# Modular monolith

Jessie Liauw A Fong

June 3, 2019

# Contents

# Chapter 1

# Preface

I am Jessie Liauw A Fong, 20 years old. I was born in Amsterdam but moved to Zaandam and still live there. I started programming in 2010 when I was in the first class of middle school. After a year of programming my interest stagnated but in 2015 I chose to begin the study software engineering and I immediately felt that passion again and I haven't lost that pasion since. In the end of 2015 I started my first software engineering job at The EsportsWall. This was a voluntary job because I did not have enough skills to get paid. 3 months later I did to start my internship at Endouble. I worked at Endouble for 1 year. 5 months as an intern and 7 months as an part time employee.

After I finished my internship at Endouble I started by own company JCB Development. Where I build high-end websites.

At the end of my time at Endouble I started a new parttime internship at Ximedes where I learned a lot about infrastructure. This is also the company I met my now co-worker Erik Schouten. He worked at CargoLedger and that is also where I work now. In the beginning of September 2018 I started a new company together with Stijn Claessen and Siebe Goos called EFFE Planning. This is also the company I will do my thesis in.

This thus means I know the ins and outs of the company.

# Chapter 2

# Summary

# Chapter 3

# Introduction

## 3.1 Motive

EFFE as a company uses the SaaS 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. But EFFE also wants to cater to the needs of bigger clients. This is why we created building blocks.

Building blocks are features that can be added/removed from the application. This can be done by the user or by EFFE. Examples of building blocks are white labeling, integration with frontend system and payroll integration. These building blocks are not required when acquiring EFFE but can be added one by one.

## 3.2 Intention

So the question is how are we going to implement these building blocks. We have a few requirements:

- They need to be interchangeable. Meaning the same building block can be changed with another one that does the same job with maybe extra functionality.
- They should be able to do everything programming related. From if else to database calls.

- They have impact on the frontend as well as the backend
- Building block should be completely separate from the application (loosely coupled)

# Chapter 4

# Research design

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

## 4.1 Research objective

### 4.1.1 The problem

Right now EFFE is developing an application for employment agencies in which those employment agencies can schedule their employees automatically. The current application is really basic and there are requests from potential clients to implement certain features. We decided to add something to the business model called building blocks.

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

EFFE is looking how to implement the building blocks in such a manner where scalability and maintainability are the focus.

### 4.1.2 Objective

The objective is to create a recommendation for an infrastructure on how to create and maintain that infrastructure. Where the focus lays on interchangeability and scalability of the different functionalities.

| Stakeholder | Interest to the objective |
|---|---|
| EFFE | The obvious stakeholder is EFFE. EFFE will enhance its business model. But not only that. We will also create a better infrastructure which means that we can implement functionalities faster and cater more to the clients' need. |
| Client | The client is also the one interested in this process. They are probably not interested into what happens behind the scenes but they are interested in the possibilities it adds for them to EFFE's application |

## 4.2 Research framework

### 4.2.1 Objects

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

#### 4.2.1.1 Backend architecture

Arguably the most important object is the backend architecture. There is already a lot of research available regarding backend architecture. The backend is also the place where the business logic will be expressed. The backend connects to the database and thus needs a lot of attention when creating this section of the application.

#### 4.2.1.2 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.

#### 4.2.1.3 Deployment lane

The backend and the frontend are the software side of the equation but the hardware is also important. Where does the software run? How does it run? The deployment lane is the section that pieces it all together. This object creates the hardware or virtual hardware. Sets this hardware up so it can then proceed to deploy the frontend and backend on the just created hardware. This process is very important and EFFE is not the first company wanting to adopt this. Which means there is already a lot of research in this area.

#### 4.2.1.4 Project manager

The last object we want to research is the project manager. Because the research is focussed on implementing business logic in a modular way it is important to research what can go wrong when the business logic is translated to code.

### 4.2.2 Research perspective

The research perspective is straight forward. Because I am one of the founders of EFFE and I am also doing this research in name of EFFE it has my best interest to approach this research from the side of EFFE. This means that I will put more emphasis on sustainability than for example on the performance. Because for now performance can be dealt with later but if you want something to be sustainable you have to think about it from the ground up. Otherwise you will need to rewrite the whole architecture.

### 4.2.3 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:** In the end everything I need to know all 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 programming:** Theory is one side of the coin. Everything can work perfectly in theory but when implementing the theoretic side you will find problems you haven't thought of before.
- **Critique from outside:** It is known that software architecture is an opinion based subject. This is because especially this area of software is fairly young. Software architecture did not have a lot of time to develop itself as far as some other aspects of software engineering such as operating systems. Because software architecture is young there are a lot of people voicing their opinions and it is important to look at the criticism on some of the architectures.
- **Researches on deployment of architecture:** I will be researching more than one architecture. Each architecture has its own development environment and deployment environment. The architecture of the servers on which the program runs is important but that will be heavily influenced

by the architecture of the software. Nevertheless should it be researched separately from the software architecture

### 4.2.4 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 business logic:** As mentioned in 4.2.1.4 Project manager there will also be considered how business logic is translated to code. Because the building blocks will eventually be different based on business logic. The research will include what can go wrong in which way.
- **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 the most 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. Which architecture has which solution for this?
- **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 an the architecture. In the research will describe which are most important to EFFE. It is then important to reflect the quality attributes we chose on the architectures.

### 4.2.5 Research framework

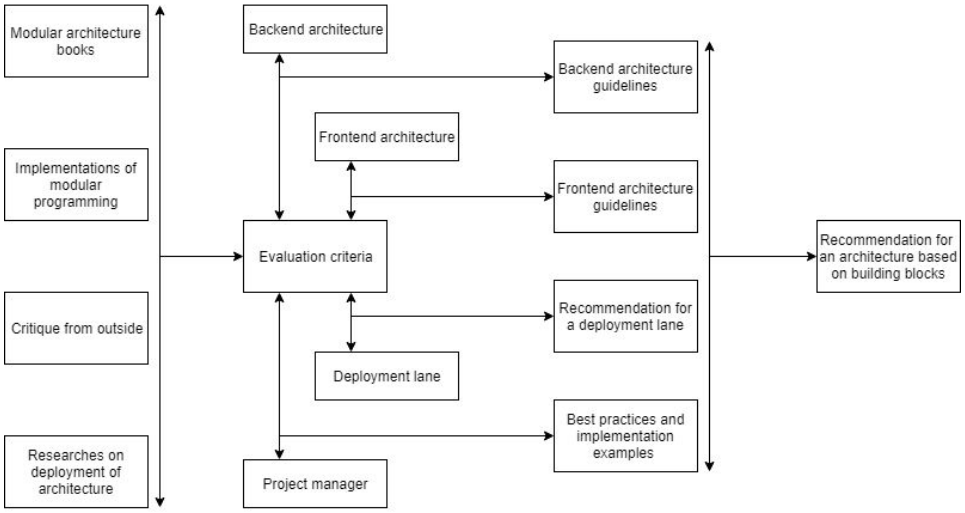


Figure 4.1: Research framework

### 4.2.6 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 on. These guidelines will indicate why I choose for a certain approach and what the specific approach is.
- **Frontend architecture guidelines:** These are the guidelines for the frontend. Such as the backend guidelines these guidelines will also contain the reasoning for a certain guideline.
- **Recommendation for a deployment lane:** As mentioned in 4.2.1.3 Deployment lane the deployment lane can impact the backend architecture and vise versa. This recommendation will be implemented and should thus work perfectly with the backend and frontend.
- **Best practices and implementation examples:** What the project manager experiences and what can go wrong is important to then again pass to the evaluation criteria.

## 4.3 Research Questions

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

### 4.3.1 Main question

From this we can derive that the main question is:

> "What is the best way to transform a monolith into a modular architecture, where the services are interchangeable from each other"

### 4.3.2 Question 1

The first question that will be asked is what is the purpose of this question. The first question is about software architecture. How does a software architect create a software architecture. The model can be found in the appendix under Creating a software architecture.

The thicker red lines show the parts of the model I want to explore in the question. Thus the question is:

> "What was the thought process behind choosing certain implementations for the quality attributes of a software architecture?"

This question will explore how a software architect chooses the architecture. This will give more insights into what they consider when choosing an implementations 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?

### 4.3.3 Question 2

"Which software architectures that focus on modularity are available?"

This question focuses on the architecture that are available. The knowledge of how a software architect chooses the architecture is answered in the previous question 4.3.2 Question 1. In this question there will be a search on the architectures that are available and how they came to be.. Because of the new perspective gotten from the previous question there can be a more nuanced look at the architecture.

Here are some of the sub questions:

- Which are the upsides and downsides of each architecture?
- On what level is the documentation and research surrounding the architecture?
- Which architecture implements the quality attributes I deemed important best?

### 4.3.4 Question 3

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

The solutions or architectures provided from question 2 will have implementations or frameworks. It is important to see which implementation implements a certain choice of the architecture in what way. 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?
- How is the community of this implementation?

### 4.3.5 Question 4

> "What are the key elements of in which a software architecture 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?
- How does the infrastructure implement our quality attributes

# Chapter 5

# Methods

# Chapter 6

# Creating an architecture

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

## 6.1 What is software architecture

First of all let's define what a software architecture is:

> "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]"

## 6.2 Priorities

As mentioned in the definition of software architecture 6.1 What is software architecture a software architecture looks at the characteristics as flexibility, scalability, ect. These characteristics and their sub characteristics are defined by ISO 25010 [5].

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

What is EFFE looking for in an architecture? As mentioned in 3.2 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.

The second focus is **compatability**. Compatibility is the degree to 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]. The system can be very modular but if the different functionalities cannot talk with each other you get nothing from the modularity.

As mentioned in first and second point the functionality may be shared between building blocks or not but each building block should be able to be functionality loose from the other building blocks. This is why **functional sustainability** will be the third focus.

With such a loosely coupled system one issue remains. The **security**. Because every functionality is loosely coupled it means that the functionalities will talk with each other over an open network or a closed network. If they talk on an open network the security needs to be checked constantly and on an closed network measures need to be taken to keep the network closed. That is the reason on why security is our fourth focus.

After running through these four 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 fifth focus.

When something is reliable it does not mean it is workable. Because if the site is not responding as fast as possible the user will get irritated and maybe leave the site. A study in 2018 of Google showed that the bounce rate between a 3 second load time and a 5 second load time is 58% [7]. Thus in order for our users to be actually able to use the application in a responsive manner **performance efficiency** becomes our sixth focus.

There are only two quality attributes left. Portability and usability. Normally there is a good argument 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 **portability** will be our seventh focus and **usability** our eighth.

### 6.2.1 Recap

1. maintainability
2. Compatibility
3. Functional sustainability

4. Security
5. Reliability
6. Performance
7. Portability
8. Usability

## 6.3 What goes into choosing or creating an architecture

### 6.3.1 Creating an architecture

So compared to choosing an architecture, creating one is something entirely different. An architecture does not exactly have a creator. This is because an architecture is just blueprint on how to create the software design. This is why the choice was made to interview current software architects and ask them the questions on how they made those choices.

# Chapter 7

# Modular architecture

Let's start with the definition of 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. [18]"

When looking at the famous architectures in software we have a few examples of non modular architectures.

An example of such an architecture is the layered architecture. In the image below is shown how the architecture operates.

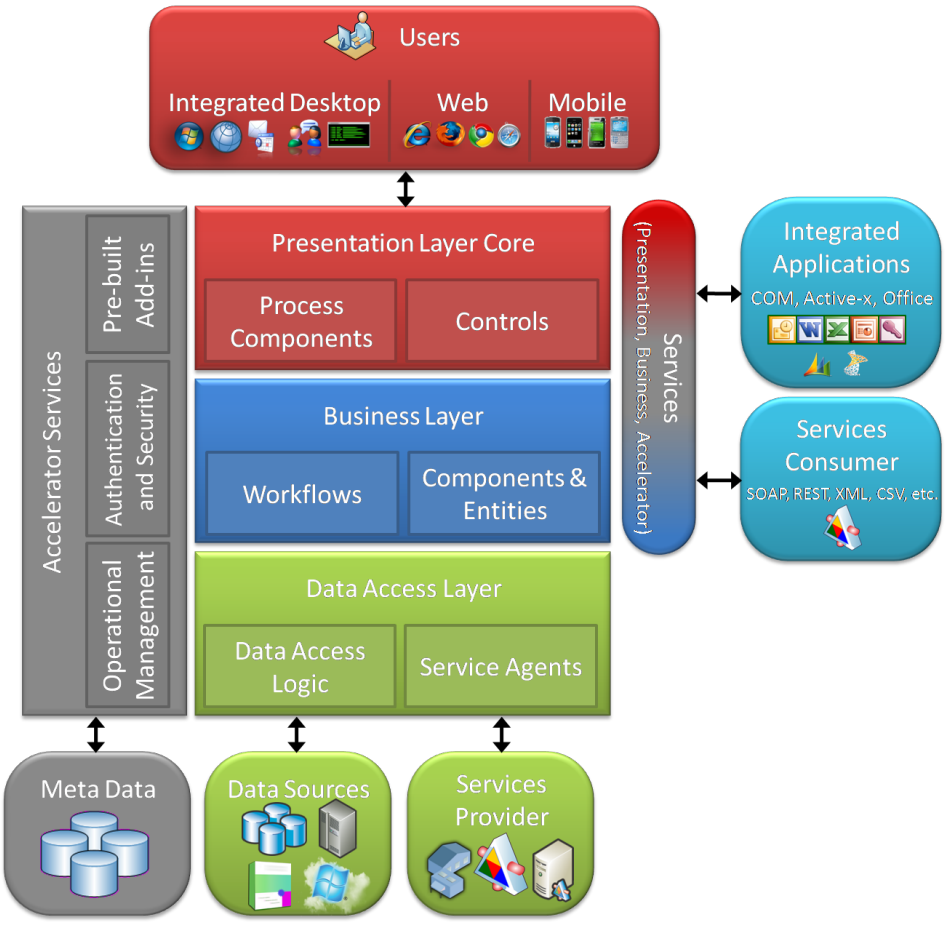


Figure 7.1: Layered architecture [19]

As you can see in this image the architectures is layered based on responsibilities. This will conclude in each layer having its own purpose. As shown in the image 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. Thus the objects are highly coupled.

So why is this architecture so different from a modular architecture? Well as the name suggests a modular architecture is based upon modules.

The definition of a module is:

"deployable, manageable, natively reusable, composable,

stateless unit of software thatprovides a concise interface to consumers" [16]"

This is eerily similar as the description of what we call building blocks in 4.1.1 The problem

## 7.1 Architectures

### 7.1.1 Microservices

If most software engineers in 2019 think of a modular software architecture the first architecture that comes up is microservices. In the last years microservices has seen a surge in usage. One of the most biggest companies that showed the effectiveness of microservices is Netflix.

#### 7.1.1.1 Definition

The best way to describe a microservice is:

"A particular way of designing software applications as suites of independently deployable services. [15]"

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 [20]

Now that we have a clear understanding of what microservices are and which principles they should follow we can pinpoint some best practices.

#### 7.1.1.2 Best practices

The first best practices is to **create a seperate datastore** for each microservice. First of all not each datastore fits each service. It may be that a message service may get more efficiency from a NoSQL database and a user service from a SQL database. A benefit stemming from this is that microservices makes you think about each datastore used for each service and why that datastore is the correct one for that specific service [17].

When creating a separate datastore for each service you run the risk of data inconsistency. For example, you have a user service which stores the user id. Also you have a message service which stores the message and the user id to whom the message is send. If the user id changes in the user service this should reflected in the message service. But with microservices this is not automatically the case because each service has its own datastore and therefore its no foreign keys that will be updated or give a warning.

Another best practices is **writing documentation** [25] for each microservice. Most importantly about how they should be used and which interface it uses. For example, we create a new service 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 and to make this easy for the new developers to connect to the existing services.

Another challenge with microservice is the **monitoring** [25] 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 a lot and duplicate 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.

### 7.1.2 Miniservices

Besides microservices which other architectures are there? 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 image below:

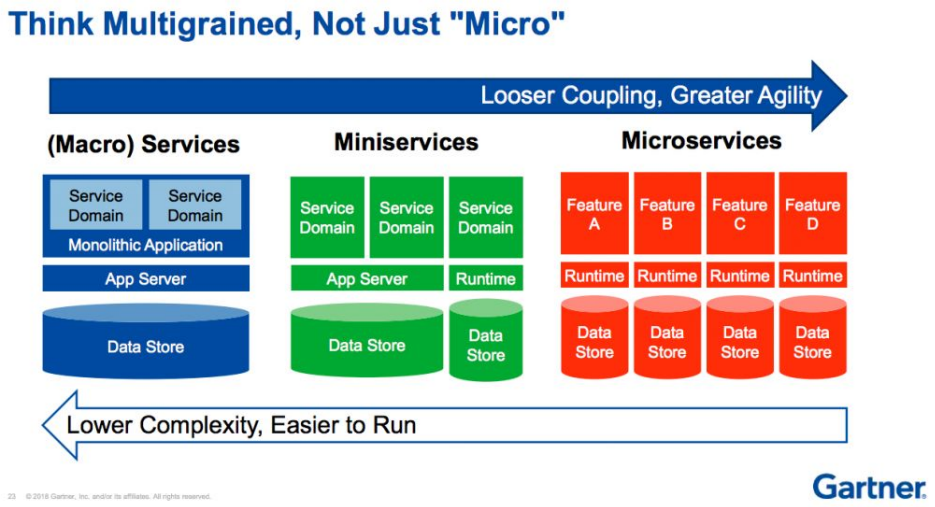


Figure 7.2: Miniservices architecture [22]

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

What does this mean for the architecture? It means that each service may contain multiple features but all of the features should be linked to the same domain. Which means that the communication inside a service is way 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 [22].

What are the advantages of miniservices over microservices? The most prominent answer 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. Also with miniservices each service connects to the same database. Which makes it easier to do complex querying.

### 7.1.3 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 way easier to manage but still having the modularity gotten from encapsulation.

Just like with minservices 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.

## 7.2 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 image below:

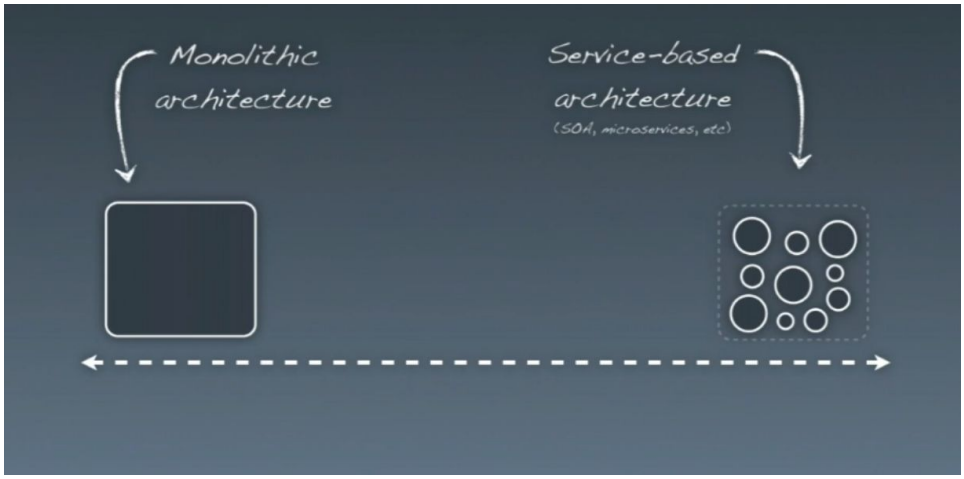


Figure 7.3: Monolith, microservices spectrum [10]

But this is not always the case as we showed in 7.1 Architectures.There are cases where microservices are the best choice and there are cases where miniservices or a modular monolith is the best choice. In this chapter I will compare the three architectures.

The question you should ask yourself is how important are these differences and why? This question can be coupled with the prioritization of the quality

attributes as 6.2.1 Recap

## 7.3 Complexity

Complexity always plays a role in choosing the right architecture. When looking at the three architectures shown 7.1 Architectures init is obvious that the complexity changes the smaller you go. Thus the most complex architecture is microservices and the least complex one is modular monolith. With miniservices right in the middle. But why?

In the image shown below there is an example of the microservice architecture. In this picture you can see 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 [21]. This is also the case with miniservices.

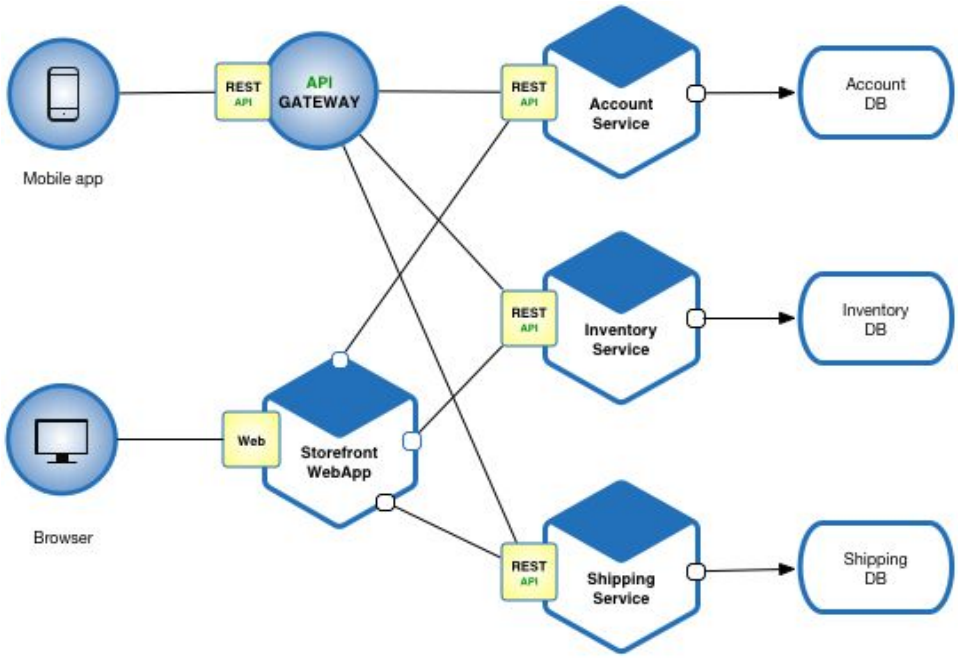


Figure 7.4: Microservice architecture

Even tho they can connect to a centralized datastore the services don't have any knowledge of each other. In a monolith 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. But there is another problem with miniservices.

A service in the miniservices does not know what is in the other model only that there is another model. So somehow it should know how to get the remaining fields. This is the same problem you have with microservices and thus is very complex.

As mentioned above why and how is this important is crucial to ask. But let's start linking this to quality attributes.

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.

## 7.4 Technology

One of the most convincing arguments for choosing microservices is the freedom of choosing the technology. You can 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 your own programming language but because all of the services talk with the same datastore the datastore technology is always the same.

Modular monoliths are the worst in this section. A modular monolith is stuck with the same technology as a programming language and with the datastore.

So which quality attributes are relevant to this attribute:

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

## 7.5 Testing

It is known that testing plays a big role in creating reliable software. There are multiple types of testing [23]. 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 we will be looking at:

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

### 7.5.1 Unit tests

1. Microservices
2. Miniservices
3. Modular monolith

This is because especially in a microservice because each function is its own service the functions are really easy to test. Because miniservices are 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 a bit tighter coupled than the miniservice.

### 7.5.2 Integration tests

1. Miniservices

2. Modular monolith
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 way more difficult. The reasoning behind this is the complexity of the service discovery. To test for example two services you need to setup the service discovery. With each service that needs to be tested it will become harder to test it because more services need to be discovered. A integration test with microservices may call 6 different services. But with miniservices it may be less if the functions that are called are in the same domain. This is why microservices gets the last place in this type of testing.

### 7.5.3 End-to-end tests

1. Miniservices
2. Modular monolith
3. Microservices

As seen with the previous test types the same problem occurs. A function that is called may need multiple microservices or miniservices called.

### 7.5.4 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.

### 7.5.5 Conclusion

So what are the quality attributes that testing influences:

- **Maintainability:** Maintainability is about effectiveness and efficiency. Both can be measured with load testing. It is an important quality attribute but because load testing is equal for all we don't consider this one
- **Compatibility:** Almost all of the test types look at if the code works and will fail if it changes. This concludes in looking at backwards compatibility and a check on it. Testing has a very high impact on this quality attribute.

- **Functional sustainability:** This is where testing started. Writing unit tests to be sure the functionality hasn't changes.
- **Performance:** With load testing performance will be tested but as with maintainability this won't be taken in consideration.
- **Usability:** End-to-end testing is specifically made for testing this and check if the usability is still the same.

## 7.6 Costs

Because EFFE is a startup costs are very important. EFFE does not have the steady money flow of a more mature company. Even though EFFE has gotten $20.000,- for Google cloud platform for one year we need to consider what happens after that year.

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

Right now EFFE employs one software engineer, Jessie Liauw A Fong. But he is also the co-founder so there is no money yet to be made by Jessie. This is because all the money EFFE makes goes right back into EFFE. But as mentioned before we should still consider what would happen if EFFE would hire more software engineers.

Development time and understanding the code go hand in hand. When you don't understand the code you can't develop. So how do these architectures hold considering development time and understanding the code?

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

Minservices 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 lot of code in one service or module. Therefore understanding the code and thus the developing time become larger.

But how big of a difference is there in this? Not much depending on your architecture. If your miniservices and modular monolith are structured in such a way that is logical it should not matter.

But when looking at the infrastructure there is a complete 180. 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 we have 5 clients who want this and 5 clients that run on the standard version we would have to run it on 6 servers which would be $30 dollars on the cheapest server which is not expensive at all.

Microservices and miniservices are the opposite side with microservice 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. But 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 we have less knowledge about our spendings beforehand.

How can you deploy multiple services automatically. There are some amazing services for this. The most known are Kubernetes, Nomad and Docker swarm. These services however cost $40,- per month with the minimum requirements. And that is if you only have three instances of such a service running. 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.

### 7.6.1 Conclusion

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

## 7.7 Scalability

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

Vertical scaling is when you are adding hardware resources 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. Because when deploying a new version of those instances you need to deploy in modular monoliths part the whole application.

When is scalability important? Well when talking about **performance**. This is also the quality attribute that is affected by scalability. When a server is going above a certain threshold it can duplicate itself and can now split the traffic along the new instance.

## 7.8 Frontend

So until now most of the comparison were for backend or did also apply to frontend but it was not mentioned. 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.

First of all let's define what EFFE considered for frontend. Right now we use Vue to create a single page application. But this does not mean other frontend frameworks are not considered.

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 ant that is the sharing of ui elements. You can 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. An talk about micro-apps (microservices frontend) gave a convincing story about why you should switch to them [14]. But when asked about the UI elements there was no answer to this question.

What happens in the current implementations of microservices and miniservices. The answer might surprise you. There is no such thing as microapps for big application. Most of them is just one single code base.

Concluding that for frontend there is only one possibility when looking at mod-

ularity and that is modular monolith.

## 7.9 Recap

There is a lot of articles written about monoliths vs microservices. In the end minservices has some of the benefits and cons of both.

This is a recap of what is discussed in 7 Modular architecture. As mentioned in 7.2 The comparison we would look at the quality attributes and how they matched up per architecture.

7.3 Complexity talks about the complexity and how it can influence the whole project in its whole. The one that came out best was modular monolith and the quality attributes that were applicable where **maintainability** and **security**.

7.4 Technology microservices came out on top with flying colors. With miniservices following and modular monolith as a obvious last. The quality attributes that are influenced by technology are **,ompatibility**, **performance** and **portability**

7.5 Testing was by far the most contested section with no clear winner. But if you looked 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**

7.6 Costs the clear winner in this section was modular monolith with miniservices following and microservices at a obvious last place. The quality attributes affected by the cost is **maintainability**

7.7 Scalability the obvious winner was microservices. In second place was miniservices andlast was modular monolith. The quality attribute applicable is **performance**.

7.8 Frontend eventually there was only one architecture that actually made sense for frontend and that was the modular monolith.

## 7.10 Conclusion

The architecture that fits EFFE best is the modular monolith. The chapters where modular monolith took the first place by far were also the ones that influ-

enced the quality attribute **maintainability** the most. As sorted in 6.2.1 Recap **maintainability** is by far most important to EFFE. EFFE also does not have a lot of money as mentioned in 7.6 Costs Thus costs play a big part in this decision as well. Last of all 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.

Microservices have the clear distinction of winning the race on technology and scalability but this is not where the focus lays. Though technology will offer some of our focusses is 24does not compete with the main focus that the modular monolith architecture touches on and the cons do not outweigh the pros.

So why not miniservices? Miniservices is a mixture of modular monolith and microservices and it does take some good parts of the both architectures but also some cons of both architectures. The biggest con here is the complexity of the datastore. Even though the same datastore is shared over the services you cannot get everything in one request. As explained in 7.3 Complexity.

And last of all both microservices and miniservices cost to much for EFFE at this stage of the company.

# Chapter 8

# Implementation of the architecture

The chosen implementation is modular monolith. Let's go over the characteristics of modular monolith. As mentioned in 7.1.3 Modular monolith a modular monolith is a domain driven design where the modules can be developed separately. Thus most of the principles we can take from domain driven design. But because the modules do not know what other modules contain. Thus there should be a kind of api on which the modules talk.

## 8.1 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 standard for domain driven design. Let's first define what a domain is.

> "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. The main distinct feature is that in a modular monolith

the module does know that another module exists but not the contents of this module. This is not the case with domain driven design.

The difference that this makes it that each module should be able to talk with each other and thus there need to be an api over which each module can talk with each other.

## 8.2 Current situation

This architecture will apply to the application of EFFE. Below is a list of the functionalities of the current application. This is so we can paint a good picture of the current situation.

Each functionality belongs to a certain domain. When looking at the functionalities we can determine that there are x domains which are the following:

| Domain | Functionalities | Is building block |
|---|---|---|
| User | • Reset password<br>• User CRUD* | No |
| Shift | • Shift overview<br>• Create shift | No |
| Skill | • Skill CRUD* | No |
| Store | • Store CRUD* | No |
| Client | • Client CRUD* | No |
| Authentication | • Login<br>• Reset password | No |
| Schedule | • Generate schedule | No |
| Hour registration | • Hour registration | Yes |
| Shift market | • Shift market | Yes |
| Shift change | • Switching shifts<br>• Calling in sick | No |
| Company | • Managing company settings | No |

* CRUD or Create Read Update Delete refers to the actions that can be called on an object

As you can see there are only two building blocks. Those are the Hour Registration

and the Shift market. But this does not mean that the other domains should not be modules. If every domain is a module it makes it easy for us to change a basic function if an enterprise wants that.

## 8.3 API

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. [26]"

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. REF TO IMPLEMENTATION

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 like a dataclass or a serializer. This shows how an object will be serialized into a JSON object and send back and forth via http. 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. But when working with the application of EFFE the company found that each user role may require a other specific serializer. For example: EFFE has three roles: the employment agency employee, the client and the temp worker. If the client and the temp worker want to retrieve the shifts the client also gets the users in that shift while the temp worker only sees the general data of the shift. This is so that temp workers won't have the biased in taking shifts with people they like or vise versa.

Therefore it is important to note that each role should have a specific serializer. So in the API there should be **base serializer** and the option to change the **serializer by role**.

## 8.4 Programming language and Web framework

It is obvious why programming languages will be researched but maybe not why there is a need for a web framework. As mentioned before in this chapter most of the web applications use a 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. [24]"

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 only thing that will be researched is the modularity of the framework or language.

### 8.4.1 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 [13]

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

What is interesting to note is that each framework uses a different programming language. There is one that jumps out of the languages that are used. That is Java. This is because Java is the only one that is statically typed. One of the requirements was being as flexible as possible and this is just not possible when working with a statically typed language. Therefore Spring falls off.

There are four frameworks left. These frameworks will be tested with this use case:

There is a shift module. The model has four attributes

- Title
- Start date
- End date
- Employees

And there is a employee module with the model having these attributes:

- Name
- Birth date

- Email

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

Last of all the modules should only talk with each other via the 8.3 API.

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

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

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

- 10.1 Creating modular monolith with Django

# Chapter 9

# 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/master/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] 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).

[14] 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.

[15] 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).

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

[17] 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).

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

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

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

[21] 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).

[22] 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).

[23] *Types of Software Testing: Different Testing Types withDetails*. Dec. 31, 2018. URL: https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/ (visited on 04/03/2019).

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

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

[26] *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).

# Chapter 10

# Appendix

## 10.1 Creating modular monolith with Django

First thing to do is install django by running:

```
pip install Django
```

Now you can start the django project with the name modular_monolith:

```
django-admin startproject modular_monolith
```

There is now a folder is created called modular_monolith. But before we start coding we have to add mysql as the database. First install the python mysql connector:

```
pip install mysqlclient
```

Now replace `DATABASES` variable in `modular_monolith/settings.py` with

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'django',
        'USER': 'root',
        'PASSWORD': 'root'
    }
}
```

Now we can create the standard tables by running:

```
python manage.py migrate
```

Django already works with encapsulated modules but they call them apps. The next thing to do is writing the api. This can be done in the modular_monolith section which

serves as the general folder. Because Django is object oriented the api can also reflect that. In 8.3 API the specific attributes are defined thus our api would look like this:

```
class ModuleAPI:
    def __init__(self, model, base_serializer, serializer_per_role=None):
        self.model = model
        self.base_serializer = base_serializer
        self.serializer_per_role = serializer_per_role
```

Because Django already supports the idea of modules we can run two simple commands to create the shift and employee modules:

```
python manage.py startapp shifts
python manage.py startapp employees
```

The first thing to do is creating the employee model in `employees/model.py`

```
from django.db import models


class Employee(models.Model):
    name = models.CharField(max_length=255)
    birth_date = models.DateField()
    email = models.EmailField()
```

Before the creation of the api for employees we create the base_serializer. For this we need a rest_framework:

```
pip install djangorestframework
```

Now add rest_framework to settings.py in `modular_monolith/settings.py`

```
INSTALLED_APPS = [
    'rest_framework',
    ...
]
```

Create the serializer in `employees/serializers/base.py` like so:

```
from rest_framework import serializers
from employees.models import Employee


class BaseSerializer(serializers.ModelSerializer):
    class Meta:
        model = Employee
        fields = '__all__'
```

Next the creation of the module api for employees. We do this by creating `api.py` in `employees/api.py` with the contents:

```
from employees.models import Employee
from employees.serializers.base import BaseSerializer
from modular_monolith.api import ModuleAPI

api = ModuleAPI(Employee, BaseSerializer)
```

Now the model of shift can be created:

```
from django.db import models
from employees.api import api


class Shift(models.Model):
    title = models.CharField(max_length=255)
    start = models.DateField()
    end = models.DateField()
    employees = models.ManyToManyField(api.model)
```

As you can see this is the first time the module api is used. The api is included and used to create a many to many relationship.

The shifts serializer is the same as the one from employees except for the model:

```
from rest_framework import serializers
from shifts.models import Shift


class BaseSerializer(serializers.ModelSerializer):
    class Meta:
        model = Shift
        fields = '__all__'
```

The api of shifts looks like this:

```
from shifts.models import Shift
from shifts.serializers.base import BaseSerializer
from modular_monolith.api import ModuleAPI

api = ModuleAPI(Shift, BaseSerializer)
```

The django application expects that in `modular_monolith/settings.py` the apps are added to `INSTALLED_APPS` like such:

```
INSTALLED_APPS = [
    ...
    'employees',
    'modular_monolith'
]
```

Now run the command makemigrations and migrate in order to create the tables for the new modules:

```
python manage.py makemigrations
python manage.py migrate
```

Add the list serializer to the `serializers/list.py` from both modules as such:

employees/serializers/list.py

```
from rest_framework import serializers
from employees.models import Employee
from shifts.api import api


class ListSerializer(serializers.ModelSerializer):
    shifts = api.base_serializer(source='shift_set', many=True)

    class Meta:
        model = Employee
        fields = '__all__'
```

shifts/serializers/list.py

```
from rest_framework import serializers
from shifts.models import Shift
from employees.api import api


class ListSerializer(serializers.ModelSerializer):
    employees = api.base_serializer(many=True)

    class Meta:
        model = Shift
        fields = '__all__'
```

Now add the views to `employees/views.py` and `shifts.views.py` respectively as such:

employees/views.py

```
from rest_framework import viewsets
from employees.models import Employee
from employees.serializers.list import ListSerializer


class MainViewSet(viewsets.ModelViewSet):
    queryset = Employee.objects.all()
    serializer_class = ListSerializer
```

shifts.views.py

```
from rest_framework import viewsets
from shifts.models import Shift
from shifts.serializers.list import ListSerializer


class MainViewSet(viewsets.ModelViewSet):
    queryset = Shift.objects.all()
    serializer_class = ListSerializer
```

Next is the addition of the views to the main application. Thus `modular_monolith/urls.py` looks like:

```
from rest_framework.routers import DefaultRouter
from shifts.views import MainViewSet as Shift
from employees.views import MainViewSet as Employee

router = DefaultRouter()
router.register('employees', Employee)
router.register('shifts', Shift)

urlpatterns = router.urls
```

Now run `python manage.py runserver` and these urls are available to see the created api

- localhost:8000/employees/
- localhost:8000/shifts/

This test uses php version 7.2.11, composer 1.8.5 and laravel 5.8.19

First install laravel via composer

```
composer global require laravel/installer
```

Then create the laravel project

```
laravel new modular_monolith
```

Unlike some other frameworks you need to create the migrations yourself by running:

```
php artisan make:migration create_employees_table
php artisan make:migration create_shifts_table
php artisan make:migration create_employee_shift_table
```

There will be a file created in `database/migrations` which ends with `create_employees_table`. Replace the `up` function with this:

```
Schema::create('employee', function (Blueprint $table) {
  $table->increments('id');
  $table->string( 'name');
  $table->date('birth_date');
  $table->string('email');
});
```

And for the file that ends with `create_shifts_table`

```
Schema::create('shifts', function (Blueprint $table) {
  $table->increments('id');
  $table->string('title');
  $table->date('start');
  $table->date('end');
});
```

Now last of all the table that connects the two models needs to be made. This is because it is a many to many relation. Laravel does not pick this us automatically. Therefore again replace the up function of the file that ends with `create_employee_shift_table`

```
Schema::create('employee_shift', function (Blueprint $table) {
    $table->bigIncrements('id');

    $table->integer('employee_id')->unsigned()->nullable();
    $table->foreign('employee_id')->references('id')
        ->on('employees')->onDelete('cascade');

    $table->integer('shift_id')->unsigned()->nullable();
    $table->foreign('shift_id')->references('id')
        ->on('shifts')->onDelete('cascade');
});
```

First the `.env` needs to know the database settings. Replace these variables

```
DB_DATABASE=laravel
DB_USERNAME=root
DB_PASSWORD=root
```

Now migrate the database by running

```
php artisan migrate
```

Then create the Employee model in `app/Employees/Employee.php`

```
<?php
namespace App\Employees;

use Illuminate\Database\Eloquent\Model;
```

```
class Employee extends Model
{
    public $name;
    public $birth_date;
    public $email;
    public $fillable = ['name', 'birth_date', 'email'];
    public $timestamps = false;

    public function shifts()
    {
        return $this->belongsToMany(\App\Shifts\Api::getModel());
    }
}
```

Then the controller for Employees will be created:

```
<?php
namespace App\Employees;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;

class EmployeeController extends Controller
{
    public function index()
    {
        return response()->json(Employee::with('shifts')->get());
    }

    public function store(Request $request)
    {
        $employee = $request->all();
        $employee['birth_date'] = \Carbon\Carbon::parse($employee['birth_date']);

        return Employee::create($employee);
    }

    public function show($id)
    {
        return Employee::with('shifts')->find($id);
    }
}
```

Next is the shift model

```
<?php
```

```
namespace App\Shifts;

use Illuminate\Database\Eloquent\Model;

class Shift extends Model
{
    public $title;
    public $start;
    public $end;

    public $fillable = ['title', 'start', 'end'];
    public $timestamps = false;

    public function employees()
    {
        return $this->belongsToMany(\App\Employees\Api::getModel());
    }
}
```

And the matching Controller:

```
<?php
namespace App\Shifts;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;

class ShiftController extends Controller
{
    public function index()
    {
        return response()->json(Shift::with(['employees'])->get());
    }

    public function store(Request $request)
    {
        $shiftArray = $request->all();
        $shiftArray['start'] = \Carbon\Carbon::parse($shiftArray['start']);
        $shiftArray['end'] = \Carbon\Carbon::parse($shiftArray['end']);

        $shift = Shift::create($shiftArray);

        $shift->employees()->sync($shiftArray['employees']);

        return $shift;
    }
```

```
    public function show($id)
    {
        return Shift::with(['employees'])->find($id);
    }
}
```

Now the urls can be mapped to the controller by adding this to the `url/web.php`

```
Route::resources([
  'employees' => '\App\Employees\EmployeeController',
  'shifts' => '\App\Shifts\ShiftController'
]);
```

The last thing to do is disable csrf for the paths. This can be done in `app/Http/Middelware/VerifyCsrfToken.php`

```
protected $except = [
  'employees',
  'shifts'
];
```

Now run the app with

```
php artisan serve
```