# Converting a monolithic application into a modular architecture

Jessie Liauw A Fong
**Student number**: 500726467
**Study**: Software Engineering
**University**: Amsterdam University of Applied Sciences

August 12, 2019

# Preface

I am Jessie Liauw A Fong. Born on April 1th 1998 in Amsterdam. When I was three years old I moved to Zaandam, in which I still live. My programming journey started in 2010 in the first grade of middle school. I was programming on the Texas Instruments TI-84 Plus calculator we got for math. Even though I loved programming this early on, my first actual experience began in 2015 when I started the study software engineering at the Amsterdam University of Applied Sciences. In late 2015 I started a voluntary job as software engineer at the EsportsWall. I decided to take this job so I could improve my skills. This job helped me get an internship and eventually a job at Endouble, where I worked for a year as a software engineer.

At the end of my time at Endouble I started a new parttime internship at Ximedes where I learned about software architecture and infrastructure. This is also the company I met my now co-worker Erik Schouten. He worked at CargoLedger where I currently work as a software & devops engineer. At the start of September 2018 I started a new company together with Stijn Claessen and Siebe Goos called EFFE Planning. EFFE makes an automated employee schedule application where clarity, ease of use and efficiency are central. This application will be the subject of this research.

# Contents

# List of Figures

# Chapter 1

# Summary

The company EFFE Planning makes employee schedule software. The business model of EFFE requires interchangeable modules, so EFFE can cater to bigger clients. The question this research will answer is:

"What is the best way to transform a monolith into a modular architecture, where the modules are interchangeable with each other"

The first course of action was to order the quality attributes. These quality attributes were used to decide which modular architecture is the best fit for EFFE. The research concluded that the modular monolith architecture fits best with EFFE's priorities.

The backend framework used to implement the modular monolith is Django. Django is domain driven out of the box. In the frontend Vue was the framework that best aligned with the order of the quality attributes. In the implementation Nuxt.js was used which is a framework on top of Vue. Nuxt.js was used because the current application already uses it.

When implementing the modular monolith there was a need to create an assembler and dissembler for the modules. This makes it easier for developers and in the deployment lane. The result fo this was a CLI (Command Line Interface) tool called https://github.com/jessielaf/modad

# Chapter 2

# Introduction

## 2.1 EFFE Planning

EFFE Planning, hereinafter referred to as EFFE, relieves the pain of inefficient scheduling with their automatic scheduling system. EFFE has build software that prevents companies from making mistakes, lowers their cost and makes it easier for both the planner and the employee. EFFE distinguishes themselves from the competition through its automated system and intuitive user interface. The competition started ten years ago and is still focused on software, where the planner need to link the employee with the shift. This is very time-inefficient. The basis of their software is build upon this philosophy, while EFFE has its focus on a 100% automated system. Automation is the future and EFFE wants to become the leading player in this market.

## 2.2 Motive

EFFE as a company uses the SaaS (Software as a Service) model in order to comply to it's expected growth. The basic SaaS model includes the basic application or MVP. This is in order to keep the application as abstract as possible. So that every company can connect their scheduling procedure to EFFE. EFFE also wants to cater to the needs of bigger clients. This is why EFFE has created building blocks. More about the actual definition of building blocks can be found in 3.1.1 The problem

Building blocks are features that can be added/removed from the application

by the user or by EFFE. Examples of building blocks are white labeling, integration with frontend system and payrolling integration. These building blocks are not required when acquiring EFFE but can be added one by one.

## 2.3 Intention

So the question is, how is EFFE going to implement these building blocks. There are a few requirements:

- The building blocks need to be interchangeable. Meaning the same building block can be changed with another one that does the same job with slightly changed business logic.
- The building blocks should be able to do everything programming related. From "if else" to database calls.
- The building blocks can be either frontend or backend related
- Building block should be completely separate from the application (loosely coupled)

# Chapter 3

# Research design

This chapter contains the information on how the research will be conducted.

## 3.1 Research objective

### 3.1.1 The problem

Right now EFFE is developing an application for employment agencies in which those employment agencies can schedule their employees automatically. To implement custom features that some clients want, EFFE has decided to add something to the business model called building blocks.

> "Building blocks are interchangeable implementations of business logic that can be reused as efficiently as possible"

### 3.1.2 Objective

The objective is to create a recommendation for an architecture, how to create and maintain such an architecture, where the focus lays on interchangeability and modularity of the different functionalities.

| Stakeholder | Interest to the objective |
|---|---|
| EFFE | EFFE will use this research to enhance its business model. EFFE will also create a better infrastructure which means that the company can implement functionalities faster and cater better to the clients. |
| Client | The client is not interested in what happens behind the scenes, but the opportunity it brings when partnering with EFFE. |

## 3.2 Research framework

### 3.2.1 Objects

This chapter describes who/what the objects are for this research and why.

#### 3.2.1.1 Backend architecture

Arguably the most important object is the backend architecture. There are already a great number of researches available regarding backend architecture. The backend is also the place where the business logic will be expressed.

#### 3.2.1.2 Frontend architecture

The second object, frontend architecture, is a lesser known subject when looking at modularity of the actual system. Most of the big companies have a single frontend application per platform.

#### 3.2.1.3 Deployment lane

The backend and the frontend are the software side of the equation but the hardware is also important. The deployment lane is the section that pieces it all together. This object creates the hardware or virtual hardware and sets this hardware up so it can then proceed to deploy the frontend and backend on the just created hardware. How does the current deployment lane change with the new architectures?

#### 3.2.1.4 Software architect

The last object to be researched is the software architect. The software architect's job is to design how the system will be build. This can be on a small

scale like naming conventions but also bigger scale like layered architecture or modular architecture. Even the programming language can be decided by a software architect.

### 3.2.2 Research perspective

The research perspective is straight forward. Because this research is written by one of the founders of EFFE it is best to approach this research from the side of EFFE. This means that there will be more emphasis on maintainability than for example performance. Because performance can be bought later on by upgrading hardware but maintainability can only be influenced at the start of a project.

### 3.2.3 Research sources

This section will describe which sources will be used when evaluating the research objects. This will not include everything but a broad spectrum of the sources that may be used in the research:

- **Modular architecture books:** This research comes down to modular programming. Modular programming is a very broad term and it is important to find how someone else may look at this term.
- **Implementations of modular architectures:** Theory is one side of the coin. Everything can work perfectly in theory but when implementing the theoretic side there will be problems that weren't considered before.
- **Critique from outside:** It is known that software architecture is an opinion based subject. There are no official accepted right way to do certain things. This is because especially this area of software development is fairly young. Software architecture did not have much time to develop itself as much as some other aspects of software engineering such as operating systems. Because software architecture is young there are a great number of people voicing their opinions and it is important to look at the criticism on some of the architectures.
- **Researches on deployment of an architecture:** There will be more than one architecture researched. Each architecture has its own development environment and deployment environment. The infrastructure of the servers on which the program runs is heavily influenced by the architecture of the software.

### 3.2.4 Evaluation criteria

These are the criteria or leading questions that will be asked to research objects. Note: not all evaluation criteria apply to all research objects:

- **What is the biggest pitfall when implementing a new architecture:** As mentioned in 3.2.1.4 Software architect the software architect makes the choices around the architecture. So it is important to look at what can go wrong when implementing a new architecture. What are the common pitfalls they have experienced when implementing a new architecture.
- **What are the most used architectures in this area:** There is always a reason why one architecture is very common and the other one isn't. In the research the reasoning will be extracted and reflected on.
- **What are upcoming architectures that are focused on modularity:** Again the whole research is based on the building blocks. These modular functionalities that can be designed via a common interface. What is the difference in solution between these architectures?
- **Which programming languages has the best attributes to complement the modularity:** Some languages are written purely for scripting or some are written to be focused on implementing algorithms more easily. Each programming language has its attributes and which of these attributes are most defining and important to a modular system.
- **Which quality attributes are deemed most important to EFFE?:** The quality attributes from ISO 205010 [5] are the backbone of the architecture. The research will prioritize these quality attributes based on what is most important to EFFE.

### 3.2.5 Research framework

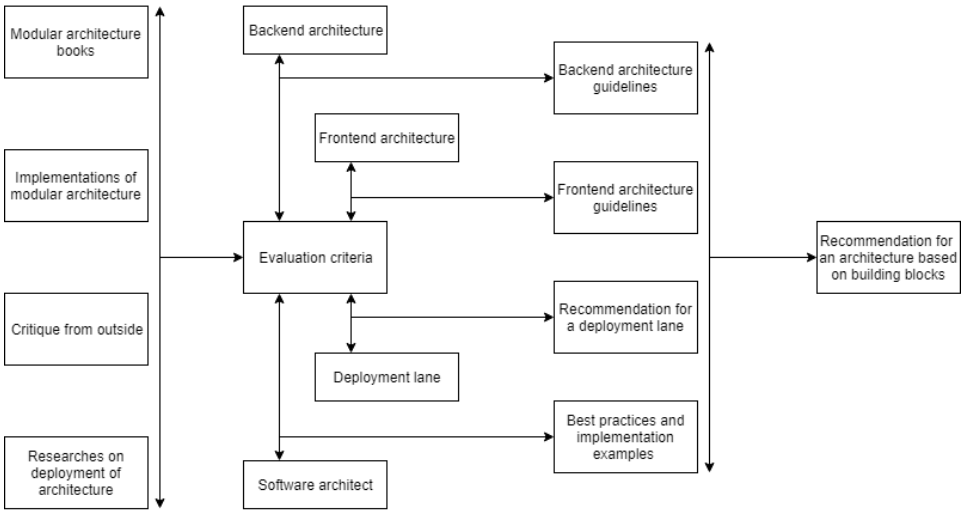


Figure 3.1: Research framework

### 3.2.6 Expected Results

The results will be the guidelines on which the practical part of this research will be based.

- **Backend architecture guidelines:** These are the guidelines on which the backend architecture will be based. These guidelines will indicate why the choice for a certain approach was made and what the specific approach is.
- **Frontend architecture guidelines:** Like with the backend guidelines, these guidelines will also contain the reasoning for each decision made.
- **Recommendation for a deployment lane:** As mentioned in 3.2.1.3 Deployment lane the deployment lane can impact the backend architecture and vise versa. How does this deployment lane differ from the current deployment lane and why?
- **Best practices and implementation examples:** The software architect's experiences when implementing new architectures together with the implementation of the chosen architecture will conclude in the best practices and the implementation examples.

## 3.3 Research Questions

Note: question 2 and 3 will be handled separately for both backend and frontend.

The main question this research will be answering is:

"What is the best way to transform a monolith into a modular architecture, where the modules are interchangeable with each other"

### 3.3.1 Question 1

The first question is about software architecture. How does a software architect create a software architecture.

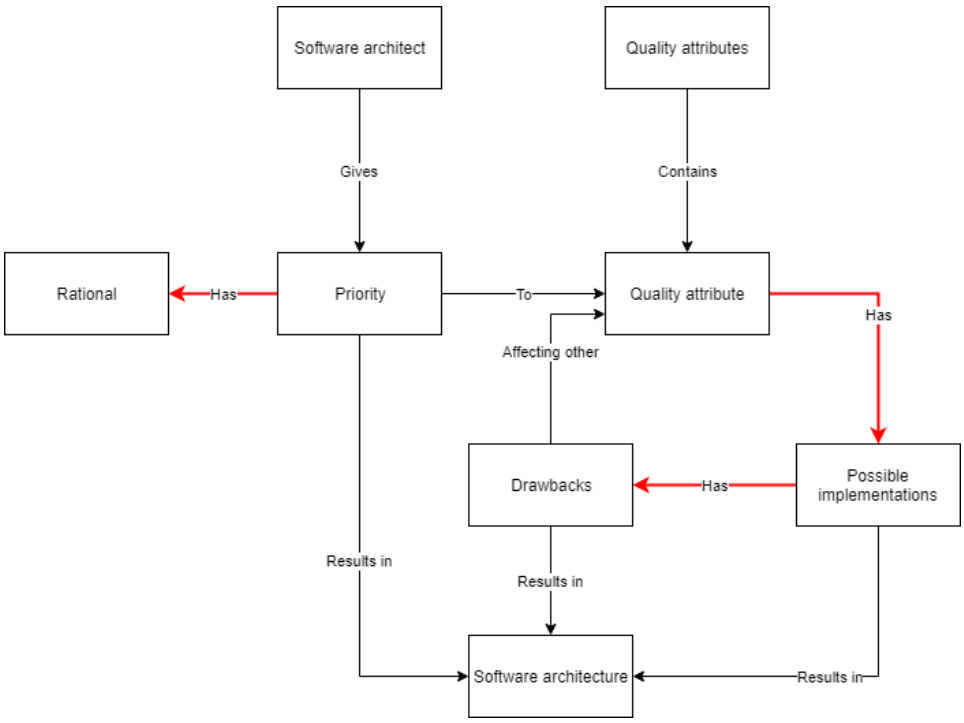


Figure 3.2: How a software architecture is chosen

The red lines show the parts of the figure that will be explored in the question. Thus the question is:

"What was the thought process behind choosing a software architecture whilst considering each implementation and its quality attributes?"

This question will explore how a software architect chooses their architecture. This will give more insights into what they consider when choosing an implementation so that their rational can be extracted and taken into consideration.

These are some of the sub questions that will be handled based on this central question

- Which techniques are used when mapping the priority and the drawbacks in order to make a decision?
- How does the priority of a quality attribute influence the end result or software architecture?
- How does the software architect combine the priority, drawbacks and possible implementations to a software architecture?

### 3.3.2 Question 2

"What are the best software architectures that mainly focus on modulairty?"

This question focuses on the architectures that are available. The knowledge of how a software architect chooses the architecture is answered in the previous question 3.3.1 Question 1. Because of the new perspective gained in the previous question there can be a more nuanced choice of architecture.

Sub questions:

- What are the up- and downsides of each architecture?
- How mature is the documentation and research surrounding the architecture?
- Which architecture implements the quality attributes that EFFE deemed important best?

### 3.3.3 Question 3

"Which implementations are there of the solutions provided for modular architecture?"

The solutions or architectures provided from 3.3.2 Question 2 will have implementations. These implementations may be a theoretic paper or a framework which implements this architecture. This question explores how and why these implementations have been made. Other questions that will be answered are:

- How mature is the architecture in contrast to the implementations?
- How does the language chosen in the implementation reflect to the architecture?
- On what level does the framework compromise which is not reflected in the architecture?

### 3.3.4 Question 4

"What are the key elements of a software architecture that will influence the deployment lane?"

This question hints at the relation between a software architecture and the deployment lane. Right now there is a limited view on how the deployment lane should be and how it can be. In order for the practical research to work there needs to be an answer to these questions:

- Which infrastructure fits best with my chosen architecture?
- What are the costs of different infrastructures?

## 3.4 Methods

The methods used to conduct this research will be explained below.

### 3.4.1 Interviews

The interviews will be conducted with pre-stated questions. These questions may require follow up questions to get a more detailed view. All of the interviews will be recorded and typed out in the 9.1.1 Questions.

### 3.4.2 Desk research

Most of the research will be desk research. There are lots of studies and small blogs written about the main question. People have shared their experience implementing architectures and their thoughts on it.

# Chapter 4

# Choosing an architecture

This chapter will view what goes into choosing a software architecture. What should you consider when choosing one and why.

A software architecture is:

> "The process of converting software characteristics such as flexibility, scalability, feasibility, reusability, and security into a structured solution that meets the technical and the business expectations. [6]"

## 4.1 Priorities

As mentioned in the definition of software architecture, 4 Choosing an architecture, a software architecture looks at the characteristics as flexibility, scalability, ect. These characteristics and their sub characteristics are defined by ISO 25010 [5]. This is the standard that will be used troughout the research.

It is important to state the order in which EFFE values these quality attributes. Every decision will be based and rationalized by this order.

As mentioned in 2.3 Intention the first point points out the modularity and the interchangeability of these building blocks. The **maintainability** quality attribute has reusability and modularity as its sub characteristic. Thus is this the first focus of the software architecture.

Replaceability is a subcharacteristic of **portability**. As mentioned in 3.1.1

The problem the building blocks are interchangeable which is a direct characteristic of replaceability.

The third focus is **compatability**. Compatibility is the degree in which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment [5]. This is directly linked with the modularity of the system.

As mentioned in first and second point in 2.3 Intention the functionality may be shared between building blocks. But each building block should be able to function without other building blocks. This is why **functional sustainability** will be the fourth focus.

With such a loosely coupled system **security** becomes a bigger risk. Because every functionality is loosely coupled it means that the functionalities may talk with each other over a network instead of code. If it is an open network the security needs to be checked constantly. On a closed network measures need to be taken to keep the network closed. That is the reason why security is our fifth focus.

After running through these five quality attributes we have an application that can function without being overtaken by unintentional users. But in order to keep the intentional users satisfied the services or functions need to be reliable. Thus **reliability** will be our sixth focus.

If something is reliable it does not mean that it is usable. If the application is not responding as fast as possible there is a chance that the user will leave the application. A study conducted in 2018 by Google showed that there is a 58% bounce rate when the load time is between 3 to 5 seconds [7]. Thus in order for our users to actually be able to use the application in a responsive manner **performance efficiency** becomes our seventh focus.

Normally there can be a solid argument made about why usability would be higher in the rankings. But because this research more focussed on the architecture of the application and not UX or UI, **usability** is our eighth focus.

### 4.1.1 Recap

1. Maintainability
2. Portability

3. Compatibility
4. Functional sustainability
5. Security
6. Reliability
7. Performance efficiency
8. Usability

## 4.2 Creating an architecture

Compared to choosing an architecture, creating one is something entirely different. An architecture does not exactly have a creator. An architecture is just blueprint on how to write the software. This is why the choice was made to interview software architects and ask them questions how they make their choices.

Before conducting the interview there is a need to have more information in order to frame the questions the right way. An architecture is assembly of implementations of certain features. These features are implemented to make the architecture fulfill a certain quality attribute. But each implementation has their drawbacks. These drawbacks can affect other quality attributes. This thought process is visualized in the figure below.

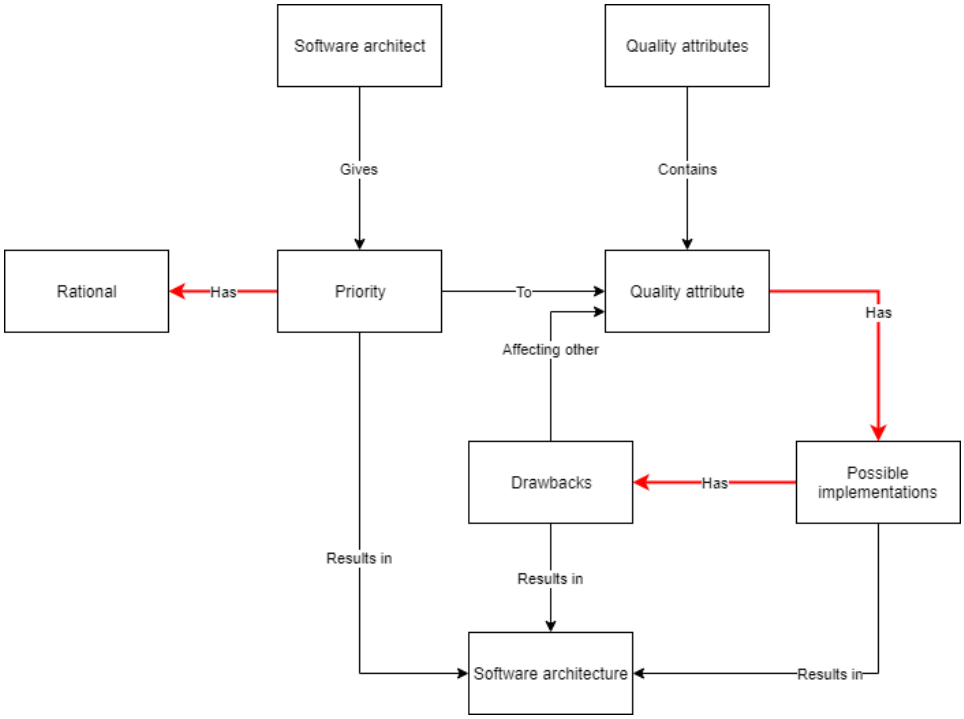


Figure 4.1: How a software architecture is chosen

The interviews showed that the understanding of software architecture portrayed in this research is the same as that of software architects. When explaining the thought process behind the image there was a positive feedback from the architects where they agreed that this is how they implement their own architectures and how they evaluate their implementations. The critique the figure got, from one of the interviewees, is that maybe the software architect does not give priority to the quality attributes but the client of the product or the product owner does. This is because these parties formulate the requirements. So the figure is changed to reflect this:

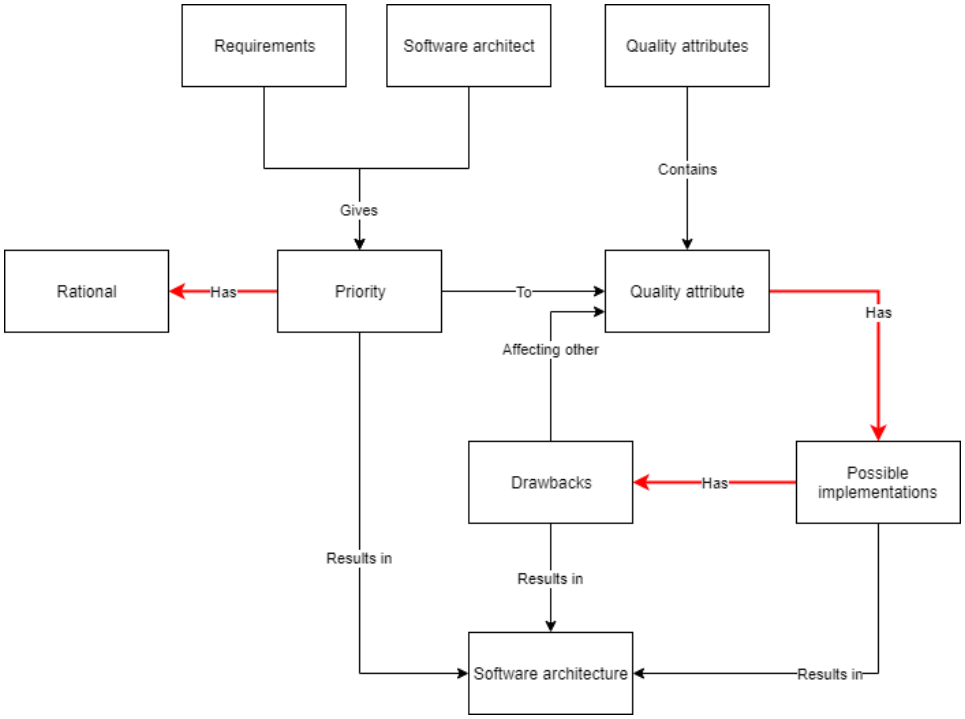


Figure 4.2: How a software architecture is chosen with requirements

# Chapter 5

# Modular architecture

A modular architecture:

> "Modular design or “modularity in design” is a design approach that subdivides a system into smaller parts called modules or skids that can be independently created and then used in different systems. A modular system is characterized by functional partitioning into discrete scalable and reusable modules, rigorous use of well-defined modular interfaces and making use of industry standards for interfaces. [22]"

When researching famous architectures in software there are a few examples of non modular architectures. Such an architecture is the layered architecture. In the image below is shown how such an architecture operates.

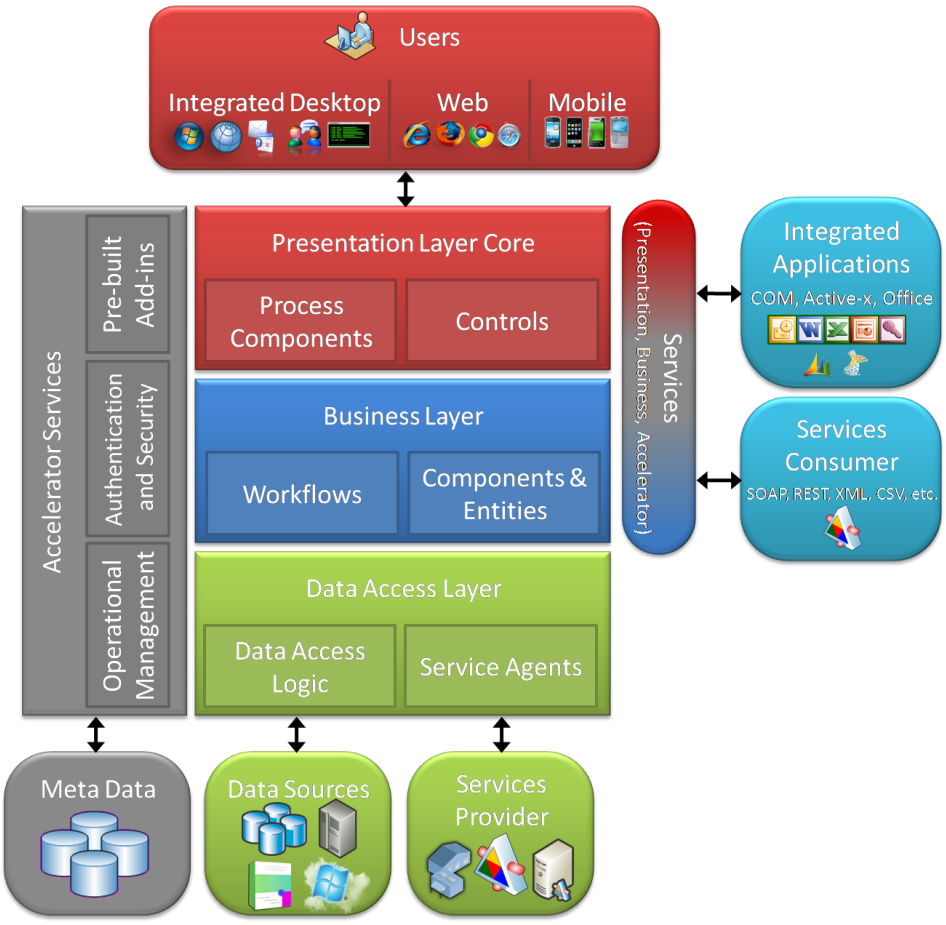


Figure 5.1: Layered architecture [24]

As shown in the figure the architecture is layered based on responsibilities. Each layer having its own purpose. The layers can talk with each other but they are intertwined. This means that a class or object in the presentation layer can talk to the same business layer object as another presentation layer class. This concludes in the objects being highly coupled.

A modular architecture is based upon modules. A module is:

> "deployable, manageable, natively reusable, composable, stateless unit of software thatprovides a concise interface to consumers" [20]"

This is eerily similar as the description of what this research calls building blocks in 3.1.1 The problem

## 5.1 Architectures

### 5.1.1 Microservices

If most software engineers in 2019 think of a modular software architecture the first architecture that comes to mind is microservices. In the last years microservices have seen a surge in usage. One of the biggest companies that showed its effectiveness is Netflix [21].

#### 5.1.1.1 Definition

The best way to describe a microservice is:

> "A particular way of designing software applications as suites of independently deployable services. [19]"

While there is no concrete definition of a microservice there are some characteristics that every definition contains.

- **Highly maintainable and testable:** enables rapid and frequent development and deployment
- **Loosely coupled with other services:** enables a team to work independently the majority of time on their service(s) without being impacted by changes to other services and without affecting other services
- **Independently deployable:** enables a team to deploy their service without having to coordinate with other teams
- **Capable of being developed by a small team:** essential for high productivity by avoiding the high communication head of large teams [25]

Now that there is a clear understanding of what microservices are and which principles they should follow. Some best practices can be pinpointed.

#### 5.1.1.2 Best practices

The first best practices is to **create a seperate datastore** for each microservice. First of all not every datastore fits every service. It may be that

a message service may achieve more efficiency from a NoSQL database and a user service from an SQL database. A benefit stemming from this is that microservices lets the team think about each datastore used for each service and why that datastore is the correct one for that specific service [21].

When creating a separate datastore for each service you run the risk of data inconsistency. For example, there is a user service which stores the user id. There is also a message service which stores the message and the user id to whom the message is send. If a user gets deleted in the user service, this should reflect in the message service. But with microservices this is not automatically the case because each service has its own datastore. Therefore the foreign keys are not native and thus will not be updated or give a warning.

Another best practice is **writing documentation** [30] for each microservice. Most importantly about how they should be used and which interface it uses. For example, when a new service is created next to our messaging and user service called file service which handles the files send in the messages. This service should know how to communicate with the message service. This makes it easier for the new services to connect to the existing services.

Another challenge with microservice is the **monitoring** [30] of the services. Because it is not known how many services are online it is important to know when they are online and what they log. For example, our messaging service is used frequently and duplicates itself. This then means that the logging of the new service needs to be picked up by your monitoring system in order to view the whole picture of the running application.

### 5.1.2 Miniservices

One of the "new" ones is miniservices. The reason new is between quotes is because most companies that implement microservices actually implement miniservices. The difference between microservices and miniservices is best described in the figure below:

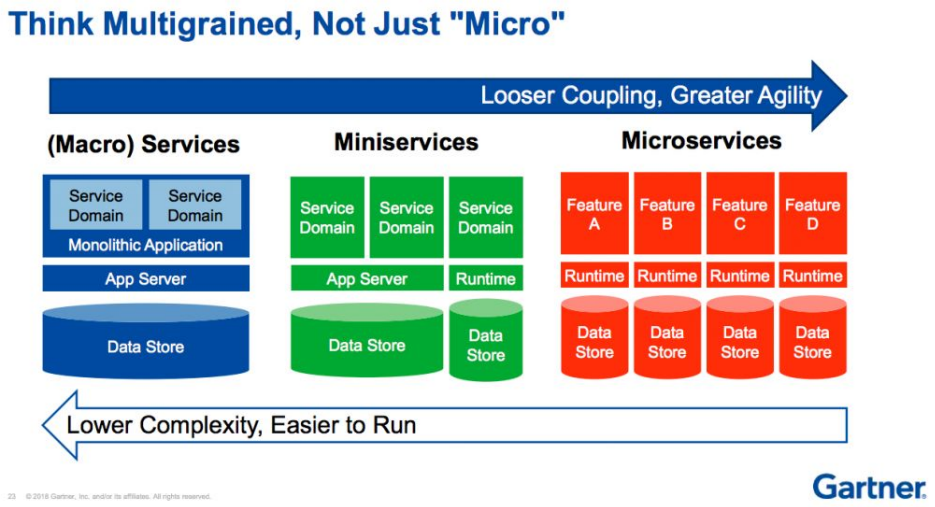


Figure 5.2: Miniservices architecture [27]

Miniservices is essentially an architecture based on breaking specific rules of microservices [1]. As shown in the picture the biggest difference between microservices and miniservices is that microservices are actual features being decoupled and miniservice is about decoupling a domain of features.

This means that each service may contain multiple features but all of the features should be linked to the same domain. Thus the communication inside a service is more fluent and needs less network design than microservices does.

Another divergence is that each microservice should have a separate datastore. This is not the case for miniservices. Every miniservice may be connected with the same datastore [27].

The main advantage miniservices has over microservices is the complexity of the network architecture. With microservices every service is singled out. Which means no service knows about each other so the protocol in which the services speak can be different and may differ from service to service. With miniservices each service connects to the same database. Which makes it easier and faster to do complex querying.

### 5.1.3 Modular monolith

The main idea behind a modular monolith is preserving the idea of encapsulation but deploying it differently [9]. Instead of deploying different services separately with each service having its own datastore, a module can be a library, plugin or namespace. This makes deploying easier to manage whilst still having the modularity gotten from encapsulation.

Just like with miniservices each module will contain the functionalities of a single domain. But unlike the miniservices the modular monolith is compiled to one application instead of multiple.

Modular monolith is perfect for smaller teams because it does not require as much setup as miniservices or microservices.

## 5.2 The comparison

A good talk about modular monoliths [10] shows that most of the time when thinking of architecture there are two extremes. The monoliths and the microservices. As shown in the figure below:

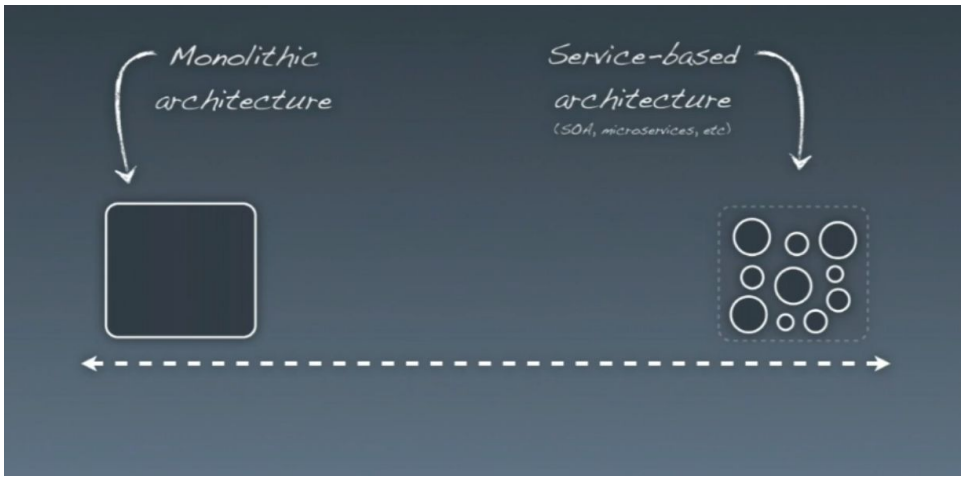


Figure 5.3: Monolith, microservices spectrum [10]

But this is not always the case as shown in 5.1 Architectures.There are cases where microservices are the best choice and there are cases where miniservices or a modular monolith is the best choice. This chapter will further compare the three architectures and decide which architecture fits

best with EFFE's priorities. This will be done with the help of chapter 4.1 Priorities

## 5.3 Complexity

Complexity always plays a role when choosing the right architecture. Looking at the three architectures shown in 5.1 Architectures it is obvious that the complexity changes the smaller the modules. Thus the most complex architecture is microservices and the least complex one is modular monolith. With miniservices right in the middle.

In the figure shown below there is an example of the microservice architecture. This shows that each service may have its own datastore but can also run on a different server. This means that each service needs to know in some way where the other services are located. This is called service discovery. Service instances have dynamically assigned network locations. Moreover, the set of service instances changes dynamically because of autoscaling, failures, and upgrades. Consequently, your client code needs to use a more elaborate service discovery mechanism [26]. This is also the case with miniservices.

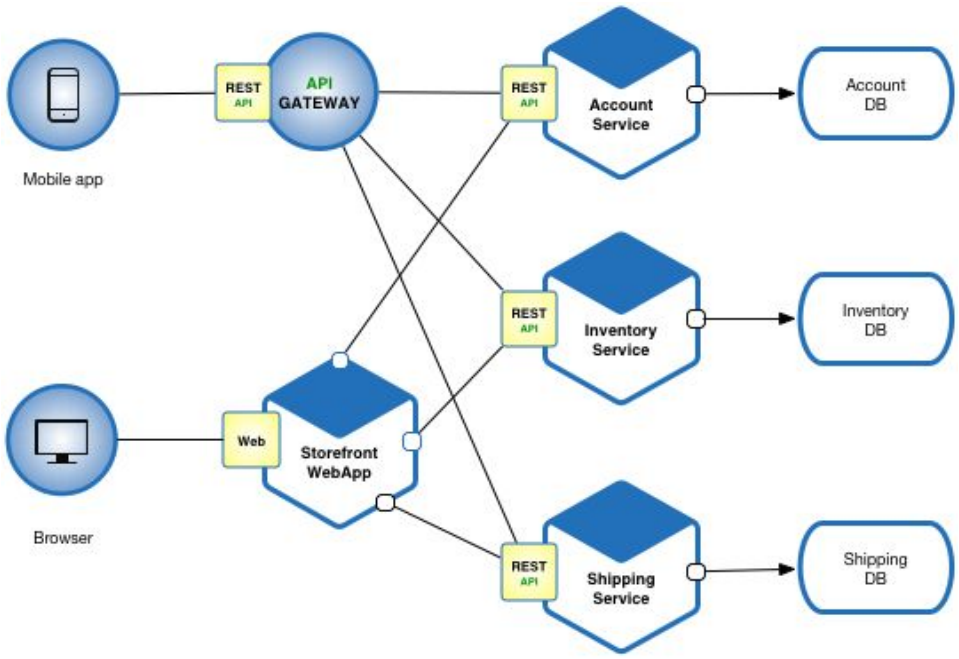


Figure 5.4: Microservice architecture

Even though they can connect to a centralized datastore the services do not have any knowledge of each other. In a monolitic application there are no different services. The modules can talk with each other via functions and imports. This means that there is knowledge of the other modules.

The other thing that makes microservices especially complex is the splitted datastores. Because the database is splitted it can be hard to handle foreign keys or pointers to other data objects. This is because the datastore does not have a direct connection to this pointer. This problem of complexity is not prevalent amongst the miniservices and modular monolith because in these architectures the datastore is shared.

The quality attributes that are applicable to this attribute are:

- **Maintainability:** The more complex an infrastructure and/or architecture is the more maintenance it requires.
- **Security:** Complexity always brings security issues with itself. This is especially the case with miniservices and microservices because of the service discovery.

## 5.4 Technology

One of the most convincing arguments for choosing microservices is the freedom of choosing the technology. There is a possiblity to write the first service with Node.js and a MongoDB database and the next service with Java and an ElasticSearch datastore. This makes it really easy when switching technology or recruiting new developers.

Miniservices does have the benefit of choosing a different programming language per service. But because all of the services talk with the same datastore, the datastore technology is always the same.

Modular monolith is the worst in this section. A modular monolith is stuck with the same technology for the programming language and the datastore.

The quality attributes that are relevant are:

- **Porability:** The portability is very high because each service can be ported separately which makes it easier.
- **Compatability:** The compatibility between technologies is extremely relevant when looking at the architecture
- **Performance:** Because the technology for each service can be different. A language can be chosen to create the optimal performance for that specific service.

## 5.5 Testing

It is known that testing plays a big role in creating reliable software. There are multiple types of testing [28]. Not all of them are useful for EFFE. That is why EFFE has created a list of tests it does on the current application. These are test types that will be looked at:

- Unit tests
- Integration tests
- End-to-end tests
- Load tests

Each section will begin with a rating from 1 to 3 where 1 is the best architecture for this kind of test.

### 5.5.1 Unit tests

1. Microservices
2. Miniservices
3. Modular monolith

In a microservice each function is its own service. So the functions are really easy to test. Because miniservices is domain based it takes a bit more effort to test the whole service but it is easier than the modular monolith. This is because the modular monolith is more tightly coupled than miniservices.

### 5.5.2 Integration tests

1. Modular monolith
2. Miniservices
3. Microservices

Because the modular monolith contains all its services it is easy to test how they work together. This can even be done with unittesting. For the miniservices and microservices it is more difficult. The reasoning behind this is the complexity of the service discovery. To test for example two services, service discovery needs to be setup. Each extra service that is added to the tests will add more complexity. An integration test with microservices may call six different services. But with miniservices it may be less if the functions that are called are in the same domain. This is why microservices is in the last place in this type of testing.

### 5.5.3 End-to-end tests

1. Modular monolith
2. Miniservices
3. Microservices

As seen in the 5.5.2 Integration tests the same type of problem occurs. A function that is called may need multiple microservices or miniservices called.

### 5.5.4 Load tests

No difference

All of the architectures are equal when it comes to load testing. This is because load testing is done on a live site. This does not mean it has to be done on production, although it can be.

### 5.5.5 Conclusion

The quality attributes that testing influences are:

- **Maintainability:** Unit testing and integration testing creates an environment that gives the developer a guideline he or she should follow. The developer knows what is expected and thus can maintain the application with more ease.
- **Compatibility:** Almost all of the test types look at if the code works and will fail if it changes. The backwards compatibility Of a application will be tested constantly.
- **Functional sustainability:** This is where testing started. Writing unit tests to be sure the functionality has not changes.
- **Performance:** With load testing performance will be tested this won't be taken in consideration because there was no difference between the architectures.
- **Usability:** End-to-end testing is specifically made for testing usability. It checks if the UI of an application still behaves the same.

## 5.6 Costs

Because EFFE is a startup costs are very important. EFFE does not have the steady money flow that a more mature company may have.

There are two parts on how costs are calculated for software. The first one contains the price of development and the second one is the price of hosting.

Development time and understanding the code go hand in hand. When a developer does not understand the code the developer can not develop. So how do these architectures hold up considering development time and understanding the code?

Microservices as mentioned before is a very complex architecture but when developing it is one of the easiest. Because each function is its own service, creating a service is really easy. There is not a great number of code in one function and therefore easy to understand.

Miniservices and a modular monolith are in the same situation the code can be more complex because they need to talk to other services or modules but there is also a great number of code in one service or module. Therefore understanding the code and the developing time become larger.

The difference between the architectures is not very big. If miniservices and modular monolith are structured in such a way that is logical it should not matter.

Modular monolith is by far the least expensive architecture. Because the application can run on one server without the expense of server discovery it can be run on a server that costs $5,- a month [2]. But the tricky parts comes when talking about interchangeability. What happens if a company wants a building block changed slightly only for them. If they are willing to pay for it it means that there needs to be a whole new server because the application is build on its own. When EFFE has five clients who want this and five clients that run on the standard version. EFFE would have to run it on six servers which would be $30 dollars on the cheapest server which is not expensive at all.

Microservices and miniservices are the opposite with microservices standing out more. These services require server discovery as mentioned before. There are open source server discovery services such as consul but those also need a seperate server. Server discovery is not the most expensive part. The most expensive part is a combination of having multiple services running on different servers that can autoscale. This means that there is less knowledge about our spendings beforehand.

There are some amazing services that handle deployment and autoscaling for microservices. The most known are Kubernetes, Nomad and Docker swarm. These services however cost $40,- per month with the minimum requirements. The cost of the servers and the orchestrator can ramp up quickly.

The quality attribute that is most affected by the cost is the **maintainability**. This is purely because if the infrastructure is this expensive there would be no money to maintain it.

Thus when looking at development time vs infrastructure costs there is a lot to say for the modular monolith. Because even tho the development time is a bit slower for modular monoliths the infrastructure is way cheaper than miniservices or microservices.

## 5.7 Scalability

It would not be fair to compare these architectures without taking a look at scalability. This is where microservices shine. Microservices are made for horizontal scaling.

Vertical scaling is when hardware resources are added to a server. For example adding 4GB of ram to a server. Horizontal scaling is when adding more instances of the service. This can be on multiple servers [3].

As mentioned before microservices is created for horizontal scaling. When a service suddenly gets a lot of traffic the service can autoscale itself. This can be done rather easily because the service itself is so small. This is also why it is harder with miniservices and even harder with modular monolith.

Scalability is important when talking about **performance**. When a server is going above a certain threshold it can duplicate itself and can now split the traffic along the new instance.

## 5.8 Frontend

When looking at the architecture there is one that stands out as easily adapted for frontend and that is modular monolith. This is because it will still be compiled to one application.

Right now EFFE uses Vue to create a single page application. But this does not mean other frontend frameworks are not considered explained in 6.4.2 Frontend.

When looking at microservices in the backend there is a similar phenomenon in the frontend called micro-frontend or micro-apps [4]. But there is one problem that persists with this solution and that is the sharing of UI elements. There is an option to share them between services but that would mean each service would use the same language and need to be deployed all together if one of those UI elements change. Therefore this is not a solution. A talk about micro-apps (microservices frontend) gave a convincing story about why someone should switch to them [18]. But when asked about the UI elements there was no answer that fixed this problem.

Concluding that for frontend there is only one possibility for a modular architecture and that is modular monolith.

## 5.9 Recap

This is a recap of what is discussed in 5 Modular architecture. As mentioned in 5.2 The comparison in this research there will be looked at the quality attributes and how they matched up per architecture.

5.3 Complexity talks about the complexity and how it can influence the whole project. The architecture that ended on top was modular monolith and the quality attributes that were applicable where **maintainability** and **security**.

In 5.4 Technology microservices came out on top. With miniservices following and modular monolith as an obvious last. The quality attributes that are influenced by technology are **compatibility**, **performance** and **portability**

5.5 Testing was by far the most contested section with no clear winner. But when looking at the types of tests that were considered (unit, integration, end-to-end and load testing) modular monolith ended with the best result with miniservices again in the middle and microservices ending last. The quality attributes for testing are **maintainability**, **compatibility**, **funtional sustiainability**, **performance** and **sustainability**

In 5.6 Costs the clear winner is modular monolith with miniservices following and microservices at an obvious last place. The quality attribute affected by the cost is **maintainability**

5.7 Scalability has microservices at first. In second place is miniservices and last is modular monolith. The quality attribute applicable is **performance**.

In 5.8 Frontend there was eventually only one architecture that actually made sense and that was the modular monolith.

## 5.10 Conclusion

The architecture that fits EFFE best is the modular monolith. The chapters where modular monolith was the best option were also the ones that influenced the quality attribute **maintainability** the most. As sorted in 4.1.1 Recap **maintainability** is by far the most important quality attribute for EFFE. EFFE also does not have much money as mentioned in 5.6 Costs Thus costs play a big part in this decision as well. Finally the modular monolith architecture is especially good for small teams and that is a perfect

description of the software team of EFFE since it exists out of one person at the moment.

Microservices have the clear distinction of winning the race on technology and scalability but this is not where the focus of this research lays. Although technology aligns with some of EFFE's focusses it does not compete with the main focus that the modular monolith architecture touches on and the pros do not outweigh the cons.

Miniservices is a mixture of modular monolith and microservices and takes some good parts of the both architectures but also some drawbacks of both architectures. The biggest downside is the complexity of the network as explained in 5.3 Complexity.

# Chapter 6

# Implementation of the architecture

The chosen implementation is modular monolith. As mentioned in 5.1.3 Modular monolith a modular monolith is based on domain driven design where the modules can be developed separately. Most of the principles can be taken from domain driven design. But in a modular monolith the modules do not know what other modules contain.

## 6.1 Characteristics

Domain driven design was coined by Eric Evans in his book Domain driven design [12]. Eric Evans himself said that there is no real definition for domain driven design but there is one for a domain:

> "A domain is a field of study that defines a set of common requirements, terminology, and functionality for any software program constructed to solve a problem in the area of computer programming, known as domain engineering. The word domain is also taken as a synonym of application domain It is also seen as a sphere of knowledge [8]"

What is important to note is that each module is linked to a domain but there is a distinct difference between only implementing domain driven design and modular monolith. A modular monolith is part of the software architecture

of the application while domain driven design part is of the software design. The difference between those two being that software architecture is about converting characteristics or quality attributes into a structured solution whereas software design is more about the responsibility each module or section inside the architecture has. [6]

## 6.2 Current situation

The table below is in order to paint a good picture of the current application, situation and map which domains there are, inside the EFFE application, and the functionalities they possess.

| Domain | Functionalities |
|---|---|
| User | • Reset password<br>• User CRUD* |
| Shift | • Shift overview<br>• Create shift |
| Skill | • Skill CRUD* |
| Store | • Store CRUD* |
| Client | • Client CRUD* |
| Authentication | • Login<br>• Reset password |
| Schedule | • Generate schedule |
| Hour registration | • Hour registration |
| Shift market | • Shift market |
| Shift change | • Switching shifts<br>• Calling in sick |
| Company | • Managing company settings |

* CRUD or Create Read Update Delete refers to the actions that can be called via the rest API

An example of a use case where it is needed to replace one of these modules is if a big client comes to EFFE but requests something small that should be different in the user module. With a modular monolith we can create this new module and place it inside the application and it will work the same as the normal application.

## 6.3 API

### Backend

When creating this API the assumption is done that a modern ORM(Object relational mapping) is used.

> "Object-relational-mapping is the idea of being able to write queries, as well as much more complicated ones, using the object-oriented paradigm of your preferred programming language. [31]"

This assumption is done because almost every modern framework uses this concept to map objects to a relational database which is what EFFE uses.

Therefore the first attribute defined in our api is the **model** itself.

Microservices talk with each other via a protocol. The most used protocols are HTTP, TCP or AMQP [11]. What all of these protocols have in common is that they return a serialized version of the response. Most of the time in JSON.

Commonly in web frameworks there is something used called a dataclass or a serializer. This class can convert an object into JSON or any other content type. Thus if the api of a module in the modular monolith can expose such a **serializer** the application can serialize all the foreign keys the module's model has.

### Frontend

The frontend API is more simple than the backend API. The frontend API only exports CRUD (Create Read Update Delete) functionalities. The class the modules call API is just a service over which the modules call the Rest API of the backend.

## 6.4 Programming language and Web framework

A web framework is:

> "A software tool that provides a way to build and run web applications. As a result, you don't need to write code on your own and waste time looking for possible miscalculations and bugs. [29]"

It is a industry standard to use web frameworks. It simply makes life easier. But which web framework? The comparison will be of the front- and backend frameworks and languages. There will not be a research of the whole framework or language. The focus of the research will lay on the compatibility of the framework with the modular monolith architecture.

The reason the focus lays on the web framework is because the modularity mostly comes from the framework. Most popular programming languages have the same capabilities when used right. The framework decides how the language is used and also how it reacts to modularity.

### 6.4.1 Backend

Even though the programming language is important most of the time their modularity comes from the framework that is implemented. According to hackers.io these are the top backend frameworks in 2019 [16]

1. Express (Node.js)
2. Django (Python)
3. Rails (Ruby)
4. Laravel (PHP)
5. Spring (Java)

All frameworks will be tested against the same use case:

The first domain is employees. An employee has:

- Name
- Birth date
- Email

The second domain is shifts. A shift has four attributes

- Title
- Start date
- End date
- Employees

The application will provide an api which can do a create, list and retrieve(single object) call for both shifts and employees.

Last of all the modules should only talk with each other via an API explained in 6.3 API.

All of these tests are done using Windows 10 on a Dell XPS 13, using the git bash terminal and the usage of MySQL as the primary database.

The assumption is made that the database exists where `root` is the username and password. The web framework used is the name of the database table.

All the code can be found at `https://github.com/jessielaf/modular_monolith`

Rails and Spring were not successful when implementing the modular monolith architecture. Rails did not have the freedom of putting, for example, controllers in different maps and Spring did not have an easy to use option for dynamic dependency injection. Django on the other hand is the only framework that has domain driven design out of the box. This already gives the edge to Django as the framework to use. But what amplifies this choice is the amount of code that is needed in order for the test to work. This was minimal in comparison to the other frameworks. Django was also the only framework with build-in database migration generation. This allows the user to create migrations based on the model. This is very important because it eliminates human error when creating migrations by hand.

### 6.4.2 Frontend

Frontend is a very fast moving lane in software engineering. On october 8th 2010 [17] the first big frontend framework was published called AngularJs. This framework has been maintained by google and received a lot of traction. Three years later at Js ConfUS Jordan Walke of Facebook gave an introduction to React [23]. This changed the frontend world. Mainly because react was not a framework but a library. Which means that it is able to be included it in existing project whereas with Angular there is solely an Angular application. In February 2014 the lastest big javascript framework would be released called Vue [32]. Vue is often seen as the perfect blend between React and Angular. This is partly because Vue can be used as a library and a framework.

These three frameworks / libraries where chosen because they are the most used and the most loved by the javascript community [15].

The scope of the test is that the frontend application should use our backend site created in 6.4.1 Backend. So the application should be able to do:

- Create a employee
- List the employees
- Detail employee view
- List shifts
- Add shifts
- Detail shift view

All the code can be found at `https://github.com/jessielaf/modular_monolith`

From the implementations it is obvious that Angular is harder to implement than Vue or React. This is the result of a combination of typescript and dependency injection which Angular uses. Vue and React on the other hand are really similar. But there are a few differences that makes Vue easier to use than React. The first one is the two way binding of Vue [14]. React does not have this feature. What this means is that the developer has to write his or her own handler for every different input type. Another difference is that with Vue the router is included. Thus the Vue router is supported by the official team. React does not have a router build in. The best frontend framework for the modular monolith is Vue.

## 6.5 Building the modular monolith

The application consists of multiple modules and these modules need to be assembled before the application can be used. Assembling the modules should be easy on build when the application is being deployed but also when a developer is setting the application up for the first time. The first option is to create a command that will clone all the modules. This can be done per project and with a cli (command line interface) tool or with code inside of the application itself. The best option is the cli because this gives a common interface that can be used in backend and frontend projects. An example of such a system is a dependency managers, such as Yarn or Composer. These managers use a JSON file to define which versions of which dependencies are used. The config files for a modular monolith could be more stripped down because the requirements are less when comparing it to a full functional dependency manager.

There are two things all dependency managers require from a dependency, a **name** and a **version**. What is unique to the modular monolith situation is that our name can be the same but where the module is **located** from can be different. Because there is a possibility that a specific module is required with a different version. This is all module level but there should also some project level settings. Such as the **directory** where the modules should be copied to.

```
copy_dir: modules
modules:
  employees:
    repo: git@github.com:jessielaf/employees-module
    version: master
  shifts:
    repo: git@github.com:jessielaf/shifts-module
    version: 1.0
```

In this example the employees module will be retrieved from `git@github.com:jessielaf/employees-module` with the latest version and will be copied to `modules/employees`.

An example how the code could work is below:

```
import yaml
from git import Repo


with open("example_config.yml", "r") as stream:
    config = yaml.safe_load(stream)
    modules = config['modules']
    copy_dir = config['copy_dir']

    for name, module in modules.items():
        repo = Repo.clone_from(module['repo'], f"{copy_dir}/{name}")
        repo.git.checkout(module['version'])
```

Of course this is a very stripped down version of what can be. The best upside to this solution is that it can be the same for frontend and backend. The other option is to add the modules with the use of the framework. This makes it easier to use for new developers because they do not need to install a plugin or dependency that does this. But the code for the frontend and the backend needs to be managed and exists in the repositories. This makes it that the frontend config can be different from the backend one and it creates a harder to understand config. This is why the choice goes to the one config meets all solution.

## 6.6 Deployment lane

As mentioned earlier the architecture that has been chosen for an application can make a big impact on the deployment lane. This is the reason why this section is chosen as an important piece for this research. But the choice of architecture came down to a solution where the deployment lane should not change or only change with building the application as explained in 6.5 Building the modular monolith this can be done with one command. The deployment lane can thus stay virtually the same as before for EFFE.

## 6.7 Implementing it in the application

### 6.7.1 Backend

First of all the module api should be added. This can be done in `effe/api` and looks like this:

```
from typing import Any
from dataclasses import dataclass


@dataclass
class ModuleAPI:
```

```
    model: Any
    serializer: Any
    serializer_per_role: Any = None
```

The first module that will be converted is shifts. This is because this is the biggest module. If this module can be converted all of them can be. The first thing to do is replace a direct link to the model to a link via the module. This means

```
form shift.model import Shift
```

Can be replaced with

```
from shift.api import api as shift
```

The use of the model looked like this:

```
Shift.objects.all()
```

And can be changed to:

```
shift.model.objects.all()
```

For the serializers the same principles apply.

```
from shift.serializers.base import BaseSerializer
```

Can be changed to:

```
from shift.api import api as shift
```

So when referring to the shift serializer:

```
shifts = BaseSerializer()
```

To:

```
shifts = shift.serializer()
```

The first problem that this created is not being able to import a serializer into a model. This happens because python imports all classes even if it does not use one. Apparently this creates a circular dependency. When python has a circular dependency it gives a `ImportError: cannot import name` error. There is an open question about this on stackoverflow that has not been answered yet [13].

The dependencies of the api need to be lazy loaded. There are two options on how to do this. The first one is to overwrite a function in the api. The ModuleApi would look like:

```
from abc import abstractmethod


class ModuleAPI:
    @abstractmethod
```

```
        def serializer(self):
            pass

        @abstractmethod
        def model(self):
            pass
```

The api would look like this:

```
    from effe.api import ModuleAPI


    class Api(ModuleAPI):
        def serializer():
            from shift.serializers.api.base import BaseSerializer

            return BaseSerializer

        def model():
            from shift.models import Shift

            return Shift

    api = Api()
```

The other method is string based imports. Where the ModuleApi would look like this:

```
    import importlib
    from typing import Dict


    class ModuleAPI:
        _model: str
        _model_package: str
        _serializer: str
        _serializer_per_role: Dict[str, str]

        def __init__(
            self,
            model_package: str,
            model: str,
            serializer: str,
            serializer_per_role: Dict[str, str] = {},
        ):
            self._model = model
```

```
            self._model_package = model_package
            self._serializer = serializer
            self._serializer_per_role = serializer_per_role

        def model(self):
        return getattr(importlib.import_module(self._model_package), self._model)

        def serializer(self):
         return importlib.import_module(self._serializer).BaseSerializer
```

And the api:

```
from effe.api import ModuleAPI

api = ModuleAPI("shift.models", "Shift", "shift.serializers.base")
```

The first option is better. This is because the model is imported directly this means that when renaming models or paths some IDE's will pick it up themselves. It is also more explicit and pylinters will pick up if a module cannot be imported.

Switching to the new api was easy until the user shift view needed to be changed. This view shows the the shifts of one user and the shift change requests of a user. This means that inside the shift serializer the shift change request serializer should be applied. But to make sure there are no circular dependencies this can not be done inside the `BaseSerializer` of the shift module. So there needs to be a function on which fields need to be serialized. Thus the api needs to be rewritten.

The django rest framework uses a private field `_declared_fields` that contains the nested serializers. The new ModuleApi looks like this:

```
from abc import abstractmethod
from typing import Dict

from rest_framework.serializers import Serializer


class ModuleAPI:
    @abstractmethod
    def _serializer(self):
        pass

    @abstractmethod
    def model(self):
        pass

    def serializer(self, serializers: Dict[str, Serializer] = None):
        base_serializer = self._serializer()
```

```
            if serializers:
                for name, serializer in serializers.items():
                    base_serializer._declared_fields[name] = serializer
                    base_serializer.Meta.fields.append(name)

            return base_serializer
```

In the shift overview the field shiftchangerequest_set should be serialized by the shift change request serializer. The serializer looked like this:

```
serializer_class = serializers.ShiftOverviewSerializer
```

And with the modular monolith looks like this:

```
serializer_class = Shift.serializer(
    {
        "shift_change_request": ShiftChange.serializer()(
          read_only=True, source="shiftchangerequest_set", many=True
        )
    }
)
```

The class `serializers.ShiftOverviewSerializer` can be removed because it is not used anywhere anymore.

In order to use the cli described in 6.5 Building the modular monolith cleaner, a project was made. It can be found at `https://github.com/jessielaf/modad`.

To implement this, there needs to be a creation of the config in `modad.yml` which looks like this:

```
dest: .
modules:
    - name: shift
      repo: git@github.com:jessielaf/effe-shift
```

Then push the shift module to github at `https://github.com/jessielaf/effe-shift` and remove the shift module from the base application. This command can be ran

```
modad assemble
```

The shift module is pulled from the github repository and placed in the application.

### 6.7.2 Frontend

EFFE uses Nuxt.js as a framework on top of Vue. Nuxt creates the routing via the folder structure. But this folder structure is decoupled from the business logic. There are two options to fix this:

- **Adding two directories**: The assembler as described in 6.5 Building the modular monolith needs to be changed so it will also handle multiple directories where it should copy to.
- **Rewriting it to only Vue**: As shown in 6.4.2 Frontend Vue is capable of having a modular monolith architecture. But to get this there needs to be a rewrite of the whole frontend application. There is a big portion that can be refactored but there are also a great number of Nuxt.js functionalities the application uses.

Even though the second option ties more into the modular monolith architecture, the first option is better for a proof of concept for EFFE.

The config in `modad.yml` that is needed to initialize our modular monolith is:

```
dest:
    - src: pages
      dest: pages
    - src: module
      dest: modules
modules:
    - name: shift
      repo: git@github.com:jessielaf/effe-ui-shift
```

This will clone the most of the code into the `src/modules` and pages into `src/pages`. Currently this is a one way street. The modules will now only be cloned. But by this logic the modules can also be extracted via this config. This makes it easy to use in development. Just load a module, change somethings and pull it back up. Thus the assembler also needs to dissemble the modules. This dissembler looks at the config and based on it, it creates a new folder with the changes made to the directories that are copied by the assembler.

The current shift module is intertwined with the change request, availability and shift market modules. The first action is to remove these from the shift module. Next up is the api for the shift module. Because Nuxt.js is used instead of only Vue the api looks a bit different:

```
export default class Api {
    static create(axios, object, options = {}) {
        console.log(object, "not saved");
        console.error("Implement the create functionality");
    }

    static update(axios, id, object, options = {}) {
        console.log(object, "not saved");
        console.error("Implement the create functionality");
    }
```

```
        static retrieve(axios, id, options = {}) {
            console.log(id, "not retrieved");
            console.error("Implement the retrieve functionality");
        }

        static overview(axios, options = {}) {
            console.error("Implement the overview functionality");
        }
    }
```

The same command as in the backend can be ran here:

```
modad assemble
```

The application works as it did before. To dissemble the shift module this command can be ran:

```
modad dissemble shift ~/shift-module
```

# Chapter 7

# Conclusion

After the introduction the research ranks the quality attributes. This is done in order to find which architecture fits best with EFFE's needs. The architectures that were considered, were modular monolith, miniservices and microservices. These architectures have in common that their main focus is modularity. The rankings of the architectures is:

1. Modular monolith
2. Miniservices
3. Microservices

The modular monolith is ranked first, because focusses on modularity while still being able to be developed by a small team with limited resources. Which is amplified when looking at complexity and costs. This is where the modular monolith architecture excelled. Because in the end the modular monolith is compiled to one application, the complexity is a lot lower than the other options. Lower complexity leads to lower costs. The service discovery together with an orchestrater can already ramp up the costs and modular monolith does not dependent on either one.

The best backend framework for the modular monolith is Django. This is because Django, by design, already implements domain driven design. A modular monolith decouples it modules by domain. So a framework that has such a functionality built in already has an advantage. The other feature of Django, that Rails, Spring, Express and Laravel did not have out of the box, was migration generation. This feature makes it easy for the developer to create migrations without human error that may be involved.

For the frontend Vue was the best framework. Both Vue and React were easy to implement the modular monolith in, but Vue has two way binding out of the box.

In React there was a need to write these two way bindings for every different input. Angular uses dependency injection. In order to make this work smoothly with the api of the modular monolith a great number of code had to be written per module.

To build these modular monoliths a new project was created called modad. This is a modular monolith assembler and dissembler. It uses a yml file as config to create the modular monolith and load its modules on build time. The choice to create a new project instead of implementing it in the codebase was made because modad can be used for backend and frontend. While if it was implemented in the codebase the frontend and backend assembler would look different and that adds complexity.

# Chapter 8

# Sources


[1] URL: https://developers.redhat.com/blog/2018/09/10/the-rise-of-non-microservices-architectures/#more-517597 (visited on 04/05/2019).

[2] URL: https://digitalocean.com (visited on 05/20/2019).

[3] URL: https://www.codit.eu/blog/micro-services-architecture/%E2%80%8B (visited on 04/05/2019).

[4] URL: https://medium.com/@lucamezzalira/adopting-a-micro-frontends-architecture-e283e6a3c4f3%E2%80%8B (visited on 04/15/2019).

[5] ISO 2500. *ISO/IEC 25010*. URL: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010?limit=3%5C&limitstart=0 (visited on 03/06/2019).

[6] M. Aladdin. *Software Architecture - The Difference Between Architecture and Design*. July 27, 2018. URL: https://codeburst.io/software-architecture-the-difference-between-architecture-and-design-7936abdd5830 (visited on 03/21/2019).

[7] D. An. *Find out how you stack up to new industry benchmarks formobile page speed*. Feb. 2018. URL: https://www.thinkwithgoogle.com/marketing-resources/data-measurement/mobile-page-speed-new-industry-benchmarks/ (visited on 03/21/2019).

[8] Dines Bjørner. *Software Engineering 3: Domains, Requirements, and Software Design (Texts in Theoretical Computer Science. An EATCS Series)*. Springer, Apr. 2006. ISBN: 3540211519. URL: https://www.xarg.org/ref/a/3540211519/.

[9] A. Bolboaca. *Modular Monolith Or Microservices?* Feb. 10, 2017. URL: https://mozaicworks.com/blog/modular-monolith-microservices/ (visited on 03/28/2019).

[10] S. Brown. *Modular Monolith.* URL: https://learning.oreilly.com/videos/oreilly-software-architecture/9781491958490/9781491958490-video284863 (visited on 04/03/2019).

[11] *Communication in a microservice architecture.* May 14, 2019. URL: https://github.com/dotnet/docs/blob/microservices-architecture-e-book-publishing/general-changes/docs/standard/microservices-architecture/architect-microservice-container-applications/communication-in-microservice-architecture.md (visited on 05/23/2019).

[12] E. Evans. *Domain driven design.* Pearson Education (Us), Apr. 2003.

[13] Jessie Liauw A Fong. *Circular dependency created by serializer.* July 29, 2019. URL: https://stackoverflow.com/questions/57253648/circular-dependency-created-by-serializer (visited on 07/29/2019).

[14] *Form Input Bindings.* URL: https://vuejs.org/v2/guide/forms.html (visited on 06/24/2019).

[15] *Front-end Frameworks - Overview.* 2018. URL: https://2018.stateofjs.com/front-end-frameworks/overview/ (visited on 06/08/2019).

[16] A. Goel. *Top 10 Web Development Frameworks in 2019.* Apr. 11, 2019. URL: https://hackr.io/blog/top-10-web-development-frameworks-in-2019 (visited on 05/29/2019).

[17] Brad Green and Shyam Seshadri. *AngularJS.* O'Reilly Media, May 2013. ISBN: 1449344852.

[18] S. Hoogendoorn. "Welcome to the new world of micro-apps. How to get the most of front-end microservices". At the codemotion 2019 conference in Amsterdam. Apr. 2, 2019.

[19] M. Fowler J. Lewis. *Microservices a definition of this new architectural term.* Mar. 25, 2014. URL: https://martinfowler.com/articles/microservices.html (visited on 03/28/2019).

[20] K. Knoernschild. *JAVA APPLICATION ARCHITECTURE.* URL: http://www.kirkk.com/modularity/2009/12/chapter-2-module-defined/ (visited on 03/28/2019).

[21] T. Mauro. *Adopting Microservices at Netflix: Lessons for Architectural Design.* Feb. 19, 2015. URL: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/ (visited on 03/28/2019).

[22] R. Nady. *When Beauty and Efficiency Meet: Modular Architecture.* URL: https://www.arch2o.com/language-modular-architecture/ (visited on 03/28/2019).

[23] Andrea Papp. *The History of React.js on a Timeline*. Apr. 4, 2018. URL: https://blog.risingstack.com/the-history-of-react-js-on-a-timeline/ (visited on 06/08/2019).

[24] P. Rengaiah. *On Modular Architectures*. Feb. 24, 2014. URL: https://medium.com/on-software-architecture/on-modular-architectures-53ec61f88ff4 (visited on 03/28/2019).

[25] C. Richardson. *Microservice Architecture*. URL: https://microservices.io/patterns/microservices.html (visited on 03/28/2019).

[26] C. Richardson. *Service Discovery in a Microservices Architecture*. Oct. 1, 2015. URL: https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/ (visited on 04/03/2019).

[27] J. Riggins. *Miniservices: A Realistic Alternative to Microservices*. July 11, 2018. URL: https://thenewstack.io/miniservices-a-realistic-alternative-to-microservices/ (visited on 03/28/2019).

[28] *Types of Software Testing: Different Testing Types with Details*. Dec. 31, 2018. URL: https://www.softwaretestinghelp.com/types-of-software-testing/ (visited on 04/03/2019).

[29] *Web Frameworks: How To Get Started*. URL: https://djangostars.com/blog/what-is-a-web-framework/ (visited on 05/29/2019).

[30] M. Weiss. *Microservices Best Practice*. May 5, 2017. URL: https://blog.codeship.com/microservices-best-practices/ (visited on 03/28/2019).

[31] *What is an ORM and Why You Should Use it*. Dec. 24, 2018. URL: https://blog.bitsrc.io/what-is-an-orm-and-why-you-should-use-it-b2b6f75f5e2a (visited on 05/29/2019).

[32] Evan You. *The first week of launching Vue.js*. Feb. 11, 2014. URL: https://blog.evanyou.me/2014/02/11/first-week-of-launching-an-oss-project/ (visited on 06/08/2019).

# Chapter 9

# Appendix

## 9.1 Interviews

### 9.1.1 Questions

In order for the interview to go smoothly it is important to define the questions beforehand. The questions will be based on the research framework and the evaluation criteria. The interviews will be conducted by Jessie Liauw A Fong and the interviewee will be a software architect.

Questions about how they view software architecture:

- What is software architecture?
- What is in your company the main job of a software architect?
- With which kind of architectures have you worked?
- What is the biggest pitfall when implementing a new architecture?
- How do you decide which architecture is best of a certain project?
- What is the architecture that you implement in most of your projects? (Frontend and backend)

Questions about how they view modularity:

- What is the first thing that you think of when I say modular architecture?
- What are the most upcoming architectures that are focused on modularity in your opinion?
- Which programming languages do you think compliments a modular architecture best?

Questions about the chosen architecture and method:

- What do you think of the image about how I went my way in choosing the right architecture?
- What is your opinion about domain driven design
- Have you ever heard of modular monolith

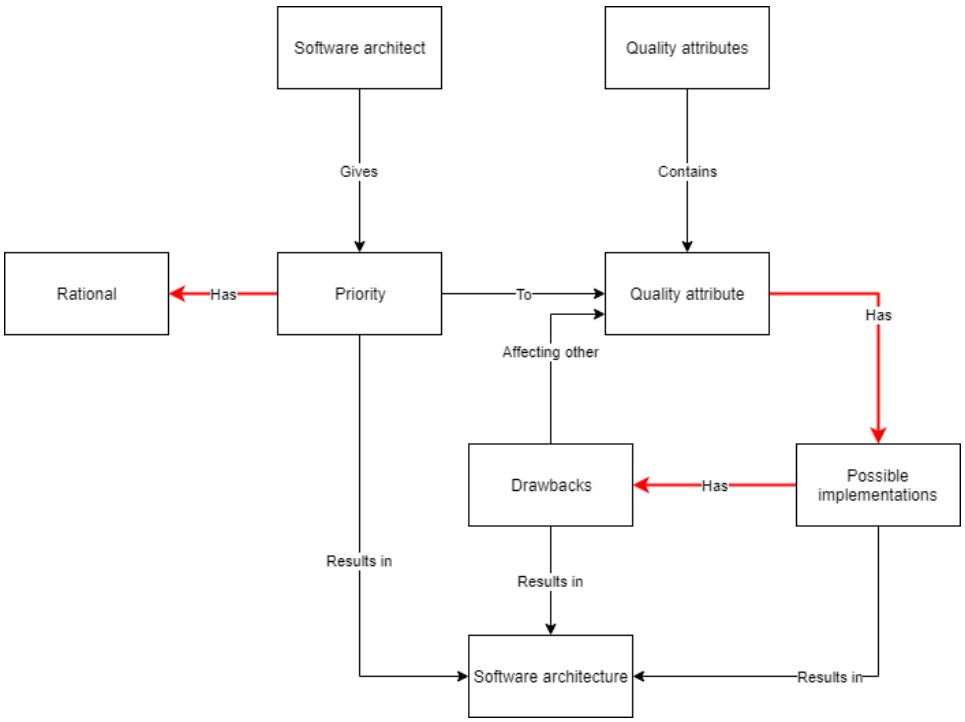


Figure 9.1: How a software architecture is chosen

### 9.1.2 Interview with Joris

Jessie: Na mijn eerste vraag aan jou is eigenlijk. Wat is in jouw ogen software architectuur?

Joris: Nou dan hadden weer meteen ook. Heb je een half uurtje?

Jessie: Als je het zo beknopt mogelijk zou moeten uitleggen aan iemand.

Joris: Dan is het sofware architectuur een model van hoe een stuk software geimplementeerd is. Het is een soort abstractie. Als je kijkt naar de architectuur van de auto bijvoorbeeld, de motor bestuur en de deur. Op dat niveau zo kun je ook de software kijken. Software verdeel je ook op een bepaalde manier in min of meer onafhankelijke delen en die delen hebben vervolgens de relatie met elkaar. Dus aan het stuur draai gaan de wielen zo. Als ik software dit hier gebeurt, gebeurt er ergens anders iets anders. En architectuur een van de definitie van architectuur is een soort gedeeld model, gedeeld met engineers en de eigenaar van de software. Over hoe die software nou opgedeeld in brokken en hoe die brokken gezamenlijk werken om de functionaliteit geimplementeerd te krijgen.

Jessie: Ja.

Joris: Korter dan dat krijg ik het niet.

Jessie: Nee dat is een goeie beschrijving. En binnen Ximedes, hebben jullie daar iemand die echt alleen op de software architectuur zit?

Joris: Nee, bij ons is er, vrijwel elk team rol van software architect. Typisch, tegelijkertijd ook de meest ervaren developer. Dus bij ons is de architect ook, die bouwt mee. Meestal wordt er op een bepaald niveau wel iets gedocumenteerd over de architectuur. Met name voor mensen die niet in het team zitten dus opdrachtgevers of mensen die nieuw komen. Ik denk dat in de meeste gevallen de architectuur nog veel meer een gedeeld begrip is binnen zo'n team of dat nou echt een los product is. We hebben ook niemand die alleen maar architect is en niet bouwt.

Jessie: Hoe documentaireserie die architectuur?

Joris: Ja dat wisselt. Ook dat wij werken voor verschillende opdrachtgevers die ook zelf verschillende mensen en eisen hebben. De meeste projecten hebben een document, word document bij wijze van spreken of markdown. Vrije teksten en daar wordt in opgeschreven wat mensen. Het is een beetje een soort cirkel definitie maar daar schrijven op wat het team nodig vindt om op te schrijven.

Jessie: Ja.

Joris: Dat dat wisselt heel erg voor detailniveau. Wat ik altijd deed, ik heb zelf ook gearchitect. Je beschrijft het systeem als geheel met de grove delen is sommige. Sommige stukken ga je even de diepte in. Omdat er bijvoorbeeld iets interessants gebeurt if iets niet voor de hand liggend. Want 9 van de 10 keer zeker bij ons geval wij schrijven code van, typisch project van 40000 regels of zo dus dat moet je. Als

je naar kijkt moet je toch wel. Het idee is dat we zulke code schrijven dat als je er naar kijkt je het toch wel snapt. Dus dan hoef je niet helemaal te gaan zitten documenteren.

Jessie: Ja.

Joris: Maar goed daar naast zijn er best een aantal klanten vroeger meer dan nu. Die het heel formeel gedocumenteerd willen hebben. Dingen als uml unified modeling language. We deden het al niet veel en we doen het nu nog minder, omdat klanten het niet meer nodig vinden.

Jessie: En als je dan kijkt naar de verschillen architecturen die binnen Ximedes worden gebruikt. Eelke zijn dat en waarom is daar voor gekozen?

Joris: Ja. Ik denk met afstand de meest voorkomende architectuur is een monoliet. Java of tegenwoordig kotlin, maar een web applicaite op JVM. Die communiceert met relationele databases. En dat of een server sided gerenderde html ui heeft of rich client zoals react, angular of vue die communiceert via de rest api. Ik denk dat dat 80 procent van de projecten zijn. En dat kiezen we omdat we, we geloven er heel erg dat je dingen zo eenvoudig mogelijk moet maken. En dat is een kunst, daar zijn geen harde regels voor maar door de bank genomen is een monoliet voor developers, voor beheerders, voor eigenaars. Eenvoudiger dan een microservices architectuur bijvoorbeeld. En dus als je kijkt naar microservices als alternatief van een monoliet. Dat is nogal wat investeringen die inkomt. Het wordt complex op te bouwen, complex om te deployen. Het lost een hele specifieke set aan problemen op die lang niet elk klant heeft. Voor de de de lood en de performance en de security die wij nodig hebben is een monoliet uitstekend. Dus dat is een beetje de defacto standaard. En wat je vervolgens ziet is dat je per project per klant daarin wordt afgeweken daar waar het nodig is en daar waar het zinnig is. Dus soms worden systemen in 2 delen op in 3 delen of 20 delen opgesplitst.

Jessie: Je wilt je kijkt gewoon naar bijvoorbeeld. Als er 1 deel is wat eigenlijk 9 procent van de loadopvang dan wil je eigenlijk alleen deel onderscheiden.

Joris: Ja, bijvoorbeeld.

Jessie: Dat je die makkelijker kan delen en dat de rest van de applicatie in een instance kan draaien.

Joris: Ja bijvoorbeeld, security is bijvoorbeeld een ander ding. Waarbij je zo min mogelijk footprint service er naar buiten wil laten zien. En 9 van de 10 keer zijn de klanten. Voor developers is er niks handigers dan gewoon een proces wat ik kan starten met dat gedoe en zit je ook weer met je git repositories. Kijk alles schaalt. We hadden ook een project draaien. Die is gestart en die had op een gegeven moment 20 git repositories en die jongens werden dood ziek. Die hebben het nu allemaal in een monorepo gestopt. Dus we zijn continu opzoek.

Jessie: Ja precies ja.

Joris: Kijk en als je facebook bent is het een ander verhaal. Dan heb je een heel team zitten die de hele dag niks anders doet dat repositories managen. Ja zijn we met een mannetje of 4/5 per team zo gaat het.

Jessie: Ja dat snap ik. Oké en als ik het goed heb begrepen hebben jullie wel microservices of hebben jullie wel andere architectuur geïmplementeerd binnen het bedrijf? Wat was de grootste challenge die pitfall die je zag tijdens het implementeren waar je die eigenlijk pas daarna dacht ik van oh ja oké als we dat eerder hadden gedaan. Was het een stuk makkelijker geweest. Of is het vooral tijd dat gewoon?

Joris: Kijk als we even teruggaan naar architectuur als mentaal model van hoe de software werkt. Los even van hoe het deployed wordt, meer aan de binnenkant. Ik kan niet zeggen dat als bedrijf er nou wel of niet iets geleerd is, maar ik heb wel software gebouwd waarvan ik achteraf de dacht. Een goed voorbeeld ik heb ooit software gebouwd voor een lease bedrijf, ING Lease. En dat automatiseerde het verkopen traject. Zo'n lease contract bestaat uit onderdelen, het is een beetje als een hypotheek, je hebt garantor en zekerheden noem het maar op. Complexe gedoe pricing dingen en dat aan de eind van de rit moesten er documenten komen. In het nadenken in het ontwerpen van het systeem. Hebben we heel lang gepraat over dat sales process. En die documentatie was een soort afterthought, oh ja aan de eind van de rit moeten het documenten worden. Ik denk dat letterlijk 3/4 jaar later toen het systeem al heel succesvol was toen merkte we opeens. Toen bleek dus dat er heel veel changes kwamen op die documenten en dat veel mensen ontevreden waren en achteraf. Hadden we met veel meer vooruit die document generatie moeten behandelen. En nu is het een systeem wat contract workflow automatiseerd en achteraf een document maakt. We hadden het als een document generator moeten bouwen die ook wat contractmanagemend deed. Dat is zo'n architectuur dingen waar je later bijna niet meer vanaf komt. Tenzij je het compleet herbouwd. Dat gezegd hebbende. We proberen hier tijdens een project gaat tijdens de bouw heel erg op te letten. Bijna agressief te refactoren als we merken dat het niet meer klopt. Uiteindelijk begin je op een gegeven moment maar te bouwen en dan ga je een kant op, dan heb je een model. En tijdens de bouw merk je. Daarom is het zo fijn dat je snel naar klanten teruggaat, omdat die klant komt met dit bedoelde ik niet. Soms blijkt gewoon dat je een verkeerd zit je domeinmodel. Ja dan los je dat op.

Jessie: Is wat je eigenlijk zegt is dat. En wat je hebt geleerd dat je gewoon eerst gaan kijken van oké wat wat is nou echt het meest grote gedeelte of tenminste de meest belangrijke functionaliteit die de applicatie daadwerkelijk heeft. Waar draait het allemaal om en daar ga je omheen bouwen in plaats van eerst het bouwen, bepalen wat je wilt bouwen en waar je eigenlijk naartoe wilt gaan.

Joris: Hier geloven we sowieso heel erg ook als wij agile werken. Dan nog beginnen altijd met noem het workshops. Gesprekken met de klant, eerste proof conceptcode. Om een beetje te spelen met de oplossingsrichtingen en daar van tevoren over na te denken. En dat gekoppeld met ijdens de bouw vrij agressief refactoren als je merkt

dat het niet goed komt en niet op z'n loop laten gaan dan wordt het een puinhoop. Maar de realiteit is dat je het gewoon ook niet altijd kan voorspellen. We hebben software die soms 10 jaar misschien nog wel langer in productie draait. De wereld veranderd ook in 10 jaar. De software wordt gebruikt voor toepassingen waar ze in 't begin nooit voor bedacht zijn. Dus al heb je het helemaal goed in het begin. De kans dat het over 5 jaar nog matched met waar dan voor gebruikt is, is heel klein. Hoe dan ook is het geen statisch ding. Continu de oplossing vinden bij de realiteit van het bouwen.

Jessie: Dan hebben we het stukje echt over de software architectuur gehad. Nu ga ik wat vragen stellen over modulariteit binne de architectuur. En eigenlijk het eerste wat ik je wat vragen is: waar denk jij aan als ik als ik het heb over een modulaire architectuur.

Joris: Modulair impliceert natuurlijk min of meer ontkoppelde onderdelen, modules. Dus als je zegt modaire architectuur dan verwacht ik dat je het hebt over hoe die bestaat uit onderdelen waarbij het voor sommige onderdelen misschien ook wel meer dan 1 implementatie is. Die dus optioneel zijn. Die je wel of niet hebt. Dat je misschien wel twee of drie versies van dezelfde module hebt. Die je min of meer als een soort legoblokjes inelkaar klikt. Nu vul ik heel veel in op basis van het woord modulair.

Jessie: Dus dat is goed. En als je kijkt naar de modulaire architectuurs die bestaan in de industrie. Welke denk jij zegmaar dat de meeste potentie heeft en daarbuiten ook goed implementeerbaar is als dat een beetje duidelijk vraag is.

Joris: De vraag is wel duidelijk. Ik weet alleen niet of ik er wat zinnigs over zeggen. Kijk dat had ook te maken met mijn achtergrond. Wij bouwen maatwerk software in opdracht waarbij we elke keer met een wit vel beginnen en zelden hoeven rekening gehouden met het feit dat dezelfde software bij het meer dan een klanten draait bijvoorbeeld. Waarbij dus ook zelden de noodzaak is om echt dingen plugable te maken. Dus in die zin heb ik er niet heel veel ervaring mee. Ik weet wel dat voor ons de paar keer dat we hebben geprobeerd dit te doet het altijd erg tegenviel. Moeilijk is om het goed te doen. En goed nogmaals voor ons de return on investment er bijna nooit was om dat in de praktijk werd module a nooit vervangen door module b. Ik heb wel eens nagedacht over plugins enzo. Het lijkt me verschrikkelijk moeilijk om in een keer goed te doen. We hebben wel een beetje ver van weet want we doen het eigenlijk nooit.

Jessie: Ja en als je kijkt naar de, dit is veel meer low level. Als je kijkt naar de programmeer talen. Welke programmeer taal denk je dat het beste modulariteit complementeerd, modulaire architectuur. Als je denkt aan die plugins.

Joris: Ik zou in ieder geval naar een statisch getypte taal gaan. Dus Java, C#, Kotlin noem het allemaal maar op, maar niet javascript, want je wilt wel echt. Ik 1 van de dingen die je natuurlijk heel strak moet hebben in een oplossing is het interface. Is het contract tussen 2 modules.

Jessie: Ja.

Joris: Een deel van dat contract zijn je types. Welke berichten gaan erover en wat zit erin, wat is optioneel, wat is een string, wat zijn de validaties op die string. Als je kijkt naar code, als je dat contract wilt uitdrukken in code. En dan zou ik helemaal gek worden als ik daar geen getypte taal gebruik. Het alternatief is natuurlijk dat je interfaces gaan over het netwerk gaan. Het is dan altijd REST of SOAP of van die dingen. Dan maakt de taal natuurlijk weer minder uit. Binnen de getypte talen zou ik niet 1 2 3 een soort voorkeur hebben dat de ene taal geschikter voor dit soort architectuur dan andere.

Jessie: Misschien ook framework specifiek?

Joris: Ja ik denk dat je meer richting de frameworks gaan. Zeker aan de javascript kant. Dingen als spring

Jessie: KTor?

Joris: Ja KTor. Dat gezegd kotlin kent ondertussen wel contracten. Contracten gaat nog een stap verder dan types bijvoorbeeld een onderdeel van wat je tegenwoordig in kotlin kan uitdrukken. Als je deze functie aanroept, dan komt er altijd een positief integer uit. En dat gaat verder dan zeggen een integer. Ja want want er is geen type in kotlin die positieve integers typed dus je hebt meer nodig dan alleen maar de types dus daar zou je nog naar kunnen kijken maar dan praat je heel erg over sdk level. En ik denk eerlijk gezegd dat het hangt van je van je project af. Als jij een plugin structuur hebt waarbij je bij runnende code, code injecteert. Ja ,dan heb je het over dit soort dingen. Als je het hebt over modules die via het netwerk communiceren dan maakt het eigenlijk niet uit.

Jessie: Dat is natuurlijk ook een van de krachten van microservices, dat je niet overal dezelfde taal hoeft te gebruiken.

Joris: Ja, exact. Dat je polyglot kan zijn. Dat je het aan de verschillende teams over kan laten.

Jessie: Dat was het stukje over modulariteit. Ik heb zeg maar een architectuur gekozen. En ik heb daarr deze eigenlijk een soort van flow diagram bij gebruikt. De Rode lijnen heb ik onderzocht. Wat ik eigenlijk heb gezegd is een software architect geeft prioriteiten aan een quality attribute. Die quality attributes zijn door ISO 25010 gedefinieerd. Die quality attributes hebben mogelijke implementaties. Al die implementaties hebben mogelijke drawbacks. En uiteindelijk doormiddel van je prioriteit te koppelen met die drawbacks kom je uit op een software architectuur. Mist er iets in dit plaatje of zeg je als ik een software architectuur kies kijk ik er anders naar.

Joris: Ja, dit is wel aardig bedacht. Ik denk zelden in dit soort formele termen, maar kijk waar ik het wel heel erg mee eens ben is dat architectuur een afweging is tussen verschillende belangen die die onder spanning staan. Security performance begrijpelijke code noem maar op.

Jessie: Allemaal dingen die je uit de ISO 25010 komen.

Joris: Ja die hebben het ook vaker gedaan. Kijk een andere definitie van architectuur zou het zou zeker kunnen zij precies dit. Een gewogen afweging van wat we voor een stuk software wat prioriteit geven en waar we dus de pijlen laten vallen. Dus als je zegt ik ga 100% voor performance. De kans dat je leesbare code krijgt is de lager en andersom ook. Dus in die zin vind ik het een mooi plaatje. Alleen het zegt meer over het proces van hoe kom ik op een architectuur dan de architectuur zelf, maar dit klinkt heel redelijk.

Jessie: Dan is mijn volgende vraag, ik neem aan dat je weet wat domain driven design is?

Joris: Ja

Jessie: Wat zijn jouw gedachtes erover? En hoe, als je dat implementeerd. Hoe implementeer je dat binnen Ximedes?

Joris: Ja hoe heet die man ook alweer? Erik Evans? Ik heb ooit een boek gelezen en ook een workshop van hem gehad. Dus ik vind dat ik vind de ideeën heel interessant en ik denk ook dat er. Ik had het een tijd geleden over die workshops die we altijd doen. Hoewel we bij Ximedes nooit dus echt domain driven design volgens het boekje hebben gedaan. Merk je wel dat, een belangrijke uitkomst van zo'n workshop is inderdaad zo'n gedeeld domeinmodel. Gedeeld besef van hoe een subset van de wereld werkt. Neem mijn lease software van daarnet. Dat is een complex ding, met contract, pricing, scenario en dat soort dingen. Een goed deel van die workshops simpelweg bedoeld om developers te laten snappen hoe die businees werkt. En ook een beetje om die business te laten verslappen wat de beperkingen van de implementatie van software. Als dat zo doen kun je dit wel, maar dat lukt niet. Dat vind ik wel echt heel mooi. Wat wat we naar mijn weten nooit gedaan hebben. Is dit zo formeel als Erik Evens het bedoeld doorgetrokken naar de implementatie? Wat ik een interessant idee vindt van hem is dat je zegt van ja, dat mag. Dus zoals ik het me nog herinner, tijd geleden, kom ik tot het domein model en dat komt 1 op 1 terug in de software en is ook 1 op 1 hoe je requirements uitdrukt. Ja ik vind het een prachtig idee, heel formeel. Wij doen het niet. Ik heb het ook nooit gedaan dat. Ook hier zie je met al dat soort dingen zal d'r een.

Jessie: Het is een heel groot verschil tussen tussen theorie en implementatie?

Joris: Nou ja en d'r zitten voor een redelijke mate van compleetheid in. Ik snap ook in het boek natuurlijk een soort extreme maar in de realiteit ga je nooit helemaal naar dat punt want het de return on investment wordt steeds lager, maar ik denk dat we dat als Erik Evans naar onze workshop zou kijken dat hij wel redelijk tevreden was. Als je naar de code kijkt een stuk minder. Een consequentie van domain driven design helemaal doorvoeren naar de implementatie is dus dat je ook al die bounded context dingen en dat dat is wel heel. Onze software is gewoon simpeler. Wij vallen toch heel vaak terug op. Ik heb objecten en ik heb value objecten dus objecten waardes in zitten. Ik heb een service laag waar ik methodes

op kan aanroepen. Dat we dan weer documenteren en ik heb wat lijm tussen die dat allemaal aan elkaar zet. Dat is geen domain driven design maar het werkt wel lekker.

Jessie: Als ik me goed kan herinneren heb ik aan jullie eigenlijk 2 implementaties. Eentje was eigenlijl layered architecture, maar jullie deden ook wel iets, ook domain driven design. Dat je 1 package had die over een domain ging.

Joris: Ja. Ja we hadden met name 1 architect. Die is weg, maar goed dat heeft hier niks mee te maken. Ja ik heb daar heb ik altijd wel mee geworsteld. Ook dat vind ik het heel idee. Alle software die ik bouw is altijd gelayered.

Jessie: Ja ja, precies

Joris: Waar ik altijd naartoe neig. Vervolgens als je het 1, 2 keer abstracte trekt. Heeft het te maken met die keuze. Java software maar ik denk elke software in de wereld valt in eerste orde uiteen in een aantal noemen het packages, noem het, maakt me niet zoveel uit wat je kiest. En wat je altijd ziet. Wat je ook kiest. Daar zullen altijd concerns zijn. Quality attributes in jouw plaatje. Die daar dan vervolgens niet lekker in passen. Als ik een layard architectuur maak. Dan is een gegeven stuk functionaliteit vanuit een use case verdwijnt opeens door allerlei verschillende plekken en allerlei verschillende lagen. Als ik zeg mijn primaire modules zijn mijn use cases. Dan vallen andere dingen weer een beetje in het niet. Bijvoorbeeld de database logica zit dan ineens op allerlei plekken tegelijkertijd. Hoewel ik het. Ik vond het een heel interessante dat een keer wat anders dan dan dan klassieke three tier layered. Maar het zal ook hier weer per project. Ook hier moet je weer een keuze maken. Eigenlijk heb hier ook weer architectuur te pakken. Hoe je dat een gegeven project een gegeven klant gestructureerd. Het hangt ervan af. Ja, het is ook dat je je hebt ook heel erg hebt te maken met de verwachting van een engineer die over 5 jaar op je project zit. Ik kan natuurlijk heel wild gaan en zeggen ik ga het allemaal anders doen vandaag. Maar dan optimaliseer je niet voor die arme ziel die opeens bugs moeten gaan zitten fixen. Dat zit ik ook wel heel erg mee. Het zijn lastige dingen.

Jessie: En? Heb je wel eens gehoord van een modulaire monoliet.

Joris: De term hoor ik vandaag voor eerst.

Jessie: Oké dus. Snel uitgelegd. Het is eigenlijk domain driven design. Dus je hebt eigenlijk verschillende lagen en dat zijn allemaal domains. Die praten met elkaar over dezelfde api. Bijvoorbeeld: User of employee heeft een shift. Dan heeft een shift een many to may relation met een employee. En die many to many relationship wordt gedefinieerd via het interface van employee. Employee heeft een interface en daarom weet shift welk moedle hij moet gebruiken. Elke module dus die exporteert ook altijd maar 1 model. En zo heb je eigenlijk de toevoeging dat elke module kan in een aparte repositories worden gedeveloped, door aparte teams. En die kunnen on build time bij elkaar komen. Snal je een beetje wat ik bedoel?

Joris: Ja, maar jouw voorbeeld als ik een shift module maak. Ik moet wel een concept van employee kennen. Is dat mijn eigen implementatie of moet ik op compile time de employee module kennen?

Jessie: Het kan op compile time. Als je niet als je niet met een compiled taal gebruikt.

Joris: Ja precies, ik moet wel het employee concept kennen?

Jessie: Ja klopt, je kent het employee concept. Hetzelfde als dat eigenlijk microservices werken. Je moet weten dat er een andere microservice is die een bepaalde actie ondersteund, maar het interface of de rest api is hetzelfde als bij andere microservices. In plaats dat je een interface hebt over de daadwerkelijke rest api of graphql of whatever. Heb je het nu over de code. Hoe je praat over de code. Een voorbeeld in spring is dus bijvoorbeeld als je dus verschillende layers hebt dat je dan een nieuwe jpa object implementeerd en dat je daarin meteen kan zeggen api.employees is de foreign key. Dus het model wat gekoppeld wordt. Dat is dus wat ik heb gekozen en ik wou je eigenlijk vragen: Wat is je eerste reactie hierop als je dit hoort? Als het nog niet duidelijk heb ik ook een code voorbeeld.

Joris: Het is half duidelijk, maar los daarvan. Mijn vraag is eigenlijk welk probleem los je op? Het klinkt complex en meer werk dan alles gewoon in een monoliet gooien. Dus waarom doe je dit?

Jessie: We omdat het idee is dus als jij. We hebben 1 basis applicatie en laat zeggen er een hele grote klant naar ons toe, maar die wilt 1 module anders hebben. Dan kunnen we nu zeggen oké we bij switchen die modellen switchen gewoon om. Want dat is nu mogelijk omdat je dus omdat ze sowieso over dezelfde api praten kan je ook gewoon een hele module eruit halen kun je de nieuwe module erin stoppen. De applicatie werkt vanzelf nog steeds het zelfde.

Joris: Maar de beperkingen zitten in je api. Daar zitten je grenzen van je vrijheid.

Jessie: Misschien ook wel weer een. Omdat die is maar 1 model mag, leveren eigenlijk. Het geeft wel weer een soort van. Je moet er wel weer meer nadenken over de architectuur voordat je eigenlijk zo'n module begint. Je moet deze wel geaccepteerd worden. Als er maar 1 model wat wat dit model gebruikt moet het dan wel een hele eigen module worden, ect, ect. Dat zijn eigenlijk de afwegingen ervan.

Joris: Ja precies.

Jessie: Dat is eigenlijk waar ik ben gekomen. Dus als je dit even snel hoort wat zijn je eerste reacties? Hoe denk jij erover?

Joris: Nou ik denk dat je over de goeie dingen druk maakt. Als je zegt, laten we zeggen dat wij meegaan met je uitgangspunt die modulariteit nodig is. Daar heb ik geen mening over. Als het nodig is dan maak je, je over de goeie dingen druk. Ja ik weet te weinig van het domein om een verdere mening of dit wel slimmer is dan iets

anders, maar het klinkt wel heel redelijk. Wat ieder geval klopt naar mijn gevoel is dat je zegt. Je maakt planning software toch? Daar gaat het om. Kijk linksom of rechtsom. Nou je zou nog een case kunnen maken. Beperk je nou niet teveel tot de plannen van de employees. Maar dat zijn precies het soort ideeën natuurlijk waar je over moet nadenken. Kijk software die alles kan kun je niet verkopen want die doet niks. Dus je moet een soort toepassing hebben. En daarbinnen weer zoveel mogelijk verschillende klanten kunnen bedienen zonder dat je allerlei if statements moet doen. Ja, dat vind ik heel goed en ik vind het een interessant idee om dit pluggability mogelijkheid te combineren met een soort domain driven design analyse. Waar lopen mijn entiteiten nou. Dat vind ik eigenlijk wel heel slim ik heb daar nooit of zo. Ik heb die 2 dingen nooit gecombineerd in mijn hoofd.

Jessie: Dat is dus het onderwerp van mijn scriptie.

Joris: Ja dat je een heel goed onderwerp te pakken hebt. Ja leuk. Goed idee.

Jessie: Dat was eigenlijk dat waren eigenlijk al mijn vragen. Dankje wel voor het interview

Joris: Ja graag gedaan. Niks leukers dan lullen over het vak.

Jessie: Dat vind ik dus ook.

### 9.1.3 Interview with Dimitrios

Jessie: Hello Dimi. So right now I'm writing my thesis to finish my bachelor's degree. In software engineering and my subject has a lot to do with architecture or it is. It is architecture. So that is why I came to you and I want to ask some questions about software architecture. So the first one is big. So what do you think software architecture is.

Dimi: There are official definitions and there are so many actually that basically I'm pretty sure no one will give you the same definition. But for me software architecture is when you think of a system how it's organized the relationship between the components you know how the application is deployed. All the principles that guide know the building and the design of the application.

Jessie: So if I'm understanding it correctly it's more like an abstract layer that is just above code. So if the actual code and then you have the architecture to right above that and then make the design above that again.

Dimi: What do you mean design?

Jessie: So you have software design and software architecture. Do you have a difference in those definitions or?

Dimi: The thing is that. Well first I to comment on. You said it's a level above like abstract level. I do not consider it as a level. Well maybe if you look at it like you have to start somewhere. You cannot just start something with that right. Because you really need to have a plan. You cannot just go build something blindly. Then again then the thing is that so we're literally splitting the collection of the principles like that those guidelines. And if you compare that with buildings you know you know the architects already would create this plan that looks and plan with their civil engineers will take and then start building them like the rest. This plan is basically software architecture.

Jessie: So in Endouble. What is the main job of a software architect. Well what what is a day to day job.

Dimi: Well we do not have a dedicated role in Endouble and that's why you have me which is basically the next round of CTO and software architect because I have the background. But usually what what we do is you know you work with this. We delegate to the tech leaders. You know you take that it takes a decision which they do that together with me when we discuss it and then you know tech leads meeting. So what does an architect do usually in Endouble? The architect ensures that he balance the non-functional requirements and the company requirements at the same time. For example you have you know you have clients that have some requirements that their website needs to be performing. It has to be able to handle that amount of requests per second. It has to be robust. You know the architect still in Endouble has to make sure that their applications can satisfy those requirements. But at the same time they satisfy the company requirements which

are meant to make it easier, you know, to understand all this kind of stuff.

Jessie: So they they're balancing so just to go back to the definition of ISO, ISO is given the 25010. I do not know if you're familiar with the quality attribute. Yeah. So what you're saying is that the software architect balances the requirements of the company with the quality attributes.

Dimi: Yes but with a special focus on the non-functional requirements

Jessie: Yes exactly. So what kind of architectures have you worked with?

Dimi: Of course the one known to all of us: the monolith. We have build the layer architecture which is that you know sometimes you know it has MVC. Pretty common in web applications. I have worked with service oriented architecture and I hate it. Microkernel you know?

Jessie: No I do not know microkernel.

Dimi: you know when you have a word with thinking. You really know it. I am gonna tell you later about it. Of course I work with them in micro services which I love. Event Driven Architecture. Yeah. Very very fond of that thing and space based. You also have worked with space based and pipes and filters is something I work with, with my personal projects, which i really love. And serverless but only experimenting. I do not have work experience with it.

Jessie: So space based and microkernel. Those are the two I am not familiar with.

Dimi: So microkernel is basically when you have a let's say common core and on top of that you can enhance it with other kind of modules. If you look at wordpress, it is exactly that. It has a kernel, something that you can see as the core and than you can add more.

Jessie: Ah that is what CSB is build on right?

Dimi: Exactly. But that is the thing about microkernel. And the thing in your questions you had written modular monolith. I wasn't familiar with modular monolith. But what you mean is actually the microkernel. It is a monolith that is modular and that you can extend.

Jessie: I could not find the official term.

Dimi: Microkernel is the official term.

Jessie: To bad I found out so late it is called that.

Dimi: The thing is that all that stuff like I also wasn't familiar with the official term and because we know you learn also by experience and I think the software architecture of o'reilly where I go. I am a veteran there. I know all of them. Since 2015 I go there. This is where I meet all the great architects and where I hear about it all standard, official terms. That is also where I learned space based. You know it because space based is nothing more than you know when you scale horizontally.

Jessie: So mircoservices is also space based?

Dimi: Kind of. Microservices has a different principle.

Jessie: That is quite a lot of architectures. So you of course have a lot of experience implementing these architectures. And because you have implemented so many what was the biggest pitfall that you saw with most with most of them. Yes. Did you underestimate the certain thing multiple times or what. Yeah. What was the biggest pitfall.

Dimi: I will tell you. Not only are you asking me when I implemented those but also when I saw to my team members implement them?

Jessie: Both!

Dimi: Well I think the biggest one is when you pick an architecture that satisfies in requirement that it's not there. This is pretty common from developers that I've seen when you know they want to pick an architecture because it is more performant or faster or whatever but you were never asked to make it more performant. And now when you do that you bring all the complexity of that architecture that solves the performance. That it was never there. You have to deal with all the complexities. That's that is for me the number one pitfall.

Jessie: What most of the times happen when people are implementing microservices.

Dimi: Yes but there is also a way to make microservices simple. So you do not have to go out of your way. But than again that is the thing. Well I think the next question is. How you decide.

Jessie: Exactly. So the let's first jump in. The better question would be then so yeah How would you decide what architectures best for your current application as you said. You have to. You have to take into account these these quality attributes but how do you go about your way in doing that.

Dimi: So what I would do is first of all I get together the requirements and there are two places that I can gather that first places the client. So the client will tell me what exactly they want and I have to go and ask them the important questions about the non functional requirements in terms of security, scalability, redundability, robustness. This kind of stuff and performance of course. Well these are you know what are important for their client. And at the same time I have to take into account the company requirements. And this is what some people actually forget to do. What are the company requirements. The company requirements are maintainability of course easy to understand. Easy to add code or remove but one more thing that people tend to forget is they do not take into account the capabilities of a company. So if your company does not have the experience how can you go on and take an architecture that requires that. Right now even with microservices. Even though they are very simple. And if you think of them alone as services if you think about it the infrastructure that you need to make them

communicate to make them deployed in that kind of stuff. It requires a lot of devops work. And if your company cannot afford it. Either because it doesn't have the experience or because it does not have the capacity right then it's wrong to pick that architecture. This is what you have to balance. You take those two a let's say a sources of non-functional requirements and then you pick the best that balances. Thats how you go.

Jessie: Yes that sounds good. So which architecturally do you prefer implementing and probably. Again it is requirements specific. So I will give an example of what I'm implementing and maybe you can give you a rationale or what you would implement of course you do not have the full picture but yeah. So right now I am building an application, EFFE, it is planning software for employment agencies. You have a main application, so you have a main application which has basic functionalities. But there is a possibility that big client comes in. He says Okay I want these basic functionalities changed. I want that a user has an extra field, when I save user should be called to my to another system et cetera et cetera. There's a lot of requirements. It should be very modular much I mean maybe the is not the best word. Flexible. Flexible it should be very flexible in order to have the basic function of basic application but also the application that is more catered to a certain client that will pay more. And then the question is how would you. Which architecture do you think it's best for backend and frontend.

Dimi: So basically what's you said right now it's a pretty typical application. But you didn't mention any of the non-functional requirements. Which are the ones that dictate. But anyway I can already understand from your case because you are the sole developer. So immediately having unpacked some equipment there you need to be able to have enough time to maintain that so it doesn't think you for life. So you can do other stuff you can enjoy your life.

Jessie: Well actually. So my the first thing I did with my thesis I was ranking quality attributes and what I found most important and I can give you here the quick recap the first one is maintain ability. The second one is compatibility. Third one is functional. Sustainability security reliability performance portability usability because yeah performance is not really important for us/

Dimi: Social performance is subjective like. You would notice that in web applications performance doesn't really matter much. It would not matter and I do not know maybe the voting applications where you really have.

Jessie: Yeah okay. So in your opinion what do you think. Yeah. Talk to just tell me what what you take in and how are you...

Dimi: Will tell you how I would approach it. For more of those architectures. There's always something that I am leaning towards and I can already tell you that of course for me microservices is the way to go. I'm fairly convinced about that and actually love the whole idea. But they will not pick it for the simple project or a small project, right. Because of the overhead that it requires in terms

of your devops and a lot of other stuff that I have to put myself in. I learned how to do it. Yeah. So I can do it and I'm doing it for my project but it's a pain in the ass. And then what I would do is try to make scaled down versions of that for example if you think about the whole microservices that you keep a one set of service independent of each other right. You can even structure that in one you know even in the monolith or you know in your backend as different packages and those packages have their own migrations, their own configuration, their own classes and everything there and you know for connections, drivers and they can potentially be taken apart and be deployed independently in another you know service but it's fine if they live, at the start of the project, there. And they can maintain it easier. I can test it also easier because the important part is that by the time you want to switch to microservices you're testing should already be in place and the testing difficulty increases as well. Because of this decoupling that is there. However when it is on the same application the existing tools that we have are pretty good. So if you take advantage of the principles of microservices but say in one application for starters for me it is always the way to go. But I have also some rules like if it's a website that you need to build it MVC I will pick that because it's a proven way. There is no need to go microservices for a website.

Jessie: Okay. Yeah. That's so so what you recommend is then micro kernel or/

Dimi: I would not go for a micro kernel at the start. Making sure it is a separate library or package or whatever you call it, is independent and decoupled from the rest. And this a micro kernel gives me at all times at times the possibility to take it that way and have it as a depedently deployable microservice. So I get best of both worlds.

Jessie: Yeah and you get you get to decoupling and the encapsulation of microservices.

Dimi: Yeah. So if there is a new requirement that comes in for let's say performance or something then you can scale up. But why you do it from the start. You know pay all of the cost.

Jessie: Exactly. OK. So yeah that was my. There was. These were the questions a bit more about server architecture. Next few questions will be about modularity. Yeah. So yeah. The first thing what comes up to you when I say modular architecture

Dimi: When I hear modular the first thing is decoupled components. With anything that might include to either independently deployable or not. Then the whole idea of building modular that is not its decoupled and you can replace it.

Jessie: I think we already touched upon this a lot but your opinion the most what are the most. What are upcoming architectures that are focused on modularity.

Dimi: I can tell you already microservices is the king. Yeah it's proven now. You would see a lot of buzz last year on the conference. Yeah but now it's pretty

standard. There is no better thing for him than microservices because microservices give you not only the ability to have components you can replace. With anything else at all times it gives you the ability to have your own whatever language you want, whatever datastore. Whatever services. It does not matter. So the king is microservices. But then if you ask me about upcoming architectures then you should already know about the evolutionary architecture right.

Jessie: No I haven't heard of.

Dimi: Yes evolutionary is a term I first heard two years ago at the software architecture conference by neal ford. Very great guy and everything and he's been doing talks on that since then and the whole idea of evolutionary architectures is that you build an architecture that really aligns with the days agile and everything. The main problem sentiment was that when you built an architecture. Systems evolve, systems change. You can never beforehand think you're the best structure for what is gonna come. You do not know. So what do you do is you make enough architecture that can support your and a little bit towards the future needs and then you build on top of that. But how do you sure of that you know your architecture are yeah really adheres or satisfies the requirements you set. This is where the evolutionary theory defines their fitness functions. The fitness functions are basically something like test automated test. That you do against the non-functional requirements so you say for example I want my architecture to be able to support 50 users per minute. Or per second or whatever. Right. Yeah. So you make this a fitness function. It's something that you can run in automated means that can show you that that system works like that. Then you can evolve the architecture and can switch from microkernel into microservices. You can always run that function. I change the architecture but it still adheres to the non-functional requirements. So yes I can evolve it without it losing the most important piece that I know.

Jessie: OK. So it's about it's more about making tests for the non-functional requirements?

Dimi: Yes it's it's actually not only making tests.

Jessie: Well you want to you want to make sure that you're non-functional requirements that you set beforehand will be adhered to even though you change architecture.

Dimi: Yes and this will allow you to change the architecture and experiment with other type of architectures while making sure that you keep those. The problem is that sometimes you cannot test all of the non-functional requirements. Performance is super easy to test.

Jessie: Usability is very hard.

Dimi: Yeah try testing maintainability. Maybe you can think of I dunno measure the time say one like. You have some metrics from quality systems say. Some are good but still it's super hard to test and you do not know if it's very representative.

But it's a good way to go and read more about that because I think this is the future. Because you have styles. But then again like you said you pick the one that fits what is best for a problem but the problem changes become greater or smaller. Other stuff so you have to change the architecture and that's a good way to make sure that you always adhere and can evolve.

Jessie: Okay so this is we we're not talking about architectures. And of course architecture just when implementing there is a need to choose like a framework and most of the time you choose a framework and language. Which language. Well let's start with the language. Which language went language do you think complements modularity most.

Dimi: I saw your question beforehand. I would not recommend any specific language. I would tell you this though I would go with a language that has enough support for modules and modularity. I will give you an example right because with microservices you can pick any language that you want doesn't mean that he can pick brain fuck right. But you can if you would like it. Because its a restful api for your service and it hides all of them you know what the Internet and what happens you can use it. Right. The thing is though that you will notice that some languages have better support for modules and packages than others. Let go look at the. You know for example javascript although it's trying to do well you will see there are problems with the modules like they're not really they're not native. First of the module concept you know. They're doing it now.

Jessie: It's native in node but not in javascript.

Dimi: Sure you will see also problems that when you do imports that you actually have to refer to the actual file location. Compare that for example to PHP where do you and you do not import do you say use and it's a class a namespace in the class. Doesn't matter where it is stored on the disk. You see different type of you know support. Now you can move the class. When now all you have to change is the namespace while in javascript you need to change the actual source code in a file. And you will see problems there. Sometimes you know you can not have modules with the same name. So I would not pick a language, but I would pick when, a specific language I mean. But I would would suggest the language that has the best support.

Jessie: Because you you read I only mentioned non statically typed languages do you think static typing has something to do with how much and how good modularity is?

Dimi: Well my previous experience with statically typed languages. Hes been many years I would take Java and C++. But because I play still a bit in unity with C#. I can tell you but I do not have that much experience in that area. I can already tell you that they have modules because I can just add another a library dynamically library to my existing program and it just works. I do not know the magic's behind it to be honest because they come in spent so much time with that but I know it's

possible. Doesn't matter if statically typed. Now if the mechanism is there and you know that statically typed languages especially C++ I love it that's how you put plug ins in the game.

Jessie: I'm you know well the you look at Scala it's one of the languages it's really built for statically typed things. So this is about There programming language and I'm thinking can go to furthering to frameworks. I know you work a lot with PHP. But then again like what do you think a framework needs to complement the modular modular system or.

Dimi: Well support for packages.

Jessie: Yes but for example if there is a framework already has these these layers of modules that will be natively or native to the framework. As such. Yeah well how would you think about it. Because it you use symphony here right.

Dimi: Yeah and laravel also.

Jessie: So when do you think symphony is a better choice when do you think laravel is a better choice when do you think express a better choice etcetera etcetera.

Dimi: Well this is all subjective nowadays. Why. Because if you asked me two years ago I would tell you go with laravel. Now if you tell me between laravel and the symphony I would tell you that it is pretty much the same. Because the thing is that there are frameworks that come from different communities with their. If they're both opinionated frameworks. Yeah right. And as with all frameworks are actually opinionated apart from Microsoft which actually goes and says Hey I'm not going to allow the community to make their own framework I will set up a framework for a start and actually they are doing well with the dotnet core. They took the best practices. It's fucking awesome. I love C# and the dotnet core it's amazing. The thing is that you know frameworks are opinionated and the things you look at the complexity. Because framework is it's nothing. It's again code that someone else wrote to make your life easier. And the goal of the framework is to make your life easier. Yeah. When you come to the point that you have to spend more time maintaining the framework you know. Instead of being writing the code to sort of the problem that you were called. Than your framework is a problem. So yeah in the past I would always recommend laravel against symphony at all times. Now if you see the symphony flex and all. It's the same thing. They basically took the whole concept of larvel. Even the way you will write the. Yeah they took the concept and it's pretty much the same right now. So if you asked me. Yeah of course it has still a bit of symphony flavor. at some parts. But then again any week now it's. And if you take a look at dotnet core. Basically you see kind laravel. Dotnet core is exactly like it was playing doing some quick start and playing around with a simple API it felt like it was running laravel.

Jessie: So it felt like very familiar?

Dimi: Yes. That's the thing that you know years have passed and people have

understood that hey we need to simply by simplify some part some frameworks do it better. Those frameworks are actually worse in other parts. So they evolve. Thats the thing.

Jessie: So the last questions are about choosing the architecture and the method. So. Yeah we all do. Again we already went over this. So did I do not know if he saw the image I sent you.

Dimi: Yeah but I had some questions.

Jessie: So yeah it was what it basically was was a software architecture. Yeah it gives priority. To quality the attributes and a priority has a rationale behind it and these quality attributes or the quality attribute has potential implementations. These these potential in implementations have drawbacks and these drawbacks are again on the quality attributes. For example when we want a lot of performance or a lot of performance let's say so I'm gonna implement micro services because of the scalability but it will it has a drawback to maintain ability for example and then when the cycle completes you. You'll end up with a software architecture.

Dimi: Well to be honest when I saw that being said it was a bit confusing. I guess how you described and what is in the picture was kind of confusing. But yeah. The thing is that I kind of agree because the job of the architect is there to balance, to balance the trade tradeoffs. Yeah because there is all of the one of the approaches have problems. So as my friend Neal Ford says. Actually they closed the letter really. Yeah actually I know him. I asked him to be my mentor he had I should sent him an email. But I never did. But he recognizes me at the conferences. Anyway and the things that he said that our job is basically to trade one problem for the simpler problem. That is the job of an architect. So you make tradeoffs. Trying to get the smallest problem for you that is easy for you to solve. So of course yes. The thing is I'm not sure about the priority part because they priorities usually given by the company of the client. They decide this has priority for me. And then of course the architect will say what can I do. Given this is my priority. Yeah. Can I satisfy them. If not of course. He has some important. He can propose alternatives so he can influence the client and the company. Because that's what you will do also in the company. You will go to your stakeholders say Okay guys there are a lot of throwbacks in this approach but if we invest a little bit more will we gain the long run so we can hire some freelancers get that knowledge that we didn't have before and it better for us. Even though at the start we do not want to do it.

Jessie: Exactly. So what you're saying is maybe the software architect doesn't really give the priority to the quality he attributes but he gets the priority from the requirements.

Dimi: And than he is the one that balances it.

Jessie: I know in Endouble there was a lot of talk about domain driven design. What are your opinions about that concept?

Dimi: I love it. For me it's a must. Like if you want to be considered a software architect you really have to know domain driven design. That is for me that's super important. Yeah and the thing is that he started with domain driven design many years ago simply because the concepts. Actually you would be surprised when I tell you that. If you go and look at them and enterprise java beans from 2000. I do not exactly remember the name. It is enterprise something. You will see the exact fucking same concepts in there but people would not understand it. And of course Java made us, as it usually does, your live harder implementing some parts. A lot of fixing things.

Jessie: Yes a lot of noise.

Dimi: A lot of noise. But the concepts where there. You will see for example when you study the structures. This is exactly what domain driven design talks about. So for me I love domain driven design because with this you clearly understand where you have to put your boundaries and if the architect understands where the boundaries are. He know how to make a modular application. he knows where to put the lines. And to make sure that those lines are kept strict and make them more robust. Make sure, you know, everything is simpler. Domain driven design is really, for me when I understood the concept, it really drag me to the next level.

Jessie: So it is not even a consideration anymore? You do not make anything that is not domain driven design?

Dimi: The thing is. From my experience I will not go full 100

Jessie: Okay so than the last question where we also touched upon is the modular monolith and what you call the micro kernel.

Dimi: I think what you are revering to is the microkernel because I am not familiar with the modular monolith.

Jessie: I'll explain a bit. It is really bound to domain driven design. For each domain you have a certain package or module, whatever. And this package can talk to other packages but only over an api. A predefined interface. In this way each package shows one model and a serializer for example. This can be bigger of course. Right now the first concept is only the model and serializer. So you can have foreign keys in there. And you can serialize those foreign keys in the rest api. What than happens is that on build you can switch these modules. For example you have a user module and there is a second user module and you can switch these user modules. But because they are talking over the same api there is nothing changing in the rest of the application. I also have an example if you need it.

Dimi: I understand it because this is what I was telling you. This is microkernel. So you have a core. For example in my core is the DHCP layer. I have a framework that does that part and it gives me a container, gives me the service provider structures for example. The router and that is basically it and maybe the logger, right. That is what you need as basis right. Because I can always deploy separate

router and service but you do not have to. Because it is its own package that you register. Your package provider or your bundle provider in symphony whatever it is called. It does not really matter. And than you can always replace with whatever you want.