UNIVERSITY OF MACEDONIA

SCHOOL OF INFORMATION SCIENCES

DEPARTMENT OF APPLIED INFORMATICS

DEVELOPING DISTRIBUTED SYSTEMS WITH MODULAR MONOLITHS AND MICROSERVICES

Bachelor's thesis

of

Tsechelidis Michail

Thessaloniki, September 2023

DEVELOPING DISTRIBUTED SYSTEMS WITH MODULAR MONOLITHS AND MICROSERVICES

Tsechelidis Michail

Undergraduate Student of Applied Informatics at University of Macedonia

Bachelor’s Thesis

submitted for the partial fulfillment of its requirements

BACHELOR'S DEGREE IN APPLIED INFORMATICS

Supervisor
Apostolos Ampatzoglou

Approved by the three-member examination committee on 14/09/2023

Apostolos Ampatzoglou	Alexandros Chatzigeorgiou	Stylianos Xinogalos
.....	.....	.....

Michail Tsechelidis

.....

Abstract

This thesis introduces the concept of a Modular Monolith and tries to standardize a way to implement this idea to enable the development of projects within a distributed context. Unlike traditional monolithic architectures, this approach allows the project to run on a single runtime while effectively addressing the development requirements.

Modular Monoliths aim to provide an easy way of controlling the granularity of services. This enables the simplification of the resulting infrastructures(team topologies, pipelines, clusters, etc) after the usage of Service Oriented Architectures like Microservices, making them more manageable, efficient and less expensive.

Currently, there is no existing architecture that fully embodies the idea of a Modular Monolith. Therefore, this thesis proposes an implementation and a Development Guide to assist developers in adopting this approach. Additionally, it explores the outcomes of research conducted to evaluate its acceptance within the industry.

Furthermore, this thesis aims to revisit certain terms. By clarifying concepts, it becomes easier to comprehend the relationship and collaboration between Modular Monoliths and Microservices.

Ultimately, this study demonstrates that embracing a Modular Monolith mindset, developers can achieve simpler infrastructures without sacrificing the development requirements.

Keywords: architecture, distributed systems, microservices, modular monolith, service oriented architecture, services, software development

Table of contents

1. Introduction

1.1. Purpose - Objectives

The purpose of this thesis is to introduce the concept of a Modular Monolith and propose a standardized approach for its implementation on systems within a distributed context. The objective is to enable the construction of projects using a single runtime, while effectively addressing development requirements.

The thesis aims to leverage the benefits of Service Oriented Architectures (SOA) and simplify the management of services by controlling their granularity. By doing so, the resulting infrastructure becomes more manageable, efficient, and cost-effective. Currently, there is no existing architecture that fully embodies the idea of a Modular Monolith, which highlights the need for this thesis to propose an implementation and provide a comprehensive Development Guide for developers to adopt this approach.

Additionally, the thesis seeks to explore the outcomes of research conducted to evaluate the industry's acceptance of the Modular Monolith concept. It also aims to revise certain terms, clarifying and demystifying concepts to facilitate better understanding of the relationship and collaboration between Modular Monoliths and Microservices. Ultimately, this study aims to demonstrate that by embracing a Modular Monolith mindset, developers can achieve simpler infrastructures without compromising on development requirements.

1.2. Structure of the Study

The thesis consists of a Theoretical Background chapter, Definitions and Arguments - which showcase a more personal view of the theory -, a development chapter - that analyzes the thought process of the creation of a Modular Monolith -, a Case studies chapter - where 2 real case systems are commented -, the Simpler Infrastructure chapter - where it showcase its importance as a result of using Modular Monoliths - and the Validation research of the proposed standardization of Modular Monoliths.

2. Theoretical Background

A simple understanding of programming is mandatory. Some experience with an Object Oriented programming language and an understanding of software that is built to communicate with other software in a Distributed manner, would be desired [1][2].

2.1. Distributed Systems

Distributed Systems are software systems that are a collection of executable software that each of them is run in a different environment inside a network [3]. All these software that run in different environments have some common characteristics. One of them is the need of communicating via a remote protocol, the most common being http - via the internet -. This communication happens using some already defined software as a library that makes the actual remote request via the internet. The usage of such software inside a system is done inside classes called Controllers. Such classes map their functionalities to URLs/endpoints and they are able to translate messages and map the information they are hiding to custom variables. One more thing are transaction/service classes, which with the help of the data provided by the Controllers, perform scenarios involving local and remote functionality. The next common thing is the need of Entity classes that map database data to software data, so the software is able to communicate with database data. The last thing is the need of Object classes that represent business Objects, and those with their interactions between them are called domain or else business logic.

2.2. Domain Driven Design

DDD is about having the business logic/domain define the design process and have it be the centerpiece of the puzzle [4]. In a system there can be domains that live inside other domains as subdomains/bounded contexts. Best practices exist on how to identify those. Domains are surrounded by functionalities that enable the software to operate in its business context. For example in distributed systems there are functionalities other than the ones that give business value, like Controllers - which perform the communication with other softwares, something that is mandatory for the domain to have -. Having the domain as a centerpiece for the surroundings can be visualized like the below image.

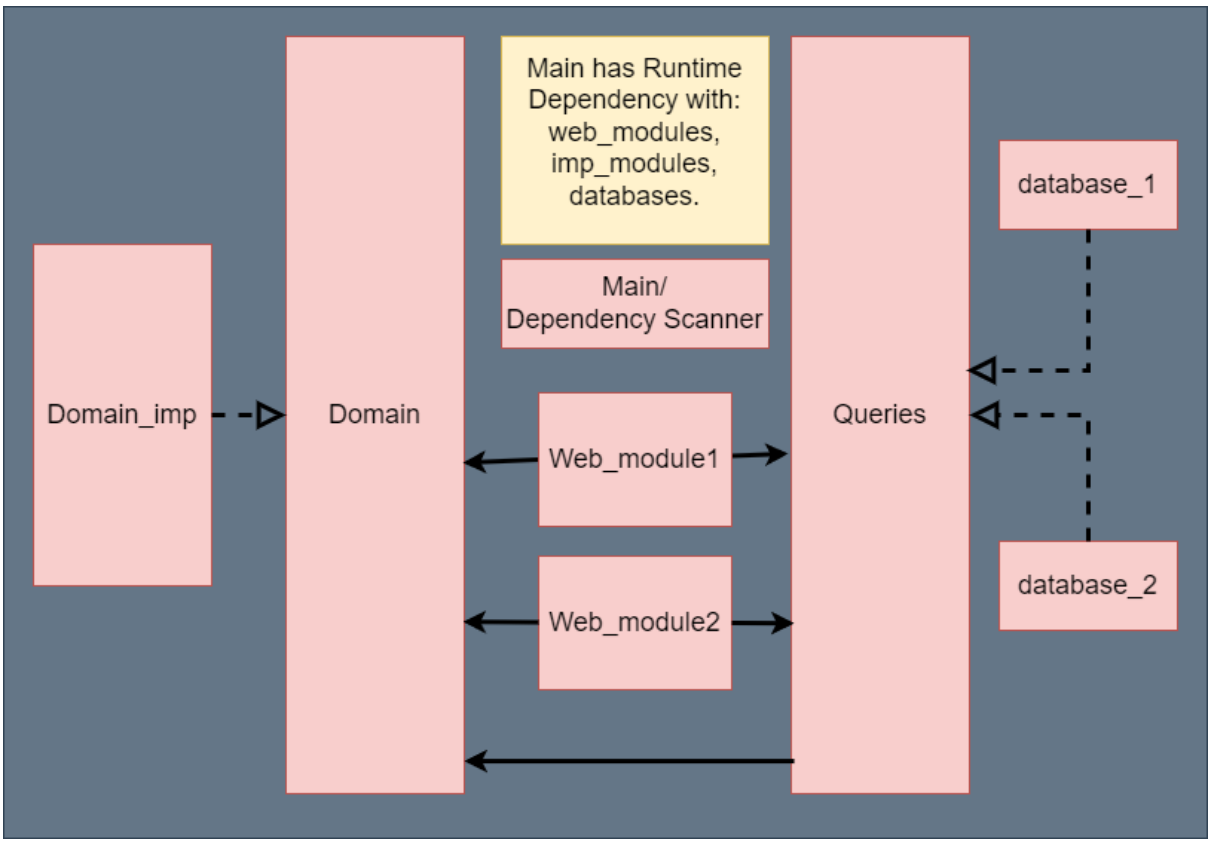


Figure 1 - Modular Monolith diagram

In a conference made by Kevlin Henney [5], *"A system is not a tree"*, it is stated that the end result always has functionalities that depend on others that are not in the same hierarchy. A domain is as complex as the business states. With the help of Dependency Injection, the system can be enforced to resemble a tree. (Martin, R., p. 193) *"Source code dependencies must point only inward, toward higher-level policies."* [2].

2.3. Dependency Injection

According to Fowler (2004) *"The basic idea of the Dependency Injection is to have a separate object, an assembler, that populates a field in the lister class with an appropriate implementation for the finder interface…"* [6], this results in the diagram of Figure 2, where HL1 has been provided with ML1 while depending only on I. (Martin, R., p. 62) *"With this approach, software architects working in systems written in OO languages have absolute control over the direction of all source code dependencies in the system. They are not constrained to align those dependencies with the flow of control… the software architect can point the source code dependency in either direction."* [2].

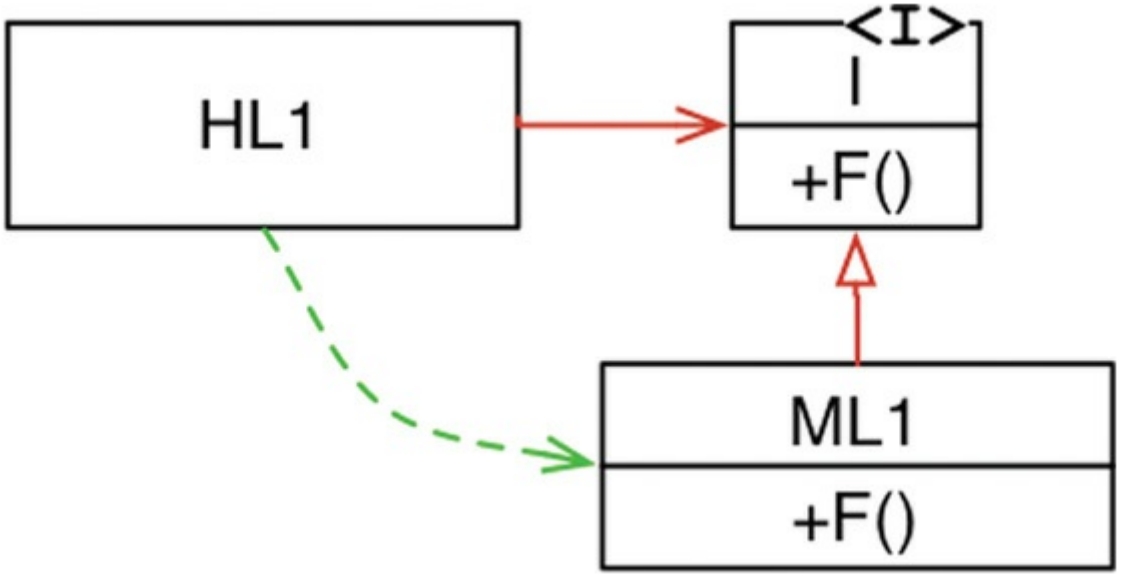


Figure 2 - Dependency Inversion

2.4. Tools

The ecosystem chosen to produce the code as a proof of concept is Java. Some pieces of the puzzle may be absent from some ecosystems like Interfaces [1]. If an ecosystem doesn't have Interfaces, then the proposal of this thesis is not applicable. One other piece is the Dependency Injection Mechanism. There must be the ability to define package paths to implicitly inject classes that other classes will have no access to. If the DI framework only works with explicitly defining the class to inject, then the proposal is also not applicable.

2.4.1. Spring Framework

Spring Framework at its core is a dependency injection framework [7]. It provides supportive software at runtime that manages the injected classes. Its ecosystem expanded to provide much more support for various things, such as providing surrounding functionalities - like a server that enables the communication, and a default configuration for the project to run -, but the core usage of it will be the ability to inject implicitly with package scanning.

2.4.2. Maven

Maven is a build tool in the Java ecosystem that enables developers to have access to software in remote repositories and manage these software as dependencies to the projects [8]. Maven compiles code and produces artifacts including executable files. Maven understands the project's structure by having pom.xml files that define what the project is, what the dependencies are and what the produced artifact is. Maven also supports its own defined modules, which can separate a project into smaller projects as groups of packages/folders.

2.4.3. Github

Github provides a remote space for developers to save code and their artifacts - with Github packages - [9]. It integrates with git enabling version control on the projects. The proposed development strategy of Modular Monoliths relies on Github packages, where each module must be in a remote space for maven to manage. It also relies on Github Actions that enables the management of git branches via automations.

2.5. Hexagonal Architecture

Hexagonal architecture provides developers with a thought process of grouping the packages/folders and their contents of a project conforming to its needs by applying the ports and adapters model [10].

This means that projects have folders for adapters(classes), ports(interfaces) and the application itself, and then the application communicates with the surrounding functionalities(adapters) via the defined ports, using along with the ports a DI framework.

Since the project is built in a distributed context, it has the application being the layer of the transactions/services, the adapters having inbound subfolders that handle the inbound communications and outbound subfolders that handle the persistence. All these functionalities are communicating via the provided ports, hence the management of the project becomes better since the architecture reflects the needs of the system.

3. Grouping Information

One characteristic of Software Engineering is grouping information. While the below image is a personal proposal of reflecting the levels of grouping information that map into actual code, similar ideas have been expressed by various engineers [11][12].

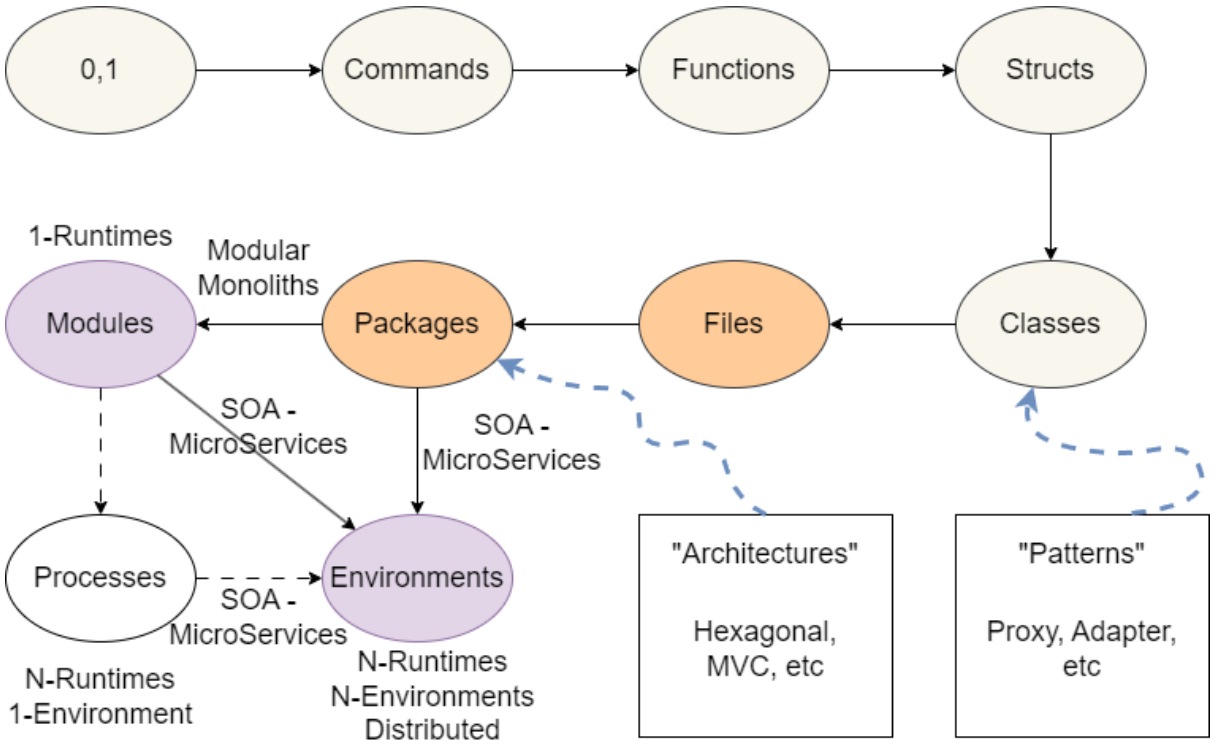


Figure 3 - Grouping Information

From the early days of software, it was understood that performance is not the only factor to create systems, else people would continue to create software in Ones and Zeros. Software Development is about codifying ideas into a human readable manner that computers can map their functionalities to [13]. At some point there was the idea of grouping functions with data structures and classes were defined [1]. Classes were originally designed with the purpose of translating abstract business components into concrete code representations. However, in the process, several recurring issues surfaced, leading to the formulation of specific best practices. Consequently, the Gang of Four (GoF) Design Patterns of Object-Oriented Programming emerged [14]. These patterns aimed to provide strategic means of organizing code at the class level, empowering developers to enhance code

reusability, comprehend the codebase more effectively, and establish guidelines for addressing common problems.

Then various techniques have been analyzed to organize code into packages like package by layer or feature [2]. The project utilizes Hexagonal Architecture which organizes code into packages based on the project, and in a Distributed context there are common surrounding functionalities that can be grouped together like ports and adapters.

The next step is to group packages into a single artifact, named module. This thesis proposes a best practice on grouping modules and using them in a way - with the help of DI - that benefits the entire software development lifecycle. The next step would be to separate functionalities into different runtimes/executables, but the cost of maintaining a separate process and its context switching isn't justified by any arising problem, except when there is a specific software requirement that the Cloud pattern *'sidecar'* solves [15].

The last level that a developer gets involved in is environments that allow functionalities to be executed with different specifications. A different environment can be a Docker container that runs a specific group of functionalities [16][17]. Once that is done, administrators take over and determine the groups of containers to create the infrastructure, which is a separate process and doesn't directly impact the development of the project [18]. These groups are commonly referred to as *'pods'* and are defined using the widely adopted standard/framework known as Kubernetes.

As seen, various levels of group information exist. The words *"Patterns"* and *"Architecture"* encapsulate so many levels and are used so extensively that without specifying the context it is confusing and misleading. Grouping information techniques are about refactoring [19], keeping the functionalities the same, while Cloud Patterns consist of best practices on implementing the environment level and patterns that enhance the already defined project to meet business requirements.

4. Types of Requirements

(Jason Gorman, February 2023, *linkedin post*) *"Thinking ... about architectural patterns/styles as responses to emerging problems ... What emerging problem would prompt us to refactor a monolith into two or more separate services?"*. The following is a personal categorization of requirements found in a distributed system that aims to guide system design choices.

- Software Requirements

They can be resolved by Grouping Information on the process or environment level, and they are usually identified at the early stages of system design. Such cases are when certain functionalities are assigned to teams that work with different ecosystems, or when an ecosystem can provide better support for the functionality that is going to be developed. The implementation of such grouping is the Cloud pattern *'sidecar'* [15].

- Hardware Requirements

These can be resolved by Grouping information on the environment level. Such requirements often reflect the need for a functionality to run with different hardware specifications than others - like running with more powerful CPUs -.

- Development Requirements

These are about enabling the teams and the software development lifecycle. The package level was used to partially resolve those [20], until teams started to use the environment level. In this thesis, it is tried to show that the module level can fully resolve those requirements. Examples of such are Team Topologies and Agile Development [21][22].

- Business Requirements

Some of them can be resolved by the environment level and others need the help of Cloud patterns that enhance the project, like the Load Balancer pattern [15]. The most popular business requirement is scaling [20].

5. Definitions and Arguments

5.1. Monoliths and Modular Monoliths

A personal definition of Monoliths is having all the functionality in a single machine, as a single program. Past generations of developers have struggled with that property since the result of the project would usually have poor quality making the project live a short term of life [20]. This was a result of mostly having trouble solving the Development Requirements.

The idea of modularizing a project was introduced from at least around 1980 [24]. The idea is to organize code but there was never agreed on what level. In this thesis a module is a group of packages that can produce their own build artifacts [10]. But in various ecosystems modules have different meanings.

Java introduced the module system JPMS. While the idea was like maven's modules, the implementation created the opposite result of modules being developed independently [25]. Every relationship had to be defined explicitly, which seems to be an obstacle to the development of a system. JPMS was supposed to help the project with Development requirements, yet managing a project which uses JPMS seemed more complicated than before.

Java has also supported another framework that provides a different approach to modularity, the OSGi [26]. Each ecosystem adapting this framework must have a tool that runs these modules/bundles, in the Java world an implementation of that is the Apache Felix container. Felix starts as a server and then developers can dynamically control - while the server is active - each module's lifecycle.

While this approach can't align with developing solutions in the Cloud, since immutability is needed [27], it fits perfectly for services that are deployed in environments controlled by the development team. That is why IoT has adopted this solution [28], since every environment is distinct, and it is wanted to be able to control each environment on their own.

An example of such an environment is a smart house where a single microcontroller is used. With OSGi there could be a possibility that an update occurs on the kitchen's functionalities while the bedroom's are still running.

In other words, Cloud deployment needs to have a centralized way to manage the environments as one, so tools like Kubernetes are able to use a defined docker image that is not going to change [18]. OSGi on the other hand prompts developers to manage such deployments while they are active and running on their own, and change what functionalities exist in the current environment.

Since the thesis is about Distributed Systems usually deployed in the Cloud, about immutable state of software, OSGi was not used and groups of modules are statically defined and create the necessary immutable docker images [17], as a resource for Kubernetes - Controlling the Granularity -.

Continuing with defining modularity, at the time of writing there has been a new term called Modular Monolith. It is a way of describing Monoliths and Modular Design specifically targeted in software that is developed in a Distributed Context -.

Modular Monoliths are often compared to Microservices, which is a wrong comparison. The correct comparison is to compare Modular Monoliths to just Monoliths. **Modular Monolith refers to a code that runs as a single unit but it can be managed as a group of smaller units [29], while Monolith refers to a code that runs as a single unit but being unstructured resulting in being managed as one unit.** A project can be structured or unstructured.

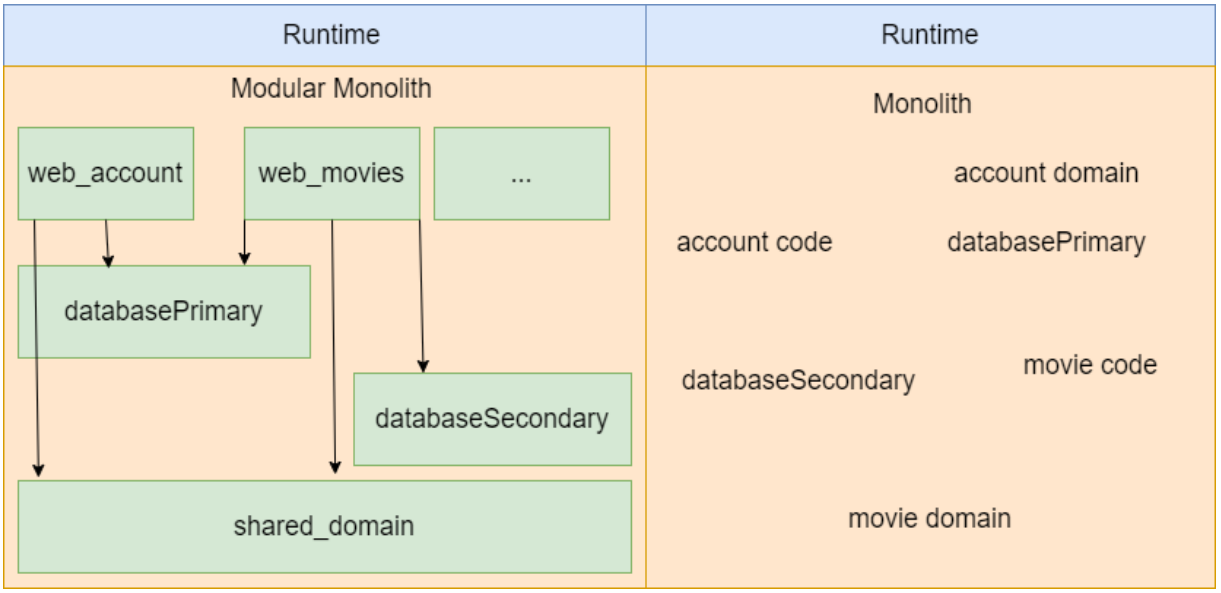


Figure 4 - Modular Monolith vs Monolith

Depending on how a project has grouped the code inside packages it can be chosen what to modularize, to solve Development requirements. In Software that is developed in Distributed Context, Hexagonal architecture packages the code nicely since it always has Controllers, Services, Persistence and stateful business Objects (domain). So, modularizing the Hexagonal architecture gives developers a starting point to create such software.

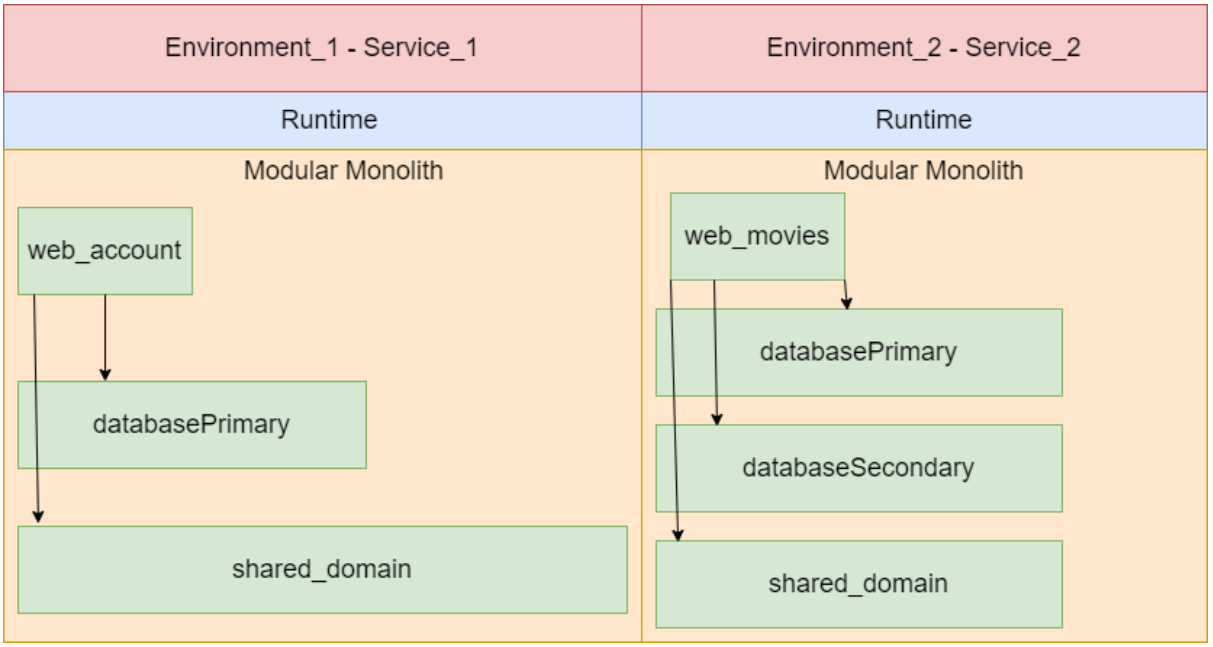


Figure 5 - Controlling the Granularity

From figure 5 the identification of functionalities that need their own environment is easy. It is also seen that web_account and web_movies functionalities don't need to communicate - out of bounds communication is avoided [30]-. Instead, each of them copies to their runtime the needed modules to function, hence hiding their relationship inside the domain. **The logic is that each service can choose which tools it needs at runtime to execute the functionality locally and not rely on external software.** (Heinrich, B., p. 23) *"...functionalities which occur multiple times in different processes can be grouped."* [31]. In the section 'Understanding the code' it is explained that this diagram is reflected by code.

Figure 6 - Controlling the Granularity mapped to code

5.2. Service Oriented Architecture

Service. This word can have different meanings, in Distributed systems it is the flow of a use case's scenario which is executed with the help of multiple distributed machines (transactions), (Coulouris, G., p. 169) *"a service is implemented as a number of different processes in different computers"* [3]. In the OSGi ecosystem it means some code that can be dynamically added to a running server to provide additional functionality.

A personal definition in the SOA context is that a Service is a Monolith (Modular or not) that is inside a Distributed System. This means that this Monolith is executable and communicates with other such Monoliths - usually via the internet -. SOA is a way to define a group of Services and how they interact to meet certain requirements.

Just like Monoliths, SOA is unstructured. That is why different types of SOA have been defined, as Children of SOA, that are structured and guide the development process, such as Microservices and Event Driven Architecture.

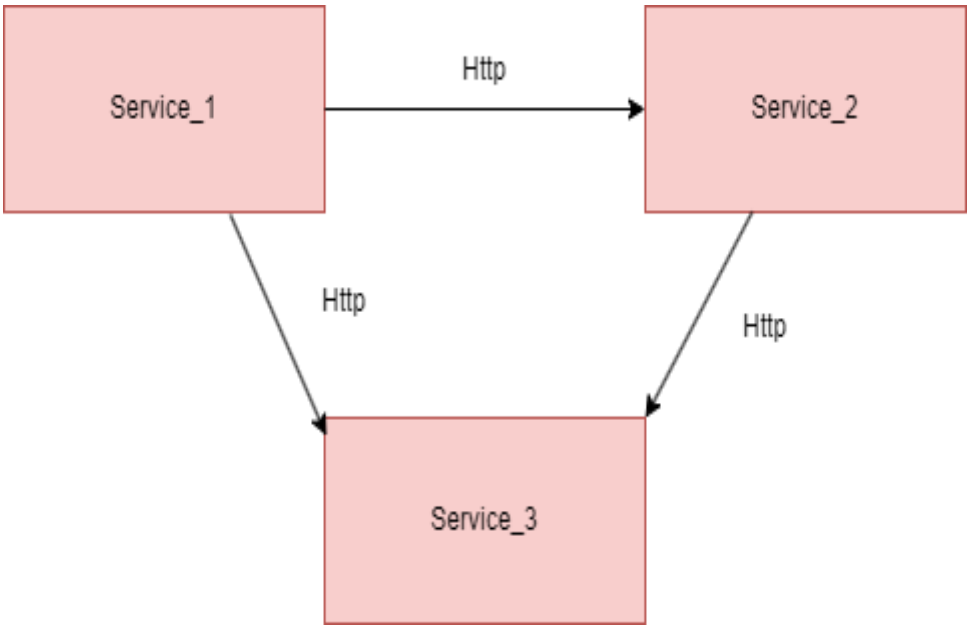


Figure 7 - SOA example diagram

5.2.1. Event Driven Architecture

Communication between systems is divided in 2 categories, Synchronous and Asynchronous [30] . Synchronous means that a system can't continue its execution flow until the communication between that and the remote system is over. Asynchronous is a way to make the system continue its execution flow and have the functionality that communicates with the remote system happen in a different time - most times when the remote system is ready -.

To create a system with asynchronous communication developers must create a very complex surrounding functionality of a queue that saves incoming messages or events. Because implementing asynchronous communication in each service is difficult and error prone, engineers decided to separate that functionality from the code, and introduced Queue Systems/Event Buses/Brokers to implement asynchronous communication. So, Figure 7 becomes like this.

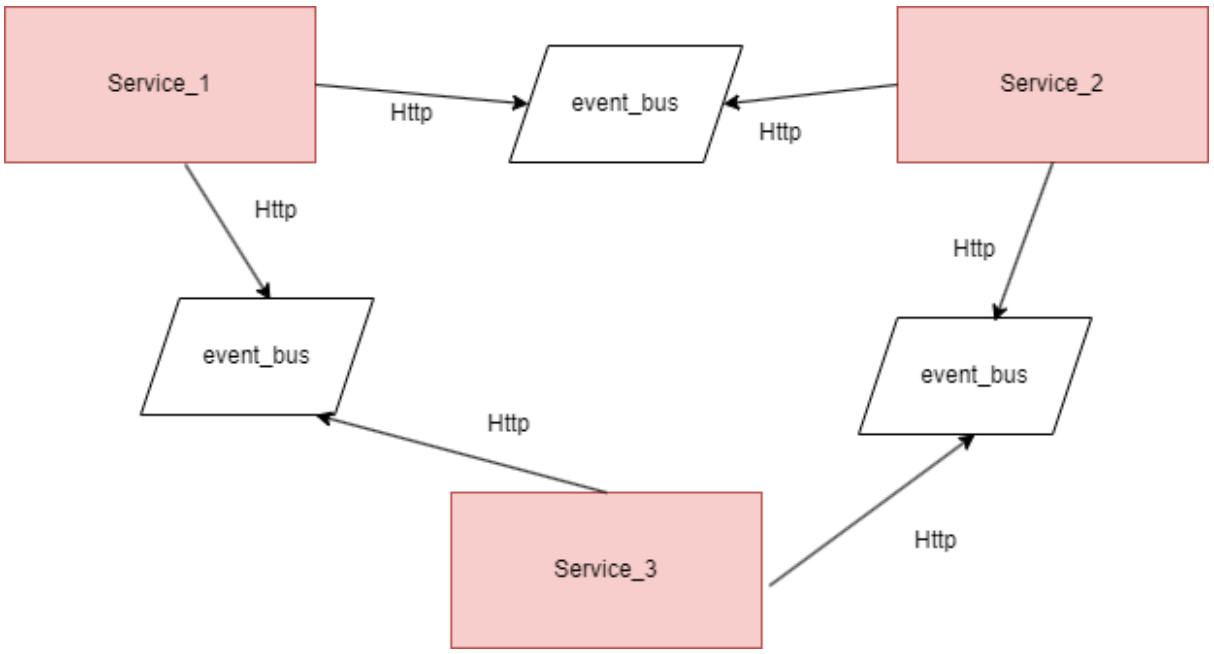


Figure 8 - Event Driven Architecture example diagram

This diagram is not following best practices on applying EDA - there could be just one event_bus - [32]. But this highlights the fact that tools, in any form of Architecture, can be misused.

5.2.2. Microservices

(Martin Fowler, 2014) *"...there is no precise definition of this architectural style..."* [33]. The missing definition of Microservices creates many problems in the industry since teams don't know what they are adapting to as architecture, causing a division in approaches to system development. The term was introduced to address two main problems:

1) Development issues of regular Monoliths
2) Scalability issues in SOAs, due to being implemented with regular Monoliths in mind.

For this architecture to solve those problems - having no definition - some attributes were defined that a system must have to be categorized as such, with the most dominant characteristics being small size and loosely coupled. But there are 2 problems:

1) What is small? How much is small? (Heinrich, B., p. 2) *"How granular should services be developed so that the effort for the development, composition, and maintenance of services is minimal?"* [31].
2) Coupling, loose or not, still exists.

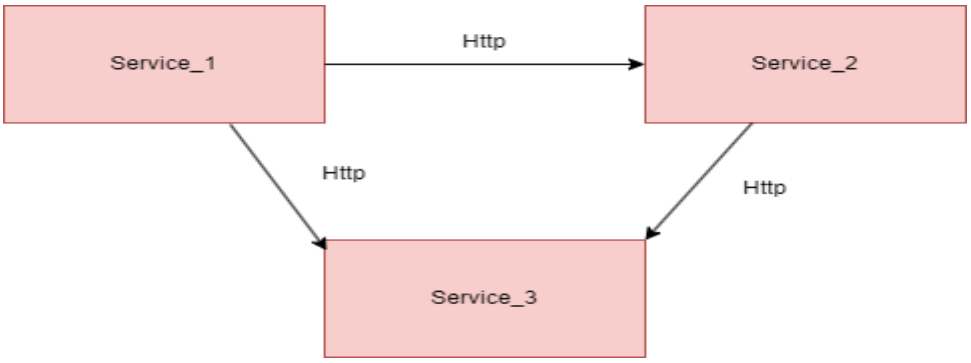


Figure 7 - SOA example diagram

These 2 attributes don't seem to change the diagram of Figure 7, having small services and using Http to enable loose coupling. To end up with such a result the *"Breaking the Monolith"* was created, which is extracting functionalities from a Monolith as services - this will later be defined as a Distributed Monolith -.

An argument arises with this approach that loose coupling isn't solving any requirements since the dependencies exist anyway [16]. An example of a problem would be changing the API of Service_3 while Service_1 depends on it. Even if a broker were put between those services, a change to the broker could still affect both services. Dependencies

create problems in development and deployment, their complexity, their management and their cost. Additionally, it is still not clear how small the services should be. At the time of writing the answer is *"as small as possible"* to have tiny functionalities for development and deployment to be easier in large scale systems. This can map to the exaggeration of having a single function - like Figure 9 - exist in its own environment [34].

```
import domain.Account;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import org.springframework.transaction.annotation.Transactional;
import queries.LoadAccount;
import queries.UpdateAccount;
import web.application.dto.CommandBalanceDTO;
import web.application.port.in.WithdrawUseCase;

@RequiredArgsConstructor
@Transactional("primaryTransactionManager")
@EnableTransactionManagement
@Service
class WithdrawService implements WithdrawUseCase {
    private final LoadAccount loadAccount;
    private final UpdateAccount updateAccount;

    @Override
    public String withdraw(CommandBalanceDTO data) {
        int money = data.getBalance();
        int id = data.getId();

        Account account = loadAccount.loadAccount(id);

        if (account.withdraw(money)) {
            updateAccount.updateAccount(account);
            int balanceResult = account.getBalance();
            return "Success! " + balanceResult;
        }
        return "failed";
    }
```

Figure 9 - WithdrawService

In this way Scalability is resolved thus bringing business value to the system. The question that comes next is at what cost [31][35]. Having shown that loose coupling can't define Microservices, the next step is to question the relevance of having small services with scaling.

5.3. Scaling

Scaling can have different meanings depending on the context. In services it references the ability for a system to Horizontally scale - creating replicas of the services' environments -. Kubernetes provides an implementation of this idea with the most common use case being replicating at a given CPU load of an environment, which is proportional to the Requests that it has [18]. The goal is to ensure Service Level Agreements (SLAs) which ensure that clients can seamlessly interact with the software without experiencing latency or downtime, bringing high availability [36]. An example of such an SLA is the *"four 9s"* (99.99%), which maps to 8,64 seconds of downtime per day. The significance of dysfunctional scaling becomes apparent when there is a business requirement to guarantee that the system must operate at a given level of uptime.

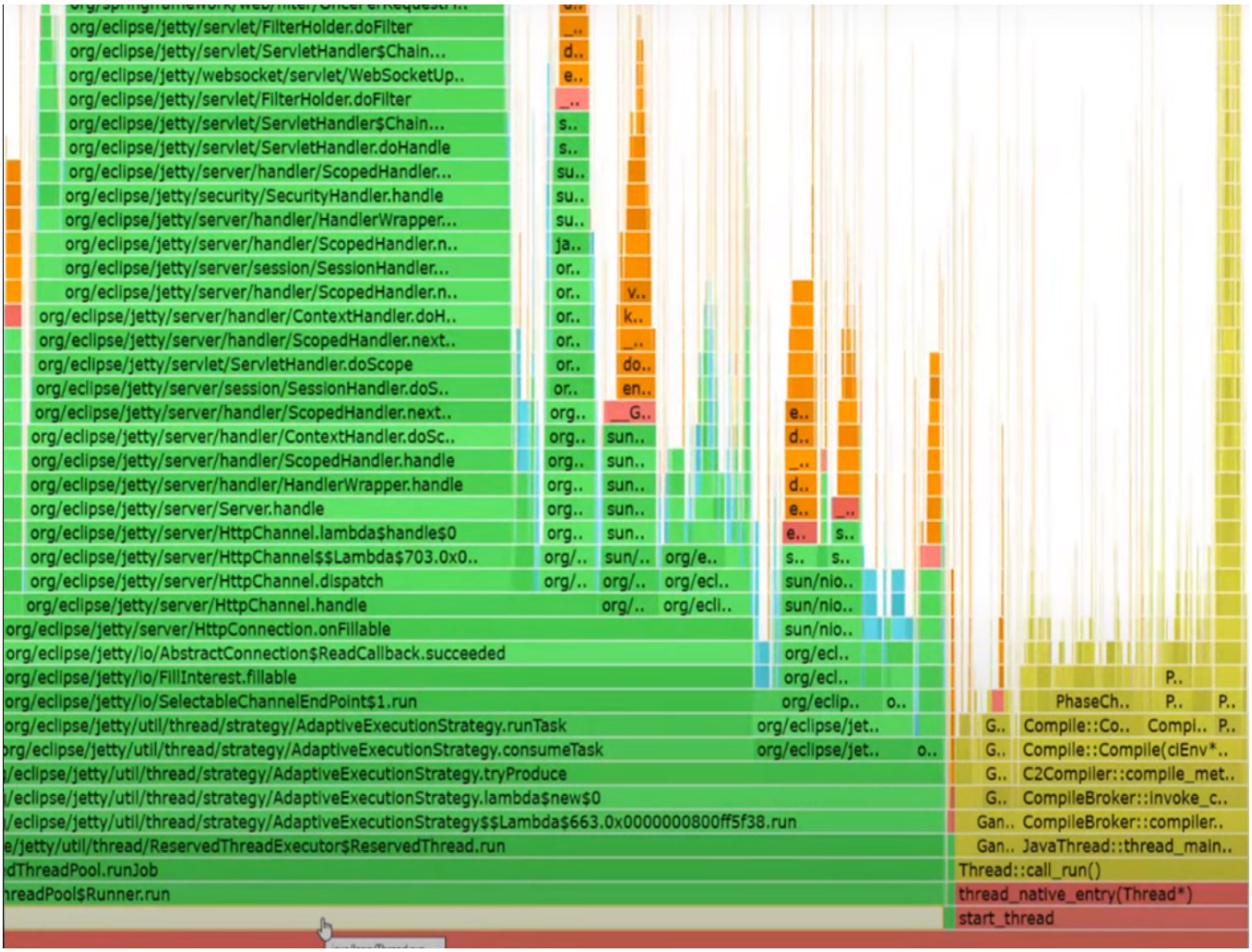


Figure 10 - Profiling (bottom)

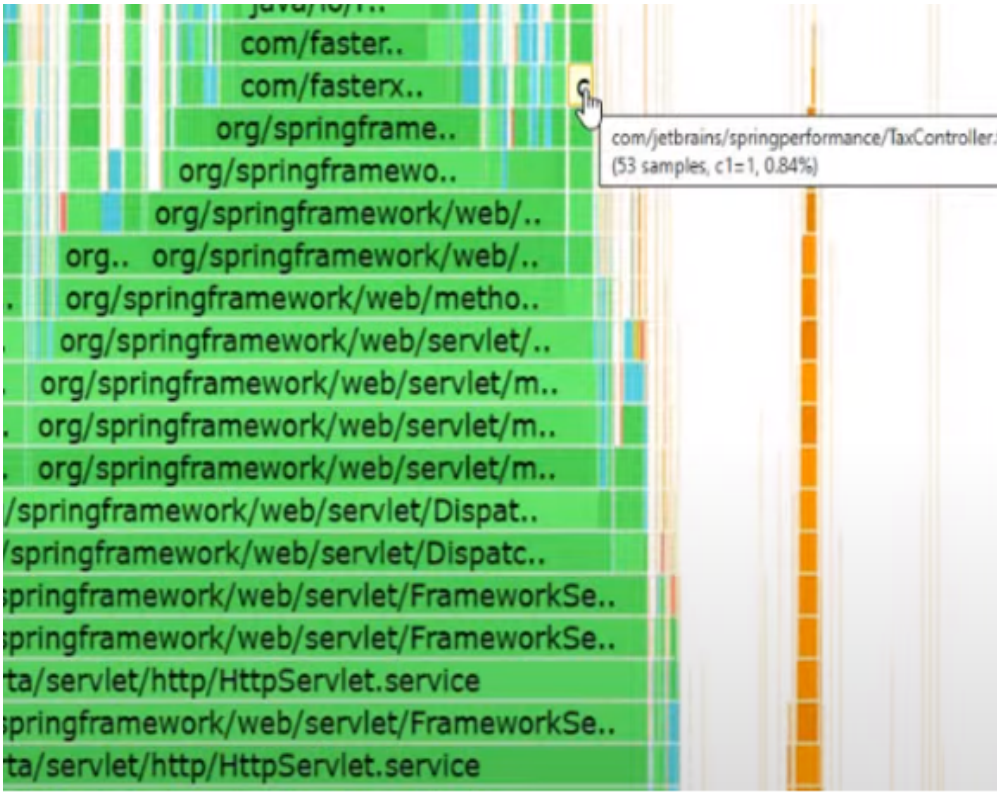


Figure 11 - Profiling (top)

The above profiling is done with a M5 machine with 4 CPUs and 16G of memory [37], which is a reasonable machine to rent in the Cloud. This simple server was tested up to 20000 Requests Per Second with a simple functionality of generating a random number – through the TaxController -. The result was that the server couldn't handle 20000RPS and it was investigated why and where the problem was. It was concluded that the CPU was the problem, since it was IDLE around 1% of the running time. The profiling on the CPU shows that only 0,84% time of CPU is spent on business logic and around 56% on a library that creates HTML templates. This shows the need to examine the systems and see potential separations, which in this case, would be of no use, since even if there were many more functionalities, each one of them would use that library. It is then reminded that 20000RPS are really not that realistic, with few exceptions being witnessed [37]. The question now is how the size of the artifact impacts the horizontal scaling of Kubernetes given that there is a business requirement to provide high availability [23]. The artifacts need to travel among the internet to be available in Kubernetes, and then Kubernetes needs to spawn environments - possibly replacing existing ones - with that artifact. So, the bigger they are, the slower the process is. This means that for Kubernetes to meet the desired SLAs at a given RPS, it needs to consider scaling at low CPU usage, wasting possible resources. The argument is that the

cost of the percentage of CPU usage that is lost could be lower than the cost of a MicroService infrastructure [35]. The bigger the artifact, the more possible functionalities it has in its runtime. A possible small vertical scaling could have less costs than a Microservices Infrastructure [31]. When a specific functionality needs to be updated, it is argued that there can be downtime for the functionalities that serve a specific use case, but the deployment strategies, such as the rolling deployment [15], can help eliminate the downtime. Modular Monoliths' artifacts can have increased size - due to the need for immutability [27] -, while deployment strategies exist to eliminate downtime, the scaling could be costly since Kubernetes may need to start scaling sooner to meet the required SLAs, but that cost could be insignificant considering the RPS needs of most systems. Even in a high RPS system [23], SLAs can be met without the absolute need of horizontal scaling, since with performance engineering and reasonable vertical scaling, most business requirements can seem to be resolvable.

5.4. Granularity of Services

(Heinrich, B., p. 3) *"First, we define granularity, according to the literature, as the number or extent of functionalities implemented by a service (Erl 2007; Galster and Bucherer 2008, p. 400; Thomas et al. 2010, p. 366)."* [31]. Granularity of a Service is about expressing how much functionality resides inside a single service hence it reflects a service's size [38]. This applies to any SOA. With controlling the granularity of services, the usage of SOA (grouping information at the environment level) is mitigated to solve what needs to be solved and then the negatives of SOA are justified [31]. As already seen from Figure 6 - Controlling the Granularity mapped to code, this thesis's proposal of Modular Monolith can control the granularity of services with ease. **This means that software can be initially developed as if it were one unit and later be separated, however it needs to be, if necessary** [39]**.** This results in Simpler Infrastructures and their positives. (Heinemeier, D., 2023) *"replacing method calls and module separations with network invocations and service partitioning within a single, coherent team and application is madness in almost all cases."* [40]. (Newman S., 2023) *"microservices should not be the default choice…Have you done some value chain analysis? Have you looked at where the bottlenecks are? Have you tried modularisation? Microservices should be a last resort."* [40].

5.5. Microservices Revisited

The thesis proposes a swift of importance, from *"loose coupling"* and the size of the services, towards independence. This term is mistaken to be equal with *"loose coupling"*, but that doesn't mean independence. Dependencies still exist, just in a different form with the same negatives as regular coupling.

Many systems end up being characterized as Microservices when they are SOA architectures using some Cloud Patterns [16]. This results in Microservices losing value as an idea and creates confusion in the industry, since everything is Microservices.

Microservices is a collection of services independent between them in a given context - eliminating out of bounds communication [30]. Their goal is to separate functionalities that could be run in the same environment, but it is chosen not to, to solve certain requirements. Similar ideas of highlighting the independence attribute already exist [16].

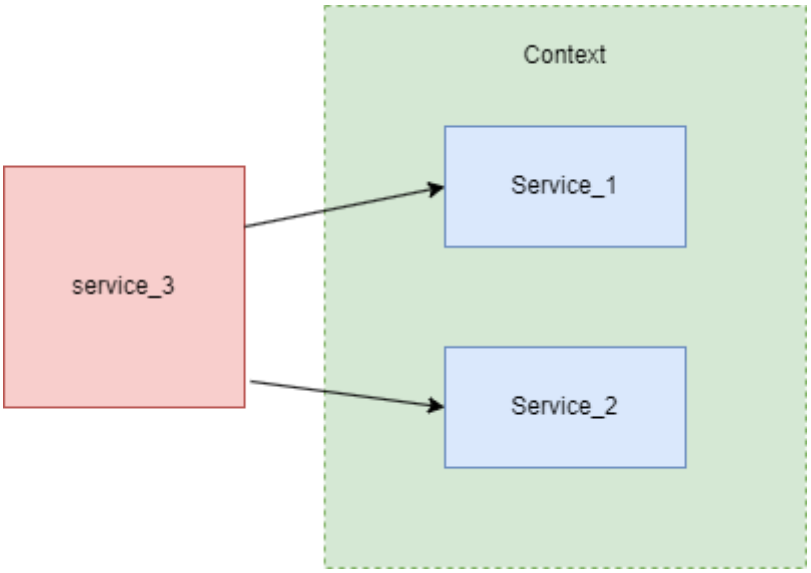


Figure 12 - Microservices Revisited diagram

The above image shows a MicroService architecture between Service_1 and Service_2. Service_3 depends on those 2 and its role is to combine the functionalities from each MicroService to perform a Use Case. The problem with the current Microservices is that they enable anti-patterns like Distributed Monoliths.

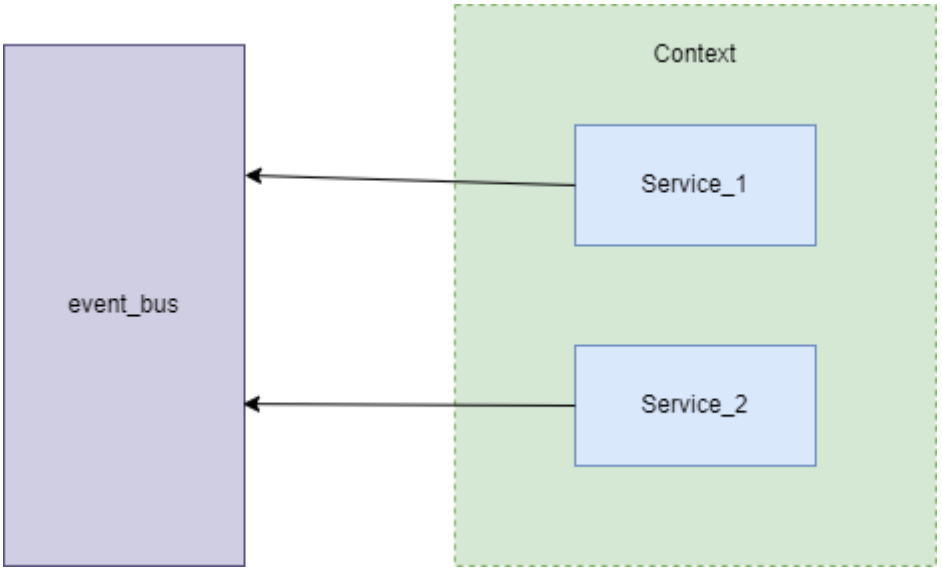


Figure 13- Hybrid architecture with EDA and Microservices

In real systems, Figure 12 has a problem of single point of failure, so the hybrid architecture of Microservices with the EDA was made. This hybrid architecture now utilizes asynchronous communication. Real systems may have their differences and may be a hybrid architecture of many things, and that is good. Adapting to the system's needs makes a project good.

5.6. Distributed Monolith

A Distributed Monolith is a collection of small - very granular - services as a result of their separation from an unstructured Monolith, keeping the dependencies the same in the system [11]. They usually are created trying to solve only the development requirements. It is an exaggeration of a regular SOA with the property of every service being small.

5.7. Anemic and Rich models

A rich model is a domain where its Objects have state that can change based on the functionalities that it has [41]. Anemic models on the other hand restrict the existence of a domain. They have a collection of Objects that are immutable and act as data holders, hence they don't have defined functionalities that reflect business logic. These Objects are often confused with Entity classes since they are also immutable and represent a collection of data without functionality that changes their state. But structures of data inside databases are different from domain Objects and how they represent their data. Create-Read-Update-Delete (CRUD) systems can work with anemic models since they have no business logic - acting as an interface of a database - and the development is faster and easier since Entities and Objects can be the same. But in most systems where business logic exists, rich models are better. As stated by Martin Fowler [41], anemic models are anti-patterns that are very common.

With the existence of Microservices as small services, functionalities in rich domains are broken down so that each service ends up with just having CRUD operations and some transactions. This means the domain functionality is also placed in the transactional layer.

One problem with this is that the transactional layer is not only responsible for scenarios but also for parts of domain functionality, hence violating single responsibility principle and having bad coherence. This leads to further difficulties in testing and problems with the collaboration of developers. Since everything is small, the domain spans across many services. This also creates distributed transactions, having the flow of information span across many services, thus the ability to understand and debug a business logic becomes harder. But with the redefinition of Microservices, rich models can become more useful and relevant.

5.8. Cloud Patterns

Cloud patterns can be applied to any type of software in a distributed context, big or small, structured or not. This means any type of SOA. Cloud patterns aim to enhance the system with additional functionalities to meet certain requirements [15]. Those requirements are divided in the following categories,

- Scalability
- Performance
- Software extensibility
- Reliability, error handling and recovery
- Deployment and production testing

An example is the Load Balancing pattern that helps with scalability. This pattern aims to distribute incoming requests among all the services of the same type based on the traffic. It consists of using dispatcher software that is deployed in the network as a service on its own. This service uses algorithms to distribute the work, and when that has some form of intelligence - adapting to the traffic dynamically - it becomes a load balancer. The services that handle the distributed requests can be structured Monoliths or not, have any size and have any form of communication between them.

There are many more patterns, but the Orchestrator and Choreography patterns are also worth mentioning since they both get confused with the Microservices and EDA architectures respectively. Those two patterns provide the best practice of implementing both architectures. The former uses a back end service to execute business logic across Microservices - as redefined -, while the latter implements the hybrid architecture that uses EDA with Microservices resulting in a single broker existing between the service and the Microservices.

6. Understanding the code

6.1. A Use Case

The following code demonstrates the thought process of creating a software system in a Distributed context. The code has an end goal to emulate banking operations like deposit and withdrawal.

6.2. Attempt: HexagonalSM

Repository Link

The first step is to build the webapp layer in a way that makes it easy to modify the server layer to explore various communication methods. Hexagonal Architecture was used to group the project into packages. The goal is to have the server layer and the webapp layer decoupled to start working towards the thought process of a Modular Monolith.

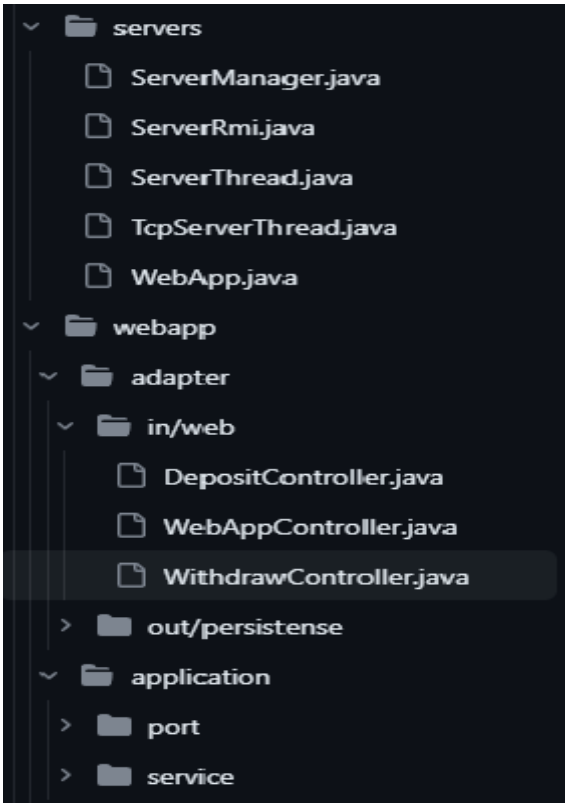


Figure 14 - Package structure of HexagonalSM

```
import webapp.application.port.in.DepositCommandData;
import webapp.application.port.in.WithdrawUseCase;
import webapp.application.service.WithdrawService;

class WithdrawController {
        private final WithdrawUseCase depositUseCase = new WithdrawService();
```

Figure 15 - Instantiation without DI

Without using DI, the decoupling of packages seems impossible since every field must be instantiated with an implementation that is in another package. The first step towards improvement is to introduce a DI framework to handle the development of such software. The idea now is to rely on a common web server to handle the communication and a library to map endpoints with controllers and take care of any coupling problems. Thus, Spring Framework was chosen to rely on the embedded Tomcat web server and the spring-boot-starter-web library to map the controllers. The next attempt will start to resemble the accompanied code [10].

6.3. Attempt: HexagonalSpring

Repository Link

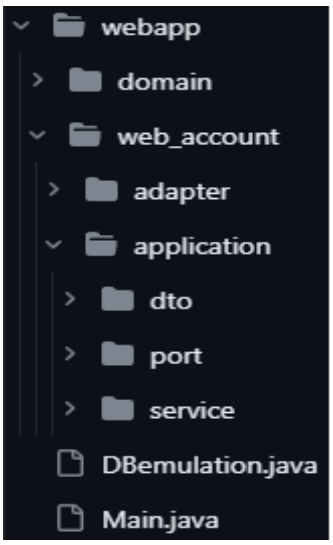


Figure 16 - Package structure of HexagonalSpring

At first, Spring-Boot eliminated the need for the server layer that handled the communication. Spring made it easy to map the inbound Controllers with the inbound requests, representing the entry point of the application.

```
import hexagonal_spring.webapp.web_account.application.dto.CommandBalanceDTO;
import hexagonal_spring.webapp.web_account.application.port.in.DepositUseCase;
import lombok.RequiredArgsConstructor;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequiredArgsConstructor
class DepositController {
        private final DepositUseCase depositUseCase;
```

Figure 17 - Instantiation with DI

Now the places that use the implementation of a port/interface don't need to import the implementations. In Figure 17, the DepositUseCase is used without providing an implementation. This enables the development of code that conforms to the ports. This is

analogous with the 'API-first' design [42]. This enables each package to be developed independently. This resulted in the following class diagram.

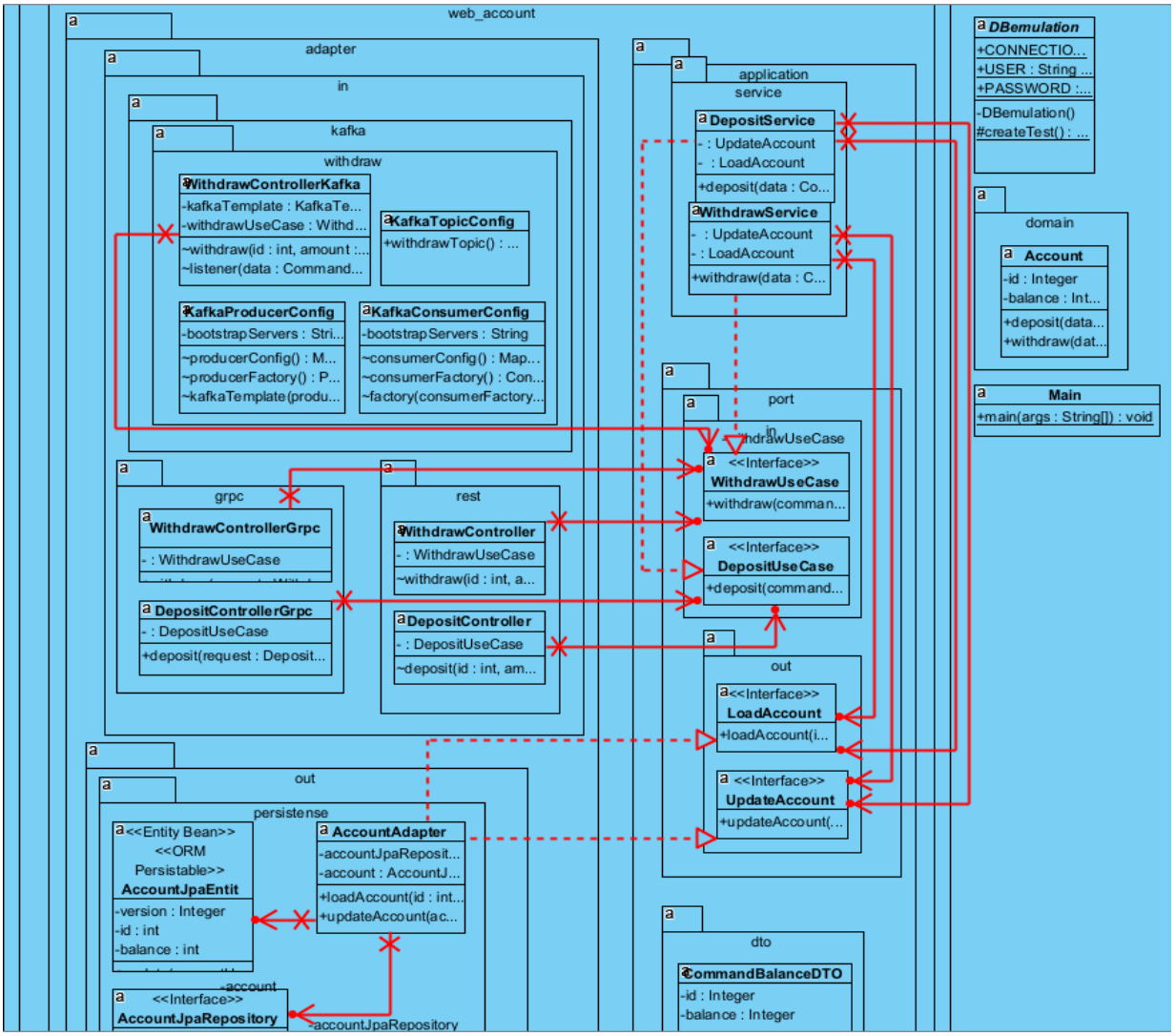


Figure 18 - Reversed class diagram with account subdomain

The project still resembles a Monolith. The first problem is the need of the entire codebase present to be able to work with a subset of packages. The second problem is that every change compiles the packages that were not affected by a single change - meaning that more subdomains would result in a difficult development process -. A solution to that problem would be to separate each subdomain as standalone projects and introduce internet communication as a way for the subdomains to collaborate. That was the starting point where the term Modular Monolith had to resolve those problems in a single runtime, without separating the project. But, as a new concept, an implementation or guide that uses a single runtime to handle the project as a collection of different pieces, doesn't exist.

6.4. Attempt: HexagonalSpring_Modular

Repository link

The tools to capitalize - towards standardizing the idea of the Modular Monolith - are Interfaces and the Dependency Injection to create Modules.

```
<modules>
    <module>main</module>

    <module>domain</module>
    <module>domain_imp</module>

    <module>web_modules/web_account</module>
    <module>web_modules/web_movie</module>

    <module>persistence_modules/queries</module>
    <module>persistence_modules/databasePrimary</module>
    <module>persistence_modules/databaseSecondary</module>
</modules>
```

Figure 19 - Parent pom modules

The picture shows the definitions of modules in the parent pom file of maven. There global dependencies are located that are common to all the children. Those modules were created after a manipulation and separation of the packages defined by Hexagonal architecture - including an additional sub-domain, movies -.

- The main module is created to initialize the server and inject the dependencies. It has the SpringBootApplication annotation and it is the executable.
- The domain module is created with Interfaces only - like APIs - containing global/core logic.
- The domain_imp which contains the implementations of the domain module.
- For each subdomain, a web_module is defined as a Hexagonal architecture.
  They handle the inbound and outbound requests of the subdomain (service).
- A queries module is defined, having Interfaces only (ports.out) - like APIs - that any service can use.

- The persistence of each web_module is separated on its own module (adapters.out.persistence). They contain schemas and query implementations.

The idea is to have functionalities collaborate with other functionalities in the same runtime using Interfaces - as rules that any functionality must comply with-.

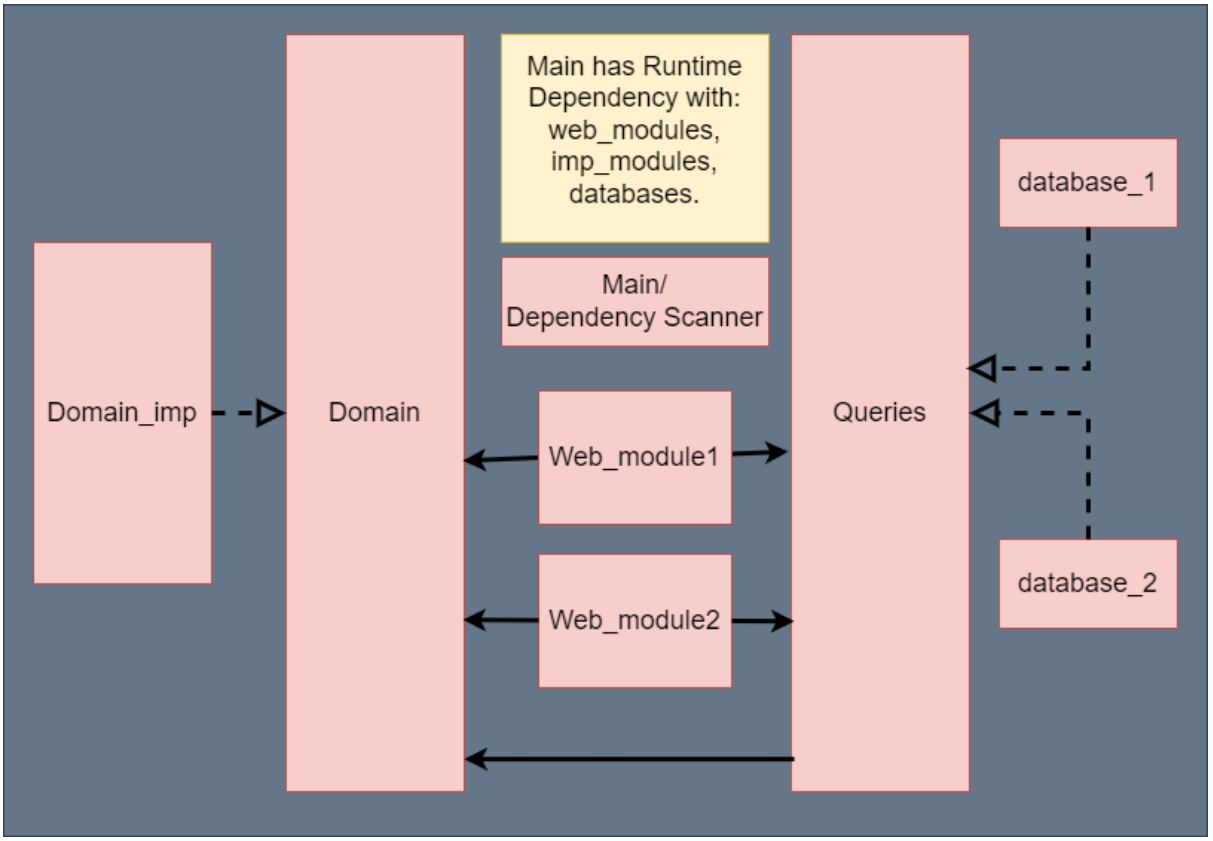


Figure 1 - Modular Monolith diagram

Figure 1 is the target dependency graph, with compile dependencies between [queries and domain] and [web_modules and (queries,domain)]. Figure 20 is a reversed dependency tree with both compile and runtime dependencies.

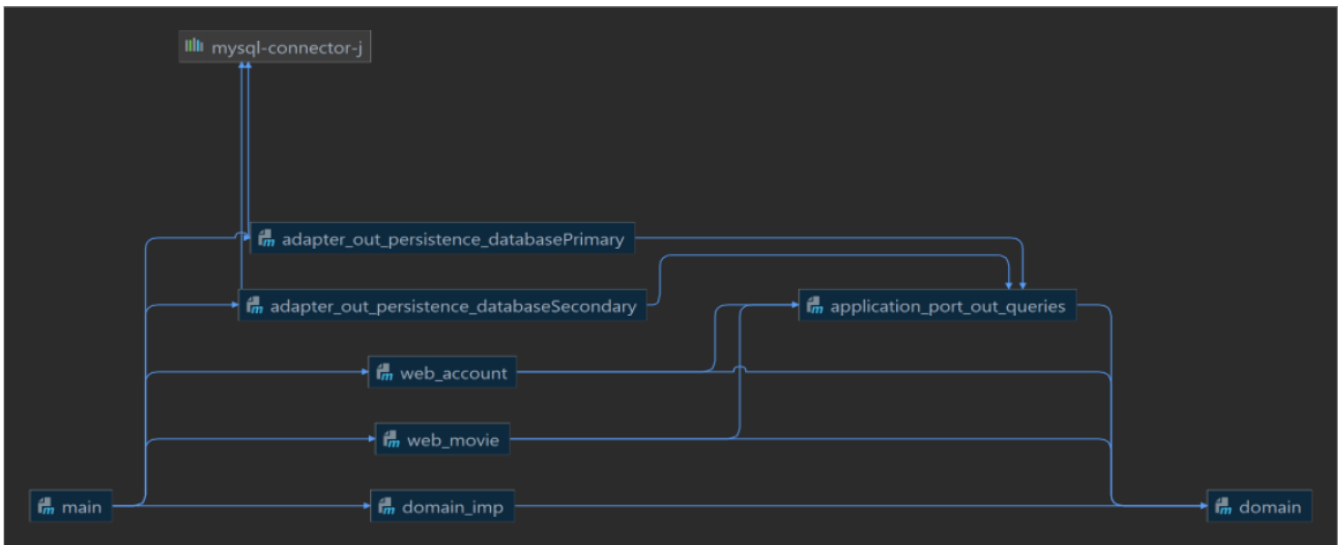


Figure 20 - Modular Monolith diagram reversed

With the help of Dependency Injection, the system can be enforced to resemble a tree. (Martin, R., p. 193) *"Source code dependencies must point only inward, toward higher-level policies."* [2].

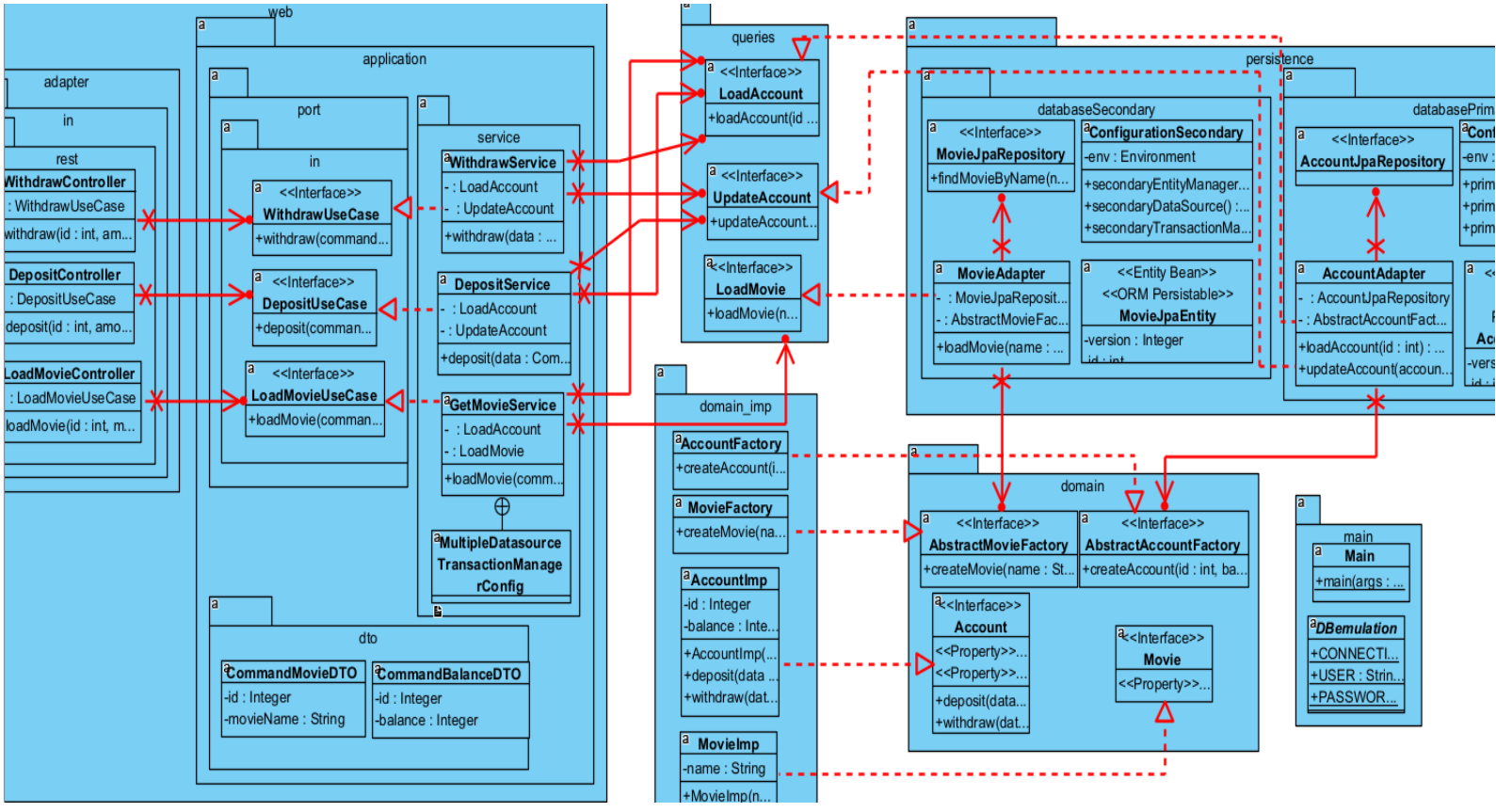


Figure 21 - Reversed class diagram with account and movies sub-domains

6.4.1. The code in depth

```java
import domain.Account;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import org.springframework.transaction.annotation.Transactional;
import queries.LoadAccount;
import queries.UpdateAccount;
import web.application.dto.CommandBalanceDTO;
import web.application.port.in.WithdrawUseCase;

@RequiredArgsConstructor
@Transactional("primaryTransactionManager")
@EnableTransactionManagement
@Service
class WithdrawService implements WithdrawUseCase {
    private final LoadAccount loadAccount;
    private final UpdateAccount updateAccount;

    @Override
    public String withdraw(CommandBalanceDTO data) {
        int money = data.getBalance();
        int id = data.getId();

        Account account = loadAccount.loadAccount(id);

        if (account.withdraw(money)) {
            updateAccount.updateAccount(account);
            int balanceResult = account.getBalance();
            return "Success! " + balanceResult;
        }
        return "failed";
    }
```

Figure 9 - WithdrawService

Analyzing Figure 9, the code implements a use case scenario - the withdrawal - that has the following flow/scenario:

- Loads an Account calling the loadAccount interface from queries.
- Uses the Account's functionality located in the Account interface to perform withdrawal, thus changing the state of the account.
- Updates the account in the database using an interface from queries if successful.
- Returns a message to the caller.

The code works without having a compile dependency with domain_imp or the databases. This results in working with the web_modules, domain_imp or the databases independently. The main module, having defined packaging paths, can inject the needed code - annotated by Component - at runtime.

```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.domain.EntityScan;

@SpringBootApplication(scanBasePackages = {"domain_imp","web","persistence"})
@EntityScan("persistence")
public class Main {

    public static void main(String[] args) {
        DBemulation.createTest();
        SpringApplication.run(Main.class, args);
    }
}
```

Figure 22 - Main module injecting with paths

In this way paths can be defined to implicitly inject classes that there is no access to.

```xml
<dependencies>

    <dependency>
        <groupId>tsechelidisMichail</groupId>
        <artifactId>web_account</artifactId>
        <version>${version}</version>
        <scope>runtime</scope>
    </dependency>

    <dependency>
        <groupId>tsechelidisMichail</groupId>
        <artifactId>web_movie</artifactId>
        <version>${version}</version>
        <scope>runtime</scope>
    </dependency>

</dependencies>
```

Figure 23 - Main module's defined functionalities as dependencies to execute

Then the web_modules define their dependencies in a similar matter. The same version is mandatory since web_modules must operate in the same context. An example would be having a Country's VAT being 20% and then having it changed to 15%. Since movies and accounts operate under the same VAT, they must have the same domain. This hints that there can be multiple global domains. There can even be local domains for each web_modules. This is the result of the combination of 2 design patterns [43]:

1) Separated Interface

   "Defines an interface in a separate package from its implementation."

2) Layer Supertype

   "A type that acts as the supertype for all types in its layer."

Those 2 combined define the common functionalities as interfaces in a package.

But there is a problem. Spring injects singleton objects/beans [14]. This means that each bean is created once in the system and every object that referenced that bean has access to its state, so singleton beans must have no state. That means that the injection domain - holding state - objects must be avoided. While there is a configuration that changes that in Spring, it would result in performance issues having the spring context handle much more beans that are created for every request. The solution to that is to also combine the Abstract Factory pattern [14], which separates the instantiation of an object with itself. Those factories having no state as a class, are those that are going to be injected in web_modules that will need access to domain objects with state. It is called abstract since they define an interface in the domain module exposing a createObject method that the concrete factory must implement in the domain_imp module.

```java
package domain_imp;

import domain.AbstractAccountFactory;
import domain.Account;
import org.springframework.stereotype.Component;

@Component
public class AccountFactory implements AbstractAccountFactory {

    @Override
    public Account createAccount(int id, int balance) {
        return new AccountImp(id, balance);
    }
```

Figure 24 - Injecting Factories

```java
import domain.AbstractAccountFactory;
import domain.Account;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Component;
import queries.LoadAccount;
import queries.UpdateAccount;

import javax.persistence.EntityNotFoundException;

@RequiredArgsConstructor
@Component
class AccountAdapter implements LoadAccount, UpdateAccount {

        private final AccountJpaRepository accountJpaRepository;

        private final AbstractAccountFactory accountFactory;

        @Override
        public Account loadAccount(int id) {
                AccountJpaEntity account = accountJpaRepository.findById(id).orElseThrow(EntityNotFoundException::new);
                return accountFactory.createAccount(account.getId(), account.getBalance());
        }

        @Override
        public void updateAccount(Account accountUpdated) {
                AccountJpaEntity account = accountJpaRepository.findById(accountUpdated.getId()).orElseThrow(EntityNotFoundException::new);
                accountJpaRepository.save(account.update(accountUpdated));
        }
```

Figure 25 - Adapting data from database to domain

In Figure 25 are the implementations of LoadAccount and UpdateAccount interfaces from queries. The module contains its respective database configuration and the schemas. The AbstractAccountFactory is also used for the adapters to have access to the account object. The adapter adapts data from the database to the domain. This enables multiple databases that comply with the domain. This enables services to explicitly define a runtime dependency with the databases that they need access to.

```java
@Configuration
@PropertySource({"classpath:application-databaseSecondary.properties"})
@EnableJpaRepositories(
        basePackages = "persistence.databaseSecondary",
        entityManagerFactoryRef = "secondaryEntityManagerFactory",
        transactionManagerRef = "secondaryTransactionManager"
)
public class ConfigurationSecondary {

        @Autowired
        private Environment env;

        @Bean
        public LocalContainerEntityManagerFactoryBean secondaryEntityManagerFactory() {
            LocalContainerEntityManagerFactoryBean em
                    = new LocalContainerEntityManagerFactoryBean();
            em.setDataSource(secondaryDataSource());
            em.setPackagesToScan(
                    new String[]{"persistence.databaseSecondary"});

            HibernateJpaVendorAdapter vendorAdapter
                    = new HibernateJpaVendorAdapter();
            em.setJpaVendorAdapter(vendorAdapter);
            HashMap<String, Object> properties = new HashMap<>();
            properties.put("hibernate.hbm2ddl.auto",
                    env.getProperty("spring.jpa.hibernate.ddl.auto"));
            em.setJpaPropertyMap(properties);

            return em;
        }

        @Bean(name = "secondaryDataSource")
        @ConfigurationProperties(prefix = "spring.datasource-secondary")
        public DataSource secondaryDataSource() {
            return DataSourceBuilder.create().build();
        }

        @Bean(name = "secondaryTransactionManager")
        public PlatformTransactionManager secondaryTransactionManager() {
            JpaTransactionManager transactionManager
                    = new JpaTransactionManager();
            transactionManager.setEntityManagerFactory(
                    secondaryEntityManagerFactory().getObject());
            return transactionManager;
```

Figure 26 - Configuring a database module

The only thing that a web module must do to use a database is to define the Transactional manager that is configured in the above code in each service function that needs it.

```java
@Transactional("primaryTransactionManager")
```

Figure 27 - Using the needed database at a transaction

6.4.2. Modular Compiles

At this point Modular Compiles can be witnessed. This means that each change is resulting in compiling only the code that is present in the changed module [2].

```
[INFO] domain ..................................... SUCCESS [  1.186 s]
[INFO] queries .................................... SUCCESS [  0.079 s]
[INFO] domain_imp ................................. SUCCESS [  0.084 s]
[INFO] databasePrimary ............................ SUCCESS [  0.106 s]
[INFO] web_account ................................ SUCCESS [  0.115 s]
[INFO] databaseSecondary .......................... SUCCESS [  0.100 s]
[INFO] web_movie .................................. SUCCESS [  0.054 s]
[INFO] main ....................................... SUCCESS [  0.057 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  2.283 s
```

Figure 28 - Compilation before any change

```
[INFO] domain ..................................... SUCCESS [  0.792 s]
[INFO] queries .................................... SUCCESS [  0.114 s]
[INFO] domain_imp ................................. SUCCESS [  0.069 s]
[INFO] databasePrimary ............................ SUCCESS [  0.118 s]
[INFO] web_account ................................ SUCCESS [  0.073 s]
[INFO] databaseSecondary .......................... SUCCESS [  0.058 s]
[INFO] web_movie .................................. SUCCESS [  1.931 s]
[INFO] main ....................................... SUCCESS [  0.108 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  3.716 s
```

Figure 29 - Compilation after a change in web_movie

Figure 29 shows a change on the web_movie module and the increase of compile time only to that module. But there is a problem with the queries and domain modules. If a change happens there, it is visible in all other modules that have compile dependency with them. This basically means all other modules.

The first question that arises is how often these modules change, since this can slow down the development process. One possibility was to have the project be in the early stages, so changes are frequent and a lot, since the domain is not established yet. But this means that the project is also small, so the negatives of not having modular compiles could not be a problem. The second possibility is to have the project be established and have modifications at a lower rate. But these are assumptions and Modular Monoliths' complexity must be worth it for the worst-case scenarios, even if those negatives can be seen in various SOA systems -changing interfaces is like changing APIs or changing queue definitions in brokers -.

6.5. Attempt: HSM-branched

Repository link

The idea is to have Git Branches separate each module to its respective branch [9]. Maven modules are not used, since there is no parent pom to compile the entire project - each module is compiled independently -. In this attempt, branches depend on code that the module needs at compile time. That means that every branch is created from a branch that has the dependent code (the interfaces of domain and queries).

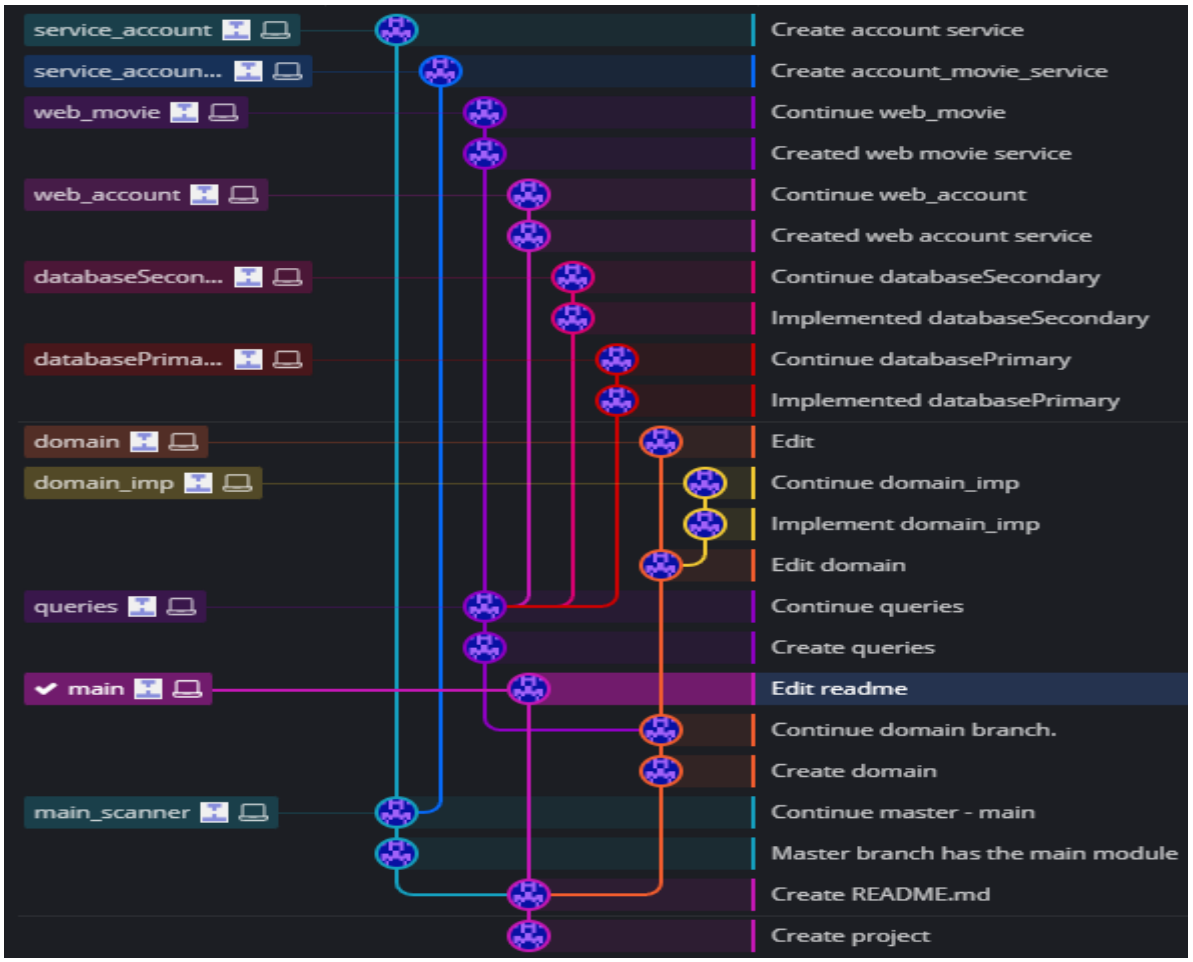


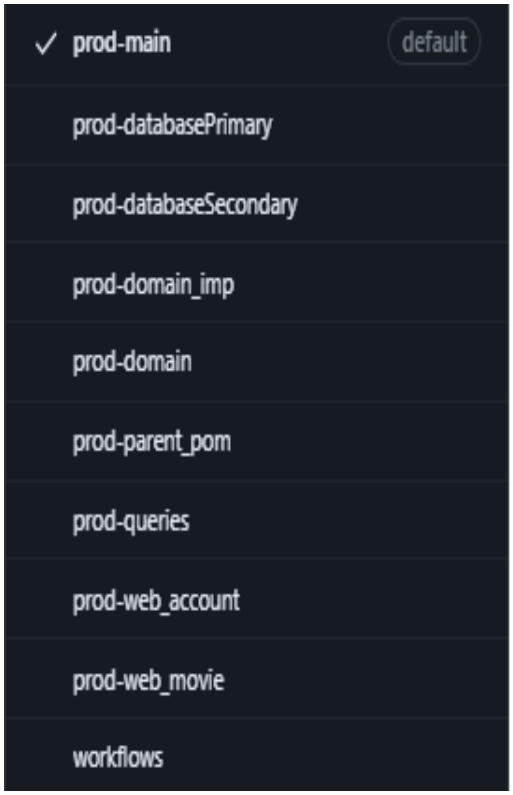
Figure 30 - Branches diagram without using artifact repository

While this can resolve the problem of a developer needing the entire codebase to operate, it creates management problems on handling branches correctly. It also isn't clear for developers what packages they can operate leading to further confusion. The final step is to have this dependent code as an available artifact in a common repository.

6.6. Solution: HSMB-v3

Repository link

Introducing Github Packages as a central repository for the modules - artifacts/jars - [9], branches are decoupled from each other. Each branch is independent and reflects the development of a module clearly.



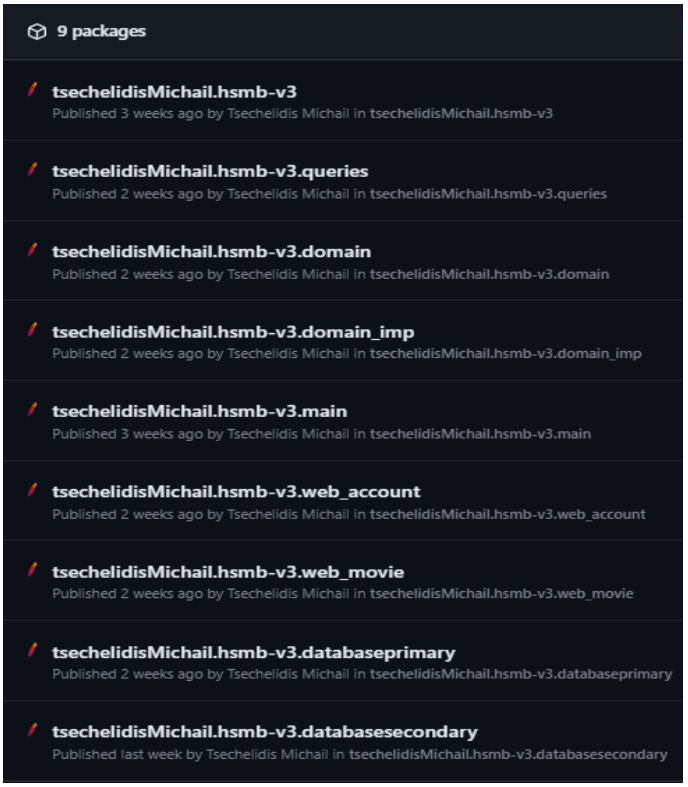


Figure 31 - Solution's branches, Figure 32 - Solution's artifacts/modules

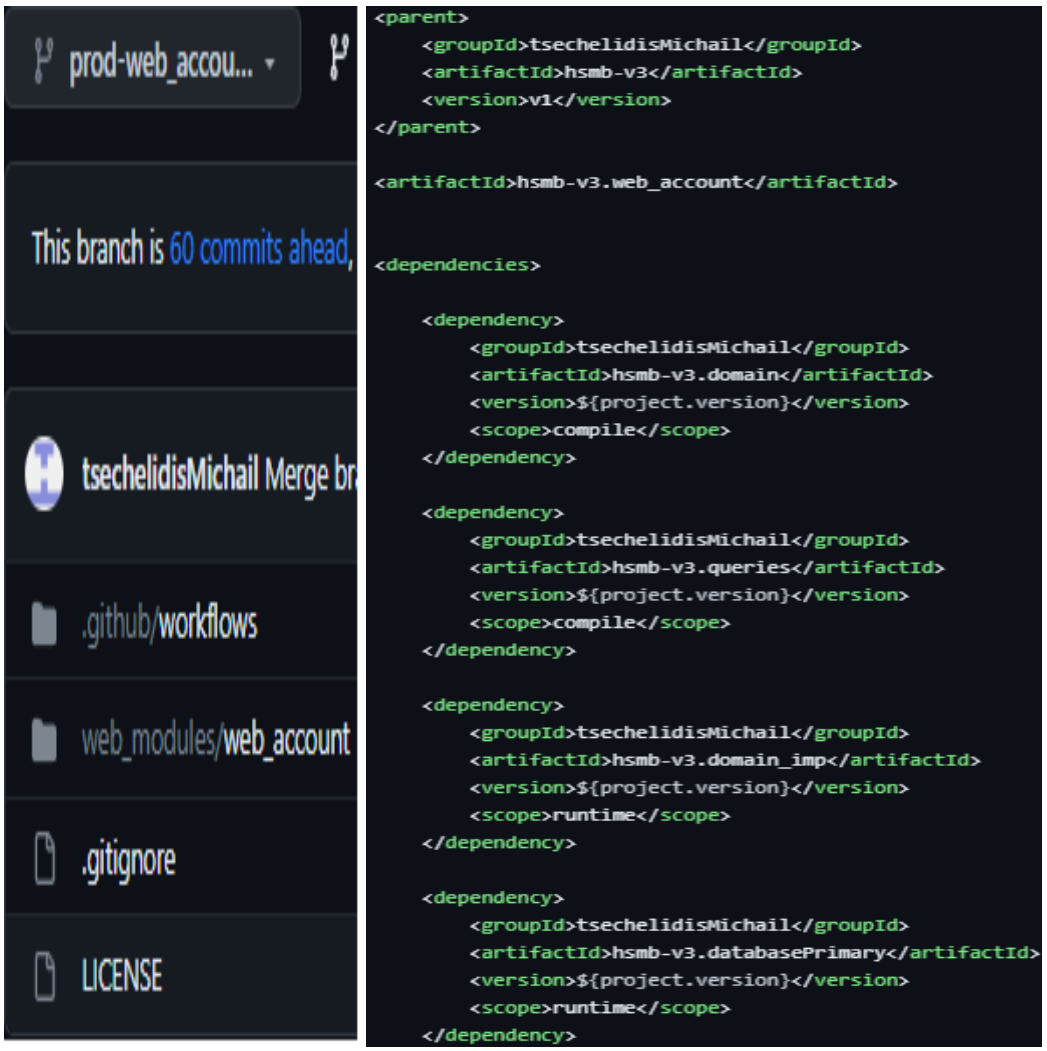


Figure 33 - Branch with no global code, Figure 34 - Module's dependencies

```
<distributionManagement>
    <repository>
        <id>tsechelidisMichail.hsmb-v3</id>
        <name>GitHub tsechelidisMichail Apache Maven Packages</name>
        <url>https://maven.pkg.github.com/tsechelidisMichail/hsmb-v3</url>
    </repository>
</distributionManagement>
```

Figure 35 - Definition of artifact repository

Each branch can be instantiated from a branch that only contains files such as licenses and readmes. Each prod-branch has its own lifecycle with additional branches that reflect the work being done by the developers. The compile dependencies are resolved from a repository defined in the parent pom by maven. Now each module has as parent that pom which is also resolved from the artifact repository. The parent does not have module definition, it just acts as a collection of common dependencies for the modules.

7. Case studies

At the time of writing Amazon announced that they switched a system of theirs from a MicroService architecture to a Monolithic one. They revealed that they reduced their system's cost by 90% [35]. They indeed had a system of small services and then they created a Monolith by combining those systems in one runtime, reflecting the previous architecture - its components and their dependencies -. Making an analogy for analysis, the before system can be represented by:
system(runtime1(functionality1), runtime2(functionality), runtime3(functionality))

When they realized that their system had been defined correctly and it had matured, they concluded that they can reflect that system in one runtime and keep the development process easy, since now the team had the experience to manage such a stable and mature system with clear components and a clear domain. The after system then became this:
system(runtime1(functionality1, functionality2, functionality3)

They kept the same domain and how they organized their components. This resulted in simpler infrastructure, now each functionality can communicate and store data locally, it also removed extra functionalities that were implementing Cloud patterns, for example now there is a single cluster of Load Balancer to manage a single cluster of Monoliths. This proves that there is value in starting by developing a Modular Monolith to enable the development process and then decide what to separate [39][40].

In addition, Shopify managed to achieve 5 nines (99.999%) SLA and achieved 1.27 million RPS with a monolithic architecture [23]. While it is not specified if it is modular or not, hence if the development requirements are solved, it shows that business requirements are resolved without the usage of Microservices - separation of environments -. Sopify's monolithic codebase did not become an obstacle to providing business value, even though the 1.27 million RPS is a big number.

8. Simpler Infrastructures

Modular monoliths make SOA not overused hence the infrastructures are simpler. This is the positive byproduct of controlling the granularity of services. What Modular Monoliths do is reduce the size of the system. When SOA is used without solving a problem, the negatives that are accompanying the SOA architectures are not justified [31][38]. Development requirements can be resolved by Modular Monoliths. The other types of requirements are situational.

8.1. Team Topologies

(Martin, R., p. 63) *"If the modules in your system can be deployed independently, then they can be developed independently by different teams. That's independent developability."* [2]. This is a Development Requirement that exists in every system. How to separate the workforce into teams [21]. The end solution of Modular Monoliths is a single codebase located in Github, that has multiple branches that reflect the modules of the system, and they can further reflect the teams and how the people inside them communicate. Conway's Law predicts: *"Organizations which design systems are constrained to produce systems which are copies of the communication structures of these organizations."* [44].

A way to map teams and developers to code is to have a branch per module. In distributed systems defining the modules is easier since every such system has some common parts of code. And the way to refactor that structure to meet more specialized requirements is easy as using the combination of patterns seen in "The code in depth". The fact that a developer can work on a branch independent of what happens in the rest of the project is important since there can be no merge conflicts. Even if there is a global/central dependency such as the domain, the nature of it having its changes reflected in an asynchronous manner to other parts of code, gives the feeling of independence to the developers.

8.2. Agile Development

This is a Development Requirement. Agile development is all about having a way to structure the development lifecycle to meet rapid changes by the business in the Use cases/requirements [22]. The independence makes it easy to accept changes and new features since the work and time being put to them doesn't affect other parts of code [2]. Even if the changes are meant to be done in a rapid fashion over a global/central domain, it allows the development of dependent parts to continue and having the changes be applied in an asynchronous manner. This enables DDD, defining business logic and redefining it whenever is needed is easy. Domain can become a reflection of most complex business logics, and yet not affect the surrounding functionalities that give value to this domain in the Distributed Context. Moreover, changes that are happening between modules are clear and transparent. The information of changes is not lost among the chain of communication between people. Developers can choose when to use the compiler to show possible integration problems.

8.3. Continuous Integration and Delivery

(Martin, R., p. 63) *"when the source code in a component changes, only that component needs to be redeployed. This is independent deployability."* [2]. The first thing that can be witnessed is the usage of a single repository as a single codebase. Pipelines - a chain of programs that build, test, deploy, etc the codebase - are also located in a single repository and can be used in a way of reusing such pipelines for different parts of the system with ease, depending on the branch. Each module uses only the needed modules to be compiled and tested. Modules' artifacts can be created and deployed to an artifact repository, such as Github packages, on their own.

8.4. Networks in the Cloud

Cloud patterns have additional software running to enhance the system, such as Load Balancers and Brokers [31]. These things cost and have a negative impact on the network. Those are mainly created by the extra network hops that a message has to make and the extra time of processing by each such extra software, increasing latency, network overhead and the complexity of the system, hence the management becomes harder, and it becomes costly to make sure the administration team can handle and optimize such a network. Modular Monoliths reduce the size of the system, since the end Services are much less, as Monoliths combined in a single runtime. The extra software that is needed is multiplied by much less Services. The entire network is smaller, Use Cases require less hops and less processing - like marshaling , dispatching, routing, etc -.

8.5. Resource savings - Green Computing

In a 2017 survey 25% of 16.000 cloud servers were doing **no** useful work [45]. A question arises of how much this survey's results would change with the proposed simpler infrastructures of Modular Monoliths, as seen in networks in the Cloud. It is very important to consider that since it affects many things. The first thing that affects is the carbon footprint of the data centers. To put this to perspective, a datacenter needs as much electricity as a medium sized country, and electricity comes at most - depending on the datacenter's location - from sources that have high carbon footprint (like power plants based on natural gasses). Additionally, this can have a significant impact on the companies' costs [31][35], meaning that going greener also benefits businesses. Lastly, resource savings hold significant importance in the telecommunications industry. Numerous studies focus on enhancing a system's network capacity to handle higher traffic through the adoption of new protocols, while Simpler Infrastructures could reduce the need of handling higher traffic.

9. Validation

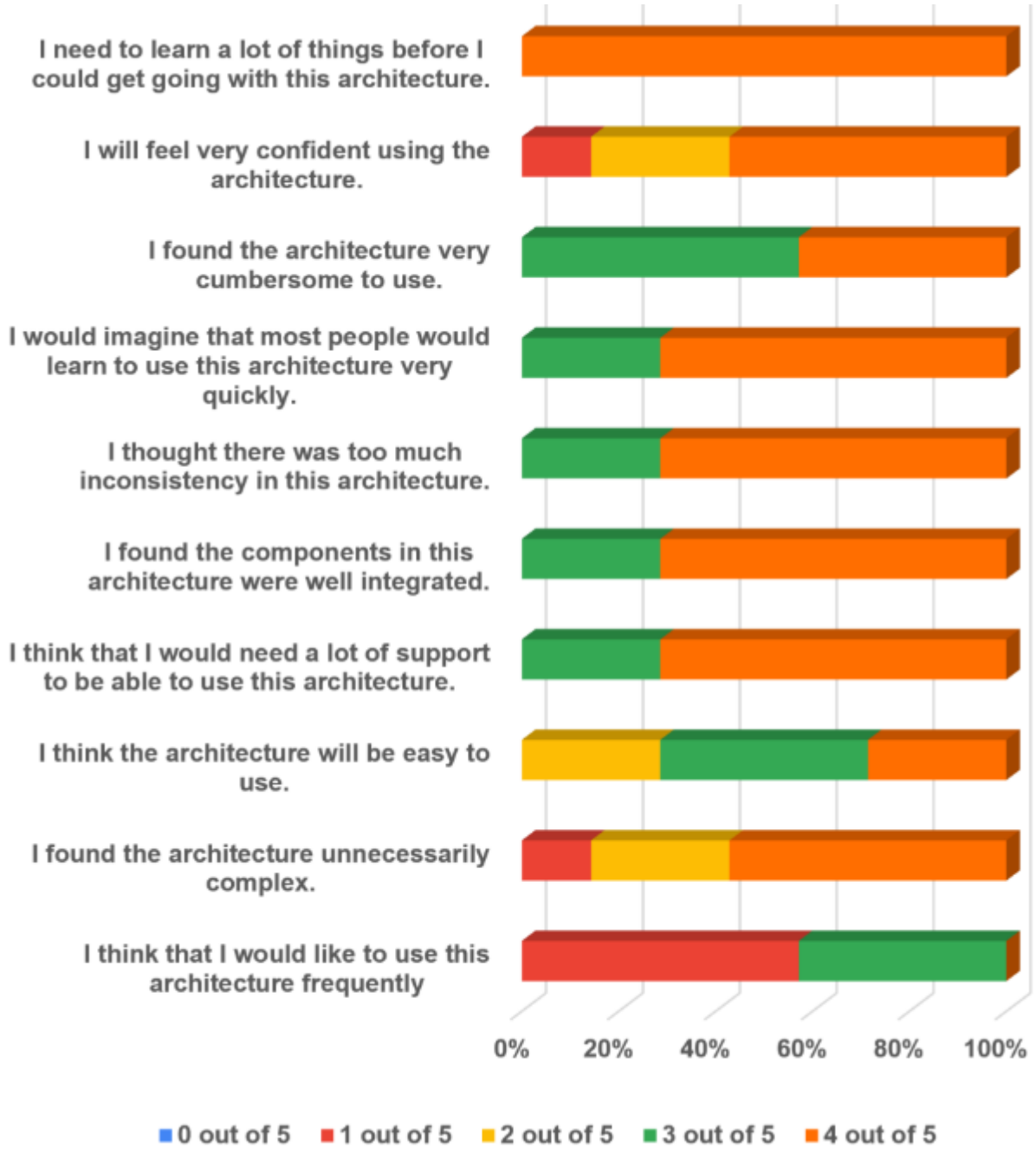


Figure 36 - User Acceptance

The 0/5 represents the Strongly Agree while the 4/5 represents the Strongly Disagree.

This graph is the result of a questionnaire that was answered by 12 architects from different companies [46]. They were asked both about the Modular Monolith but also the proposed development strategy. The research gave positive results, and it was perceived as a very interesting approach, with little answers being negative and many of them being

skeptical and considerable if not positive. Specifically, from the SUS' questions from Figure 36, it seems that the negative of the proposal lies in the fact that, as a new concept, developers would have to learn an architecture that is built with many different things to make it work. While Modular Monoliths could be beneficial, it seemed that Microservices were still the dominant choice to build large scale and complex systems.

It seemed that the Modular Monolith was not enough to convince them that the development requirements would be solved in real case scenarios, since Git branches being used in such a way would require much more effort to manage the system as a single huge codebase.

Some positive responses to certain questions suggest that implementing Modular Monoliths in large-scale systems could be worthwhile for companies. One question focused on the benefits of easily defining a subset of functionalities to create executables. This relates to the idea of "Controlling the Granularity of Services" and the potential cooperation of Modular Monoliths with SOA architectures. The second concern was about such Modular Monoliths being unnecessarily complex. Some strongly disagreed with this, arguing that the used techniques are solving arising problems. There were also architects who were interested in trying this approach in a real case to test its feasibility.

10. Future work

Lastly, I will mention some ideas for potential future research.

- Modular Monoliths and their proposed development strategy alongside Microservices in a large-scale system.
- Data analysis of the impact of services' size on RPS and SLAs
- CI/CD tools that enhance the SDLC of Modular Monoliths.
- Modular Monoliths with native executables and unikernels.
- Modular Monoliths with embedded functionalities to eliminate standalone systems in the network.
- Resource savings in IoT and networks with embedded systems with the help of Modular Monoliths.
- Data analysis on how much impact exactly Simpler Infrastructures have in Team Topologies, Agile Development and CI/CD.
- OSGi with Modular Monoliths to have dynamic control over the system.
- Mutable services with Modular Monoliths.

11. Conclusion

In this thesis it is proposed the potential standardized implementation of a Modular Monolith and its suggested development strategy. It is argued that the usage of Modular Monoliths resolves the development requirements before the usage of SOA.

It has redefined some terms to make it clear how and when to use certain system design techniques. It shows that the introduction of controlling the granularity of services with ease enables the cooperation of Modular Monoliths with SOA, which then can have the end infrastructures smaller and simplified. This showcased the potential positive impacts of having Simpler Infrastructures.

Lastly, the acceptance of Modular Monoliths is examined, and we conclude that while this thought process can be interesting, the nature of real systems - as large and complex - can make Modular Monoliths not feasible.

While the Modular Monolith adaptation to distributed systems was questioned, it seemed that there was a genuine interest in having a team apply that in the real world. So, until there is a working, large-scale system that uses Modular Monoliths in such a way, we can only have this thesis - and its contents - as a potential point of reference for future ideas to be researched and not as a source of truth.

12. List of Images

13. Bibliography - References

[1] Mclaughlin, D., Pollice, G. and West, D. (2006) *Head First Object–Oriented Analysis and Design: The Best Introduction to Object Oriented Programming*. Sebastopol, California: O′Reilly.

[2] Martin, R. (2017) *Clean architecture: A craftsman's guide to software structure and Design*. Upper Saddle River: Pearson.

[3] Coulouris, G. (2012) *Distributed systems: Concepts and design*. Harlow: Addison-Wesley.

[4] Fowler, M. (2020) 'DomainDrivenDesign', *Martinfowler*. Available at: *https://martinfowler.com/bliki/DomainDrivenDesign.html.*

[5] Henney, K. (2019) 'A system is not a tree', *Youtube*. SINFO, Portugal. Available at: *https://www.youtube.com/watch?v=je29nMb7fYM&ab_channel=SINFO.*

[6] Fowler, M. (2004) 'Inversion of Control Containers and the Dependency Injection pattern', *Martinfowler*. Available at: *https://martinfowler.com/articles/injection.html#FormsOfDependencyInjection.*

[7] Walls, C. (2022) *Spring in action, Sixth edition*. New York: Manning Publications Co. LLC.

[8] Redmond, E. and O'Brien, T. (2013) *Maven: The definitive guide*. Beijing: O'Reilly.

[9] Bell, P. and Beer, B. (2017) *Introducing github: A non-technical guide*. Beijing: O'Reilly.

[10] Hombergs, T. (2019) *Get your hands dirty on clean architecture: A hands-on guide to creating clean web applications with code examples in Java* [PDF]. Birmingham: Packt.

[11] Gorman, J. (2023) 'Climbing The Mountain of Modularity', *linkedin*, 4 June. Available at: *https://www.linkedin.com/pulse/climbing-mountain-modularity-jason-gorman/.*

[12] Brown, S. (2011) 'The C4 model for visualizing software architecture', *c4 model*. Available at: *https://c4model.com/*

[13] Henney, K. (2014) 'Seven ineffective coding habits of many programmers', *Youtube*. DevWeek. Available at: *https://www.youtube.com/watch?v=oyyFKHpzL0Q&ab_channel=DevWeekEvents*

[14] Gamma, E. et al. (2000) *Design patterns: Elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley.

[15] Pogrebinsky, M. (2023) ‘The Complete Cloud Computing Software Architecture Patterns’, *Udemy*. Available at: *https://www.udemy.com/course/the-complete-cloud-computing-software-architecture-patterns/* .

[16] Levan, M. (2023) 'Microservices: A Perspective and Overview’, *Github*, 4 January. Available at: *https://github.com/AdminTurnedDevOps/100DaysOfContainersAndOrchestration/blob/main/days/day-1-Why-Containers-And-What-Are-They.md#where-do-microservices-fit-in*.

[17] Kane, S. (2023) *Docker - up & running: Shipping reliable containers in production*. S.l.: O’Reilly Media.

[18] Burns, B. *et al.* (2022) *Kubernetes up & running dive into the future of Infrastructure*. Beijing China: O’Reilly Media.

[19] Henney, K. (2023) ‘Refactoring Is Not Just Clickbait‘, *Youtube*. NDC Conferences, London, United Kingdom of Great Britain and Northern Ireland. Available at: *https://www.youtube.com/watch?v=NMPeAW2RWdc&ab_channel=NDCConferences*

[20] Clandra, M. (2023) ‘Debunking monoliths' bad reputation’ *medium*, 17 February. Available at: *https://medium.com/geekculture/debunking-monoliths-bad-reputation-65d648c6a47*

[21] Skelton, M. and Pais, M. (2019) *Team topologies organizing business and technology teams for fast flow*. California: IT Revolution Press.

[22] Shore, J. (2022) *The Art of Agile Development, second edition. Sebastopol*: O’Reilly Media.

[23] Mihalcea, V. (2022) ‘Shopify monolith served 1.27 Million requests per second during Black Friday’ *reddit*, November. Available at: *https://www.reddit.com/r/programming/comments/z90juf/shopify_monolith_served_127_million_requests_per/*

[24] MacQueen, D. (1984). *Modules for Standard ML, LFP '84 Proceedings of the 1984 ACM Symposium on LISP and functional programming* , February. [PDF]

[25] Colebourne, S. (2018) ‘JPMS modules for library developers - negative benefits’ *joda*, 22 March. Available at: *https://blog.joda.org/2018/03/jpms-negative-benefits.html*

[26] IBM Team. (2023) ‘The OSGi Framework’ *ibm*, 13 February. Available at: *https://www.ibm.com/docs/en/was/8.5.5?topic=applications-osgi-framework*

[27] Hornsby, A. (2020) ‘Immutable Infrastructure’ *medium*, 30 March. Available at: *https://medium.com/the-cloud-architect/immutable-infrastructure-21f6613e7a23*

[28] OSGi Team. (2020) ‘OSGi Alliance IoT Vision’ *osgi*, 17 March. Available at: *https://blog.osgi.org/2020/03/osgi-alliance-iot-vision.html*

[29] Jrebel Team. (2018) ‘What Is a Modular Monolith?’ *jrebel*, 18 June. Available at: *https://www.jrebel.com/blog/what-is-a-modular-monolith*.

[30] Thomber (2017) ‘Communication Between Microservices: How to Avoid Common Problems’ *stackify*, 21 September. Available at: *https://stackify.com/communication-microservices-avoid-common-problems/*

[31] Heinrich, B. et al. *Granularity of Services An economic analysis* [PDF]

[32] Taylor, H. et al. (2009) *Event-driven architecture: How SOA enables the real-time enterprise*. Upper Saddle River, NJ: Addison-Wesley.

[33] Fowler, M. and Lewis, J. (2014) ‘Microservices a definition of this new architectural term’, *Martinfowler*. Available at: *https://martinfowler.com/articles/microservices.html.*

[34] IBM Team ‘What is FaaS (Function-as-a-Service)?’ *ibm*. Available at: *https://www.ibm.com/topics/faas*

[35] Kolny, M. (2023) ‘Scaling up the Prime Video audio/video monitoring service and reducing costs by 90%’, *Primevideotech*, 22 March. Available at: *https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90.*

[36] Wieder, P. et al. (2011) *Service Level Agreements for Cloud Computing*. Springer Science & Business Media.

[37] Behler, M. (2023) ‘Why you don’t need to worry about scaling your Java webapp‘, *Youtube*. Devoxx UK, United Kingdom of Great Britain and Northern Ireland. Available at: *https://www.youtube.com/watch?v=Ne2-tyL_SjQ&ab_channel=DevoxxUK.*

[38] Haesen, R. et al. (2008). *On the Definition of Service Granularity and Its Architectural Impact*. [PDF]

[39] Fowler, M. (2015) ‘MonolithFirst’, *Martinfowler*. Available at: *https://martinfowler.com/bliki/MonolithFirst.html*.

[40] Anderson, T. (2023) ‘Reduce costs by 90% by moving from microservices to monolith: Amazon internal case study raises eyebrows’, *devclass*, 5 May. Available at:

https://devclass.com/2023/05/05/reduce-costs-by-90-by-moving-from-microservices-to-monolith-amazon-internal-case-study-raises-eyebrows/.

[41] Fowler, M. (2003) 'AnemicDomainModel', *Martinfowler*. Available at: *https://martinfowler.com/bliki/AnemicDomainModel.html.*

[42] Postman Team 'Guide to API-first' *postman*. Available at: *https://www.postman.com/api-first/*

[43] Fowler, M. (2015b) *Patterns of enterprise application architecture*. Boston, MA, USA: Addison-Wesley.

[44] Brooks, F.P. (1995) *Mythical man month: Essays on Software Engineering*. Reading: Addison-Wesley.

[45] Cummins, H. (2023) 'Writing Greener Java Applications', Youtube. Devoxx UK, United Kingdom of Great Britain and Northern Ireland. Available at: *https://www.youtube.com/watch?v=kwnnbvwXVXY&ab_channel=DevoxxUK.*

[46] Tsechelidis, M. et al. (2023) 'Modular Monoliths the way to Standardization', 3rd Eclipse Security, AI, Architecture and Modelling Conference on Cloud to Edge Continuum (ESAAM 2023), Germany, 2023. Available at: *https://www.researchgate.net/publication/373195134_Modular_Monoliths_the_way_to_Standardization*