.
4. Implement a usable part of the backend using the procedures you have designed.
5. Design and execute suitable testing methods for the backend portion of your implementation.
6. Test the final implementation either in a production or a testing environment.
7. Evaluate the final solution and suggest future modifications.

Electronically approved by Ing. Jaroslav Kuchař, Ph.D. on 13 February 2024 in Prague.

Master’s thesis

Migration of Atlantis inventory system’s PHP backend to C#

Bc. Pham Minh Duc

Department of Software Engineering
Supervisor: Ing. Jiří Hunka

May 8, 2024

Acknowledgements

I would like to thank Ing. Jiří Hunka for his guidance during the creation of this thesis, my thanks also go to my family, who supported me during my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 8, 2024 …………………

Czech Technical University in Prague
Faculty of Information Technology
© 2024 Minh Duc Pham. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Pham, Minh Duc. *Migration of Atlantis inventory system's PHP backend to C#*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Abstrakt

Projekt Atlantis je skladový systém původně vyvinutý v PHP. Systém zahrnuje frontend, backend, databázi a další doprovodné komponenty, které více než 3 roky spravuje společnost Jagu s.r.o. Systém je úspěšný, ovšem setkává se s problémy týkajicí se rozšiřitelnosti, výkonu a udržitelnosti kvůli výběru platformy. Tato diplomová práce se bude zabývat přechodem projektu Atlantis z PHP do jazyka C# na platformě .NET. Hlavním cílem je pečlivě naplánovat a provézt částečnou migraci backendového systému, přičemž zajistit kompatibilitu s původním systémem, jeho frontendem a databází, a to bez narušení jejich funkcionality. To zahrnuje analýzu současného systému a úzkou spolupráci s členy týmu. Zaměření této práce je technický setup projektu, architektura a technologické myšlenky zaměřeny na přechod mezi zmíněnými technologiemi. Vzhledem k rozsahu práce nebude práce pokrývat kompletní implementaci celého backendu. Výsledkem bude pevný základ, na kterém bude migrace prováděna, včetně strategií a nástrojů potřebných pro process migrace. Přínosy nové architektury byly demonstrovány prostřednictvím částečné implementace, která byla spuštěna na prostředí podobném produkčnímu.

Klíčová slova .NET, Čistá architektura, CQRS, DDD, Strangler Fig pattern, UML, Návrhové vzory

Abstract

The Atlantis project is a warehouse inventory system originally built in PHP, the system includes a frontend, backend, database and other accompanying components maintained by Jagu s.r.o for over 3 years. While the system has been successful, it has faced some extensibility, performance, and maintainability challenges due to the platform choice. This thesis discusses the transition of Atlantis from PHP to C# on the .NET platform, aiming to take advantage of the enterprise features of .NET. The main goal is to carefully plan and carry out the migration of the backend system, making sure it can still work with the old PHP system to avoid disrupting current functions. This involves analyzing the current system and working closely with team members. The focus of the thesis is the technical setup of the project, architecture and technical knowledge to support a seamless transition between the mentioned technologies. Due to the sheer scale of the system, it will not cover the complete implementation of the entire backend. This thesis will ensure that the new backend can still communicate with the existing components like the frontend and database without issues. The final outcome will be a foundation upon which the migration will be carried out, including strategies and tools needed for the process. We demonstrated the benefits of the new architecture through partial implementation which was showcased on a production-like environment.

Keywords .NET, Clean architecture, CQRS, DDD, Strangler Fig pattern, UML, Design patterns

Contents

Introduction 1

An important note - NDA 3

I Analysis 5

1 Analysis of Atlantis 7
1.1 Qualitative and Quantitative research approaches 7
1.1.1 Phases of analysis 8
1.2 Analysis of previous works - Ing. Pavel Kovář 10
1.2.1 System architecture - Authorization 10
1.2.2 Used technologies 12
1.3 Analysis of previous works - Ing. Oldřich Malec 14
1.3.1 Existing system 14
1.3.2 Used technologies 14
1.4 API Categories 15
1.5 Code observations 22
1.5.1 Cross-cutting concerns 24

2 Symfony vs .NET 27
2.1 Troubles with the current system 27
2.2 Previous efforts 27
2.3 Performance 28
2.3.1 Symfony (PHP) 28
2.3.2 .NET (C#) 29
2.4 Compatibility 30
2.5 Conclusion 31

3 Migration strategies 33
3.1 Full migration 33
3.2 Phased migration 34
3.2.1 Strangler Fig Pattern 34
3.2.2 A façade as the interceptor 34

3.2.3 The phases of Strangler fig pattern 35
3.2.4 Migration by features 37
3.2.5 Migration by blocks 37
3.3 Phased migration - database 38
3.3.1 New database 38
3.3.2 Backwards-compatible database 38
3.4 Conclusion 39

II Design **41**

4 Architecture **43**
4.1 Why Architecture matters 43
4.2 Current architecture 44
4.3 Enhanced architecture 45
4.3.1 SOA/Microservices 45
4.3.2 Monolith 46
4.3.2.1 Selecting the architecture 47
4.4 Clean Architecture 48
4.4.1 Understanding Clean Architecture 49
4.5 CQRS and Domain Driven Design (DDD) 51
4.5.1 Final note on system design 52
4.6 30,000-foot Overview 53
4.6.1 Naming the new system - Squid 53
4.7 10,000-foot Overview - Strangler façade 54
4.7.1 Key factors 56
4.8 10,000-foot Overview - Squid API 57

III Implementation **59**

5 Implementation - Squid API **61**
5.1 Motivation 61
5.1.1 Cooperation with Max 62
5.2 Clean architecture in .NET 62
5.3 CQRS and DDD implementation 63
5.4 Validation 65
5.5 Database 67
5.6 Authentication and Authorization 67
5.6.1 Porting Octopus Authentication and Authorization . . . 68
5.6.2 Permission-based Authentication and Authorization . . 69
5.6.3 API Documentation - Swashbuckle library 73
5.6.4 Logging 73

6 Implementation - Strangler façade **75**
6.1 Modes of the Strangler façade 77
6.1.1 Using Octopus mode 79
6.1.2 Using Comparison mode 80
6.1.3 Using Squid mode 81
6.2 Implementation details - Integration issues 83

IV Testing **85**

7 Testing **87**
7.1 Manual testing 87
7.2 Automated testing 88
7.2.1 Basics of automated testing 88
7.2.2 Unit testing 88
7.2.3 Integration testing with throwaway Docker containers . 89
7.3 Resetting of database between tests 91
7.4 Seeding database - for testing 92
7.5 Snapshot testing 94
7.6 Tying it all together 96
7.7 Test Driven Development - TDD 98
7.7.1 Summary 99
7.8 E2E Testing 99
7.9 Developer feedback 99
7.9.1 Feedback Conclusion 103

8 Conclusion **105**

Bibliography **107**

A Acronyms **113**

B Contents of enclosed SD-Card **115**

List of Figures

1.1 Introspection authentication flow 11
1.2 Swordfish - current state 15
1.3 Decision tree - classification 16
1.4 UML Diagram Centered on Supplier 17
1.5 Activity diagram of External Order POST endpoint 18
1.6 Activity diagram of Move Products POST endpoint 19
1.7 Activity diagram of Location Transfer POST endpoint 20

3.1 Strangler Façade migration phases [1] 35

4.1 Application layer - Service layer 45
4.2 Microservices[2] 47
4.3 Monolith[3] 48
4.4 Clean architecture[4] 50
4.5 DDD - Bounded context[5] 52
4.6 High-level overview of the Octopus system 53
4.7 Squid system - draft 54
4.8 High-level overview of the Squid system 55
4.9 Strangler façade modes 56
4.10 Squid Api - Clean architecture 58

5.1 Squid - Implementation Dependency structure 63
5.2 CQRS basic approach 64
5.3 CQRS with Mediator approach 64
5.4 Squid - Implementation ”thin” controllers 64
5.5 Validation pipeline behavior 66
5.6 Authorized back-channel httpclient 69
5.7 Keycloak - Permission policy 70
5.8 Keycloak audience validation 72

6.1 Middleware - order of execution [6] 75
6.2 Middleware structure [6] 76
6.3 Middleware - Strategy selector 77
6.4 Middleware - Strategy selector sequence diagram 78
6.5 Strangler - Octopus mode 79

6.6 Strangler - Grafana Dashboard 82
6.7 Integration - Authorization headers 84

7.1 Test Containers usage 90
7.2 Conflict of tests - Database state 92
7.3 Snapshot testing - snapshot example 96
7.4 Snapshot testing - Diff window 97
7.5 E2E testing - Production clone environment 99

List of Tables

1.1 REST Endpoints for Authorization Token Introspection Flow . . . 12

3.1 Criteria for block/feature Extraction [7] 36

4.1 Comparison of Monolithic and Microservices Architectures 48

7.1 Simple CRUD migration estimate 101

List of Code listings

5.1 Role-based authorization using Angler 69
5.2 Permission-based authorization using Keycloak 71
6.1 Traditional Assert Example 78
6.2 Strangler Configuration - Time window 82
6.3 Strangler Configuration - Percentage 83
6.4 Strangler Configuration - User 83
6.5 Strangler Configuration - Enabled 83
7.1 Traditional Assert Example 95
7.2 Assert with Verify example 95
7.3 Sample test - improved 96
7.4 Sample test - fully optimized 97

Introduction

The project in focus of this thesis is the Atlantis project. Atlantis is a warehouse management application used by companies across the Czech Republic. It implements end-to-end processes for many concurrent actors in a small to mid-sized warehouse.

Atlantis is a culmination of several components, including the backend, frontend and database system. Which have been maintained and developed for over three years by the owning company - Jagu.

Beginnings

The project has been in use by paying clients since its creation and serves them as a critical part of the infrastructure. The beginnings of the project were defined as part of Ing. Pavel Kovař's Master's thesis [8]. The outputs of which were very close to being finalized and usable as a stand-alone warehouse management system. The User interface (UI) was delivered by Ing. Oldřich Malec in his Master's thesis [9]

Background

Developed using PHP, the system has been serving well. It was adapted to work with the domain requirements and served as a stable platform for the initial phases of the application lifetime. After some time the requirements started shifting towards features for which PHP was no longer the optimal choice. It struggles with scalability, maint-inability and performance in complex high-load business processes.

Thus, a decision was made to migrate the Atlantis warehouse system to a more optimized and robust platform: C# on the .NET platform. Such a decision was not made lightly as it is not only a change of programming language but rather a long-term strategic choice for the team, whose composition highly favors the .NET ecosystem with experience in many different projects. Further advantages and disadvantages will be discussed in chapter 2.

The problem

The Atlantis warehouse system has been reliable, but the slowly growing complexity has made the project hard to maintain and even harder to update the codebase. Key issues being the difficulty of working with database entities, their mapping to objects, and general reply latency. Unfortunately, switching to the C# platform is not instant, and despite adequate and deep analysis, it is impossible to be perfect.

A critical part of the development of the application will be its backward compatibility with the PHP system. As we slowly migrate parts of the system, it is crucial to maintain the original processes and logic, allowing parallel functionality with the original warehouse system. This requirement ensures uninterrupted service, safety, and integrity during the migration period.

Project goals

The most important goal of this thesis is to carefully plan and implement the transition of the Atlantis warehouse system backend from PHP to C#. Part of this will be a deep analysis of the current system, working with colleagues from the team who will be adopting this project, and the implementation itself using .NET framework to rebuild the architecture, implement critical solution infrastructure, and provide a rich and seamless experience for future developers.

Part of the goal is to implement a part of the system to demonstrate the functionality and advantages of the new architecture. This thesis will aim to provide a strategies for migration, as well as ideas, and tools to be used in the process.

Scope

This thesis focuses on the architectural redesign and technical setup for the transition of the Atlantis system backend to C#. While it includes implementing a portion of the system, it does not cover the complete reimplementation of the entire backend.

We will be analyzing strictly the backend system, leaving out the frontend and other user interfaces. Precautions ensuring interface compatibility will be in scope of the backwards compatibility aspect of the solution.

An important note - NDA

This thesis is accompanying work done for Jagu s.r.o. for which I have signed a Non-Disclosure Agreement (NDA). I will be more general and objective while describing problems during the project and the roadblocks ahead. The code produced will not be available in the attachments, rather it will be accessible only through the internal company source code management system. This approach safeguards the company's intellectual property while allowing me to discuss the theoretical and methodological aspects of my work.

Part I

Analysis

CHAPTER 1

Analysis of Atlantis

The Atlantis system, developed in PHP was initially made to improve on the ”Sysel” inventory system. Overall it has succeeded in bringing many new features which has aided its growth and its marketability, serving as an all-in-one package for the whole process management of any small to mid-sized company.

Atlantis does not only serve as an inventory management platform, but it handles many parts of merchandise lifetime.

1.1 Qualitative and Quantitative research approaches

During the analysis of the Atlantis project I have discussed many topics with many of the engineers working on the project. I have employed mostly Qualitative methods of interview. Let us take a look at what these methods are and how to approach these interviews effectively. Inspiration was taken from the article ”Guidelines for conducting and reporting case study research in software engineering” by Per Runeson and Martin Höst [10].

The article suggests a methodological approach to the analysis by first defining research questions:

- What are the current challenges faced by the Atlantis project?
- What can we do to make this project successful and create an impact?

Methods of data collection

1. Interviews: The interviews are conducted in semi-structured manner with preset agenda and timeframe.

2. Analysis of documents: Read and analyse project documentation, previous project works and the code base
3. Observations: Observation of meetings, company priorities and common daily issues

Out of these three, observations were not sufficiently reached due to not receiving permission to join the main company communication channels. This has led to communication issues with many of the team members. Nonetheless, interviews were conducted with the most active current members of the project - the main stakeholders of this project, which were in no particular order: The CEO of the company, the Atlantis project team lead, the project mid-level engineer and junior engineer.

Quantitative

Quantitative analysis is usually statistical and requires some way to quantify data to find patterns and relationships. This type of analysis is used when data can be numerically measured and is structured. This can be, for example, system performance, which can be properly quantified. Quantitative analysis helps in validating findings from qualitative analysis by providing statistical evidence.

Qualitative

In the qualitative analysis, the main goal is to find patterns, themes and meanings from data collected. This involves processing data collected from interviews, personal notes and other sources like the codebase. This type of analysis requires the analysts to immerse themselves into the problem domain to understand the various contexts and decisions that were made.

1.1.1 Phases of analysis

Phase 1: Ensuring system compatibility

The first phase was following a qualitative approach, which involved a deep analysis of the existing Atlantis system's codebase. This step is extremely important to understand the various architectural decisions, dependencies and potential challenges that might affect the migration to a new architecture. The analysis here was mostly document-based, it involved understanding system documentation, code comments and the structure of the code. This analysis helps to make sure that the new architecture would be compatible with the existing system, this was a fundamental step in the process of creating a new architectural design.

Phase 2: Team meeting to propose new Architecture

After the first phase a prototype of the architecture was drafted, and a semi-structured team meeting was conducted to propose the new architecture and to collect feedback from all the team members. Several concerns were raised as the result of this meeting:

1. Maintainability: Will this architecture be more maintainable than the one implemented in Octopus.
2. Phased migration by features or blocks: The system will need to be deployed in parallel with Octopus, as there are insufficient resources to fully migrate to the new architecture.
3. Permission-based authorization: The system must support an authorization system which is more granular than the one which is implemented in Octopus.

These issues are discussed in detail as part of this thesis, including the decision-making process, alternatives, and final implementation. This phase was extremely important for addressing any issues before implementing the new architecture.

Phase 3: Interviews with the sources of Concerns

Phase 3 involved conducting detailed one-on-one interviews with the team members who had raised concerns from the previous phase. In these interviews, I have proposed ideas or implemented prototypes on the new architecture as solutions to each of the raised concerns.

- Permission-based authorization: An interview was held with the team member raising the concern. The results of the meeting are discussed in subsection 5.6.2
- Maintainability: There was a one-to-one interview with the lead developer regarding the direction of the architecture, but this will mostly be the focus of Phase 4, after the implementation is mostly finished
- Phased migration by features or blocks: One-to-one interview was conducted with the corresponding team member

Phase 4: Developer feedback on the new architecture

In the final phase the architecture was tested and deployed. This involved final feedback for the outputs of this thesis. I will be discussing the results of this phase at the end of the thesis in section 7.9.

Each of these phases contributed to the project analysis, ensuring that the final architectural plan is carefully considered and inclusive of input from the various members of the team. This phased approach makes sure that the project will progress with the acknowledgment of the team, in a direction that is expected.

1.2 Analysis of previous works - Ing. Pavel Kovář

Ing. Pavel Kovář's work [8] set the grounds for the Atlantis project. Pavel's work included much of the domain design, which has been implemented either as part of his work or later after the thesis concluded. I will not be going too deep into the domain-specific language and usage, as that is not the focus of this thesis. For the meaning of some of these domain terms, please refer to Pavel's thesis. Architectural design and the component interactions within the technology stack are what we wish to gather in this chapter.

Domain model

The domain model has since changed and been expanded upon by many developers. Therefore it would not be benefitial to take Pavel's thesis as a complete source of truth in this department. I will summarize the used domain structure and architectural characteristics in chapter 4.

1.2.1 System architecture - Authorization

The system is designed to be modular and reusable. It consists of the mobile/web interface, authorization server, and the main backend part of the system. The components communicate via REST API. In the same spirit of modularity, the PHP framework chosen is Symfony [11]. Symfony is a modular PHP framework that defines sets of reusable open-source components available through Composer[12].

This part is dedicated to the design of the system's authorization and authentication components. The migration of the authorization server itself to C# is not in the scope of this thesis. However, its integration into the new system's authentication and authorization flow is critical and unavoidable. Alternatives will be discussed as part of section 5.6.

Pavel goes deep into the technical aspects of OAuth 2.0 [13] protocol. For a more detailed look please check the thesis [8]. I will try to analyze and summarize the relevant information for this project.

The Authorization server was implemented and was code-named *"Angler"* by which I will refer to it from now on. Angler has two interfaces:

1. User interface (UI)

2. Application Programming Interface (API)

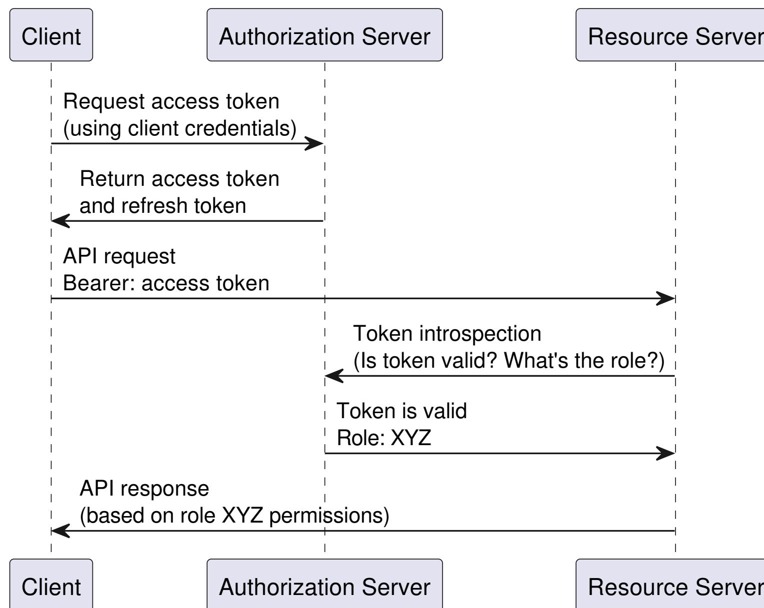

Figure 1.1: Introspection authentication flow

We will be focusing on the API implementation. Upon further analysis of the code of Octopus the interactions are described with the diagram* 1.1.

Interestingly JWT tokens are used, but not as one would expect. Usually, JWT tokens are signed by the Authorization server using a secret key. The clients (the resource servers) should be able to verify the signature by using the Authorization server's public key and a matching algorithm.

However, here JWT tokens carry the role information, but cannot be trusted as the signature cannot be verified. Therefore an introspection endpoint (listed at Table 1.1) is used which will provide trusted claims from the authorization server directly. This can become a bottleneck. How to improve this will be discussed in section 5.6.

*This is from the view of the backend or the "Resource server" (from OAuth terminology) we do not distinguish by the origin of the token

Table 1.1: REST Endpoints for Authorization Token Introspection Flow

Endpoint	**Expected Response**	**Description**
`/oauth/token` Body: `grant_type`, `client_id`, `client_secret`	Access token and refresh token.	Obtain access token using client credentials.
`/oauth/introspect` Body: `token` (access token to introspect)	JSON containing token validity, role claims, and other metadata.	Introspection endpoint to validate access token and get role information.

There are several roles the users of the system are categorized into:

- ROLE_STOREKEEPER
- ROLE_CHIEF
- ROLE_CUSTOMER

Since then, several others have been added:

- ROLE_PACKER
- ROLE_ORGANIZER
- ROLE_ESHOP
- ROLE_EXTERNAL_HAMSTER

One of the additional requirements expressed by the team was the future compatibility with a more granular permission-based authorization provider, like Keycloak.

1.2.2 Used technologies

Pavel has taken care to choose languages that fit the requirements, his experience and technology stack used at Jagu s.r.o at the time. Here's an analysis of the technologies used:

PHP

PHP[14] was chosen as the primary language for developing of the backend logic. It has great compatibility on many platforms. PHP has been chosen here due to its simplicity and speed of prototyping.

Symfony

Symfony, is an open-source framework for PHP. It consists of many small, reusable components. It has a very active and large community of volunteer contributors.

PHP and Symfony are the language and framework of choice for this project, its advantages and disadvantages are later discussed in chapter 2

PostgreSQL

An open-source object-relational database system, PostgreSQL[15] is one of the most popular database engines. It has stood the test of time on high-traffic websites, and ensures users can access data without conflicts and data loss.

Doctrine ORM

Doctrine is an Object Relational Mapping (ORM) for PHP. It allows object-relational mapping in PHP, by utilizing its proprietary database abstraction layer (DBAL) to create a unified API for interaction with the database regardless of the specific system (MySQL, PostgreSQL, SQLite, etc...)

Redis

Is used for caching and session management, Redis[16] optimizes the system performance by reducing load on the database.

Docker

Used for containerization, Docker[17] simplifies deployment and while ensuring consistency allows the scaling of the application across different environments.

Deployment - CI/CD

The thesis provides details of the deployment and testing procedures to ensure reliability and stability of the system. Automated deployment tools like Gitlab CI/CD is used to automatically build and test the solution.

Nginx

Nginx[18] is chosen as the webserver/reverse proxy, it performs well and is very stable.

1.3 Analysis of previous works - Ing. Oldřich Malec

While my thesis does not focus on the frontend part of the application, a user-facing interface is a vital part of the application and one of the main consumers of the backend in the tech stack.

Ing. Oldřich Malec's work[9] aimed at analyzing, designing and prototyping a solution for the frontend of Atlantis. The thesis was done in coordination with Ing. Pavel Kovář's work and resulted in a front-end system used and developed to this day, code-named *"Swordfish"*

As Swordfish is not to be rewritten to C#, it is very important for the new system to be compatible with this frontend application. For this integration, proper technical analysis and testing is required. We shall continue with the analysis part here.

1.3.1 Existing system

The user is greeted with the main dashboard of the system as seen on Figure 1.2. The interface is clean, reactive and intuitive. It utilizes modern practices, both in design with the application of material design and technologically. Oldřich did a comprehensive analysis on the subject of selecting the framework of choice and has chosen Vue.js. As I am not experienced with frontend development, I cannot comment on the strategic impact of choosing Vue.js and its viability in the future, but as one of the selected criteria for selection of the framework over the others: Github stars doubled since 2018[19], and the project is receiving continual updates, Vue.js has shown great potential.

The application uses HTTP REST API calls to interact with the backend. Authentication and security are done via Angler, which utilizes the UI part of the authorization server, yielding an authentication token of the user at the end of this interaction. This token is then used in all of the interactions within the session.

The thesis mentions some performance issues during the development, which required special precautions for the proper performance of the system. Caching is therefore heavily utilized.

1.3.2 Used technologies

Vue.js

Vue.js, along with Webpack are the main technologies used in the implementation of the frontend. Vue has RESTful API design, meaning it interacts with the backend mainly using Representational State Transfer (REST) calls. This requires the backend to be stateless and cacheable while adhering to the usual set of standard REST principles.

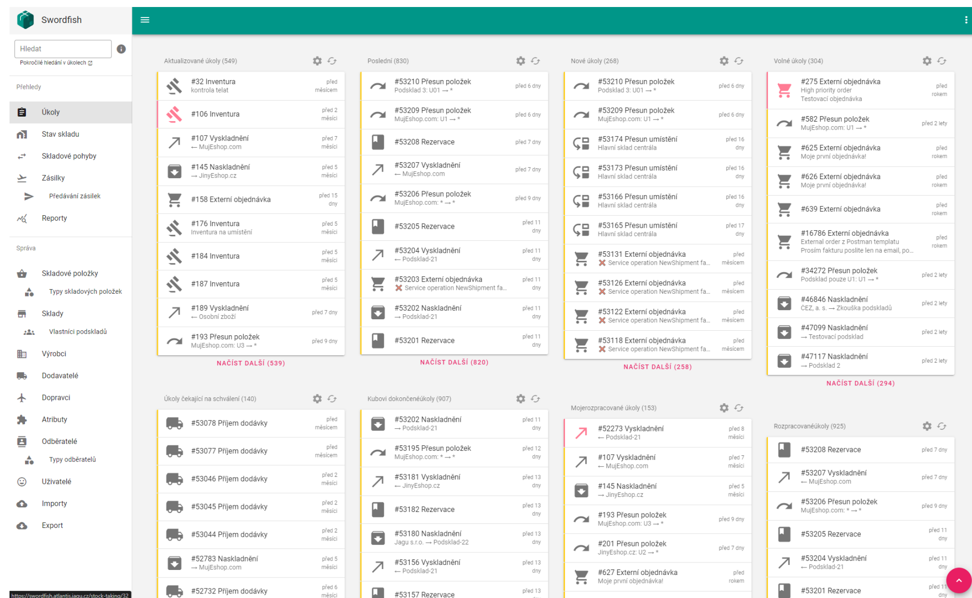

Figure 1.2: Swordfish - current state

JSON is the main data format for communication between the fronend and the backend. Serializing and deserializing from JSON needs to be standardized between these two components.

Authentication and authorization is handled using JSON Web Tokens (JWT) these tokens are retrieved from an OAuth server. This interaction must be supported by the backend to correctly validate the given tokens.

1.4 API Categories

It is important to note that since the initial work of Pavel and Oldřich, the system has grown drastically, over ten times the size in the span of the application by estimates of a team member. Atlantis - Octopus now has hundreds of endpoints (366 at the time of writing). In this section, we will be decomposing and analyzing a subset of these endpoints and classifying them based on these criteria 1.3. To respect the agreement with the company, I will intentionally not go into too much detail with the business processes. I will describe the endpoints from a purely technical perspective.

OpenAPI endpoint documentation has been used as a high-value source for cohesive blocks or microservices in the case of the work by[20], though I will not be using tool-based approaches as discussed in the article. To compensate, I have consulted these findings with the current lead developer for the Atlantis project.

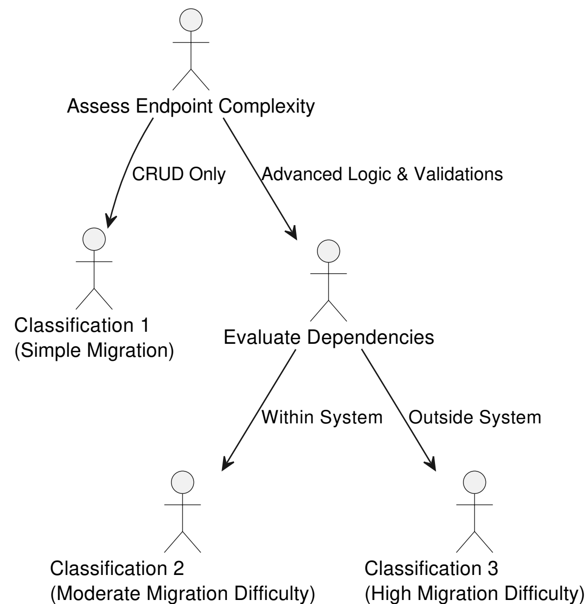

Figure 1.3: Decision tree - classification

Manufacturer

Classification 1 - Create, read, update and delete (CRUD). A simple, straightforward implementation. This endpoint contains basic operations. And can be used as a first implementation example.

Stock

Classification 1 - CRUD. It has blameable trait, and contains many validations.

Supplier

Classification 1 - CRUD. Has blameable trait and many validations. Can be seen on Figure 1.4.

Delivery accept task

Classification 1 - One of the simpler concrete task implementations. Task type: DELIVERY_ACCEPT. Standard Task management, but only approve and reject.

External order task

Classification 3 - Get endpoints are straight forward. Task type: EXTERNAL_ORDER. Post endpoint is very complex. Includes creation and validation of aggregate addresses for delivery address from buyer contact. Carrier

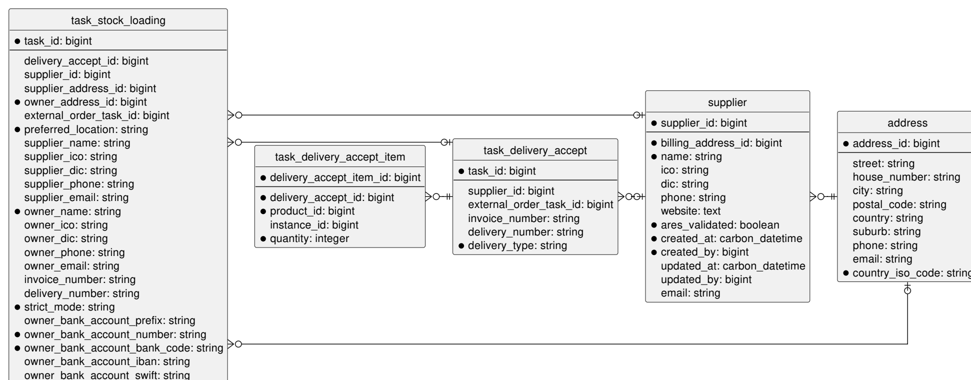

Figure 1.4: UML Diagram Centered on Supplier

service properties validation, other properties initialization, order items validation. Items are reserved, and message sent on message bus. Can be seen on Figure 1.5.

Location transfer task

Classification 2 - Standard task handling, task type LOCATION_TRANSFER. Move from source to destination is complex multi-step process with heavy validation.

Move products task

Classification 2 - Moving of products from inside one warehouse/subwarehouse to another location within that warehouse/subwarehouse. Transfer modes: Free, NoExtra, Exact. A simplified diagram can be seen on Figure 1.6

Move products task Items

Classification 3 - Moving of items within the move products task, very complex with heavy validation. Can be seen on Figure 1.7.

Stock picking task

Classification 3 - Task type: STOCK_PICKING. Has capability to merge two different stock picking tasks, Creates shipments if external order. Message bus utilized to communicate context to shipment handlers. Reservations created for items in scope.

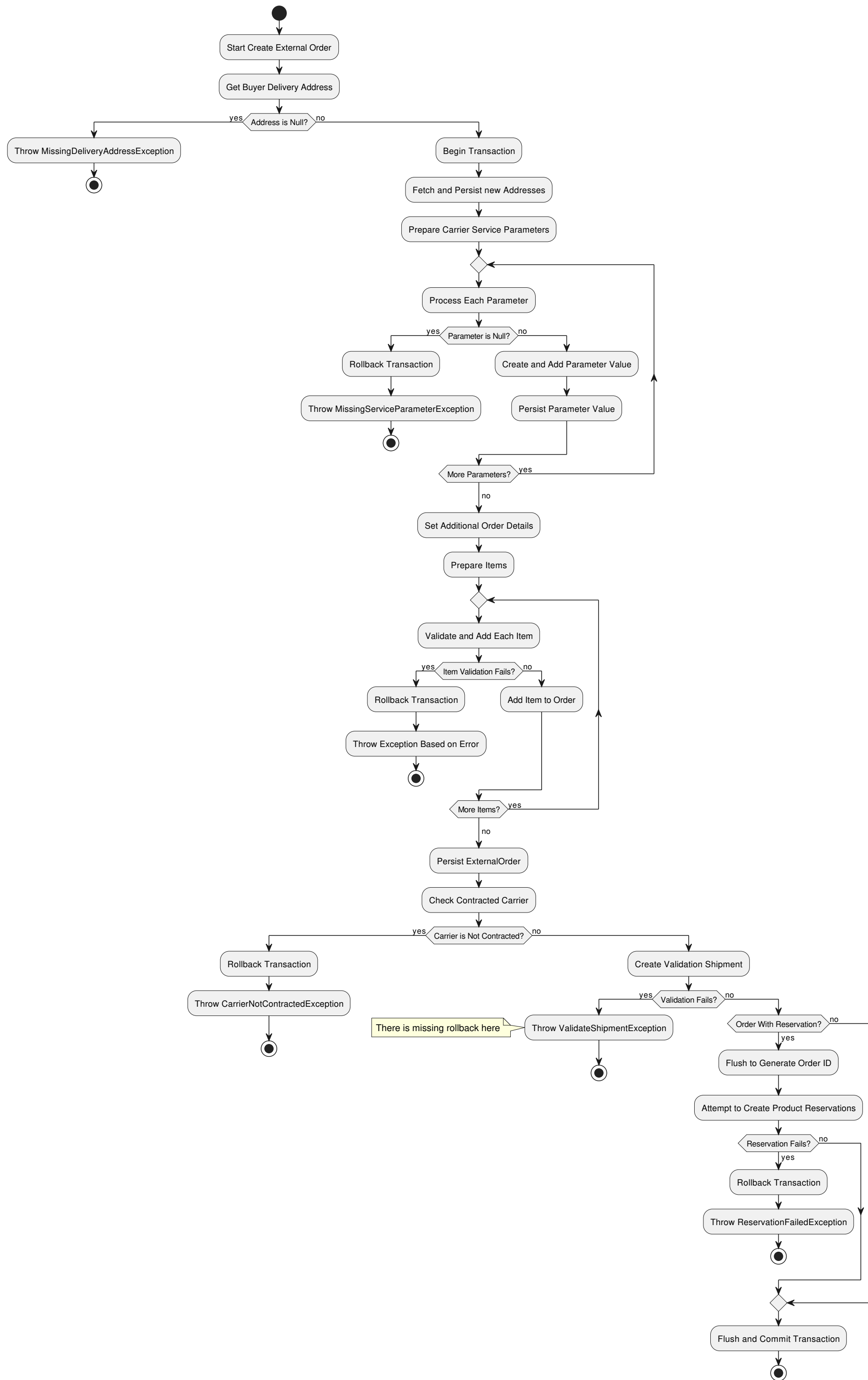

Figure 1.5: Activity diagram of External Order POST endpoint

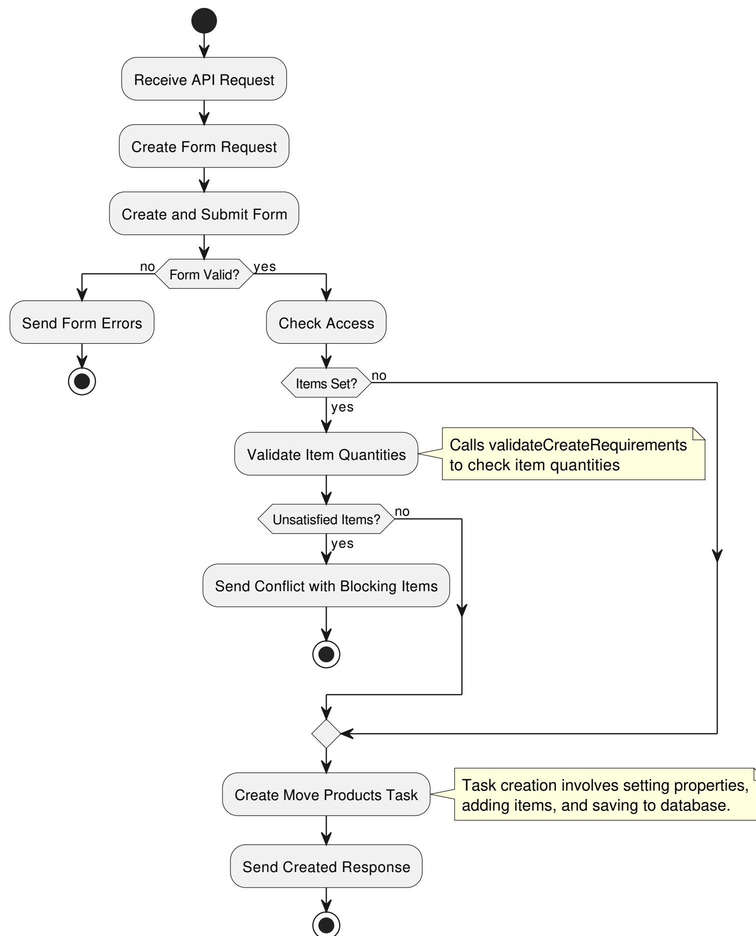

Figure 1.6: Activity diagram of Move Products POST endpoint

Stock picking task set items

Classification 3 - Complex multi-step processes with the moving of items, many state checks, and permission validations. Global item locking, Moving to temporary locations.

Stock taking task items

Classification 1 - CRUD with heavy validations. Permission validation, Form validation, Duplicate Serial validation, Location validation, duplicate validation.

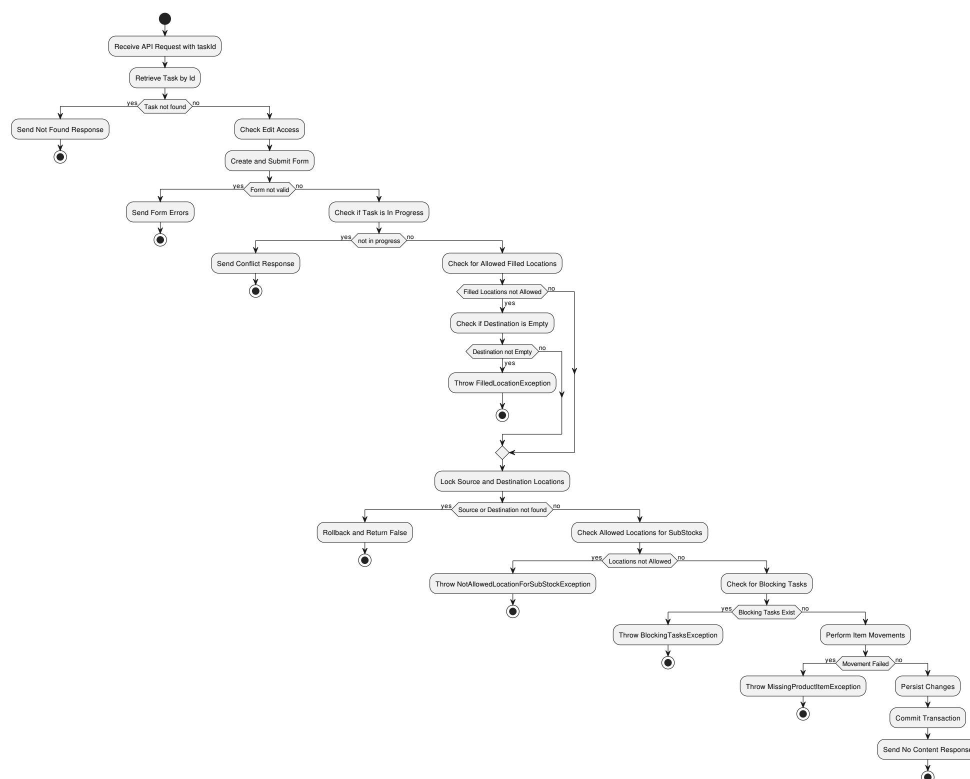

Figure 1.7: Activity diagram of Location Transfer POST endpoint

Subordinate stock transfer task

Task type: STOCK_TRANSFER. Classification 2 - Classic task implementation management of task state. Creation of Substock Transfer creates reservations for each of the transferred items. Reservations lower the perceived quantity.

Subordinate stock transfer task Items

Classification 3 - There are many moving parts, transaction begins, item is locked, decreased from source, increased in target, and validations scattered across whole process. At the end is there check whether items are blocked by different stock picking tasks.

This process is very complex, with each step there are failure states, short-circuiting the process. Global item locking

Task attachments

Classification 2 - CRUD for file attachments. Handling of files uploaded by users is required. Attachments can be images, csv files, pdf files. Files retain

name given from user. These files are to be handled with utmost diligence, file safety is a very important and often underestimated problem. According to [21], Caution is needed when providing users with the ability to upload files to a server. Attackers may attempt to:

Execute denial of service attacks. Upload viruses or malware. Compromise networks and servers in other ways.

This endpoint was a topic of discussion within the development team, including the lead management. The decision was made to move this object handling to MinIO[22] in the near future.

Task

Classification 2 - There are two endpoints that represent the base class of different task classifications. Each task can have different states, types, and priorities. Object link creation reflects the concrete type, allowing for a detailed view.

Task Notes

Classification 1 CRUD - Notes are added per each task, they are viewed in generic task aggregate. They represent message of assignee currently working on such task.

Task report

Classification 2 - 2 get endpoints - Requires generation of signed links which are later accessed to download HTML or PDF formats of task details. Purely read function, possible to be deferred asynchronously.

Task Entries - Time entries

Classification 1 - CRUD - Serves as time spent management on each task, it is on generic task. Which is the base class of all other types of tasks. Time management is handled in central datatable, referenced by task ids. Discussed with a member of the team regarding the inefficiencies of such a design, a better approach was discussed, but ultimately, it is not in the scope of this thesis, as database changes are not possible.

User

Classification 3 - This endpoint acts as an adapter to the Angler authentication endpoint. Due to current bottleneck issues with Angler, some changes may be necessary. The aim is to either re-implement a similar solution or improve it by using the existing User database table. The User table can be periodically synced with Angler at certain intervals alleviating constant calls by using

the database instead. Based on input from the development team, Angler is expected to be migrated soon.

Permissions with a more granular structure are to be designed in 5.2. Leveraging greater control over all users of specific resources can be implemented via integration with Keycloak, which is part of the stack and has been on the horizon for future development.

Currently, the user data is cached in memory. With possible changes coming from external applications going undetected. Polling and/or webhooks are to be implemented. Sadly, the current infrastructure cannot support this approach.

This was a subject of discussion with the team, including members who implemented the founding projects. It was agreed to go in the direction of optimizing these interactions by moving the data into the database, with the possibility of webhook sync triggers in the future.

Export & Import

Classification 3 - This part of the application handles the import and export of the state of the application. The handling is deferred over the message bus, signaling some separation. This part of the application can perhaps be separated as a cohesive block rather than be migrated as a feature in the same place. The difference will be discussed in subsection 3.2.5.

Shipments

Classification 3 - Shipments are another part of the application deserving of similar treatment as Export & Import. The logic is complex and already separated using the message bus. A similar approach to migrate as a cohesive block to a separated component can be taken.

1.5 Code observations

This section contains observations of some code/architecture smells or anti-patterns after a detailed analysis. Note that these opinions may be subjective and not necessarily always negative. Some of these problems are problems with the language itself (PHP, Symfony and its modules) which might force developers to use certain practices that are not very friendly.

Connascence

Connascence is a term that I first discovered in the book ”Fundamentals of Software Architecture: An Engineering Approach” by Mark Richards and Neal Ford [23]. In this book the authors use connascence to describe the amount of coupling between different components of the software system. Connascence

defines a vocabulary for describing how changes in one part of the system might require changes in another, therefore impacting the system's maintainability and flexibility. Connascence is highlighted as one of the key concepts in understanding relationships within the system, allowing developers to write more decoupled code.

An example of connascence can be two methods in different objects are so closely related that changing one requires changes to the other, they indicate a form of connascence. The relevant types are:

- Connascence of Name: Multiple components must agree on the name of an entity or property, this is one of the most prominent in the solution. The naming of individual properties is often hardcoded, in many places, in the database filter definitions, in the mapping of forms to specific types, and in the setters in the individual entity repositories.
- Connascence of Type: Components must agree on the type of an entity or property. While the language is dynamically typed, types are needed in many places, like validation and persistence. Type definition is also not straightforward, often requiring a comment block with @var components, multiply this by the need to duplicate the type definition in many places.
- Connascence of Position: Components must agree on the order of values. Positional parameters are one of the examples of this type of connascence. An example provided was array_walk(parameters, function) and array_map(function, parameters)

While these types of connascence are undesired, they are also the least urgent. It is good to be conscious of connascence when designing systems or refactoring solutions due to the complex nature of coupling.

CRUD operations

Create, Read, Update and Delete operations. Most of these endpoints fall into this simple category. The implementation revolves around managing the entities in the database, without much domain logic.

Complex operations

There seem to be many fairly complex operations that handle states across different operations. Perhaps it would be a good idea to move this state handling to a separate component, some workflow engines could be used for the handling of these complex long-lasting multi-step processes. This would simplify the application layer by focusing mostly on business logic instead of state handling. These workflow engines often have fairly simple syntax and/or interfaces that can be updated by people focusing on the business side.

Blameable trait

Blameable, commonly known as Auditable, is a desired characteristic for frequently changed objects. The task object tracks the user who modifies it, the user who created it, and the respective datetimes.

Validations

Validation is observed on many endpoints, varying in complexity. Either static value validations, like string length. There are validations using the database, like the uniqueness of some properties. There should be focus on simplifying this layer in the new implementation, perhaps treating it as a cross-cutting concern.

Task operations

Task is the aggregate root for many of the deriving task implementations. The task manager manages task notes, attachments, and the state of the task.

Message bus

The message bus acts as the interface for out of process calls, these should be respected and carried over to the new implementation.

”Reservations” and Moving of items

These rely on pessimistic system-wide locking, which can be prone to deadlocks and invalid states. Locking can perhaps be rewritten in event sourcing approach, I have found some articles online which have used this approach [24] or a simpler approach with optimistic locking can be used.

1.5.1 Cross-cutting concerns

Cross-cutting concerns are software aspects that affect the whole application. These are application-wide functionalities that might span several layers, but they should be centralized in one location. Cross-cutting concerns must be implemented on the architectural level. These include:

- Security: Authentication and Authorization
- Logging: The persistence and availability of logs and tooling to invoke logging during execution
- Monitoring: The monitoring of performance and wellness of the system
- Exception handling: Handling errors or invalid states during the execution

- Performance: This can be in the form of caching or scaling of the application

Code smells

Here we will discuss some code smells, please note these are opinionated. However, I will try to be constructive and offer solutions

- Seems there are way too many attributes and cluttered controllers
- Inability to call controllers directly from code, requires service to be implemented, thus coupling to the service, but without validation, very error prone when called from different request scope.
- Transactions are not global, therefore are easy to forget. Mistakes are made when rollback is not initiated.
- Access to view tasks is done in controller - should be moved to generic validator

FR1: Backwards compatibility with the frontend

This series of requirements specifies the importance of the new system to support the other components within the technology stack. One of the main parts being the frontend written in Vue.js code-named Swordfish.

FR2: Backwards compatibility with the OAuth server

Octopus uses a proprietary OAuth server codenamed Angler. Support of this technology enables the integration of components using Angler as the primary authentication service in their workflow.

FR3: Backwards compatibility with CI/CD methods

Octopus runs in Docker[17] running alongside the database, Redis[16] and Minio[22]. They are deployed to production environments via Docker swarm. The new system shall not deviate too much from these used technologies.

FR4: Improved maintainability

The goal of this requirement is to improve the maintainability and deployability of the system. To learn from Octopus and provide the developers with a productive environment for efficient programming. The new system should have a clean, modular codebase to simplify future maintenance and reduce the effort for testing and implementing modifications.

FR5: Permission based authorization system

Feedback was given from the development team to support other authorization providers in the future. This requirement specifies the need for the architecture to support multiple authorization providers. The old, backwards-compatible Oauth system, and a different permission-based system in the future.

FR6: Architectural support of domain processes

The new architecture should support the processes of Octopus. It will support the extracted Cross-cutting concerns from subsection 1.5.1 and Domain-level functionalities from section 1.5

NFR1: Safety of the new system - risk management

Octopus has been developed over a long period of time. The features work as expected and are depended upon by clients. Therefore a sufficient plan and development strategy needs to take place for a safe and efficient migration from Octopus.

NFR2: Performance of the new system

Developed in PHP the system was not designed primarily with performance or scalability in mind. The new system shall be more efficient than Octopus. Having less response time the application can serve more requests better utilizing the system resources.

NFR3: Documentation

The system should provide documentation of the interfaces similar to Octopus.

NFR4: Reliability

The new system should be highly stable, with mechanisms to fall back on the old system in case of failures.

CHAPTER 2

Symfony vs .NET

The Atlantis system, developed in PHP-based architecture served as a great platform for the first implementation for such a complex system. In its creation PHP posed as a simple, but powerful language allowing for quick drafts of complex functionality. In its nature it allowed for fast delivery and deployment of new features and updates.

However, as the system grows in complexity, the limitations of the technology start to show up.

2.1 Troubles with the current system

While effective in its early stages, Atlantis soon found challenges with maintainability and performance, due to its PHP backend. Performance issues became apparent when the system struggled to serve higher request volumes, leading to worse performance and response times.

Maintaining the solution also became increasingly more challenging, with the codebase becoming cumbersome and modifications requiring more and more effort and testing. This, and the nature of the project requiring more features as time went on, unavoidably leads to a less architecturally sound project with less desirable outcomes.

2.2 Previous efforts

Attempts have been made to fix some of the limitations (bottlenecks) of the Atlantis system. Improvements and optimizations to the PHP codebase have been made on a recurring basis to improve stability and performance. These fixes included refactoring of code, improving database calls, and interactions with other parts of the system.

However, these improvements provided only temporary fixes without handling the root cause: the inherent nature and unsafety of the PHP framework. Thus ideas were made to rebuild on a more suitable and safe platform, with .NET being a good alternative. The .NET framework with its type safety, better performance and maintainability could serve Atlantis in its complex business requirements and future growth.

2.3 Performance

Talking about performance is not an easy topic. Both frameworks Symfony (PHP) and .NET (C#) are mature frameworks, that have been incrementally improving on this front since their inception. It is generally believed that .NET is the more efficient of the two, supported by independent research like the TechEmpower benchmarks [25]. Some research on a smaller scale has been done by Andrei Descalu [26]. While the results are also clearly in favor of .NET, Andrei points out the difficulty of making PHP perform as well as it can.

While PHP has become significantly more efficient with the coming of PHP 8 and its numerous performance enhancements, it still is playing an unfair game compared to .NET, but why?

2.3.1 Symfony (PHP)

Interpreted language

PHP is a dynamically typed interpreted language [14], it is read and executed on the fly by the interpreter. This allows for quick development and deployment, which leads to slower execution times compared to compiled languages. There is an inherent overhead with interpreting code at runtime.

Runtime

Modern PHP versions (version 8 at the writing of this thesis) have massively improved in terms of performance, thanks to Just-In-Time (JIT) compilation. This optimization is targeted to CPU-bound applications.

According to Nikita Popov (one of the lead contributors to PHP code on this front), the introduction of JIT in PHP 8 leads to *“Significantly better performance for numerical code, slightly better performance for 'typical' PHP web application code, and the potential to move more code from C to PHP, because PHP will now be sufficiently fast.“*[27].

Note Nikita is referring to PHP 8, not Symfony, which has even less of an impact due to many optimizations done by the project on top of the PHP code. In the case of Atlantis there are no significant performance improvements

because web application performance depends more on I/O operations and database calls than on raw computation speed.

Development practices

According to Andreid Descalu's article [26], PHP is very difficult to run well on production, requiring careful configuration and attention to detail. To run PHP well, third-party dependencies need to be added Apache/fpm, PHPUnit, xdebug, etc... Which are not trivial to configure and maintain.

Writing well-performing code in Symfony requires a deep understanding of all of the framework components. Developers need to do their due diligence in providing optimized database queries and managing object serialization/deserialization. Symfony does provide tools for optimizing these processes. Unfortunately, using these tools in an unoptimized way (which is common) will greatly reduce performance.

2.3.2 .NET (C#)

Compiled language

C# is a strongly typed compiled language, meaning it can be compiled AOT (Ahead-Of-Time) into processor-specific machine code, which allows it to outperform many interpreted languages. However, this is often not used by default. By default, .NET C# is compiled into Intermediate Language (IL), which is then compiled by a JIT compiler at runtime into machine native code. This is generally good for longer-running processes, as the code is optimized on the fly, which is not possible with AOT as the code is generated only once. [28]

Runtime

ASP.NET Core is designed to support high-demand scenarios, making it a great choice for web applications that are needed to perform thousands of requests per second. These claims are supported by credible sources like [25], according to which .NET ranks high compared to other frameworks.

Development practices

C# and the .NET framework are designed with performance in mind. It has built-in asynchronous interfaces supporting async/await pattern, which allows non-blocking truly asynchronous function execution. This is perfect for I/O-focused tasks, as asynchronous calls can be executed in parallel regardless of being run on the same thread, allowing .NET to serve many parallel requests.

C# is a strongly typed language which helps prevent bugs related to dynamic typing. However, developers are required to adopt more disciplined coding practices to benefit from this.

2.4 Compatibility

In this section, I would like to discuss the compatibility between these two technologies. PHP and C# on .NET are completely different languages with very different syntax. The migration between these languages will involve more than just translating code, there will be new tools, frameworks and development practices that will need to be learned and adopted by the team. Let us explore some frameworks and their alternatives in the .NET ecosystem

Doctrine - Entity Framework Core

Let us get the most important comparison out of the way first. These ORM (Object–relational mapping) frameworks are used to map database objects to entities in the application. Entity Framework Core is the .NET alternative that is not only more powerful and faster but also benefits from the strongly typed language. Migrations are very similar and offer in-code database state management. As several members of the team pointed out, this change alone will act as a dramatic improvement of the system's performance and maintainability.

cURL, JSON, Mbstring, XML - Built in

Many PHP extensions are built in the .NET Framework itself, like System.Text, System.Xml, HttpClient, System.Globalization or System.Text.Encoding to name a few. Microsoft has a fairly hands-on approach in developing their platform, many of the core functionalities of applications are available by using the built-in namespaces.

Serialization is also fairly important in a backend application, a JSON serializer is usually used both at the input and output of the application. The .NET framework provides its own (fairly new) implementation in System.Text.Json or Newtonsoft.Json can be used in more complex scenarios.

Symfony Security, Console, Form, Dotenv - Built in ASP.Net Core

Symfony is the PHP framework of choice in the solution and it brings in many available functionalities. Many of which are built in the ASP.Net Core framework, these include the

- Security: Security abstractions are built into the ASP.NET framework with great extensibility. The interfaces can be implemented for various identity providers, more details can be found at [29]. This will be discussed more in-depth later
- Dotenv: Envrionment configuration is done using appsettings.json and appsettings.Production.json files, this is also built in functionality and is also very extensible. Further reading at [30]

- Logging: Logging abstractions and several providers are also provided by the framework, this is extensible to many providers, both delivered by .NET, .NET foundation projects, or the community [31]

- Messaging: Currently this is not built in the framework, but will be coming in .NET 9 in the future, currently a very popular (and very well supported) package is MassTransit [32]

- Deployability: The application in our case is deployed on Docker, where recently (.NET 5/6) has greatly improved upon and supports the platform out of the box with images and tooling maintained by Microsoft[33]. The applications built on newer .NET versions can also run on Windows, Linux or MacOS which greatly improves developer setup.

Redis, MinIO, PostresSQL, Keycloak, Angler clients

These components are used by Octopus to fulfil various tasks like caching, object persistence or OAuth providers.

- Redis: StackExchange.Redis library is used for Redis communication

- MinIO: Though the component is not currently used, the team showed interest in moving to this technology soon. MinIO is built to be compatible with AWS S3, which is supported on .NET

- PostgresSQL: The database is supported by open-source library Npgsql.EntityFrameworkCore.PostgreSQL, which implements the interfaces for Entity Framework Core

- Angler, Keycloak: The OAuth security flow is supported directly by the framework using the Microsoft.AspNetCore.Authentication.OpenIdConnect package, though some custom logic will be needed to implement some details

2.5 Conclusion

Due to a fairly pragmatic technology selection on the side of Octopus, many of these technologies are easily adopted on .NET side. This aligns with my preference to rely on external frameworks as little as possible. Making software rely on external frameworks is a recipe for fragile applications, which have drastic implications when some core libraries are decommissioned. We will go one step further with this idea by structuring the application to decouple the functionality from the frameworks as much as possible in the section 4.4.

FR7: Performance optimization

The new .NET system must serve requests with improved performance compared to the old PHP system.

FR8: Performance monitoring

To facilitate and track the progression of the first task, implement a tracking mechanism for the performance of the system.

FR9: Use compatible technologies

Utilize packages discussed in section 2.4 to implement key functionalities of the system like data persistence, logging, and authentication/authorization.

NFR5: Knowledge transfer

Provide knowledge transfer to the development team on .NET best practices and the structure of the new system to allow better future development.

CHAPTER 3

Migration strategies

The migration of Atlantis from PHP to C# .NET will be challenging. A strategic plan is required to make sure the transition will be as smooth as possible. The goal will be to research the best path of action for this migration to minimize risk and downtime. We also need to make sure that the resources spent building these migration tools do not outweigh the resources needed to just migrate all in one go.

3.1 Full migration

The optimistic but naive approach is to migrate fully to C# in one go. Arguably this is the simplest approach and often is a go-to with some smaller projects, but why would we think it might not work here?

The positives

1. Simplicity: A full migration is straightforward. During the migration changes can be made which are not backwards-compatible. Architecture, database, and interface changes can be made extremely quickly. When completed, the team can fully focus on the new solution without maintaining the old one.

2. Consistent: The state will always be fully on one platform, no integration is required between the old and new implementation. The codebase will be fully in C# which might allow better coding practices, more performance and easier maintenance.

The negatives

1. Risk: This approach is very risky. A bad migration can have significant consequences for the stakeholders of the system: the clients. Which might have a very damaging impact on the company. A failed migration might cause long downtime, inconsistent functionality, long feature delivery time, or bad performance.

2. Resource intensive: Requires a big investment in time and resources from the managerial point of view. The complete application must be rewritten, tested, and deployed before any benefits can be drawn. This can put a big strain on the development team budget.

3.2 Phased migration

Phased migration is an approach to migrating of an application from one type of platform to another, in this case from PHP to C# in a manageable, incremental process, hopefully minimizing downtime and integration problems with the existing system.

By breaking the problem into smaller chunks - phases, the developer can focus on a specific part of the application, working in parallel with other team members. Members make sure each part is fully migrated before continuing onto the next. In this part we will be discovering the concept of phased migration, the granularity of such an approach, and the strategy for data handling during migration of the database.

We shall now define the Strangler Fig Pattern, which shall guide us in the migration of the project.

3.2.1 Strangler Fig Pattern

The Strangler Fig Pattern is a type of phased migration. *"Incrementally migrate a legacy system by gradually replacing specific pieces of functionality with new applications and services. As features from the legacy system are replaced, the new system eventually replaces all of the old system's features, strangling the old system and allowing you to decommission it..*"[1]

The name comes from the strangler fig tree. This tree grows around another tree, eventually replacing it

3.2.2 A façade as the interceptor

The Strangler Fig Pattern defines a gradual migration approach to a new system, while keeping the old system to handle features that haven't been migrated yet. This presents the challenge of managing two distinct implementations. Having two implementations of the system means we need to

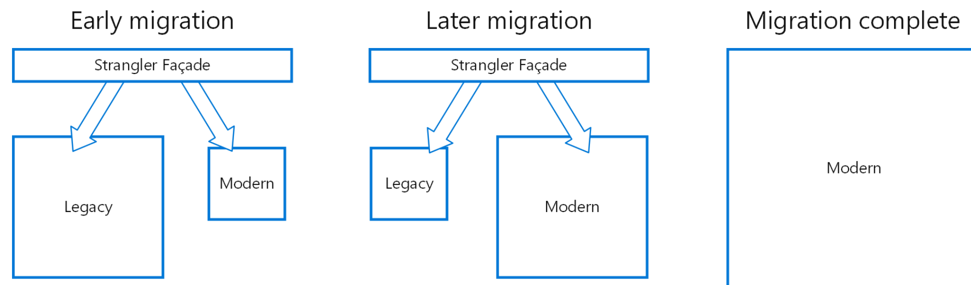

Figure 3.1: Strangler Façade migration phases [1]

define for the client interfaces, which implementation to use for which use case.

The strangler façade comes as a solution to this problem. Defining a routing layer that intercepts the requests and routes requests to where they should be served. As seen in the illustration 3.1, the strangler façade serves until migration is complete, after which the layer is no longer needed.

It helps minimize risk and allows for any speed of migration. Over time, as features are migrated to the new system, the legacy system is eventually "strangled" and is no longer necessary. Once the whole process is complete, the legacy system can be fully retired.[1]

A great benefit is the optimization of resources of this approach. We maximize usage of the legacy system, while focusing on high-priority replacements in the new system.

3.2.3 The phases of Strangler fig pattern

Based on [7], there are five steps to applying the Strangler Fig pattern:

1. Service identification

To identify sections to be migrated, it is recommended to use Domain Driven Design (DDD) to create a universal language to describe the terms and concepts in the business domain. It tries to involve both technical and business knowledge to identify the entity classes and relationships.

Several techniques can be applied during this process. The research mentions event storming [34], domain storytelling [35], and user story mapping. These are useful for retrieving requirements and gathering a higher-level domain understanding. While they are helpful and very well-documented, they are not useful in our case, as the general domain knowledge is contained within the team and can be retrieved using qualitative interviews.

At the end of this step, we shall have an understanding of how to split the solution into cohesive blocks or features, by which we will track the migration.

Table 3.1: Criteria for block/feature Extraction [7]

Criteria	**Score**	**Rationales/Benefits**
Function soon to be Modified	{0, 40}	Extracting this kind of services can avoid making changes directly to the legacy system.
Performance Problem	[0, 20]	If a block/feature consumes many resources and may affect the performance of other modules, it should be extracted early.
Core Function	[0, 20]	If the extracted service includes core functions, the extraction usually brings more benefits, like scalability or maintainability
Related Service	[0, 10]	It would be better if an extracted feature/service collaborates with other migrated blocks/features.
Separate Service	[0, 10]	Extracting this feature/block is easier than other candidates.

2. Preparatory task

In this step, the communication and technical aspects of the migration platform are implemented. As an example, an API gateway or some other Strangler façade is implemented to route requests to and from the legacy and new implementation. The façade is then deployed alongside the new solution.

3. Prioritization

Prioritizing which blocks/features to migrate first is an important step. The research suggests a reference approach for this step.

When a block or feature is identified and expected to be migrated, a score is given based on Table 3.1. The feature/block with highest scores should bring the highest benefit when migrated first. Note that the ”Function soon to be modified” score is not a range like the rest but rather a given value based on true or not. If the feature is soon to be modified the score is 40, else it is 0.

4. Implementation

After prioritizing and choosing which feature or block to migrate, comes implementation, integration, and testing. Features and blocks implemented use models and interfaces of the legacy system, creation and reusing of infrastruc-

ture is part of the implementation, but it shall not interfere with the legacy system at this time.

5. Removing legacy code

This is the final phase of the Strangler Fig pattern. The legacy system and the new system have been working concurrently without effect on users. The new implementation has been fully tested and integrated into the solution in a production environment. Therefore, the legacy code, which this new code has replaced, can now be safely removed.

3.2.4 Migration by features

Now that we defined the steps of migration, let us define how we might want to split the solution into cohesive units. One approach is with features. A "feature" is a specific piece of functionality that delivers some service to the user.

Based on [36], a feature is a group of requirements that describe what the software should do. The definition focuses on what the feature is supposed to achieve without getting into details of how it will be implemented in code.

In a warehouse management system, it might be user authentication, stocking of items, listing of products, etc... While migrating by features, one would select one of these features and migrate it from the legacy system to the new one, while the rest of the application stays the same.

Each feature should be isolated, allowing for rigorous testing for that specific feature. It shall not affect any other features. With this approach, it is necessary to maintain and have infrastructure in place for handling of this migrated feature. Part of the system will be on new codebase, and part will be legacy.

3.2.5 Migration by blocks

Migration by blocks is similar to the previous approach but divides the application into larger units or "blocks". These blocks are a sum of several tightly coupled features that represent some part of domain logic.

With a warehouse system, it might be some batch order processing, report export or delivery processing. With block migration, the whole block will be migrated at once.

The block is larger and less granular than features. It usually contains multiple functionalities. The migration of a block might require more starting commitment and effort but could save on resources by implementing tight coupling without splitting into individual features. There is an opportunity here to shift into more decoupled communication between modules, using messaging or event-driven architecture.

3.3 Phased migration - database

The database is an important part of any system. It represents the state of the application. Having multiple databases means multiple concurrent states.

3.3.1 New database

A migration often includes the creation of a new database schema. This has benefits and drawbacks.

Benefits

1. Performance: We can optimize the database to improve the performance of the application, this might improve scenarios where database was a bottleneck.
2. Modernization: This is inherently ties to the first point, but there are different approaches to databases that might be better supported in .NET and could not work on the legacy system. Like multi-tenancy or more optimized database entity inheritance.

Disadvantages

1. Complexity: This will add much complexity to an already resource-intensive migration
2. Testing: There will be much more testing needed to ensure data integrity.
3. On-boarding: The team might need more time to adjust since the database might be very different.
4. Resource-intensive: More time, budget, and coders needed.

3.3.2 Backwards-compatible database

In some cases the legacy database can be used, especially if it succeeds to serve clients and there would not be enough benefit to develop a new database.

Benefits

1. Simpler transition: When using legacy database, the old system can act as fallback - a safety net, when new system fails
2. Real data: Since the legacy system uses the database, we can test the new solution on real production data, allowing much improved testing.
3. No database sync: Using one database, there is no need for synchronization of databases.

Disadvantages

1. No (limited) schema changes: This was touched on above, there cannot be any big changes to the database, as they would affect the legacy system. Such changes would also need to be synchronized to be supported at the same time on both systems.

2. Locks - Transaction management: It is important to avoid transaction conflicts between the two systems. In some cases, there might be custom locking, which needs to be migrated fully.

Special care needs to be taken with the migration of a backward-compatible database. Automated testing and monitoring are critical for this approach to work. We need to ensure that changes in one system do not break the other.

3.4 Conclusion

The migration of Atlantis from PHP to C# .NET, will be complex. It should be managed through a phased approach using the Strangler Fig Pattern. This method minimizes risk by allowing incremental replacements of the old system, ensuring the project remains stable and functional throughout the transition. It provides an iterative way to implement changes, which will optimize resource usage and maintain operational continuity.

FR10: Backwards compatibility with the database

The database (PostgreSQL) will be shared between the old system (Octopus) and the new system. The new system will be required to support the used schema.

FR11: Façade Implementation

Develop a strangler façade to intercept requests and route them appropriately during migration phases.

FR12: Implement parallel deployment of the two systems

Facilitate the communication between the old and new system on one environment.

FR13: Feature or Block Migration

Implement migration on a feature-by-feature or block-by-block basis.

FR14: Testing and validation

Implement rigorous testing to ensure that migrated components are functional

NFR6: Modularity of the new system

The system's architecture will need to be at least as modular and organized as the Octopus system. The system will be divided into features or blocks based on which it will be slowly migrated.

NFR7: Use Strangler Fig Pattern

Use the strangler fig pattern as the guideline for the migration.

NFR8: Minimize downtime

The result of the migration strategy shall impact the system downtime as little as possible so as not to disrupt the business.

Part II

Design

CHAPTER 4

Architecture

Software architecture is the foundation upon which software systems are built. It defines not only the structure and dependencies between components, but also the interactions within to create a cohesive and effective system. One of the more influential books for me in this field was The Clean Architecture by Robert C. Martin [4], where he mentions one of the main objectives of good architecture:

"The goal of software architecture is to minimize the human resources required to build and maintain the required system."[4].

Let us set the goals of this chapter:

1. Assess Current Architecture: Analysis was part of chapter 1
2. Apply Modern Architectural Practices: We will discuss these in this chapter
3. Use Migration Strategies: Analysed as part of chapter 3.

4.1 Why Architecture matters

Reducing risk

- Technical debt: Without a proper, well-defined architecture, the lack of initiative from the developers, or lack of funding, the project can easily spiral into a "Big ball of mud" [37]. A robust architecture provides a framework for making good decisions that prevent impulsive solutions likely to cause problems later.
- Scalability: As the business grows, the software needs to grow along with it. With increasing loads and new functionalities, the system must be flexible enough to accommodate this without sacrificing performance.

- Integration: Large systems usually involve integrating multiple systems together. With a well-planned architecture, these integrations shall be simplified by creating well-defined structures to accommodate these interactions. (clean architecture, separation by layers)

Resource optimization

- Modularity: When designing architectures it is important to keep modularity as one of the key values. A modular system divided into manageable components or services facilitates ease of maintenance, faster development and allows teams to work independently of each other, greatly enhancing productivity.
- Agility: A good architecture will align with the needs of the business, allowing it to respond to changes in an agile way. By facilitating new features for new opportunities in a timely manner without inducing more risk and increasing human resources costs.
- Future-proofing: By anticipating future needs of the system, a well-defined architecture can ensure it remains relevant. This foresight can help prevent overhauls in the future.

4.2 Current architecture

Due to the switch to another language (C#), the code will need to be rewritten. We plan to use many strategies used in the development of Octopus as a starting point. However, Octopus has been developed for a considerable period, we can learn from the mistakes made and create a better architecture. Analysis was made in chapter 1, which we will use to build a more efficient system.

The solution follows a standard Monolithic MVC approach structure:

- Controllers: Controllers handle the incoming HTTP requests, process them by invoking application services, and return the appropriate responses. The controllers are structured by the provided features/use cases for example: handling Users, Delivery Accept tasks, etc... The controllers parse the input, validate the forms, invoke the respective application service and format the output.
- Models: Models represent the business logic of the application. They are responsible for retrieving data, handling the ORM entities and returning data to the controller. The structure of the models folder is organized into the feature-specific subfolders making the folders fairly organized and easy to navigate.
- Views: Views are not represented in the solution.

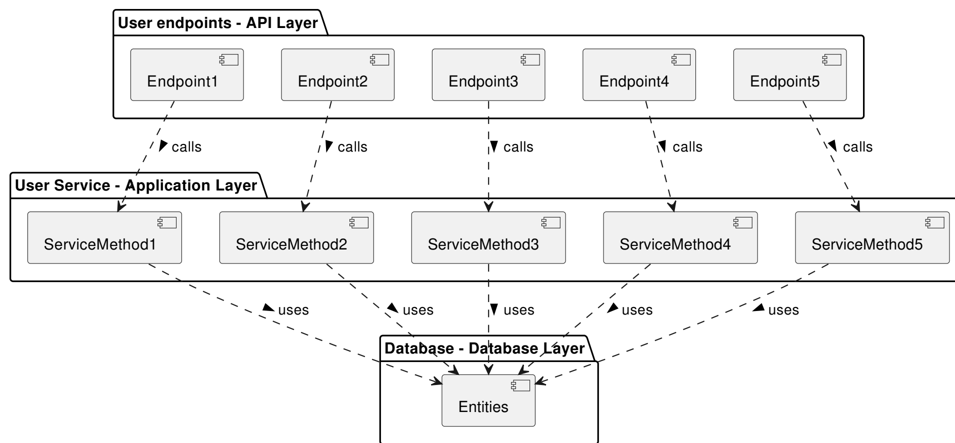

Figure 4.1: Application layer - Service layer

The solution, in general, is well-structured and easy to navigate. Unfortunately, a lot of the code is cluttered due to the constant data retyping and a fairly hands-on approach to entity handling, but this is mostly due to the Symfony platform.

A fairly common pattern of an API-Layer (Controller), and Application/-Database layer (Models) is shown on Figure 4.1.

4.3 Enhanced architecture

In this section, we aim to introduce some modern practices when building new architectures. Architecture is a fairly subjective topic. Each developer is used to some way of doing things, and dictating how a developer should structure their code is divisive and often leads to prolonged, passionate discussions on what is better.

The main values for the new architecture:

- Maintainability (FR4): The application shall be simple to develop and maintain
- Testability (FR14): The application shall be very testable
- Loose coupling (NFR6): The architecture should be loosely coupled, to enable migration and the possible separation of some modules

4.3.1 SOA/Microservices

One of the important considerations is the migration straight to microservice-focused architecture. Microservices are very testable and loosely coupled, as they have fairly firm interfaces to be able to communicate with each other,

a simple representation is at Figure 4.2. Unfortunately they are often very hard to maintain. Developers often find writing the code for microservices to be very straightforward. The problem comes with deployability and maintenance.

Benefits

- Scalability: The microservices allow parts of the application to scale independently, which can be more cost-effective than a monolithic approach
- Resilience: The services are fairly independent, failure in one might not bring the whole system down.
- Mix match technologies: It can be possible to have multiple languages and frameworks deployed at the same time
- Team independence: Teams working on the project can work independently

Disadvantages

- Complexity: Managing microservices is incredibly complex. Dealing with distributed systems involves working with network latency issues, message handling. A huge stack is harder and costly to maintain
- Databases: With microservices, each microservice manages its own database, which brings the difficulties with transaction handling and eventual consistency.
- Testing: Unit testing is not sufficient, usually replicating the whole stack is necessary to sufficiently test the application
- Deployability: Deployment in microservices is often overlooked, but it is one of the most challenging problems with microservices. Maintaining continuous deployment, orchestrating service initialization and whole application monitoring can be very resource-intensive

4.3.2 Monolith

Benefits

- Simplicity: Monolithic applications are simple to develop, test and deploy. Scaling is also simple, usually requiring to copy the setup to another machine.

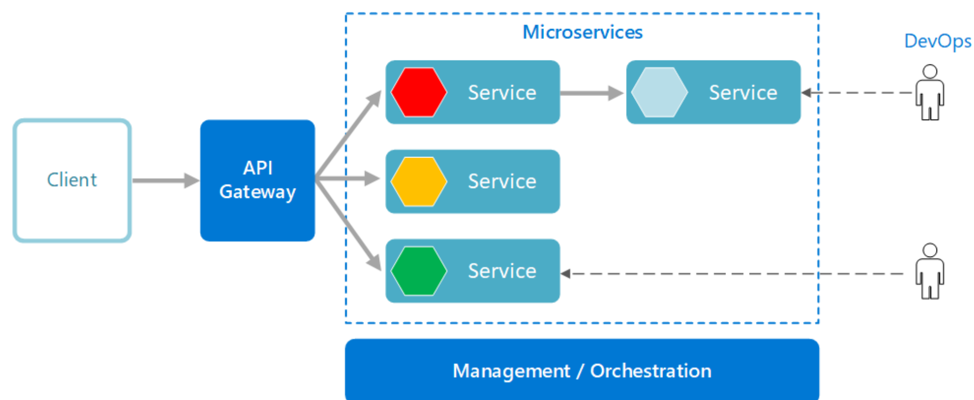

Figure 4.2: Microservices[2]

- Better for small teams: The simplicity leads to the system being viable for a smaller 4-7 member team. The work is usually possible to be split when the application is well structured.
- Testing: No need to create large networks of services, debugging is simple

Disadvantages

- Scalability: This can be an issue if some parts of the application need to be scaled rather than the whole application. This can be expensive
- Slow development: In teams of more people and in a system with tight coupling, it might become problematic to work in parallel due to large amount of possible conflicts, extending the time of delivery.

4.3.2.1 Selecting the architecture

Selecting the right architecture is not always about having the most features, but about having the appropriate structure for the task at hand.

The comparison is summarised in Table 4.1. But to reiterate, with the microservice architecture, the issues start with additional components in the application stack, such as messaging and transaction handling to ensure eventual consistency, API gateways/proxies, and service bus implementations. The issues continue with DevOps, like the order of startup initialization, error handling, and versioning.

It seems that with this project, we should lean more towards the Monolithic architecture, given the scale of the team and the requirements of the system do not indicate the benefits of a microservices architecture would be fully utilized. Also, many sources often overlook the upfront cost of investment into a microservice architecture. In my opinion, starting a project with

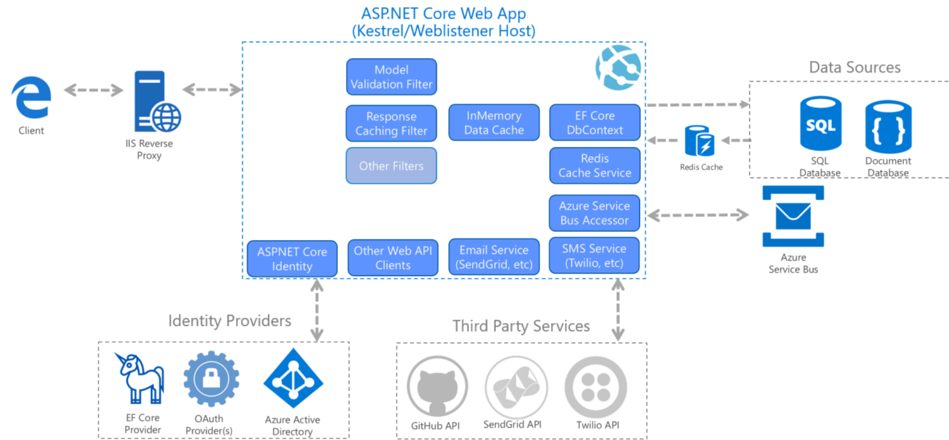

Figure 4.3: Monolith[3]

Trait/ Architecture	Monolithic	Microservices
Simplicity	Easier to develop, test, and deploy. Suitable for small teams.	Complex setup and operations, needs advanced CI/CD.
Performance	Faster in-process communication.	Possible latency with service communication.
Scalability	Scales as a single unit, less efficient.	Services scale independently, more flexible.

Table 4.1: Comparison of Monolithic and Microservices Architectures

microservices from the start can actually be a significant operational and financial burden.

With that in mind, we shall not be disregarding the idea completely. A more modular approach to a monolithic architecture could mean that in the future a migration to microservices is possible. We should allow such an architecture to incrementally evolve from the monolithic architecture if required.

4.4 Clean Architecture

Clean architecture is a term coined by Robert C. Martin, the author of the book "Clean Architecture: A Craftsman's Guide to Software Structure and Design" which is mentioned at the start of the chapter. Previously named the

”Hexagonal architecture” or ”Ports-and-Adapters architecture”, Clean Architecture has become a standard in well-structured solutions, which has been recognized by Microsoft themselves[3]. In this section, I would like to explore the design psychology behind the Clean architecture and why going with this route could bring some well-needed restructuring to the Octopus system.

4.4.1 Understanding Clean Architecture

Clean architecture is modeled as a set of concentric circles representing different areas of a software solution as seen on Figure 4.4.

1. Entities: Is at the core of the model, entities represent the enterprise business rules. An entity is an object with methods and sets of data structures. They are the least likely to change when there are external changes. For example, one would not expect the domain entities to change when the UI (or in our case a DTO) changes.

2. Use Cases: One layer above are the use cases, which contain the application logic. The use cases orchestrate the entities based on the inputs and outputs of the application.

3. Interface adapters: In this layer external dependencies (interfaces) which are defined in the layer closer to the center are implemented, for example, the database interactions, application services, etc... One would also find JSON serializers which are used to convert Hypertext Transfer Protocol (HTTP) requests to internal data structures to used by the use cases.

The dependency rule

The dependency rule is one of the fundamental concepts behind the Clean architecture, it dictates that code dependencies must point inwards. Nothing in the inner circle should know anything about anything in an outer circle. For example, a domain entity should not know about the database engine, web frameworks, or external services used. By following this rule, it is impossible to fall into the circular dependency trap, which can create very hard to maintain code.

Benefits of Clean architecture

- Testability: The architecture promotes high testability by separating the use case and domain layers from the UI and the database. This isolation allows the creation of simpler and targeted unit tests, which do not rely on external components. Having high testability is the key to having reliable and dependable software development cycles.

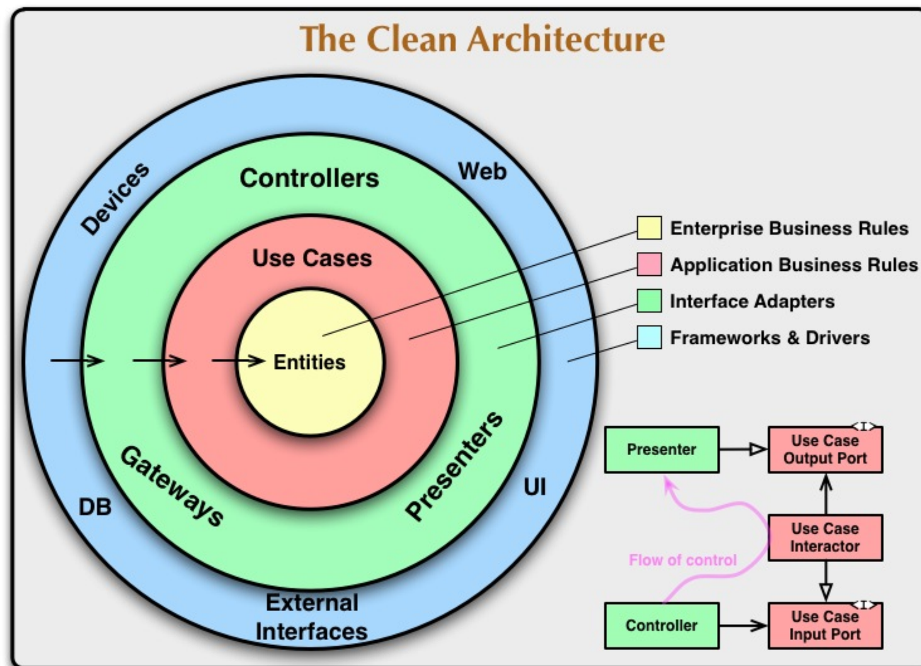

Figure 4.4: Clean architecture[4]

- Maintainability: With clear separation of layers and the encapsulation of business logic, Clean Architecture simplifies maintenance. Developers can work independently on different layers of the architecture.

- Flexibility: By applying separation of concerns on the level of the Clean architecture, one can create software that is very flexible to changes. This is further enhanced by following DDD and Command and Query Responsibility Segregation (CQRS), which is fully supported by the architecture.

Important considerations - Learning curve

With the clean architecture, it is important to get used to the decoupled nature of the system. Developers might be used to a standard MVC (Model View Controller) structure, even MC (Model Controler) structure in backend systems. It is important to understand the responsibilities of each layer to avoid misplaced application logic that could lead to a violation of the dependency rule.

4.5 CQRS and Domain Driven Design (DDD)

DDD and CQRS (Command Query Responsibility Segregation) are architectural patterns that can provide great benefits to a complex system like Atlantis, especially in combination with Clean architecture. The integration can improve maintainability and scalability by separating responsibilities.

CQRS

CQRS splits the application into two parts:

- Commands: The command side handles the creation, updating, and deletion of data. The commands encapsulate the operations which change state or mutate the data.
- Query: The queries handle data retrieval (the non-mutating) operations.

This separation increases performance and scalability by allowing each side to be scaled independently. For example, the queries can all be cached, while the commands cannot. This can be abstracted using the mediator pattern, allowing generic caching mechanisms. Microsoft has a nice article on this in its design pattern section which can be found here [38]. The article shows some further extensions that can be made to this pattern with event-sourcing and messaging. We will be implementing this pattern in its basic form, allowing these extensions to be made in the future.

DDD

DDD focuses on complex domain logic that encapsulates business processes. It aims to provide a place to define high-level processes (that are encoded in code), inside the individual ”bounded contexts”. Bounded contexts are logical boundaries around a coherent and consistent domain. This is illustrated by Martin Fowler, another great software architect, in diagram Figure 4.5.

There are several types of objects which are usually at the core of a DDD system:

- Entities/Value Objects: Entities carry identity - a user, an order. Value objects do not, for example color, currency etc...
- Aggregates: An aggregate is a cluster of domain objects (entities and value objects) that can be treated as a single unit for data changes.
- Domain events: These events reflect state changes within the domain, for example Task completed, Task assigned, Task created, etc... These are events which are propagated through the domain *after* the action was made. These events can be captured within the application layer to propagate messages, perform actions etc...

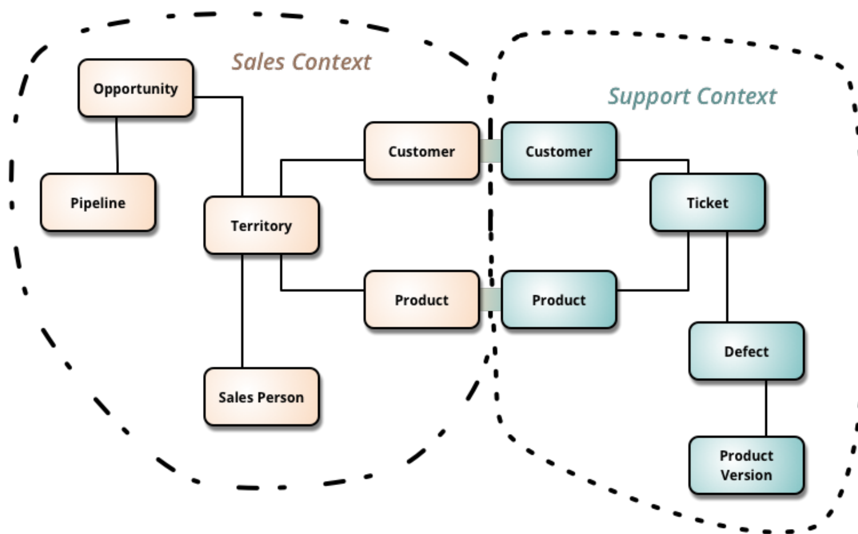

Figure 4.5: DDD - Bounded context[5]

- Repositories: Repositories are used to encapsulate the logic needed to access data for aggregates (entities, value objects)

4.5.1 Final note on system design

For me, designing a good architecture means being defensive by keeping as many doors open as possible, allowing future expansions to be added in a way that affects the core system as little as possible. This means implementing modular code, minimal dependency on specific frameworks, and adhering to the SOLID principles. Even though we will be discussing specific technologies in further sections, the code will be built to rely on as few specific providers as possible, fulfilling one of the core principles of the Clean Architecture.

The most daunting part of the Clean Architecture is the initial setup, creating firm interfaces for databases, identity providers, etc... After the initial investment in the solution setup is done, the long-term benefits of a cleaner, flexible, and more maintainable solution are undeniable. In my opinion, the complexity that is introduced is greatly outweighed by the benefits. Implementing the Clean architecture from the start sets a solid foundation for the future of the software system.

With that in mind let us take a look at the current architecture of Octopus and showcase the direction we will take the new system.

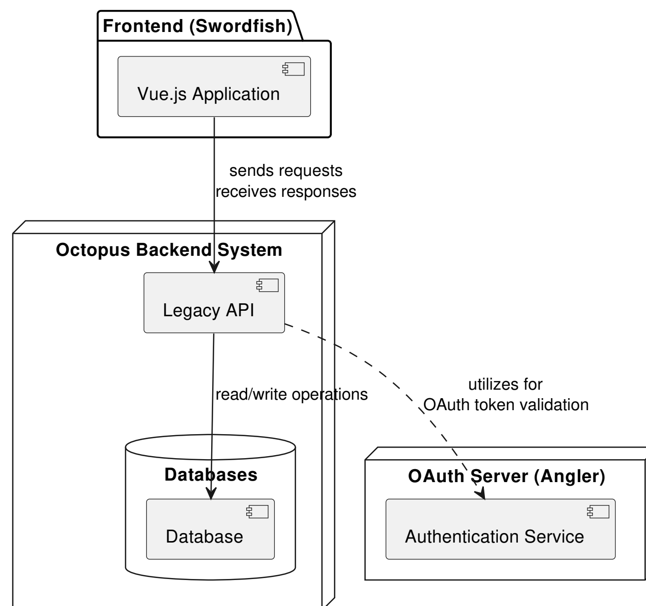

Figure 4.6: High-level overview of the Octopus system

4.6 30,000-foot Overview

Atlantis in its current state contains 4 major components:

1. Swordfish: The frontend application
2. Octopus: The backend application
3. Angler: The OAuth authentication application
4. The persistence: PostgreSQL and Redis

There are some other components in the ecosystem like Pufferfish (ETL Application), Redis and Minio (Databases) which I am omitting from the diagrams for the sake of simplicity. The system we wish to design for the new application as part of this chapter is the "Octopus Backend System" in Figure 4.6.

4.6.1 Naming the new system - Squid

Let us name the system "Squid". The name is not of my own making but has been a product of team discussion and my colleague Max's initiative. Let us, from now on, use "Squid" as the codename for the backend system built in .NET. Squid will be packaged as a direct replacement of "Octopus" - the current backend system.

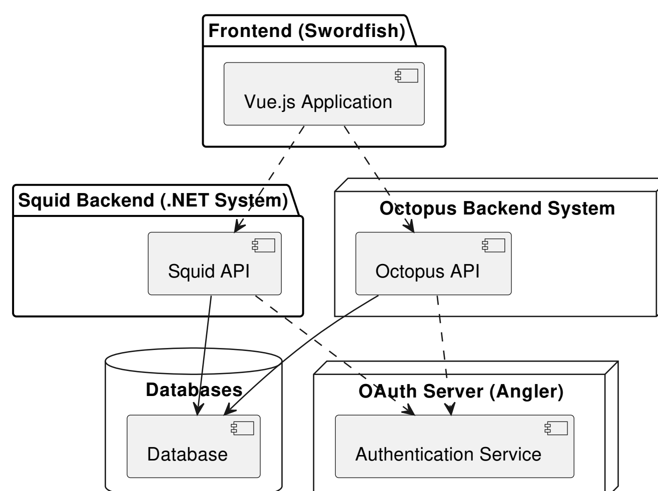

Figure 4.7: Squid system - draft

First let us address FR1, FR2 and FR10. These requirements specify the backward compatibility with the old system components. With these requirements, we could simply replace ”Octopus Backend system” from Figure 4.6 to ”Squid Backend system” and simply develop ”Squid API” from the beginning. However, FR12 specifies the requirement of deploying these systems in parallel with each other. Therefore, we shall draft this simple approach at Figure 4.7.

With this schema, we violate requirements NFR1, NFR4, and NFR8 by coupling the migration to the frontend. Basically requiring the frontend application to facilitate routing based on the migration phase[†], this will induce more risk on an already functioning system. With these requirements, we intend to shift all of the migration handling to the Squid system, this allows us in case of critical failure to transfer back control to the old system as a sort of kill-switch. How this would look can be seen in Figure 4.8.

With this structure, we expect the new application to be deployed alongside the rest of the stack. We therefore, need to fulfill (FR3) for which we will use Docker compatibility is confirmed in section 2.4.

4.7 10,000-foot Overview - Strangler façade

In this section, we shall dive deeper into the structure of the Strangler façade and our commitment to NFR7.

The strangler façade needs to have at least 3 modes:

1. Octopus mode (Bypass mode): The façade redirects the request to the legacy Octopus system.

[†]The front end would need to decide which application to go to for each request.

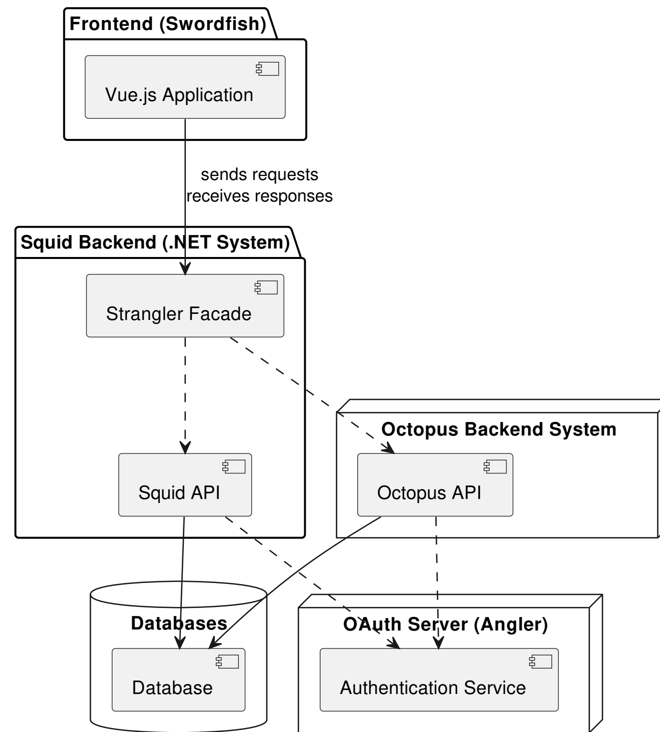

Figure 4.8: High-level overview of the Squid system

2. Ghost mode: The façade clones the request to both systems, returning the response from legacy system.

3. Squid mode: The façade redirects the request to the new Squid system.

It will need to choose which mode to execute based on the migration phase of each feature or block, fulfilling NFR1 and NFR7. Let us look at how we should design the modes in Figure 4.9. The strangler facade should act as a sort of proxy between Octopus and Squid systems.

This façade allows us to go forth with FR13 (Feature and Block migration) and allows us to use Squid without having all features migrated from Octopus. Given a request which comes targeting feature X endpoint the facade will execute:

1. Feature X is not migrated -> Octopus mode

2. Feature X is implemented but is not fully trusted -> Ghost mode

3. Feature X is implemented and is fully trusted -> Squid mode

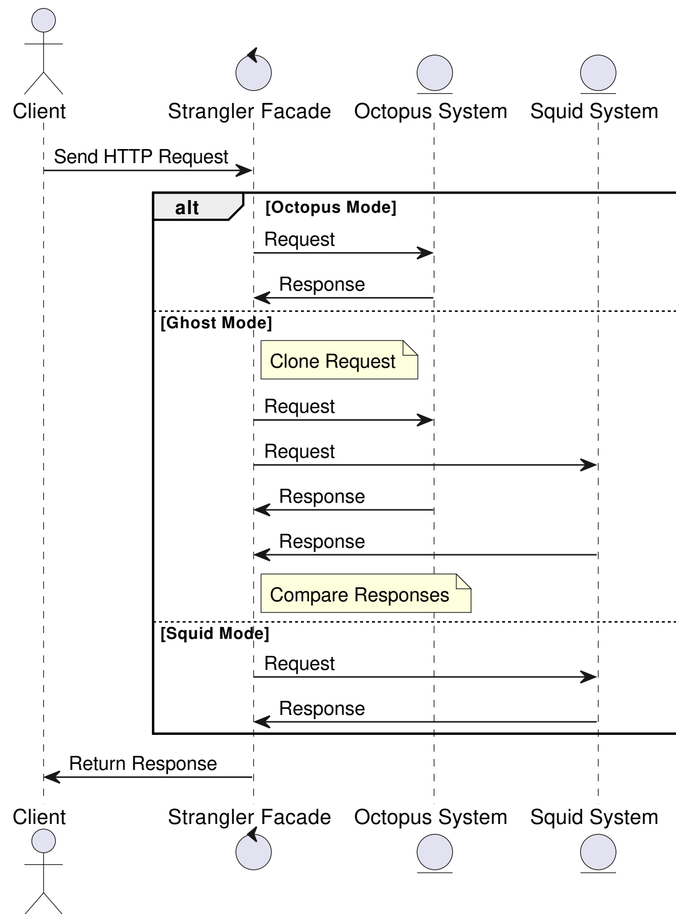

Figure 4.9: Strangler façade modes

4.7.1 Key factors

Aspects to focus on ordered by priority:

- Reliability (NFR4): The façade in its various modes, especially Octopus mode, should be fully reliable. Ideally it shall contain as little points of failures as possible. The façade will become part of the critical infrastructure and therefore, should be reliable.

- Compatibility with Frontend (FR1): This façade will become a new layer between the frontend (Swordfish) and the backend (Octopus) as shown in Figure 4.8. We shall make sure the systems will be compatible.

- Feature or Block Migration (FR13): The façade shall direct the request based on its migration phase.

- Monitoring (FR2): The façade in its Ghost mode will provide valuable data when comparing the responses, which can be used by the development team to resolve bugs and inefficiencies.

With these factors in place, NFR7 shall also be partly fulfilled.

4.8 10,000-foot Overview - Squid API

The Squid API shall follow the modern practices aligned in the chapter 4. It will allow us to:

1. Modularity (NFR6): Facilitate all required architectural features, enabling compatibility of all features from Octopus to be migrated in cohesive modules (features or blocks)

2. Maintainability (FR4): Keep open doors for architectural features that might improve some of the processes

3. OAuth Compatiblity (FR2): It shall be compatible with Octopus authentication providers

4. Documentation (NFR3): The architecture will be self-documenting or shall allow the documentation level possible with Octopus

The structure of the SquidAPI will follow the Clean architecture structure as seen in Figure 4.10. The colors are chosen to represent corresponding layers as in Figure 4.4. This structure follows the Dependency rule of all dependencies pointing inwards, with no cyclic dependencies and with the domain logic at the core of the application.

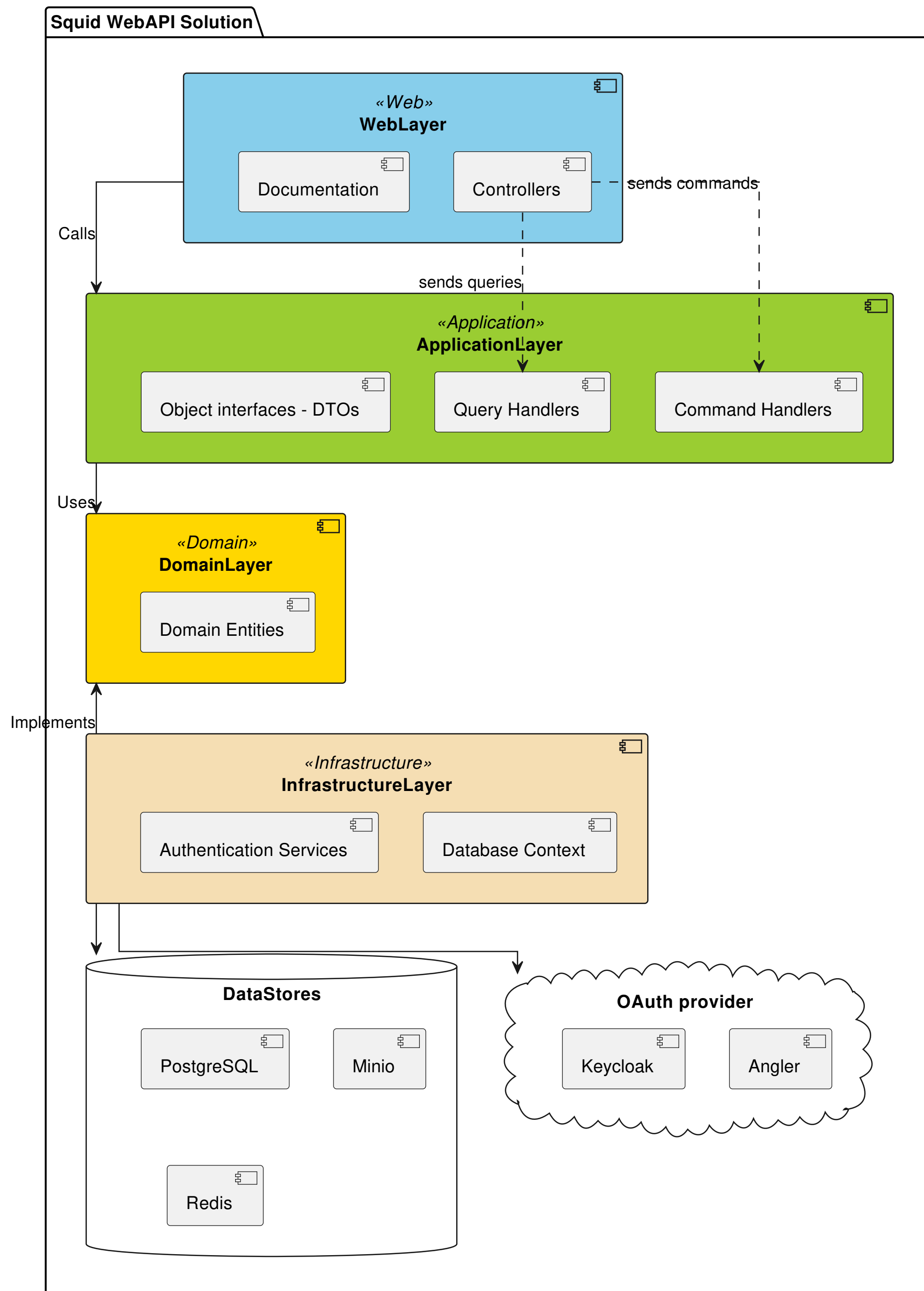

Figure 4.10: Squid Api - Clean architecture

Part III

Implementation

CHAPTER 5

Implementation - Squid API

In this chapter we will focus on the actual implementation of the backend, the future replacement of the Octopus backend API. We will be adhering to the concepts outlined in the previous chapters like Clean Architecture, CQRS etc... The goal of this work is to set a solid foundation for the solution in .NET using well-established practices. By the end of the chapter, we will have a functioning system with many of the features and infrastructure adopted from Octopus in a production-ready state. A crucial component which is in scope of the implementation and will sit conceptually above the backend API is the strangler component. This component will allow this system to work in tandem with the legacy Octopus system, we will discuss the implementation in the next chapter.

5.1 Motivation

One of the books that were very influential for me and my road as a developer has been "The Clean Coder: A Code of Conduct for Professional Programmers" [39] by Robert C. Martin (again!). The author was able to distill the essence of being a professional developer into a book, he was able to give experienced insight into what being a professional means. The general points were:

1. Professionalism: The word used throughout the book has many implicit meanings. The author defines professionalism to include taking responsibility for one's work, producing high-quality code, and thinking about how it affects clients, colleagues, and the organization.

2. Code Quality: The book highlights the importance of writing clean, manageable, and maintainable code. The author, Martin, promotes practices that improve code quality, such as Test Driven Development (TDD), continuous, persistent refactoring, and commitment to the SOLID principles.

3. Discipline and Continuous Learning: The field of software development is still evolving, and the author highlights the importance of learning and updating knowledge with the current new technologies and best practices. Having good discipline is very important to being a valuable asset to the team, to commit to doing the best job possible.

4. Communication: Effective communication is a key skill of a professional developer. This includes clear communication with team members, clients, and business stakeholders. Martin points out the unfortunate circumstance in which developers do not expect to communicate much when starting their career working at a computer, but the opposite is the truth. Communication as a software developer is one of the most important aspects of the job, be it by providing project status, admitting mistakes, or standing by own principles even when under pressure to compromise on the quality.

5.1.1 Cooperation with Max

I have worked closely with Bc. Max Hejda during the Analysis and Implementation phases. Max has been instrumental in the DevOps department, helping the project by implementing the CI/CD pipelines, retrieving access for various shared components, and ultimately deploying the project. Max will be further expanding upon the project in his Master's thesis.

Scrum - project collaboration

We have utilized the scrum agile framework as the collaboration template for this project. As there were only two of us, we have decided to use a simpler version:

1. Sprint planning and retrospectives: We held a planning/retrospective meeting at the beginning of each sprint. This involved closing finished tickets and identifying prioritized and feasible tasks for completion within the next sprint. The planning occurred in 2 week intervals.

2. Weekly stand-ups: As we were in constant communication, this served more as a generic check-up on the progress of the tickets or by discussing any obstacles we were encountering.

3. Tooling: For tracking tickets, we used the company Redmine instance, and for communication, we used the company Slack messaging instance.

By incorporating scrum practices we improved the workflow of planning future work. This was structured and flexible enough to work for our use case.

5.2 Clean architecture in .NET

The application is divided into 4 projects (not including tests)

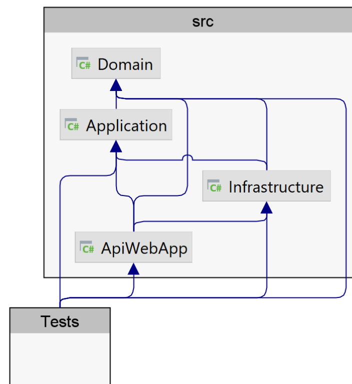

Figure 5.1: Squid - Implementation Dependency structure

1. Domain: Equivalent to the "Entities" layer in the Clean architecture, it encapsulates the domain logic and rules.

2. Application: Equivalent to the "Use Cases" layer, it contains the application logic of the application. It interacts with the domain entities to fulfill the specific usecases.

3. Infrastructure: This project implements the interfaces in the lower inner layers, like identity and other external services.

4. ApiWebApp: This project serves as the "frontend" of the backend application, in the sense that it contains the controllers. It also serves as the startup project and ties all of the projects into an executable.

5.3 CQRS and DDD implementation

The CQRS pattern was implemented using the MediatR library [40] this is a lightweight implementation of the mediator pattern (discussed in my Bachelor's thesis [41]). We can see the simple CQRS pattern at Figure 5.2 and the enhanced version using MediatR at Figure 5.3. Please note that the illustration makes it seem as the mediator is some singleton service and a single point of failure, this is not the case. The mediator is scoped to the request lifetime, a new one is created for each request - like with the Command/Query handlers. It's job is to abstract away the communication between the Controller and the Application layer, creating incredibly light controllers (as seen on Figure 5.4) and also allowing additional logic to be added in the form of "Behaviors".

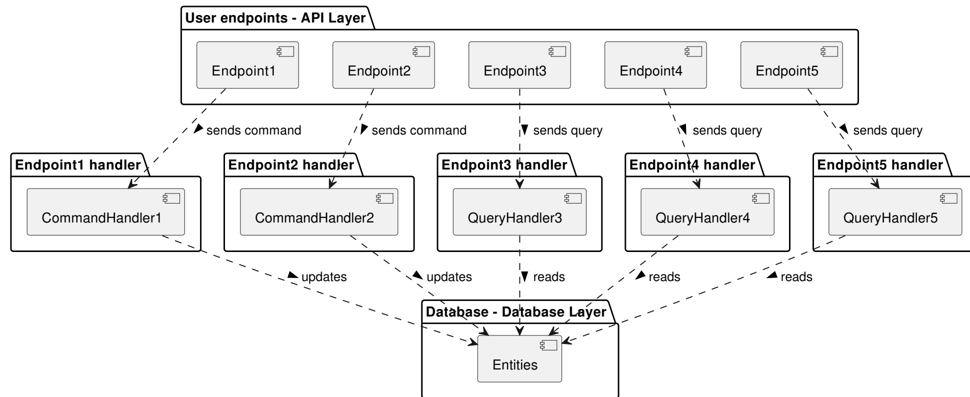

Figure 5.2: CQRS basic approach

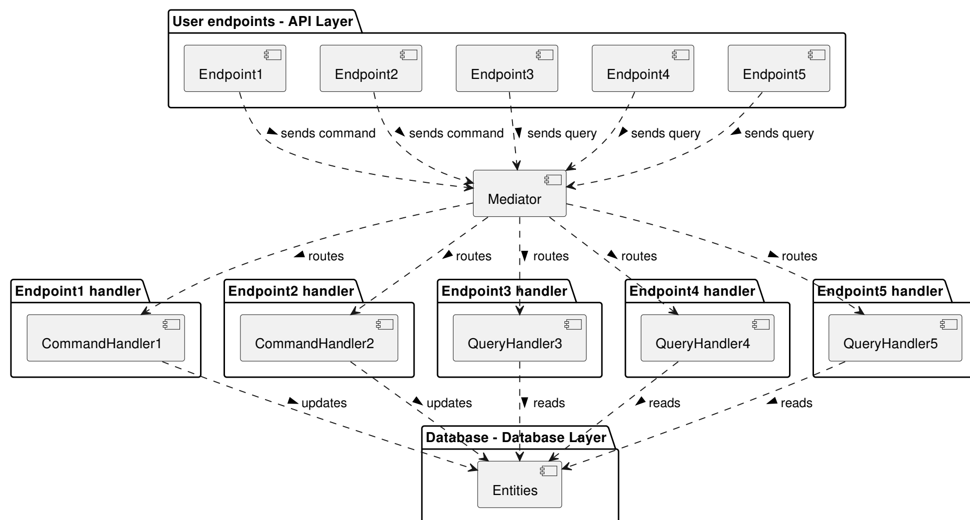

Figure 5.3: CQRS with Mediator approach

```
[HttpGet(template: "{taskId}/items/{itemId}")]
[ProducesResponseType(statusCode: StatusCodes.Status200OK)]
[ProducesResponseType(statusCode: StatusCodes.Status404NotFound)]
[AuthorizeRoles(Role.Chief, Role.Storekeeper, Role.Organizer)]
Duc Minh Pham
public async Task<DeliveryAcceptItemDTO> GetDeliveryAcceptItemInfoById(GetDeliveryAcceptItemInfoQuery query)
{
    return await _sender.Send(query);
}
```

Figure 5.4: Squid - Implementation ”thin” controllers

When the controllers are this light-weight, we can basically "call" the endpoints from within another use-case by creating a new Command/Query object. This is a much better approach than implementing an application service, because these internal requests cannot skip the validation steps. In our implementation, all the controllers do is swagger documentation (return codes, comments) and authorization role specification.

What are behaviors?

MediatR behaviors are "decorated" (Decorator pattern referenced in my previous thesis[41]) around the logic of the handlers. They are a powerful feature that allows developers to implement cross-cutting concerns across all requests that go through MediatR. Behaviors are essentially middleware components that can execute code before and after the next component in the pipeline, which is typically the (Command/Query) request handler. This capability is very similar to ASP.NET Core middleware, which is discussed in the next chapter section 6.

An implemented use case is global validation handling, which is executed at the start of any request (if the request has a validator). But this can be expanded like global logging per request, exception handling or caching of Query requests etc...

DDD with MediatR

While unfortunately, I was not able to define much of the Domain in DDD, mostly because of the commitment to FR10, which disallows database changes. The support is there to define higher-level processes inside the "Domain" layer of the clean architecture. MediatR (IPublisher) can then be used in conjunction with EFCore's database transaction interceptors to dispatch domain events for specific operations with the database. This creates a clean environment to define the processes without coupling to lower-level interactions like messaging, this can rather be handled in the Application layer with INotificationHandlers which can handle the specific domain events.

It is important to note that with DDD it is easy to go the route of the "Anemic Domain Model" anti-pattern, where the developers define most of the logic inside the application layer and use the entities only for data. This is discussed in an article by Martin Fowler [42]. In my experience, this anti-pattern is a common pitfall, but not a big problem in simple use-cases.

5.4 Validation

One of the use cases for the mediator pattern is validation. Validation is a cross-cutting concern within the application architecture that must be consistently applied independent of application business logic.

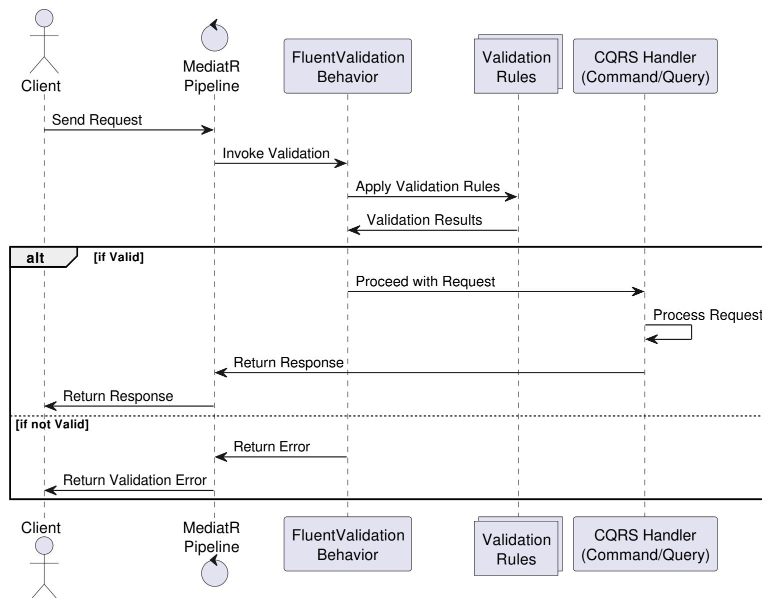

Figure 5.5: Validation pipeline behavior

These validators are fully integrated with dependency injection, giving them access to a wide range of services, including the database necessary for complex validation scenarios. To match the error handling customizability of Octopus, custom error codes and messages can be returned from the validations, reflecting the exact validation error. Validation should not be handled in the controller. It shall be handled using generic pipelines allowing further extension like more validators, validator inheritance and recursive validation.

The validators are implemented as plug-ins, and are automatically invoked if defined for any command/query objects. This utilizes the mediator and CQRS patterns to full potential. The developer can create a validator for any request coming into the system, including requests coming from the application itself! By defining validators in its own separate layer we follow the single responsibility principle and remove the responsibility from the controllers. An example flow can be observed at Figure 5.5.

FluentValidation

A .Net library with fluent API allowing verbose and customizable validation definitions - [43]. This library has implemented many common validation rules, which are very robust. They also implement asynchronous interfaces allowing support of IO operations like validation via the database. It supports automatic documentation generation, allowing the rules to be reflected in the Swagger UI.

5.5 Database

The database is an integral part of the solution, but as we discussed in previous chapters, one of the requirements is FR10 - the backward compatibility with the Octopus database. In this solution we are using Entity Framework Core (EFCore) [44] developed by Microsoft as the ORM of choice. This ORM is incredibly powerful compared to Doctrine on the PHP side and therefore can greatly simplify many of the interactions.

We are also leveraging the Automapper and Entity Framework synergy with projections (ProjectTo Queryable extensions). This minimizes SELECT N+1 query problems, which are fairly common when using ORMs.

Scaffolding the database

The database needs to be adopted to be used with Squid. Thankfully, EFCore supports "scaffolding" or "reverse-engineering" [45], this is a functionality by the library to generate supporting code that can be used to interact with the database. In our case, this generates entities and the DBContext, which is used to interact with the database.

As the application will be executed in parallel with Octopus (FR12) the application needs to also support the database in future states. Therefore, the application needs to support future Octopus database migrations. A script was provided to re-run the scaffolding when needed. This approach is not fool-proof and requires some effort to sync up the changes from Octopus. This information was communicated to the team, as Squid needs to be synchronized with Octopus at the database level.

However, from my observations, Octopus has only made minor changes to the database recently. Therefore, it might be a better approach to synchronize manually by reflecting the changes in the EFCore code.

5.6 Authentication and Authorization

When talking about security it is important to distinguish the differences between these two terms.

- Authentication: Determines "who" someone is
- Authorization: What the person can do in the scope of the application (permissions)

In ASP.NET Core there are abstractions encapsulating "Claims" of a user, usually properties like "name", "email", "date of birth" are formed around the user's identity. This is a fair bit more granular than only role-based authentication, it is named a "ClaimsIdentity" which can contain several claims. This concept goes one step further in something called "ClaimsPrincipal",

which can contain several ClaimsIdentities. This is a set of Identities one user might have which together serve as the identifier of the user. In code, we have abstracted away from this, enabling DDD to be applied to the user as a stand-alone entity. I recommend this well made article by Andrew Lock on the subject [46].

With Authentication out of the way, Authorization is no less extensible with Claims-based authorization which builds on top of the Claims authentication, by using the data from the ClaimsPrincipal to authorize the user. However here we can discover that Claims-Based authorization is built on top of Policy-Based authorization for which Claims are verified using a Claim Requirement and Roles as Role Requirement.

5.6.1 Porting Octopus Authentication and Authorization

Using the theory above we shall implement what is currently used in Octopus.

Authentication - Angler

Angler uses a fairly proprietary authentication flow using introspection which is drafted at Figure 1.1. To implement this flow, I have chosen to create an application-wide HTTP message handler which can be used to communicate with the Auth service. The handler can create an authorization token for itself to request introspections on user tokens. Making this generic allows this logic to be reused across different use cases to communicate within the stack as an authorized user.

The result is a back-channel Httpclient that is used for the introspection requests, allowing us to retrieve ClaimsIdentity for each access token, a sequence diagram is at Figure 5.6. This allows us to consolidate a ClaimsIdentity for each user.

Authorization

In Octopus there are a set number of roles, outlined in Table 1.2.1. Using the ClaimsIdentity object consolidated in the authentication part we can use the role-based authorization system [47]. To replicate the authorization on an endpoint-per-endpoint basis. For example "/manufacturers" GET is allowed for roles "ROLE_CHIEF", "ROLE_STOREKEEPER" and "ROLE_PACKER", but the POST endpoint allows only "ROLE_CHIEF". This replicates the functionality of role-based authorization of Octopus. An example of the GET endpoint is shown at Code listing 5.1 along with the attribute needed to replicate the functionality from Octopus.

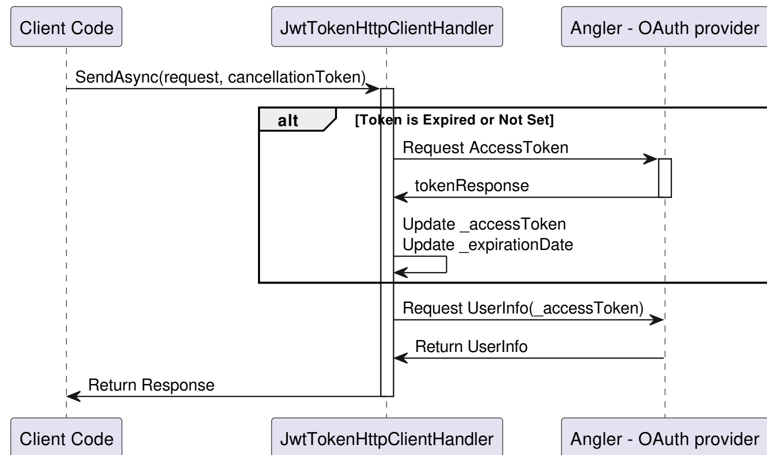

Figure 5.6: Authorized back-channel httpclient

```
[AuthorizeRoles(Role.Chief, Role.Storekeeper, Role.Packer, Role.
Organizer, Role.ExternalHamster)]
public async Task<PagedResult<ManufacturerDTO>> GetManufacturers(
    [FromQuery] GetManufacturersPaginatedQuery query)
{
    return await _sender.Send(query);
}
```

Code listing 5.1: Role-based authorization using Angler

5.6.2 Permission-based Authentication and Authorization

As part of FR5, we are implementing a proof of concept permission-based authentication and authorization using Keycloak.

What is Keycloak

Keycloak is a standalone authentication server, not dissimilar to Angler, where applications can delegate authentication and authorization. Keycloak supports standard protocols like OAuth for integration with web-based applications.

Authentication

Instead of Angler, the user is routed through Keycloak's login page. After authentication, Keycloak provides access tokens in the form of JWT, which contain their roles and permissions. Keycloak's JWT differ from the Angler tokens by having a verifiable signature, therefore not requiring a separate (introspection) call to the authentication service from the backend to verify the access token.

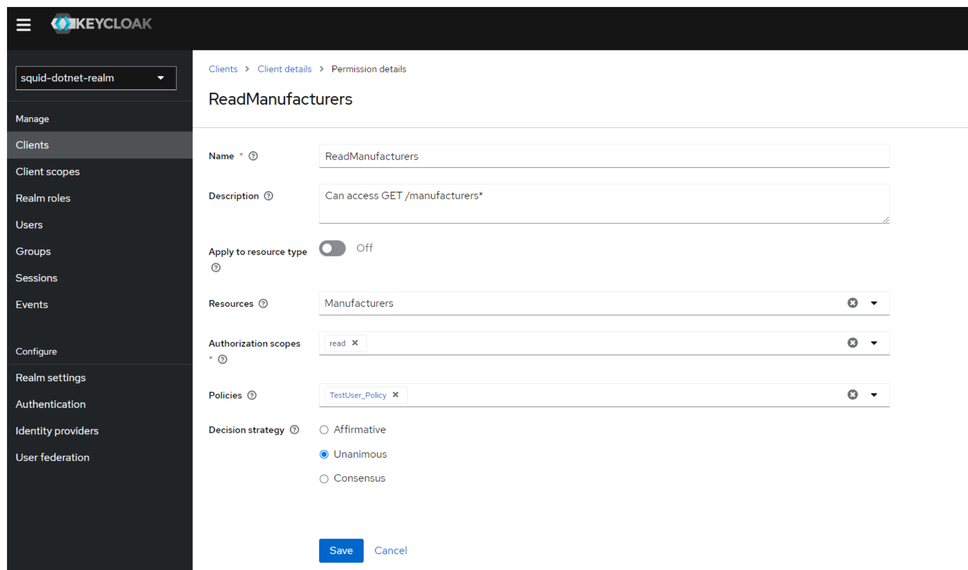

Figure 5.7: Keycloak - Permission policy

The verification is completely handled by .NET's built in support of JWTs, in the background the the JWT is decrypted revealing the "authority" of the token, the application then goes to discover the public key and algorithm (this can be cached) of the authority to verify the signature.

Authorization

Keycloak supports the standard role-based authorization that Angler supports, meaning it can at least replace the functionality of Angler. On top of that it supports claims-based authorization, where trusted claims can be added to the JWT and permission-based authorization. Using permission-based authorization Keycloak uses defined policies - which use a specific policy language documents configured on the Keycloak admin console.

An example of such a policy can be seen on Figure 5.7. This permission is defined as a combination of 3 components:

- Resource: Manufacturers endpoint - this is what Keycloak is protecting
- Scope: Read - I have defined read/write, but can be different operations
- Policy: This is where one can specify for whom this permission applies, either a role, a claim or even a specific user as I have set here "TestUser"

The permission is easily defined in code, thanks to the authorization abstractions in place, an example shown at Code listing 5.2. You might note that the role-based authorization can be kept from Angler because roles can be supported solely by Keycloak if the data are replicated over from Angler.

```
[AuthorizeRoles(Role.Chief, Role.Storekeeper, Role.Packer, Role.
Organizer, Role.ExternalHamster)]
[Authorize(Policy = Permissions.ReadManufacturers)]
public async Task<PagedResult<ManufacturerDTO>> GetManufacturers(
    [FromQuery] GetManufacturersPaginatedQuery query)
{
    return await _sender.Send(query);
}
```

Code listing 5.2: Permission-based authorization using Keycloak

Audience - security measure

An example of unlimited audience - which is default behavior:

1. JWT is requested by the an attacker
2. The JWT was meant to be used on Octopus, but due to the unlimited audience, the attacker uses the JWT to access a different application (Application B)
3. Application B might interpret the roles and claims differently (perhaps roles are reused), the attacker receives access to application which was not intended

Keycloak has support for audience, its documentation is available at [48]. With this set up the roles in Keycloak carry a ”scope” for which the role is used (for example squid-atlantis). The audience can be set up to be added automatically based on the roles of the user requesting the JWT, however the protocol states that an audience cannot be granted to the application that is requesting it, in cases of low trust like this. Therefore there are two clients, one for the frontend to issue new JWT tokens for Squid (Audience: atlantis-squid) and the second client is for Squid to communicate with Keycloak on permission validation. A diagram describing the flow is at Figure 5.8.

Why not use it now?

It is proof of concept because it will not be used in the foreseeable future, due to the coupling with Octopus and its authorization system - Angler. If this was used as the sole authorization mechanic, the Strangler implementation would become unusable due to the mismatching access tokens used during the request.

An example: The client uses an access token retrieved from Angler on endpoint ”/manufacturers”, the strangler implementation will clone the request to Octopus, and to Squid. If we used the Keycloak integration here, the request on Squid will return 403 (Forbidden) due to the unrecognized access token.

This keycloak integration can be used if:

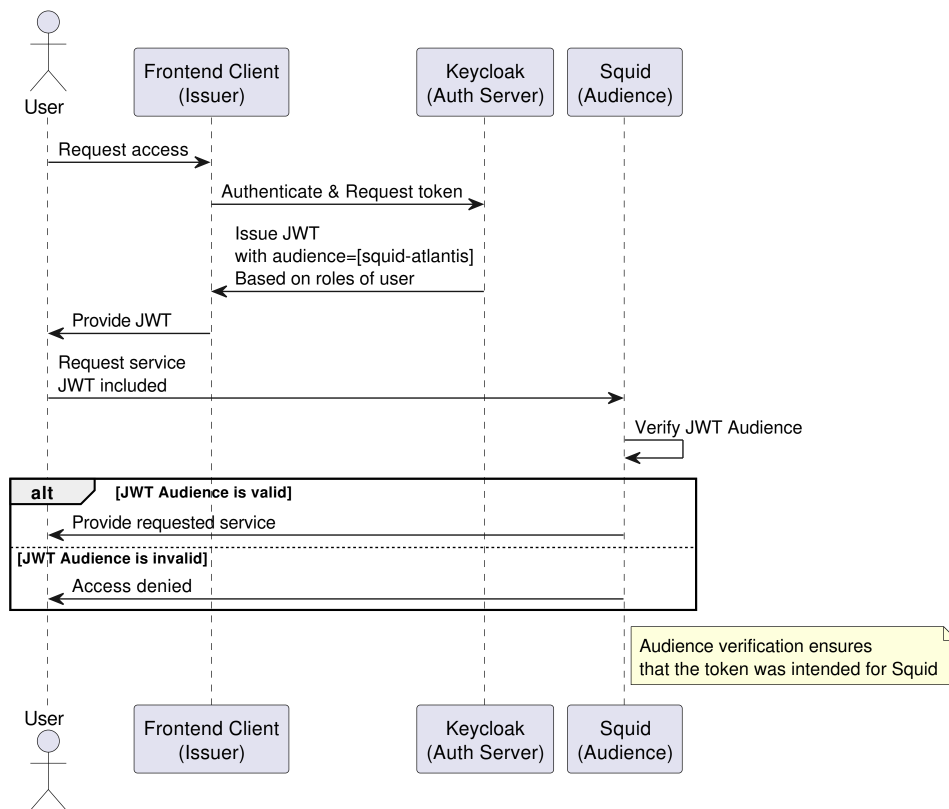

Figure 5.8: Keycloak audience validation

1. Octopus is fully migrated to Squid, Angler will be decommissioned
2. Octopus will be changed to support Keycloak instead of Angler, decommissioning Angler
3. Swordfish - the frontend will be changed to allow some hybrid authentication approach with access token in headers (Angler) and access token in cookies (Keycloak)

Developer feedback

This was implemented to fulfill one of the concerns raised during Phase 2 of the Atlantis analysis in chapter 1. The concern was that a new system that could not support this more granular permission system was doomed from the start, as this was the natural progression of the authorization system to support further growth of the system. The previous role-based system was not sufficient.

A one-on-one interview was conducted with the senior developer who raised the concern, and this implementation fulfilled his expectations.

Conclusion

The functionality is implemented and it is highly abstracted thanks to .NET interfaces like ClaimsPrincipal. I have implemented a simple kill-switch in configuration named "UseKeycloak" which will switch between the OAuth providers Angler/Keycloak. When set up correctly, as shown here, no code change is needed to onboard permission-based authorization.

5.6.3 API Documentation - Swashbuckle library

The goal of the API documentation was to at least match the granularity and flexibility of Octopus's API documentation, in addition making it as automatic as possible. This was achieved by using the natively supported Swashbuckle.AspNetCore [49]. The library has a simple setup and is very configurable. It automatically generates an interactive Swagger documentation which can be used as a basic interface for the backend. The implementation matches the flexibility of Octopus with some added benefits:

- The documentation site acts as a usable interface, needing only a JWT token to be fully functional. (OAuth login is also ready to be used[‡])
- The model is auto-generated using the validation rules of FluentValidation which are used in the validation layer, providing detailed rules for each property.
- The documentation can be expanded by using XML comments around the code [50], which are automatically rendered in the SwaggerUI.

The API Documentation can be leveraged as a tool for showcasing the application, like how it served Octopus. But, with the additional functionalities it can be fully utilized for testing and development.

5.6.4 Logging

Logging will be a very important part of this application. In general, logging provides insight into the behavior of the application when deployed. Not only does it help with debugging, but it allows us to monitor the application's performance, fulfilling (FR8) and possibly (FR7).

We will be implementing the current best practice in the field by using structured logging. With this approach, the log data is formatted in a way that is easy to query and analyze.

[‡]The current OAuth provider - Angler does not support CORS on its token endpoint. This can be used with a different provider like Keycloak.

Implementation in .NET

We will be using the Serilog library [51]. This library ingegrates structured logging into a .NET environment. It supports various sinks (outputs) which can be configured to output the logs in various formats. For example A console sink outputs to console, a file sink outputs to a file etc... The library integrates with the built-in .NET logger abstractions [31] and expands them.

Grafana integration

I have asked the team for access to the logging platform used by the other applications developed by the company. Which was Loki in combination with Promtail, these components gather logs directly from the docker container logs. These logs are consolidated and persisted in Loki, which can be queried using Grafana. I was surprised to learn that structured logging was not utilized, rather the logs were collected in plain-text form.

This type of logging enriched by additional data enables detailed monitoring and custom dashboards to be built in Grafana. This is heavily utilized in the next chapter about the Strangler façade implementation, allowing us to quickly troubleshoot migration implementations compared to Octopus.

CHAPTER 6

Implementation - Strangler façade

In this chapter we will discuss the technologies used to implement the Strangler façade. The concept of the strangler façade has been discussed in chapter 3. Let us summarize how middlewares works in .NET in particular and how they can be utilized for the Strangler façade implementation.

Key characteristics of a .NET middleware

Middlewares are components in the execution pipeline as seen on Figure 6.1.

- Modular: There can be several (or zero) middleware registered in the execution pipeline, adding and removing a middleware is done during the startup of the project

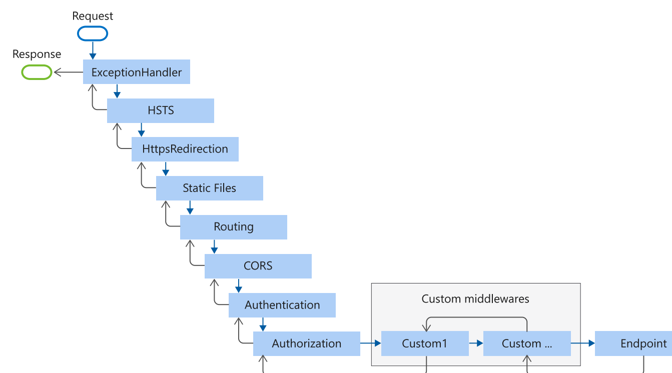

Figure 6.1: Middleware - order of execution [6]

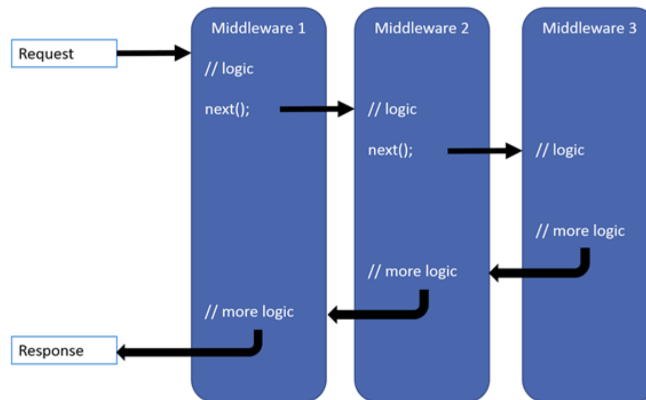

Figure 6.2: Middleware structure [6]

- Ordering: The order can be depended on, and it reflects the order in which the middleware were registered. It is important to consider this as one middleware might depend on another
- Dependency injection: Middleware has access to objects and configurations available through dependency injection

It is important to consider that we will be implementing o ne of the few middlewares of the solution, but many of the functionalities of the .NET platform like authentication, authorization, or Cross-origin resource sharing (CORS) handling, are done through already implemented middleware. Which are implicitly part of the execution pipeline.

Middlewares are incredibly powerful tools for handling application-wide cross-cutting concerns, but they should, in my opinion, be used for very specific scenarios where one needs to operate on the HTTP headers, status codes, or responses directly. For use cases tied more to the application logic, MediatR behaviors should be used.

Let us list what the middleware can do Figure 6.2:

1. It can pass to the next component in pipeline
2. Can execute code before and after the execution of the next component in the pipeline
3. The middleware can short-circuit the execution and not call the next component

We will be using all three of these methods in the implementation of the Strangler façade middleware.

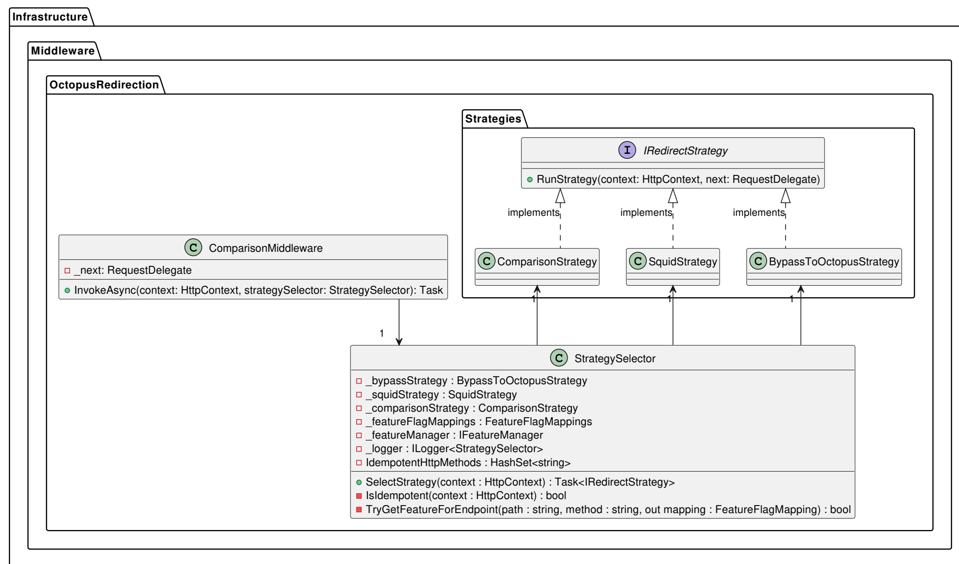

Figure 6.3: Middleware - Strategy selector

6.1 Modes of the Strangler façade

The strangler façade has 3 modes:

1. Octopus mode: The façade redirects the request to the legacy Octopus system, the middleware short-circuits the pipeline and the rest of the Squid application is no longer executed.

2. Comparison (ghost) mode: The façade clones the request and sends an identical request[§] both to Octopus and to Squid. The responses from the applications are intercepted and compared. The response from Octopus is sent back as response regardless of the comparison result.

3. Squid mode: The façade allows the request to pass through the middleware, executing the rest of the pipeline as if the façade was not there.

Strategy selector

The Strategy selector component selects which mode to use on an endpoint to endpoint basis. The component follows the strategy design pattern. This is one of the design patterns that I have expanded upon in my Bachelor's thesis[41]. The structure is at Figure 6.3, and an example sequence diagram for a GET endpoint on "/manufacturers" is at Figure 6.4.

An example configuration:

[§]The requests are not quite identical, the headers need to represent from which Host the request originated. Allowing correct URL generation.

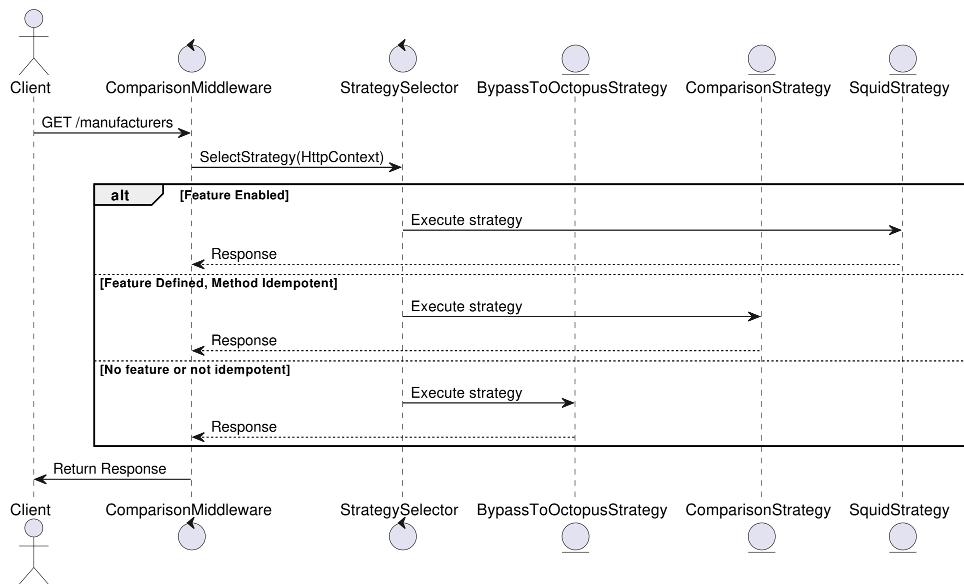

Figure 6.4: Middleware - Strategy selector sequence diagram

```
"FeatureFlagMappings": {
  "Mappings": [
    {
      "Flag": "Manufacturers",
      "Endpoints": [ "/manufacturers.*" ],
      "Methods": [ "GET" ]
    },
    {
      "Flag": "GetTasks",
      "Endpoints": [ "/tasks$", "/tasks/\\d*$" ],
      "Methods": [ "GET" ]
    },
    {
      "Flag": "GetDeliveryTasks",
      "Endpoints": [ "/tasks/delivery-accept.*" ],
      "Methods": [ "GET" ]
    }
  ]
},
"FeatureManagement": {
  "Manufacturers": false,
  "GetTasks": false,
  "GetDeliveryTasks": false
}
```

Code listing 6.1: Traditional Assert Example

One might note that the endpoint HTTP paths and corresponding methods are completely flexible and are defined using regex. With this type of adjustability, it is entirely up to the developer on the scope of endpoints to divert to Squid. The developer is fully in control of which strategy to execute for which endpoint. This configuration can be changed at any time, even

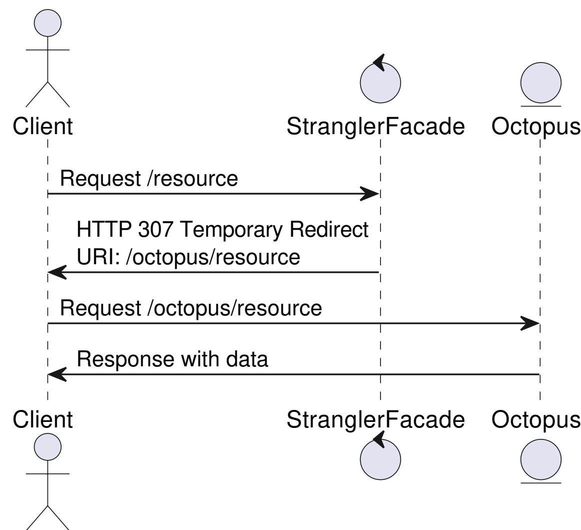

Figure 6.5: Strangler - Octopus mode

during live execution of the application, but I would recommend keeping this configuration as part of VCS to ensure consistency and versioning.

6.1.1 Using Octopus mode

The Octopus mode is the default fallback for the implementation. If the endpoint is not implemented or not configured in the configurations. The strangler façade will automatically choose the BypassToOctopus strategy, completely skipping all of the logic in the application. This is by design.

This mode is critical to the whole idea of the Strangler pattern and enables requirements NFR1, NFR4, FR13 and NFR8. Implementation-wise, it is by far the simplest (also by design). Having bugs here would be unacceptable and a massive risk to carry. We therefore, went through many hoops to allow the simplest implementation possible by returning a 307 Moved Temporarily HTTP status code. With this implementation, it would be up to the client to redirect to another API endpoint. This came with some integration issues, which I will discuss later in section 6.2.

Nonetheless, this solution utilizes the nature of redirects built into many clients, including Axios, which is used in the Frontend Vue javascript application - Swordfish. An exemplary sequence diagram is at Figure 6.5. For the readers who are concerned with performance of this ”jump”, adding an additional round-trip does have an effect on network latency (though minimal). There are solutions to this, for example, by changing the 307 Temporary redirect to a 308 Permanent Redirect, which is cached. Note that this might make the changes less reactive as the cache needs to be invalidated on the client side for changes to take effect if this approach is taken.

6.1.2 Using Comparison mode

The implementation reflects the proposed design at Figure 4.9, but the implementation is a fair bit more complicated than expected and consists of several steps and each of them needs nontrivial implementation:

1. Request cloning: When a request is received, the system clones the request and sends one copy to Octopus and another to Squid. This step is required to ensure both systems get identical input requests.

2. Response intercepting: After the requests are executed, the responses are intercepted from both systems.

3. Deep JSON comparison: The responses are analyzed and result of this step is a validation report, with the differences of the two responses if there are any.

4. Logging and Visualisation: The results of the previous step are logged to Loki and visualized using Grafana in a dedicated dashboard, enabling excellent monitoring capabilities.

5. Response routing: For safety, the response that is returned to the client is always the response from Octopus. This makes sure that the system behaves as expected during the transition process, while collecting real-time valuable data.

We might notice that this workflow is impossible with non-idempotent requests. An example would be a HTTP POST request, which would be replicated to both systems, but because they operate on the same database, this would result in duplicate records. This is a limiting factor for this approach, and there is a check in code for this not to happen unintentionally. However, the developer is free to override these safety precautions.

Request Cloning

In this step, the incoming request is cloned and sent to Octopus, special care is taken here to be as safe as possible. Let us break down the considerations taken:

- Handling headers: During the cloning of the requests we must copy the headers. One would expect that copying all headers will be beneficial, but opposite is true. We instead implemented a whitelist header system, which is configured currently to copy Authorization headers and Host headers, for why this is discussed later in section 6.2. Host header is required because Octopus utilizes URL reflection to generate new URLs like ”object_link”.

- Network: Also discovered after the mishap with Nginx self DDOS in section 6.2 is the use of internal docker network instead of an exposed domain.

- Reliability: One of the main requirements are NFR1, NFR4 and NFR8. These refer to the safety of the system when in production, we want to minimize the impact of these processes on the overall functionality of the system. We have implemented this to be as robust as possible to handle exceptions during Squid requests. All occurring exceptions are logged and traffic is redirected through Octopus, ensuring that errors do not impact the quality of the service.

Logging and Visualisation

In this step, the differences in the responses between Octopus and Squid are logged and visualized. Here, we are bearing fruit from the integration with Grafana and the structured logging we provided in subsection 5.6.4. We are able to create dashboards and group by the endpoint as seen on Figure 6.6.

The dashboard shows how many requests were successful and how many were not, grouped by the endpoint path. The reason why the validation failed can also be discovered by expanding on the logs, which are fully searchable by the endpoint path. This is thanks to the structured logging, which allows Grafana to index the various properties instead of a full-text search and pattern matching.

The dashboard also logs the performance of the Octopus and Squid systems, fulfilling our performance requirements FR7 and FR8.

6.1.3 Using Squid mode

The selector utilizes the incredibly powerful and dynamic Feature management tooling provided by Microsoft [52]. This is represented by the ”FeatureManagement” section in the Code listing 6.1. In the example, this feature is not used and set to ”false”, but this can be enabled to fully migrate to Squid for specific endpoints:

Specific Time window

This will direct all traffic for the GET method on endpoints ”/tasks/delivery-accept.*” for a set time window Code listing 6.2. It can be utilized to test during nonbusiness critical times.

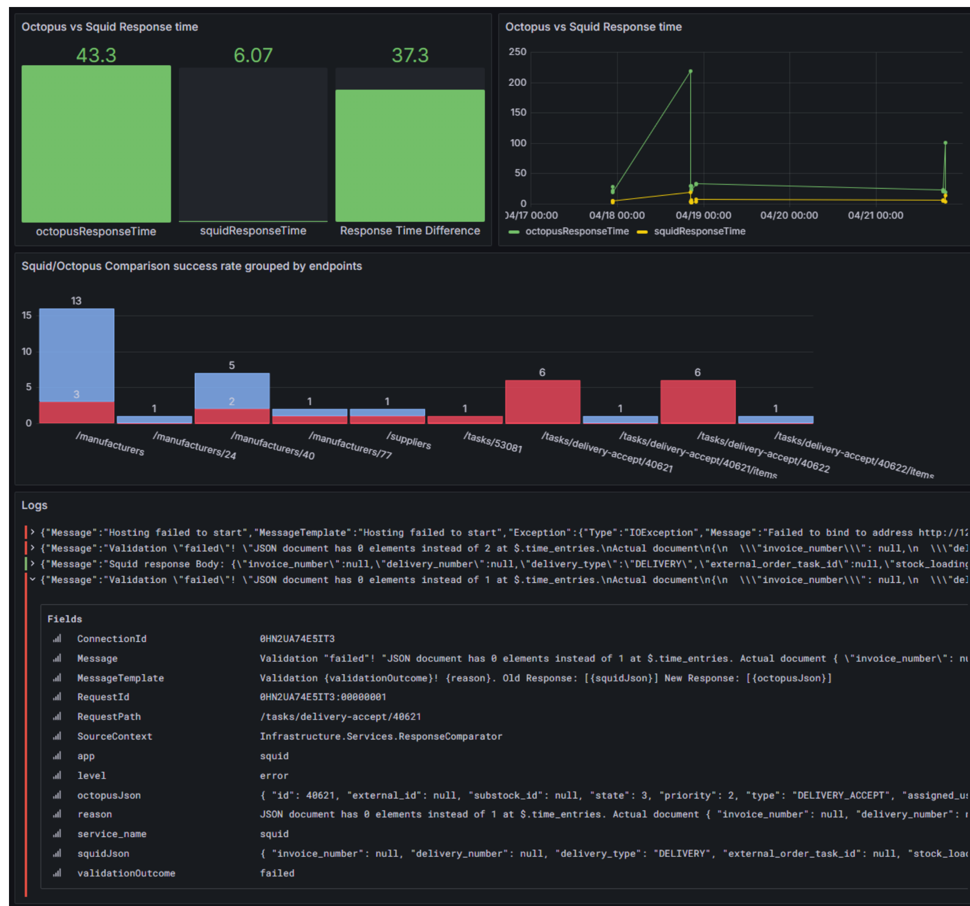

Figure 6.6: Strangler - Grafana Dashboard

```
1  "GetDeliveryTasks": {
2    "EnabledFor": [
3        {
4            "Name": "Microsoft.TimeWindow",
5            "Parameters": {
6                "Start": "Mon, 22 April 2024 13:59:59 GMT",
7                "End": "Tue, 23 April 2024 00:00:00 GMT"
8            }
9        }
10    ]
11 }
```

Code listing 6.2: Strangler Configuration - Time window

Specific percentage

This is for the same path and method, but instead of a time frame, it uses a percentage to determine whether to direct the request to Squid Code listing 6.3. Note that when this is used, the decision is cached, and whether the

feature is enabled or disabled is saved for the duration of the session.

```
 "GetDeliveryTasks": {
   "EnabledFor": [
       {
           "Name": "Microsoft.Percentage",
           "Parameters": {
               "Value": 50
           }
       }
   ]
}
```

Code listing 6.3: Strangler Configuration - Percentage

Specific targeting - User

Again, the same method and path, but this time, the testing is targeted to a specific user or group Code listing 6.4. This works without issue, as we are using Microsoft's authentication and authorization abstractions.

```
 "GetDeliveryTasks": {
     "EnabledFor": [
       {
         "Name": "Microsoft.Targeting",
         "Parameters": {
           "Audience": {
             "Users": [
               "vedouci"
             ]
           }
         }
       }
   ]
}
```

Code listing 6.4: Strangler Configuration - User

Fully migrated

When we are satisfied and want to let Squid handle the requests for the endpoint, simply setting the flag to true will permanently enable the feature Code listing 6.5.

```
 "GetDeliveryTasks": true
```

Code listing 6.5: Strangler Configuration - Enabled

6.2 Implementation details - Integration issues

Here, I would like to point out some integration issues that showed up after the deployment. This was done in collaboration with Max, who facilitated the deployment of the application. The monitoring tooling, especially logging via Loki/Grafana was instrumental in debugging these issues.

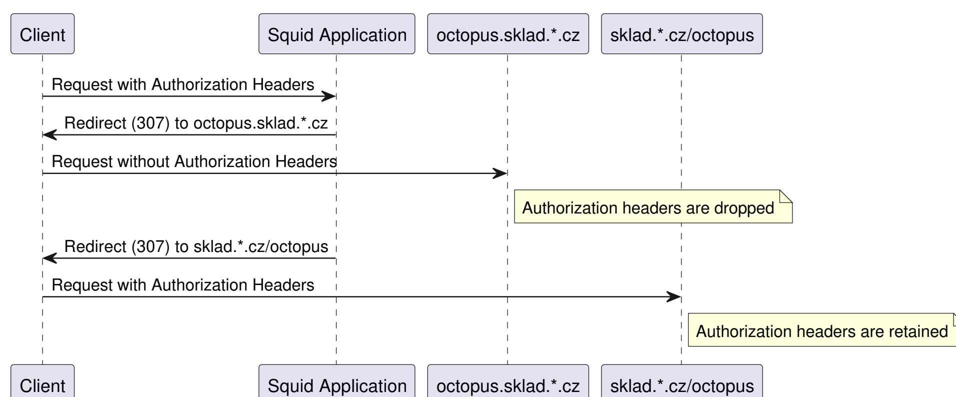

Figure 6.7: Integration - Authorization headers

Integrating redirect calls

Redirecting has worked well when tested on localhost (Octopus bypass mode), but modern browsers have fairly strict rules for redirect, especially when it comes to authorization headers. This manifested in the redirect call to Octopus, always returning 401 Unauthorized even when the Authorization header was sent to Squid. A simplified sequence diagram is shown at Figure 6.7. This was later fixed by not using a different subdomain for the redirect to Octopus.

Self DDOS - Nginx header overload

After deploying the environment and using the Comparison mode. There were weird overload 500 errors coming from the application, followed by hundreds of logs per request. The culprit turned out to be the one component which was not replicated on the development machine - the Nginx reverse proxy. The proxy works by adding headers to each request coming through. Due to a similar issue as in previous section the proxy continually redirected the same request hundreds of times, which rendered the application useless.

The solution was to avoid the reverse proxy completely when using the Comparison mode by using the internal docker network. This additionally greatly improves the performance by reducing the number of jumps to reach Octopus.

Part IV

Testing

CHAPTER 7

Testing

The availability and correctness of the endpoints is the utmost priority. Therefore testing has been critical during the development process. We went above and beyond to provide a complete testing and monitoring suite with quick feedback, simplicity of usage, and flexibility in its employment. Let us start with some testing strategies which were used during the development of the solution.

7.1 Manual testing

While manual tests are not ideal, they are inherently part of any developer's toolbox. It is the simplest and fastest way to test functionality. It is important when doing a proof of concept implementation or a quick prototype application.

Manual tests provide quick feedback for newly written code without the overhead of creating code for the test. Therefore, it is used the most during the early stages of development when features are being fleshed out and rapid iteration is important. Manual testing is useful for catching unexpected behavior or errors that automated tests are not designed for at this phase.

Manual tests were done by myself and my colleague Bc. Max Hejda during our development of the application. They allow us to quickly check whether the various parts of the infrastructure are connected as expected, the authorization does not fail, or whether the documentation is generated how it should be. These tests make sense when they are done rarely and do not benefit from repeated executions. However, as the application becomes more mature, the need for automated testing is needed.

Manual tests are not reliable, they are not repeatable, and are inefficient. They grow out of control as the project grows larger.

7.2 Automated testing

Automated testing is one of the most important parts of the modern software development process, especially as the application grows and the complexity along with it. This method ensures consistent results, repeatability and can massively reduce regression issues.

7.2.1 Basics of automated testing

The benefits

- Speed: Automated tests are quick to execute, which provides quick feedback to the developer, usually before deployments are made. This feedback loop allows programmers to deliver safe code without the need to wait for long manual tests.
- Consistency: Automated tests can be executed the same way every time. This gives consistent results with every test run, which helps to identify regressions and errors quickly, as any change is reported immediately when there is a difference from the expected results.
- Scalability: When the software inevitably grows, it becomes impossible to manually (consistently) test every part of the solution. Automated tests can easily cover large amounts of code with incomparable precision.

Types of automated tests

- Unit tests: Tests focusing on the smallest individual units of code, usually these tests are isolated and can be optimized to be executed in parallel.
- Integration tests: These tests are made to test the interfaces between components within the stack, like the database, Redis, and the Docker network itself. These tests are important to make sure that different parts of the system work together as expected.
- E2E tests: End-to-End (E2E) testing is testing of user interactions with the system. These tests are important to verify the functionality of the whole application from the perspective of the user.

7.2.2 Unit testing

The unit testing is implemented using xUnit.net. *"xUnit.net is a free, open-source, community-focused unit testing tool for the .NET Framework."*[53]. This testing framework is also used by Microsoft's .NET Core team and is (at the time of writing) the most popular unit testing framework for .NET.

The AAA Pattern

Widely used and considered a standard in the industry is the AAA pattern of unit tests:

- Arrange: The test environment and test data are prepared
- Act: The functionality that is being tested, is executed
- Assert: The outcome is verified and compared to expectations

There are exceptions to this rule, but usually, all three sections are needed to validate the functionality of a feature. We will be referring to these sections of the unit tests later, as we optimize and apply more advanced techniques.

Integrating into the development process

Unit testing is most effective when fully integrated into the development process, let us discuss some methods:

- Test-Driven Development (TDD): In TDD, developers are encouraged to write tests before they write the related code. We shall discuss this methodology later in section 7.7.
- Automated test execution: The power of Unit testing comes from the repeated execution and verification. Therefore the development environment (MSBuild) can be configured to run these tests after a successful build.
- Continous Integration (CI): In CI environment (GitLab) the unit tests are run automatically on a shared pipeline, every time a change is committed to the git repository. This helps catch errors before they are merged to the shared branch. The implementation of this will be part of my colleague Bc. Max Hejda's thesis.

7.2.3 Integration testing with throwaway Docker containers

In this section, we will discuss how a reproducible testing environment or "Integration" environment is set up for integration tests. The goal of these tests is to test the external dependencies of the application.

The stack that is replicated contains three images 1. Redis 2. PostgreSQL 3. Octopus

These containers are configured to mimic the environment of a real production instance. Environment variables are modified to be able to run Octopus in this environment.

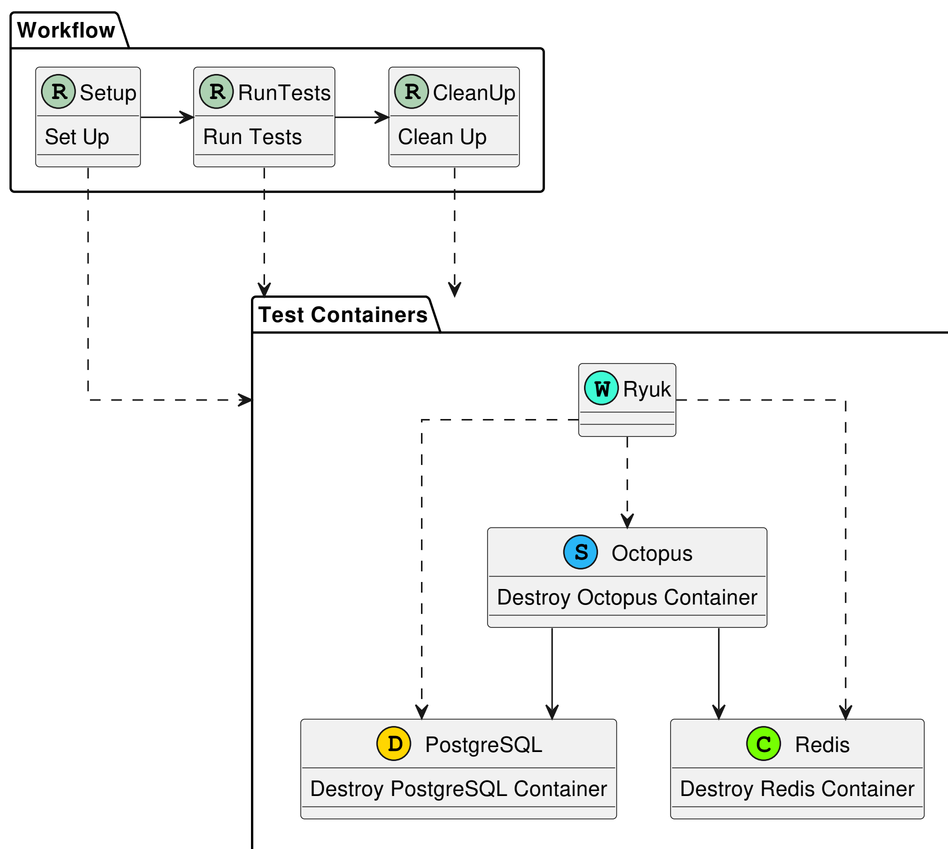

Figure 7.1: Test Containers usage

Testcontainers

We are utilizing the Testcontainers library[54] for the initialization of the docker stack. This library uses the available docker API to orchestrate the setup of the docker network and to manage the lifetime (creation and disposal) of the managed docker container instances. Let us explore the container lifecycle (illustration at Figure 7.1):

- Initialization: Testcontainers spawns a watcher container named Ryuk[¶], and the containers within the configured stack.
- Test execution: Tests are executed on the wired-up containers without any intervention from Testcontainers (the application communicates directly with the containers).

[¶]The name references the Japanese Anime/Manga "Death Note" where Ryuk is the god of death and orders death by writing names into the "Death Note". Similarly, the container keeps track of the created containers and destroys the containers after the tests are finished.

- Cleanup: After the testing is done, the containers are removed. Ryuk makes sure that even if the application crashes, the containers are deleted.

The considerations:

- Environment variables: Surprisingly many environment variables are replicated for the correct startup of Octopus.
- Networking: The Testcontainers library, by design does not work with hardcoded ports. To allow asynchronous execution (to avoid port conflicts), it randomizes ports given after container creation. But the ports within the container itself can be static. Therefore docker network is set up to facilitate communication between containers.
- Waiting strategy: Testcontainers provides some waiting strategies that make sure the container is ready to serve requests before proceeding with the tests. The order of activation is critical for the correct functioning of the Octopus container. Thankfully, Redis and PostgreSQL are implemented by the Testcontainers library as they are fairly common in many applications. However, Octopus's configuration needs to be created manually.
- Healthcheck: The Octopus CI/CD pipeline defines a liveness check that determines when the Octopus container has started successfully. However, this check is insufficient for our requirements. As our tests are run by a machine, we need Octopus to start serving requests immediately. To ensure this, we have implemented a real readiness check that confirms a successful connection to the database, authorization server, Redis and it physically confirms the readiness by polling for a valid answer on one of the endpoints.

Thanks to Testcontainers, we have created a significantly better integration pipeline than by managing the integration environment manually. The integration testing is possible on any docker-compatible API. Therefore, it is simple to run on the local machine or in the GitLab CI/CD pipeline, allowing quick feedback on integration failures.

7.3 Resetting of database between tests

While Testcontainers helps with the environment setup, where it facilitates the availability of a running database docker container to execute tests on. After the throwaway database is set up, its state is on the tests themselves to handle. Let us imagine the following scenario in Figure 7.2. To mitigate the illustrated problem, we can instruct Testcontainers to create a database for each test, but that might be hundreds of tests, which can be extremely resource-intensive.

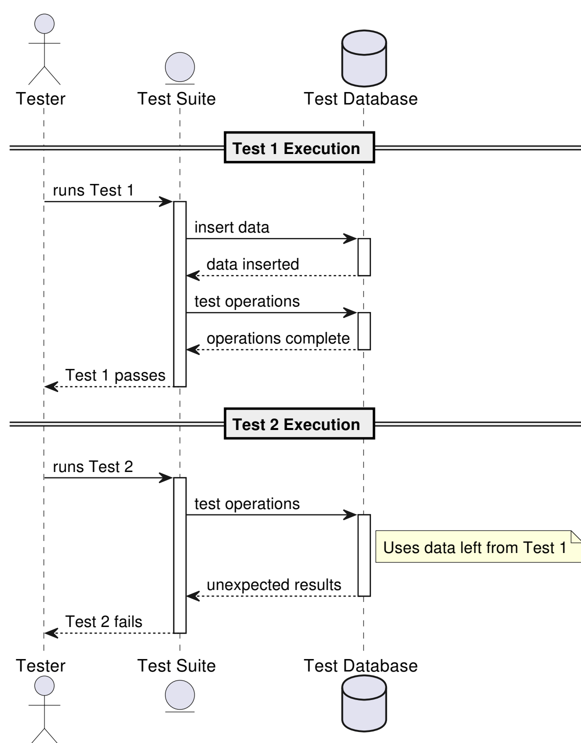

Figure 7.2: Conflict of tests - Database state

Using Respawn to manage Database state

Respawn [55] is a library that is designed to clear databases between tests, which is the exact use case we are looking for. In the background, Respawn connects to the database and keeps track of the tables that are utilized, after a reset is invoked, Respawn generates and executes SQL on the test database to return it back to a clean slate.

7.4 Seeding database - for testing

Since Octopus works with very complex data, with many interconnected relationships in it's database. It is important to mimic these complex datasets for testing purposes for Squid to ensure compatibility with the Octopus database. Unfortunately, as one might expect, manually creating this data is very time-consuming and repetitive after some time.

Let us take the task from the supplier entity diagram as an example Figure 1.4. To test a GET Query, we need to stub the complex object into the database, to test against. An Address, a Supplier, and some example Tasks, ideally more of each in different configurations to demonstrate the 1:n relationships. Each of the entities contains around ten properties, where the

properties need to correctly match valid business format constraints like the correct length of string values, unique email, etc... There are several solutions that might come to mind

1. Automate the generation: There are libraries like Autofixture[56] or Bogus[57], which can be used for this purpose. But after much fiddling around, programming these libraries to produce valid production-like data is very tedious
2. Use POST functionality: We can use the Task create command, which needs to implement the validation of these constraints, but that would couple the GET testing with the POST testing. It would not be possible only to test GET without POST for scenarios where the POST endpoint is not expected to be delivered.

Overview of the problem

1. Fake data is generated for an object - "Arrange" section
2. Data is inserted into the database - "Arrange" section
3. Data is fetched and tested with application logic - "Act" and "Assert" section

Note that the 1. and 2. steps are irrelevant to the testing. They pose as boilerplate code written purely for the test to work. The goal of the test is to verify the functionality of the application layer - the logic behind step 3. However, without steps 1 and 2, the testing is impossible. How can we improve this?

Proposed improvement

As Octopus is already a fully functioning system, we can collect data from the database for testing purposes on Squid. The plan is to create a backup of the database and utilize it to run tests. This approach would greatly simplify the "Arrange" phase of the testing process.

1. Create a Database dump: Capture the current dump using tools like pg_dump to gather the state of the database.
2. Use the database dump: Use the dump to seed the database inside the scope of the test. This would serve as a baseline dataset for all integration tests and unit tests in Squid. This approach would greatly simplify the "Arrange" phase of testing by using a realistic dataset that is captured from a valid database on which Squid will work on.

With the database dump available when the developers need it, the testing becomes very simple. Let us summarize the benefits of this approach:

- Reduce time to set up a test: This will reduce the "Arrange" part of the test drastically by removing the need to insert data manually.
- Tests will be more accurate: Testing data will be the exact data that is used on the development environment, which is proven to be stable with Octopus and, therefore, can be used as a reference.
- Octopus rebasing: Octopus is bound to change at some point in time, along with its database schema. With these tests, simply updating the dump shall provide valid testing of the new database schema on Squid.

For this to work, the database dump will need to be upgraded based on the current live Octopus version to validate the compatibility between the two solutions. The developers are aware of this approach and will need to synchronize the database releases in the future. This is regardless of this database testing approach, this shall only ease the transition.

7.5 Snapshot testing

Due to the complex data structure testing that is needed to confirm compatibility with Octopus, snapshot testing has proven as a very powerful technique to improve efficiency with writing tests. The technique allows us to shorten the "Assert" phase of the unit tests, many times reducing it to a single line of code.

Principles of Snapshot Testing

The specific implementation that is used is the Verify package[58]: *"Verify is called on the test result during the assertion phase. It serializes that result and stores it in a file that matches the test name. On the next test execution, the result is again serialized and compared to the existing file. The test will fail if the two snapshots do not match: either the change is unexpected, or the reference snapshot needs to be updated to the new result."*[58].

Test writing workflow

1. Test creation: Create tests with the standard "Arrange", "Act" and "Assert" structure.
2. Initial run and Fail: On the first test run without an existing snapshot, a failure is expected. This triggers a diff window that highlights the differences between the expected and the actual output.

3. Subsequent runs: If changes were approved and snapshot was created or updated, the subsequent runs will be compared against that snapshot. Tests pass if the current output matches the snapshot, they will fail if not.

```
1   // Arrange
2   var manufacturer = new Manufacturer
3   {
4       ManufacturerId = 2,
5       Name = "test1",
6       Abbreviation = "TM1",
7       CreatedAt = DateTime.Now,
8       CreatedBy = 0
9   };
10  await Scope.AddAsync(manufacturer);
11
12  var query = new GetManufacturersByIdQuery { ManufacturerId =
13      manufacturer.ManufacturerId };
14
15  // Act
16  var result = await Scope.SendAsync(query);
17
18  // Assert - this is what we aim to improve
19  result.Should().NotBeNull();
20  result!.Id.Should().Be(manufacturer.ManufacturerId);
21  result.Name.Should().Be(manufacturer.Name);
22  result.Abbreviation.Should().Be(manufacturer.Abbreviation);
23  result.ExternalId.Should().Be(manufacturer.ExternalId);
24  result.CreatedBy.Should().Be(1);
25  result.CreatedAt.Should().BeCloseTo(DateTime.Now,
26  TimeSpan.FromMilliseconds(10000));
```

Code listing 7.1: Traditional Assert Example

```
1   // Arrange
2   var manufacturer = new Manufacturer
3   {
4       ManufacturerId = 2,
5       Name = "test1",
6       Abbreviation = "TM1",
7   };
8   await Scope.AddAsync(manufacturer);
9
10  var query = new GetManufacturersByIdQuery { ManufacturerId =
11      manufacturer.ManufacturerId };
12
13  // Act
14  var result = await Scope.SendAsync(query);
15
16  // Assert - Improved
17  await Verify(result);
```

Code listing 7.2: Assert with Verify example

The result will be a snapshot similar to Figure 7.3. This serialized object will be used for subsequent tests. If there is a discrepancy, a diff window will be prompted Figure 7.4, which will show the exact difference. Note that the window shown is the Rider diff tool, but any diff tool can be used, in fact even

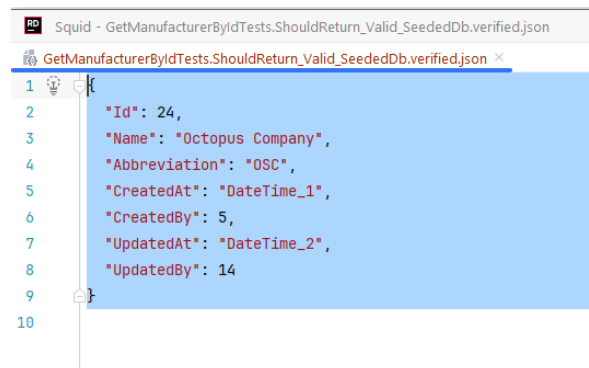

Figure 7.3: Snapshot testing - snapshot example

a manual text editor can work here. The output is a verified serialized JSON representation of the tested object.

The snapshot is now also part of the git repository and is available to be checked as part of code reviews by other team members.

7.6 Tying it all together

Let us summarize the implemented techniques:

1. Database seeding - Dev environment database dump
2. Testcontainers - throwaway Docker containers for testing
3. Snapshot testing - first result is serialized, subsequent tests are compared with the serialized version

With this setup we have very powerful tests which do not require much work at all - with code at Code listing 7.3, and snapshot generated in Figure 7.3.

```
[Fact]
public async Task ShouldReturn_Valid_SeededDb()
{
    // Arrange
    await Scope.SeedDatabase();

    var query = new GetManufacturersByIdQuery { ManufacturerId = 24 };

    // Act
    ManufacturerDTO result = await Scope.SendAsync(query);

    // Assert
    await Verify(result);
}
```

Code listing 7.3: Sample test - improved

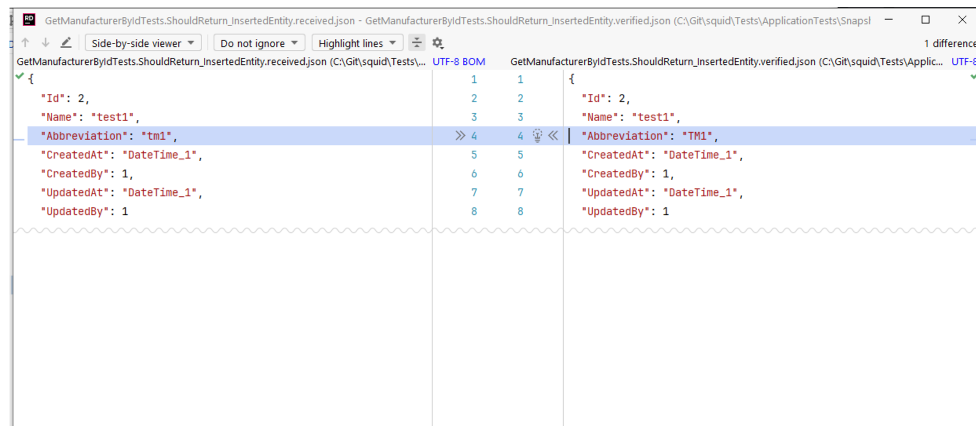

Figure 7.4: Snapshot testing - Diff window

For tests written for the migration effort, we can take this one step further. Let us imagine we want to test the response from Squid for endpoint "/manufacturers". We can leverage all of the techniques above by using a snapshot generated by Octopus.

An example of how a test like that would look like is at Code listing 7.4. In this example test, the initial snapshot is generated by a fully functioning local Octopus instance, which is spun up on Testcontainers (on-demand) for the purpose of this test and wired up to the database and Redis. This snapshot is then used as a reference when compared with the response from Squid.

This setup, along with the incredibly powerful techniques above, utilizes the incredible .NET in-memory test server [59] to serve requests for testing.

```
1   [Fact]
2   public async Task ShouldReturnAllManufacturers_Paginated()
3   {
4       // Arrange
5       await Scope.SeedDatabase();
6
7       // Act & Assert
8       await Scope.VerifyAgainstOctopusApiResponse(
9           $"/manufacturers?itemsPerPage=10");
10  }
```

Code listing 7.4: Sample test - fully optimized

With the ongoing migration, the endpoint responses from Squid are required to be fully backward-compatible with Octopus. With this testing suite, we can achieve this goal with much greater speed, while mitigating risk by human error.

7.7 Test Driven Development - TDD

With TDD, one defines the test first before writing the tested code. The test can be extremely simple, such as the existence of a function or the return of a specific reply based on input. But even these simple tests provide important insight into the development process. The methodology pushes the developer to design the components first before diving into the code.

TDD provides a more disciplined approach to programming. By following the principles, the developers are encouraged to think about the structure and the naming before the functionality of the components. Martin, the author of the book ”The Clean Coder: A Code of Conduct for Professional Programmers” [39] highlights TDD as an essential part of a professional programmer’s toolkit.

With the implementation of several testing strategies, we shall motivate the developer working on the migration, to go with TDD.

Fluent assertions

The assertions are enhanced by incorporating ”Fluent Assertions”[60].

Traditional Assertion Example:

```
string ValueA = "Some text";
string ValueB = "Different text";
Assert.IsTrue(ValueA == ValueB);
```

```
Error:
    Expected: True But was:  False
```

Fluent Assertion Example:

```
string ValueA = "Some text";
string ValueB = "Different text";
ValueA.Should().BeEquivalentTo(ValueB);
```

```
Error:
    Expected ValueA to be equivalent to "Different text" with
    a length of 14, but "Some text" has a length of 9, differs
    near "Som" (index 0).
```

In my experience, the more complex the test, the greater the effort put into communicating with future developers about why the test fails, the initial constraints, and why this test must succeed. The verbose and readable messages greatly improve the process of debugging, sometimes removing the need to execute the test with a debugger active.

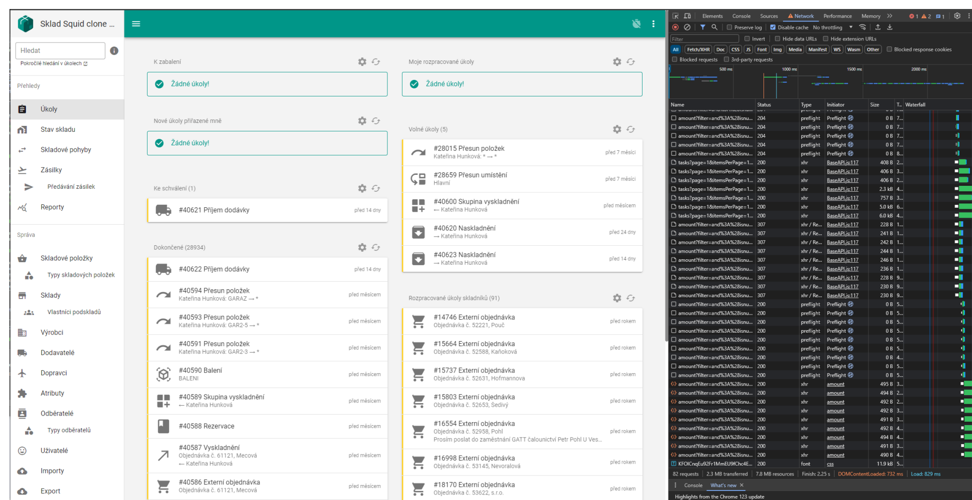

Figure 7.5: E2E testing - Production clone environment

7.7.1 Summary

By utilizing Snapshot testing, Database seeding, and Octopus test containers, the integration testing consists of only defining which endpoint of Squid and Octopus to test. There is a symbiosis that emerges from the use of these very powerful technologies the result is greater than the sum of its parts.

7.8 E2E Testing

As a final test, we are performing full end-to-end testing by deploying the whole stack, including the front end, onto a production clone environment. This environment contains a clone of the database with real data. But instead of Octopus the front end is pointed at Squid.

The testing was successful, showcasing the redirecting approach working as expected, which can be seen on Figure 7.5 as 307 redirects. These redirects instruct the front end to fetch data from Octopus for endpoints that are not fully migrated.

7.9 Developer feedback

After the architecture was implemented and the project was set up, I conducted several interviews with the team on feedback for the implementation to gather ideas on what was done well, what to improve, etc. This was the phase 4 of the overall analysis done as charted in chapter 1.

In the following section, I have given each participant of the research a code change to perform in Squid. The goal is to gauge how maintainable the new architecture is and whether productivity and testability were improved.

The questionnaire

1. What is your experience with .NET and PHP, if relevant provide other languages you are familiar with. Do you have any experience with Octopus/Atlantis? If so, please describe how extensive it is.

2. When performing the code change please note down information that you needed to look up, what interested you, what you needed to find out. - does not need to be structured

3. Describe the process of testing your implementation, the process of finding out that it really works as it should.

4. Please describe how long it took you to make the changes, whether it is sustainable or not, whether you found it comfortable to work in this environment, and what could be improved?

The three developers who participated in the questionnaire are developers who will be maintaining the solution in the future for the company.

The Lead developer of Octopus

Due to time pressure, change was not made on Squid, but thanks to the extensive knowledge of Octopus, the developer provided some estimates of the time needed for the implementation. I was on hand to show the relevant code that needed to be added in C#.

1. Experience with PHP/Symfony for 4 years, .NET 1 year of experience. Worked on Atlantis for 4 years and is responsible for the application

2. The user looks for documentation of the project, struggles with startup of the project

4. An estimate was given for a fairly straight forward CRUD endpoint migration at Table 7.1. While these are rough numbers, much of the time saving comes from the powerful database framework, Automapper, and validation pipelines which can save a (proportionally) large amount of time compared to PHP/Symfony.

Table 7.1: Simple CRUD migration estimate

Endpoint	Octopus estimate	Squid estimate
DTO object	40 minutes	15 minutes
GET Paginated	1 hour (+ 30 minutes documentation)	30 minutes
GET by Id		10 minutes
POST	1 hour 10 minutes	30 minutes (+ 10 minutes validators)
DELETE	20 - 30 minutes	10 minutes
PUT	40 minutes	15 minutes
Total	4 hours 30 minutes	2 hours

Mid-level engineer

For this category, I asked my colleague Bc. Max Hejda, to provide his point of view on a change he made during the project.

1. ”I have minimal experience with .NET (actually, this is the first major work I’ve done in it - I did a few minor things on another .NET project). With PHP, I have quite extensive experience from both work and school-related and pseudo-school projects. I work with Nette, Symfony. I did a few smaller development tasks on Atlantis but didn’t know the Atlantis domain much. Additionally, I am actively involved in DevOps operations at Atlantis.”

2. ”I had to get quite familiar. It is actually the second strongly typed language I have used for web application development (the first was Java in Distributed Systems). It was very interesting to get acquainted with EF (probably the most well-developed ORM I’ve encountered so far), AutoMapper.”

3. ”I did the testing using a prepared methodology with snapshot testing against Octopus. Using this method, I tested the GET requests. The remaining operations were tested using integration tests, to ensure the logic matches the underlying code of Octopus.”

4. ”The Stocks CRUD took me about 10 hours including tests. A relatively large part took understanding how CRUD works in Octopus and figuring out how to properly implement this in .NET. The actual implementation was quite comfortable.”

Junior engineer

1. "I have never programmed in .NET or PHP. My primary language is TypeScript, the closest I've come to .NET would be Java and Kotlin, in which I've only worked on school projects and assignments. I have no experience with Atlantis."
2. - "The first issue was with the JaguFilterable dependency (unlike others, it didn't import automatically and even with the correct GitLab login, it required additional manual steps which I needed help with, but eventually, it was resolved according to the readme here)."
   - "The README instructions for importing the DB dump (in Octopus) had underscores instead of hyphens in the container name, which tripped me up for a bit (it's hard to notice)."
   - "On the Wiki of Octopus, it's hard to figure out what each access is for, and there was a wrong link, which was very confusing (thus it wasn't clear at all from where I should copy the access data into Postman)."
   - "Generally, there was a lot new, but I mostly understood it intuitively from the context of the project."
3. "I tested the Squid API through Swagger and Postman - trying to enter both invalid and valid values, etc. (the address was interesting). I did the same on Octopus, where I really appreciated the comparison strategy because it saved a lot of work, and I would like it if something similar could be done for commands, though I understand that it wouldn't be as straightforward. Then I also uncovered a few errors while implementing unit tests on handlers." "I think the current testing method covers the vast majority of problems, but it relies a lot on having people who are not lazy to explore Octopus behavior and resolve specific behaviors (see creating an empty default address in the DB if null is received in the create request)."
4. "The changes took me about 12 hours, but that includes getting the project up and running and understanding everything necessary to write it. It was a good work environment, the project is clear enough for a newcomer to orient themselves in a reasonable amount of time. The only thing that might be worth documenting is the flow from controller to DB (i.e., everything that needs to be considered and what it affects along the way), because that was sometimes a bit of black magic."

The junior engineer accomplished one of the CRUD migrations, which ended up taking around 12 hours, including project setup. Unfortunately, many of the issues that arose were caused by the environment setup on the Octopus

side. Altogether, this is a favorable result. The implementation can be immediately deployed thanks to the layer of safety with the strangler façade, which immediately started testing the traffic with the new solution.

7.9.1 Feedback Conclusion

With the varied developer backgrounds, this research strives to find the common issues among all levels of seniority. The team has a diverse set of skills from various different technologies. Let us explore some of the issues that arose.

- Documentation: The documentation seems to be lacking, which is understandable. My experience revolves around .NET. Therefore, the setup of the development environment came naturally to me. However, this was not something that is quite as simple for people with less experience with .NET. As a remediation, I have contributed to the team Wiki with various aspects of the solution, and I have created a detailed step-by-step setup guide for the environment, which is available in the solution README.
- Learning curve: The migration to .NET has a learning curve, especially for those less experienced with strongly typed languages. However, Automapper and Entity Framework were highlighted as beneficial for simplifying database operations.
- Onboarding: For new developers, some structured initial approach with frequent feedback sessions and detailed walkthroughs of the architecture could be beneficial to lessen the initial challenges.

Overall, the migration project shows promise, with solid expectations from the experienced developers and successful initial implementations by the team. By addressing the challenges highlighted here, the company can look forward to a smooth transition to the .NET platform.

CHAPTER 8

Conclusion

In this thesis, we tackle the challenge of migrating the Atlantis inventory system from a PHP backend to .NET. This migration was more than translating code, rather it was a strategic upgrade that makes the system more robust, scalable, and easier to maintain, preparing it for future growth.

We have done our due diligence in maintaining compatibility with existing systems used by the company, like the frontend, the database, and authentication services, to ensure the transition does not disrupt the system's ongoing operations. This, in conjunction with the implementation of the Strangler Fig pattern, will make sure the business runs smoothly during the migration.

The new system is designed to be organized and easy to maintain through the use of the Clean Architecture, CQRS, and DDD. These architectural patterns improve on the previous architecture by dividing responsibilities and creating a more modular design. During the implementation of the new system, the status was closely communicated with the team, who's members later participated in a survey by making changes and confirming the improvement in the maintainability of the solution.

The implementation of the architecture has been tested on a production-like environment, with all of the components deployed. This critical step has shown that the implemented Strangler façade is stable and can be used as a foundation for fully migrating to the new system. The monitoring capabilities also showed improved performance of the new system.

The project still has a long way to go. The tooling provided as part of the thesis can be used as a foundation to migrate the rest of the system in a controlled, phased manner.

In conclusion, the thesis provides a detailed plan for system migration that evaluates both the old and new technologies. It serves as a useful guide for similar migration projects, showing how to update complex systems with minimal impact on daily operations.

Bibliography

[1] Microsoft Azure Architecture Center: Strangler Fig pattern. 2024, accessed: 2024-03-23. Available at: https://learn.microsoft.com/en-us/azure/architecture/patterns/strangler-fig

[2] Microsoft: Microservices architecture style. 2024, accessed: 2024-04-24. Available at: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices

[3] Microsoft: Common web application architectures. 2024, accessed: 2024-04-24. Available at: https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures

[4] Martin, R.: *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Martin, Robert C, Prentice Hall, 2018, ISBN 9780134494166. Available at: https://books.google.cz/books?id=8ngAkAEACAAJ

[5] Fowler, M.: Bounded Context. 2014, accessed: 2024-05-04. Available at: https://martinfowler.com/bliki/BoundedContext.html

[6] Microsoft: Middlewares in ASP.NET Core. 2024, accessed: 2024-04-19. Available at: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware

[7] Li, C.-Y.; Ma, S.-P.; Lu, T.-W.: Microservice Migration Using Strangler Fig Pattern: A Case Study on the Green Button System. In *2020 International Computer Symposium (ICS)*, 2020, s. 519–524, doi: 10.1109/ICS51289.2020.00107.

[8] Kovář, P.: *Backend skladového systému*. Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2019. Available at: https://dspace.cvut.cz/handle/10467/82591

[9] Malec, O.: *Frontend skladového systému*. Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2019. Available at: https://dspace.cvut.cz/handle/10467/86593

[10] Runeson, P.; Höst, M.: Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 2009, ISSN 1573-7616, doi:10.1007/s10664-008-9102-8. Available at: https://doi.org/10.1007/s10664-008-9102-8

[11] Community, S.: Symfony PHP Framework. 2024, accessed: 2024-03-27. Available at: https://symfony.com/

[12] Adermann, N.; Boggiano, J.: *Composer: Dependency Manager for PHP*. 2024, accessed: 2024-03-26. Available at: https://getcomposer.org/

[13] Hardt, D.: The OAuth 2.0 Authorization Framework. RFC 6749, RFC Editor, Oct 2012, accessed: 2024-03-27. Available at: https://www.rfc-editor.org/rfc/rfc6749.txt

[14] Group, T. P.: PHP: Documentation. 2023, accessed: 2024-03-19. Available at: https://www.php.net/docs.php

[15] Group, P. G. D.: PostgreSQL: The World's Most Advanced Open Source Relational Database. 2024, accessed: 2024-03-27. Available at: https://www.postgresql.org/

[16] Labs, R.: Redis. 2024, accessed: 2024-03-27. Available at: https://redis.io/

[17] Inc., D.: Docker: Empowering App Development for Developers. 2024, accessed: 2024-03-31. Available at: https://www.docker.com/

[18] Sysoev, I.: nginx: HTTP and reverse proxy server. 2024, accessed: 2024-03-27. Available at: https://nginx.org/

[19] Vue.js: The Progressive JavaScript Framework. 2024, accessed: 2024-03-27. Available at: https://vuejs.org/

[20] Baresi, L.; Garriga, M.; De Renzis, A.: Microservices Identification Through Interface Analysis. In *Service-Oriented and Cloud Computing*, editace F. De Paoli; S. Schulte; E. Broch Johnsen, Cham: Springer International Publishing, 2017, ISBN 978-3-319-67262-5, s. 19–33.

[21] Microsoft: File uploads in ASP.NET Core. 2024, Accessed: 2024-03-14. Available at: https://learn.microsoft.com/en-us/aspnet/core/mvc/models/file-uploads?view=aspnetcore-8.0

[22] MinIO, I.: MinIO: High Performance, Kubernetes Native Object Storage. 2023, accessed: 2024-04-05. Available at: https://min.io/

[23] Richards, M.; Ford, N.: *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly Media, 2020, ISBN 9781492043409. Available at: https://books.google.cz/books?id=wa7MDwAAQBAJ

[24] Engineering, S.: Event Sourcing for an Inventory Availability Solution. 2024, accessed: 2024-04-12. Available at: https://engineering.salesforce.com/event-sourcing-for-an-inventory-availability-solution-3cc0daf5a742/

[25] TechEmpower Framework Benchmarks. Accessed: 2024-03-19. Available at: https://www.techempower.com/benchmarks/

[26] Dascalu, A.: Benchmarks: .NET vs PHP vs Go vs NodeJS. 2020, accessed: 2024-03-19. Available at: https://andreidascalu.medium.com/benchmarks-dotnet-vs-php-vs-go-vs-nodejs-98a0ec9b56ef

[27] Popov, N.: Discussion on PHP 8 JIT. 1 2019, accessed: 2024-03-19. Available at: https://externals.io/message/103903#103927

[28] Microsoft: ReadyToRun Overview. 2023, accessed: 2024-03-19. Available at: https://learn.microsoft.com/en-us/dotnet/core/deploying/ready-to-run

[29] Microsoft: ASP.NET Core Security. 2024, accessed: 2024-04-28. Available at: https://learn.microsoft.com/en-us/aspnet/core/security/

[30] Microsoft: Configuration in ASP.NET Core. 2024, accessed: 2024-04-28. Available at: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/configuration/

[31] Microsoft: Logging in ASP.NET Core. 2024, accessed: 2024-04-28. Available at: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/logging/

[32] Patterson, C.; other contributors: MassTransit. 2024, accessed: 2024-04-28. Available at: https://masstransit-project.com/

[33] Microsoft: Building .NET Docker Images. 2024, accessed: 2024-04-28. Available at: https://learn.microsoft.com/en-us/aspnet/core/host-and-deploy/docker/building-net-docker-images

[34] Event Storming: Event Storming Resources. 2024, accessed: 2024-03-20. Available at: https://www.eventstorming.com/#resources

[35] Domain Storytelling: Quick Start Guide. 2024, accessed: 2024-03-20. Available at: https://domainstorytelling.org/quick-start-guide

[36] Banerjee, M.; Roy, S. R.; Kumar, C.: Feature Oriented Programming: A step towards flexible composition of modular programming. In *2012 1st International Conference on Recent Advances in Information Technology (RAIT)*, 2012, s. 369–373, doi:10.1109/RAIT.2012.6194448.

[37] Foote, B.; Yoder, J.: Big Ball of Mud. 1997, accessed: 2024-04-24. Available at: http://www.laputan.org/mud/mud.html

[38] Microsoft: Command and Query Responsibility Segregation (CQRS). 2024, accessed: 2024-05-04. Available at: https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs

[39] Martin, R.: *The Clean Coder: A Code of Conduct for Professional Programmers*. Robert C. Martin Series, Pearson Education, 2011, ISBN 9780132542883. Available at: https://books.google.cz/books?id=ik0qCTVzl44C

[40] Bogard, J.; other contributors: MediatR. 2024, accessed: 2024-05-04. Available at: https://github.com/jbogard/MediatR

[41] Pham, M. D.: *FitLife - game about studying at FIT*. Bachelor's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2022. Available at: https://dspace.cvut.cz/handle/10467/1016022

[42] Fowler, M.: Anemic Domain Model. 2003, accessed: 2024-05-04. Available at: https://martinfowler.com/bliki/AnemicDomainModel.html

[43] Skinner, J.; other contributors: FluentValidation. 2024, accessed: 2024-05-03. Available at: https://github.com/FluentValidation/FluentValidation

[44] Microsoft; other contributors: Entity Framework Core. 2024, accessed: 2024-04-27. Available at: https://github.com/dotnet/efcore

[45] Microsoft: Scaffolding - Entity Framework Core. 2024, accessed: 2024-04-12. Available at: https://learn.microsoft.com/en-us/ef/core/managing-schemas/scaffolding

[46] Lock, A.: Introduction to Authentication with ASP.NET Core. 2024, accessed: 2024-04-29. Available at: https://andrewlock.net/introduction-to-authentication-with-asp-net-core/

[47] Microsoft: Role-based authorization in ASP.NET Core. 2024, accessed: 2024-04-29. Available at: https://learn.microsoft.com/en-us/aspnet/core/security/authorization/roles

[48] Keycloak: Audience Support in Keycloak. 2024, accessed: 2024-05-01. Available at: https://www.keycloak.org/docs/latest/server_admin/#audience-support

[49] Morris, R.; other contributors: Swashbuckle.AspNetCore. 2024, accessed: 2024-05-03. Available at: https://github.com/domaindrivendev/Swashbuckle.AspNetCore

[50] Microsoft: Getting Started with Swashbuckle in ASP.NET Core: XML Comments. 2024, accessed: 2024-05-03. Available at: https://learn.microsoft.com/en-us/aspnet/core/tutorials/getting-started-with-swashbuckle?view=aspnetcore-8.0&tabs=visual-studio#xml-comments

[51] Blumhardt, N.; other contributors: Serilog. 2024, accessed: 2024-05-04. Available at: https://github.com/serilog/serilog

[52] Microsoft: FeatureManagement-Dotnet. 2024, accessed: 2024-04-21. Available at: https://github.com/microsoft/FeatureManagement-Dotnet

[53] Newkirk, J.; other contributors: xUnit.net. 2024, accessed: 2024-04-12. Available at: https://xunit.net/

[54] Hofmeister, A.; other contributors: Testcontainers for .NET. 2024, accessed: 2024-04-12. Available at: https://dotnet.testcontainers.org/

[55] Bogard, J.; other contributors: Respawn. 2024, accessed: 2024-04-15. Available at: https://github.com/jbogard/Respawn

[56] Seemann, M.; other contributors: AutoFixture. 2024, accessed: 2024-04-17. Available at: https://github.com/AutoFixture/AutoFixture

[57] Chavez, B.; other contributors: Bogus - Faker for .NET. 2024, accessed: 2024-04-17. Available at: https://github.com/bchavez/Bogus

[58] Cropp, S.; other contributors: Verify. 2024, accessed: 2024-04-14. Available at: https://github.com/VerifyTests/Verify

[59] Microsoft: Integration tests in ASP.NET Core. 2024, accessed: 2024-04-17. Available at: https://learn.microsoft.com/en-us/aspnet/core/test/integration-tests

[60] Doomen, D.; other contributors: Fluent Assertions. 2024, accessed: 2024-04-12. Available at: https://github.com/fluentassertions/fluentassertions

APPENDIX A

Acronyms

API Application Programming Interface.

CORS Cross-origin resource sharing.

CQRS Command and Query Responsibility Segregation.

CRUD Create, read, update and delete.

DDD Domain Driven Design.

HTTP Hypertext Transfer Protocol.

JWT JSON Web Tokens.

ORM Object Relational Mapping.

REST Representational State Transfer.

TDD Test Driven Development.

UI User interface.

APPENDIX B

Contents of enclosed SD-Card

```
readme.txt ........ the file with SD-Card contents description
text .............. the thesis text directory
  thesis .......... the directory of LaTeX source codes of the thesis
  thesis.pdf ...... the thesis text in PDF format
```