Expertise
and insight
for the future

Hoang Vo

Applying microservice architecture with modern gRPC API to scale up large and complex application

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

28th April, 2021

Author Title Number of Pages Date	Hoang Vo Applying microservice architecture with modern gRPC API to scale up large and complex application 46 pages 28th April, 2021
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Janne Salonen, Head of School of ICT Department

The main objective of this thesis is to explain how microservices and gRPC works and discuss its implementations, limitations and usages. Reliable online articles were used to support the demonstration of a microservice with gRPC servers.

The complexity of modern applications has significantly increased in the past decades. Traditional applications have limitations with the monolith server architecture and the traditional HTTP/1 transportation layer. Using this information, two small projects were designed based on the microservices methodology and gRPC.

The first project utilized NodeJS and ReactJS to show the usage Event Bus in microservice communication. The second project used gRPC on top of HTTP/2 with different programming languages. Data communication and latency were two key things to consider in both of the projects.

In conclusion, the projects demonstrates the usage of microservices and gRPC in a software application and its effectiveness in software scalability.

Keywords	Microservice, scaling, gRPC, REST, API, architecture, server, backend, frontend, protocol buffers, Javascript, NodeJS

Contents

List of Abbreviations

1 Introduction 1

2 Monolith architecture in a nutshell 2

2.1 What is monolith architecture? 2
2.2 Advantages of monolith architecture 3
2.3 Disadvantages of monolith architecture 4

3 Microservice in action 5

3.1 Introduction to microservice architecture 5
3.1.1 Data in microservice system 6
3.1.2 Synchronous communication in microservice system 8
3.1.3 Asynchronous communication in microservice system 10
3.2 Advantages of microservice 12
3.2.1 Deployment 12
3.2.2 Scalability 12
3.2.3 Easy adaptability of different technology stacks 12
3.2.4 Fault tolerance 13
3.2.5 Parallel development 13
3.3 Disadvantages of microservice architecture 13
3.3.1 Challenges for local environment set-up 13
3.3.2 Unnecessarily complex DevOps operation 14
3.3.3 Expertise problems 14
3.3.4 Strictness in following rules 14
3.3.5 Overlooking the complexity of service communications 15
3.3.6 Challenging versioning process 15
3.3.7 Giant monolith service 15
3.3.8 Networking 16
3.4 Micro-frontend applications 16
3.4.1 Advantages of micro-frontend 17
3.4.2 Disadvantages of micro-frontend 18
3.5 Dockers and Kubernetes for microservices 19
3.5.1 What is container? 19

3.5.2 Why Docker? 20
3.5.3 Why Kubernetes? 21
3.6 Monorepo for microservices 23
3.6.1 Monorepo and Polyrepo 23
3.6.2 Monorepo usage in microservice application 24
3.7 Implementing a simple microservice application 24
3.7.1 Setting up 24
3.7.2 Basic data flow and implementation 26
4 When to use monolith approach or microservice approach? 32
4.1 Team size 32
4.2 State 32
4.3 Dependency 32
4.4 Expertise 33
5 Introduction to gRPC 33
5.1 What is protocol buffer? 34
5.2 What is HTTP/2? 35
5.2.1 Disadvantages of HTTP/1.1 35
5.2.2 Advantage of HTTP/2 35
5.3 gRPC in a nutshell 36
5.3.1 Different types of gRPC APIs 36
5.3.2 Scalability in gRPC 39
5.3.3 Security in gRPC (SSL) 39
5.4 gRPC versus REST 39
5.4.1 Protobuf versus JSON 39
5.4.2 HTTP/2 versus HTTP/1.1 40
5.4.3 Streaming versus Request-Response 40
5.4.4 API oriented versus CRUD oriented 40
6 Building large-scale application using microservice and gRPC 40
6.1 Project overview 40
6.2 Project's architecture 41
7 Conclusion 44
References 46

List of Abbreviations

MS	Microservice
gRPC	gRPC Remote Procedure Call
RPC	Remote Procedure Call
CI	Continuous Integration
CD	Continuous deployment / Continuous delivery
REST	Representational State Transfer
API	Application Programming Interface
URL	Uniform Resource Locator
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
IDE	Integrated Development Environment
CRUD	Create, Read, Update, and Delete
MVP	Minimum Viable Product
VCS	Version control system
Protobuf	Protocol buffer

1 Introduction

In 2000, Roy Fielding introduced REST (Representational State Transfer) to the world when he was working on his doctoral dissertation in the University of California, Irvine, the United States. Since then, REST, called RESTful architecture, has become the most popular architecture in modern web and mobile applications. REST defines a set of strict constraints that every API (Application Programming Interface) should follow in order to communicate between a client and a server to manipulate all data resources. [1.]

The REST API uses the HTTP protocol with URLs (Uniform Resource Locator) and HTTP methods as a medium to transfer data and, hence, creates a separate layer called client (or frontend) and the server (or backend). Therefore, the term monolith (or monolithic) architecture has also been introduced. Monolith comes from the Greek word "mono", which means "single". The monolith architecture is a system design in which every service is composed of one single large component. [1.] As the complexity of the modern software increases, the monolith architecture becomes more and more impracticable and unscalable, and that is when microservices come in handy. Microservices attack the problem of scalability from a different angle where large applications are divided into several completely separate services (or loosely coupled services), each application serving its own business logic and purposes and using its own resources. With this approach, each service has its own capability to scale up without worrying about other components.

With the introduction of microservices and the RESTful API, the world of software architecture has to face another critical issue, communication. An extremely large system is a good example. For instance, the Facebook application has several thousand smaller services for both the client side and the server side. In order to communicate internally, each component needs to obey certain rules to talk to their desired component, which means things can get complicated. gRPC solves all the issues with its feature-rich and performant framework on the basis of the RPC communication protocol that was first developed in the R&D department of Google in 2015. [3.]

A modern application always aims to be performant; however, overusing or underusing all the latest technology and architecture might worsen the software. The purpose of this

thesis is to give an in-depth overview of the microservices architecture and gRPC, including all the different aspects that developers need to consider adopting either one or both of the technologies.

2 Monolith architecture in a nutshell

2.1 What is monolith architecture?

As mentioned above, monolith design is an approach in which one large system consists of all the business logic handlers. Usually, components in monolithic system are interconnected and dependent on each other. [2.] Each component has to be presented and be executable in order for the whole system to work. If one component fails its job, usually the code of the whole project will not be compiled.

In the monolith architecture, one system will have one database and, therefore, all the services point to that single database [2]. An example architecture of the monolith system for an e-commerce site is presented below.

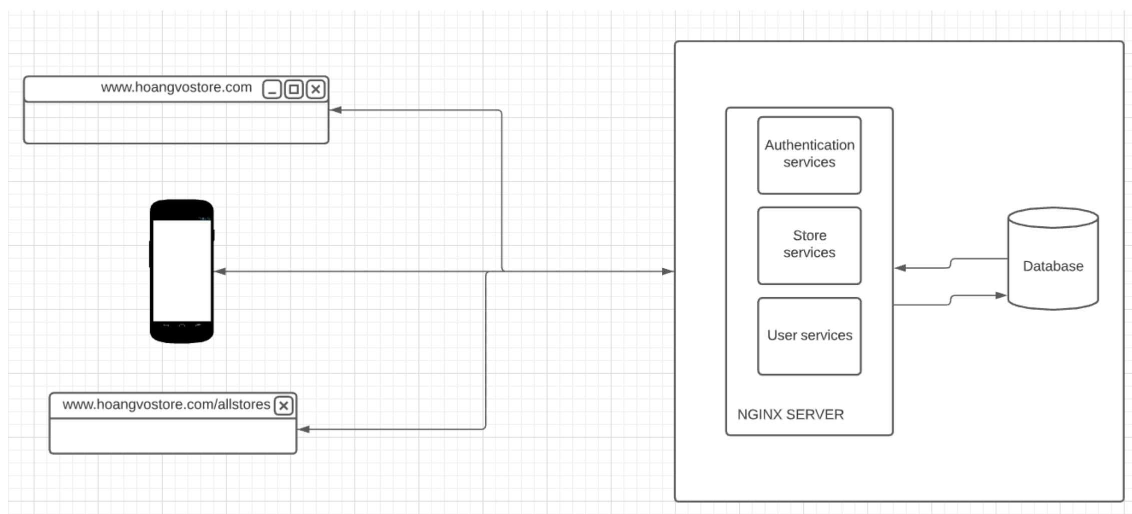


Figure 1. Simple monolith architecture in an e-commerce application

As shown in Figure 1, data resources are transferred back and forth only between one backend server and clients. The whole backend service interacts with one single database system. All the business logic handlers, such as the authentication handler and

Metropolia University of Applied Sciences

store handler, are tightly coupled. Despite having multiple modules, the whole code base is compiled and deployed as one big application.

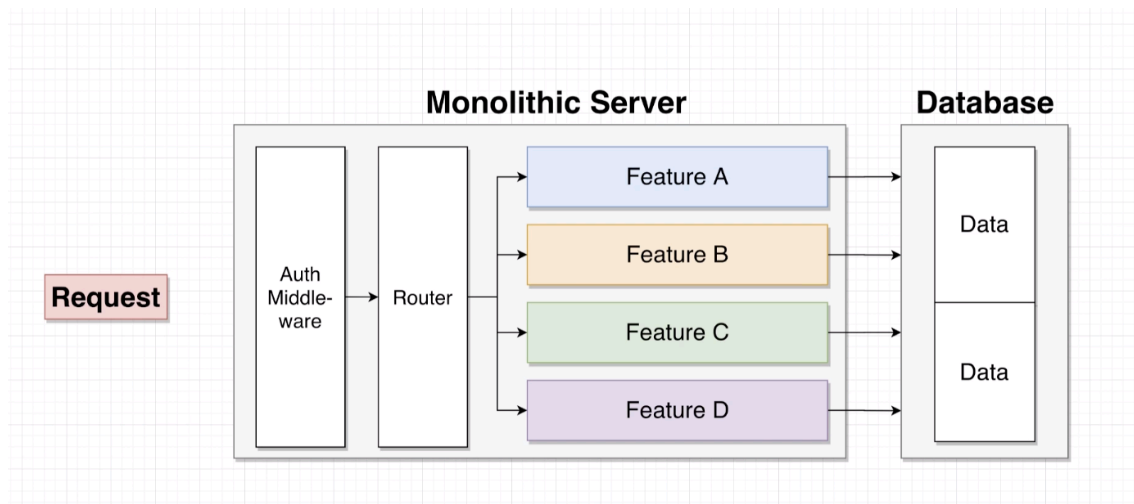


Figure 2. More in-depth flow of a monolith server. Data gathered from Haq (2018) [2].

As in Figure 2, any kind of RESTful request will go through some authentication middleware before reaching a router. A router decides the appropriate feature component to further process the request. Then that feature makes some CRUD calls with its database and sends the response back to the clients.

To sum up, the monolith server will contain all the routings, middleware, business logic and database access code required to implement all features of an application.

2.2 Advantages of monolith architecture

As being one of the most common architectures for modern applications, the monolith architecture has its own advantages that would make it a feasible choice in certain scenarios.

First and foremost, setting up a monolith application is smooth and straightforward, and developers can start coding almost instantly. Therefore, it is usually a starting point for the most greenfield projects. In this modern world, people usually make a minimum viable product (MVP) or a quick demo before starting the project properly and the monolith architecture comes in handy in these cases.

Simple development leads to simple testing. A monolith application also provides an easy approach to do end-to-end testing, such as Selenium and Cypress. As everything is running in one big service, end-to-end testing does not need to communicate between different services. [2.]

Scaling up a monolith system is extremely easy with a horizontal approach by simply making the load balancer to run multiple more servers in case of emergencies.

Last but not least, the main advantage of using the monolith approach is its simplicity in the deployment process as we only have one single package.

2.3 Disadvantages of monolith architecture

Regardless of how straightforward the initial setup that the monolith architecture offers, it has many drawbacks that make it a no-go for a large and complex system.

Maintaining a monolith system is a burdensome job as the system gets deeper. Unlike microservices, each component in a monolith architecture is used in different places and affects many areas within the system. Fixing one small component can lead to a whole system refactoring. Furthermore, reliability is not at the highest level when it comes to the monolith approach. One small bug in one component can be lethal to the entire system. To make matters worse, all the instances are affected by a single bug as scaling a monolith system is done horizontally.

Despite deploying, the monolith system is straightforward. It is not effective as the whole system needs to be re-deployed regardless of what has been changed [2]. This also means that adopting new technology is almost impossible to do on the fly with the monolith approach. Usually, the whole code base needs to be rewritten in order to have new technology.

3 Microservice in action

3.1 Introduction to microservice architecture

As described in the monolith architecture section, the drawbacks of the monolith system outweigh its advantages for a large enterprise application. Therefore, microservice architecture is introduced to tackle the issues that monolith application carries. [2.]

The microservice architecture is a method of splitting one very large system into a set of several smaller interconnected services which are executed independently on their own environment. [2.]

Each smaller service is usually built on the basis of its dedicated features or functionality, and it can be reused throughout the entire application or even different application within an organization. For example, order management service can be used within the e-commerce customer application or the e-commerce administrator application, and user management can be used in an external application or an internal application to track users. Each microservice is deployed and run on its own platform, so it is completely independent from other services and can be maintained, updated and deployed without touching other parts of the system.

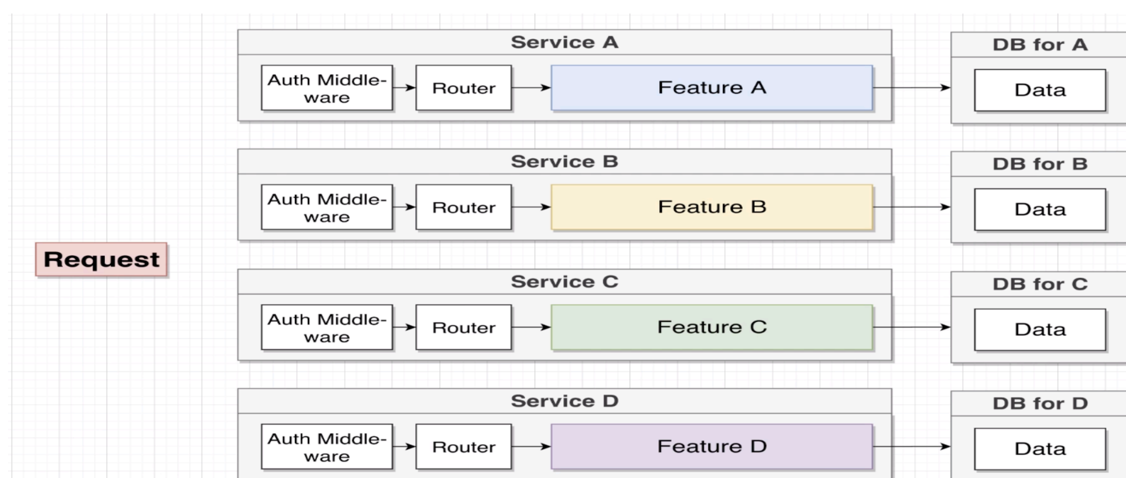


Figure 3. General flow of a microservice server. Data gathered from Haq (2018) [2].

Figure 3 shows how microservices would look like in case of approaching the same monolith system as shown in Figure 2. As can be seen from the diagram, each feature has its own service and its own database, router and authentication middleware.

To sum up, a single microservice contains routing, middleware, business logic, database access and a database.

3.1.1 Data in microservice system

From the microservice notion, it seems to be straightforward to break down a monolith system into a microservice system as reusing the same routing techniques and the same middleware is possible throughout the application. However, there is one important issue that needs to be well-thought prior to implementing a proper microservice system, and that is data management.

Storing data and accessing data in a microservice system is different from the monolith approach. As stated in the previous section, each small service should have its own database and data access so one service reaches only one dedicated database without ever touching another service's database. It does not sound effective, and it goes against the scaling ability that the microservice gives; therefore, the big question is what is the reason why separating the database is the only approach?[3.]

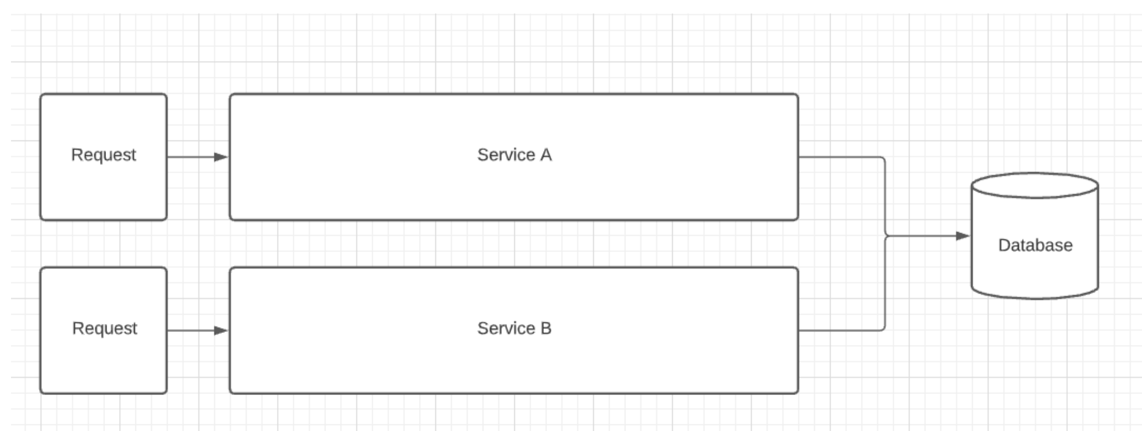


Figure 4. Data flow in case of using one database for all microservices

The first reason is demonstrated in the figure above. Figure 4 shows a microservice system which both service A and service B use the same database. In order to scale up

all the required collections or tables that are used by service B, the whole database needs to be horizontally scaled up and that leads to cost inefficiency as service A requires no scaling. Considering other scenarios where one collection or one table that is used by service B is broken, the whole database is unusable and both service A and service B are crashed.

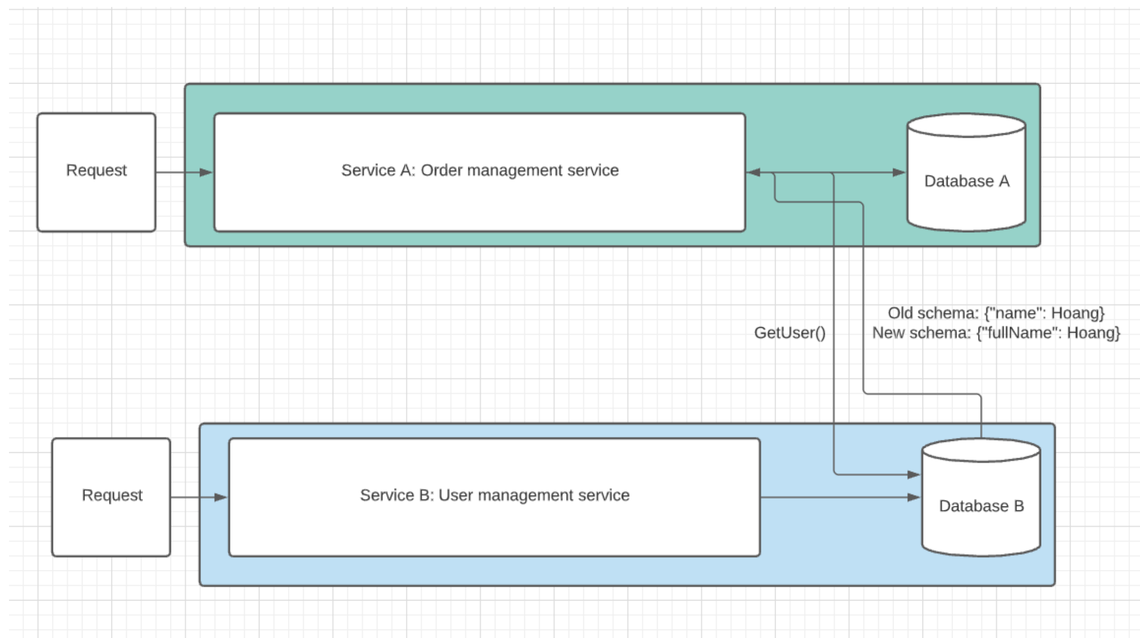


Figure 5. Data flow in case of one service reaches other service's database

The second reason is demonstrated in the above diagram in which Service A has access to Database B which belongs to service B. In the above example, service A tries to reach Database B to ask for the information of one user by calling "getUser()". Database B responds with "{'name': 'Hoang'}" and service A has configured so that it will expect the property 'name' in the response. However, in case of schema changes in database B, "getUser()" is responded with "{'fullName': 'Hoang'}". This scenario would break the whole service A as service A is not configured to handle 'fullName' as a return property.

Last but not least, using one database for a different service is not effective in a complex system as each small service might work best with certain type of database management systems (NoSQL or SQL).

To summarize, a single microservice will only have access to one single database system and it should never reach out other databases of the microservice.

3.1.2 Synchronous communication in microservice system

As stated above, each service should only communicate within its dedicated database. This can lead to an issue in case multiple pieces of information across different services are required for one single request. Solving these issues is the biggest problem for every effective microservice structure.

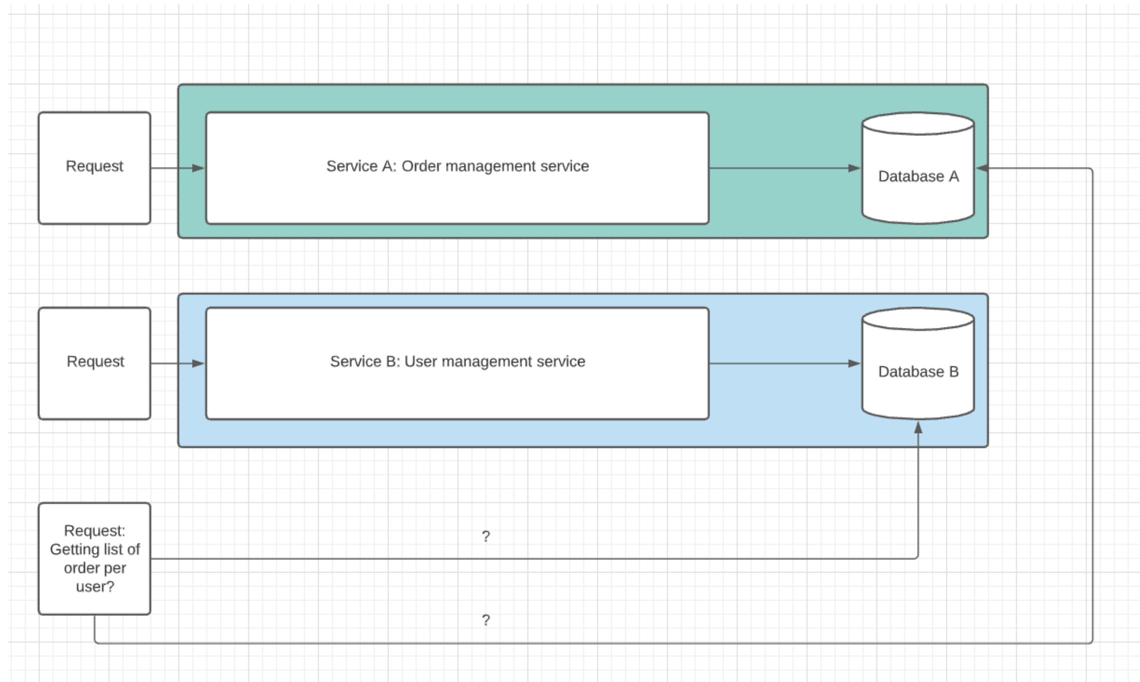


Figure 6. Diagram showing one request requiring information from a different database

Taking the example presented in Figure 5 into consideration, Service A is an order handler and Service B is a user handler. Trouble arises if the request requires getting the list of orders per user. In theory, information from both database A and database B needs to be in use without breaking the rules of data flow in the microservice. “Sync communication” is one way of tackling down this issue.

Metropolia University of Applied Sciences

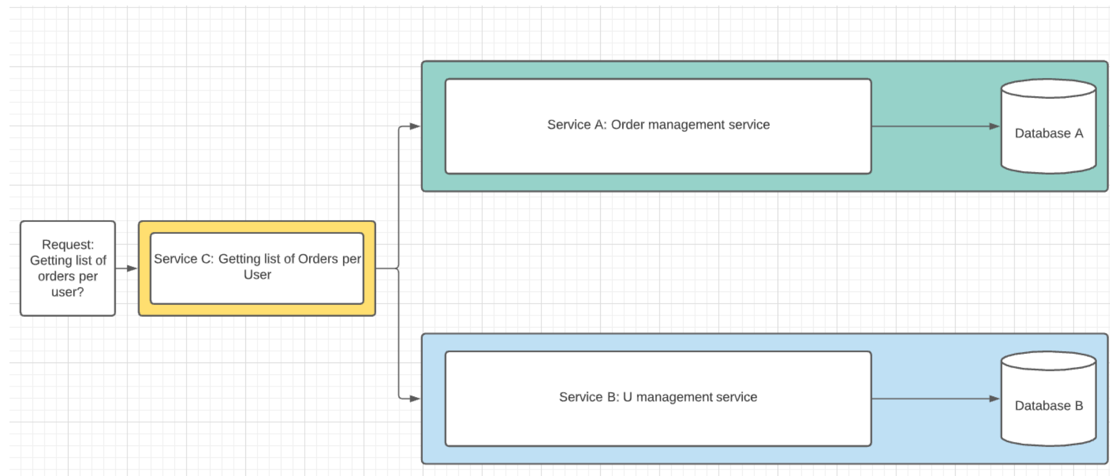


Figure 7. Simple diagram showing sync communication in a microservice system

In the microservice notion, “sync communication” means each internal microservice should use “direct request” to communicate with others. In the above example, in order to get a list of orders per user, service C is created. Its job is to make a direct request (it can be an HTTP request or any other type of request) to service A to get a list of orders, then make another request to service B to get a list of users and combine the two results as a response to the origin of the request.

This “sync communication” approach does not break any rules of microservice as service C does not reach either database A or database B. However, there are advantages and disadvantages with this type of service communication.

The upside is that this method is conceptually straightforward and easy to understand. Furthermore, no database is required for service C.

On the other end, this approach introduces a new set of problems. Dependency between services is initialized, and therefore, if one single request fails or some data is unexpectedly returned, the whole service C will be broken. The more complex the system becomes, the easier it is to have several deep webs of requests and the performance also becomes extremely slow.

3.1.3 Asynchronous communication in microservice system

In terms of microservices, async communication refers to a method where microservices talk to each other via events and event buses [4].

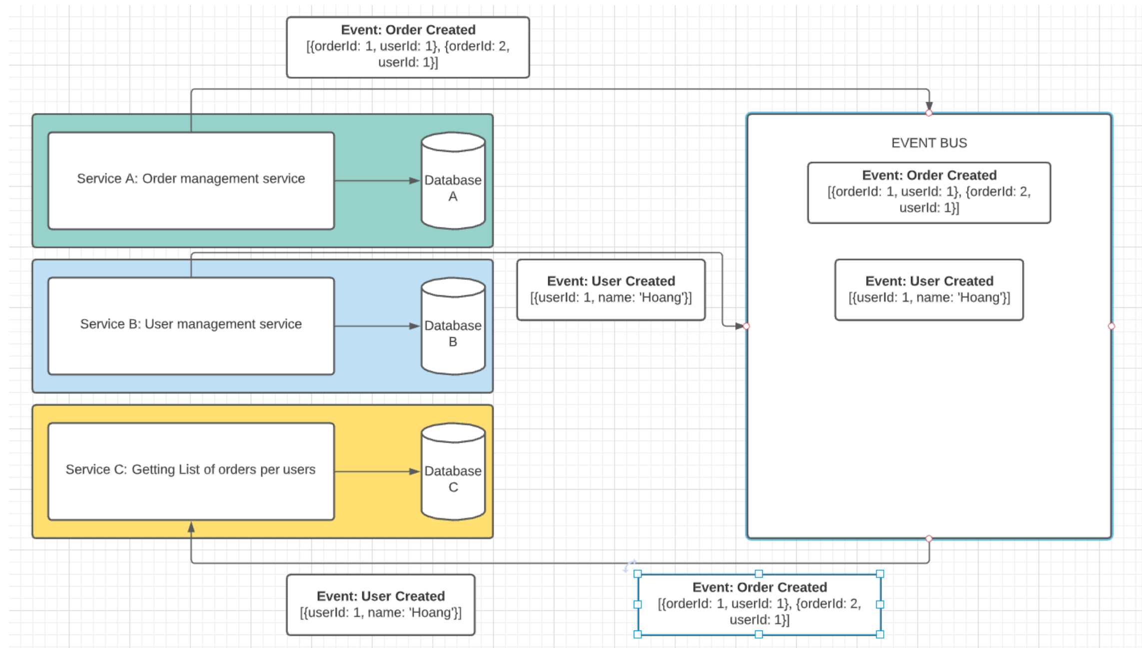
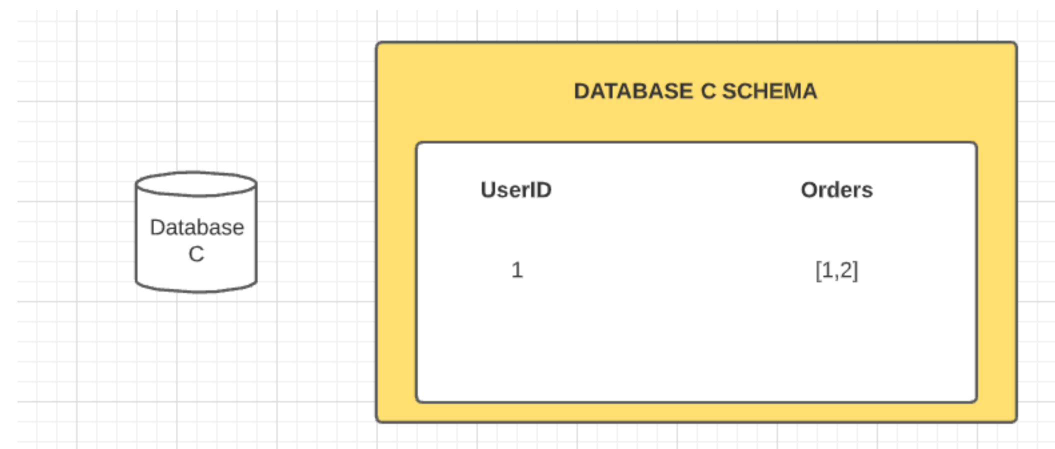


Figure 8. Basic flow of async communication in an e-commerce application

Figure 8 shows the basic architecture of an microservices e-commerce application. As can be seen from the Figure, Event Bus is used to handle all transactions happens between each service and no direct dependencies are introduced between services.



Metropolia
University of Applied Sciences

Figure 9. Example of a schema for Database C

Considering the example presented in the previous section that service C wants to get the information that is related to service A and service B. In a normal sync communication, direct request from service C to service A and B is used; however, it introduces some dependencies between services and the issues would get complicated. Async communication tackles this problem by using “Event Bus”. Every service in the application needs to notify and subscribe to an “Event Bus” every time some actions take place and to react accordingly.

Figure 9 shows the basic schema for database C, which only has connection to service C; therefore, database C should contain all the required information such as “UserId” and “Orders”.

Figure 8 shows how “Event Bus” works in this scenario. Every time a user is created, service B stores that user data into its database B. At the same time, service B emits an “Event” in this format *“{ type: ‘UserCreated’, data: { userId: 1, name: ’Hoang’ } }”* to “Event Bus” to notify all “Event Bus” subscribers that an action with type “UserCreated” has happened. Service C is interested in that event type, so it listens out for “Event Bus” whenever “UserCreated” appears and stores the data to database C *“{ userId: 1, orders: [] }”*. The same thing applies to service A; service A emits an event at the same time it stores the information to its own database. Service C listens to that event in the event bus and updates its database accordingly. At the end, service C should only retrieve information from database C without touching any other service; hence, no direct dependency is created between microservices.

There are advantages and disadvantages with this type of approach. An immediate disadvantage is about data duplication. Data is repeatedly passed around the system; for example, a single piece of data is transferred from service A to database, from service A to event bus, and from event bus to service C. A second disadvantage comes from its complex concept that is hard to understand and, therefore, developers require longer onboarding time with the project. However, the benefits outweigh the drawbacks. With this architecture, there is zero direct dependencies between services. Service C relies on the information from service A or B in order to store data in its database; however, that process happens asynchronously in the background, not when a request is made like in “sync communication”. If service A and service B are broken down, service C

works independently on its own as its database is already up to date. Another advantage is that service C is extremely fast as it does not need to wait for any returned data from other services every time a request is made.

3.2 Advantages of microservice

The main advantage of adopting a microservice in a large and complex system comes from independence and modularization characteristics.

3.2.1 Deployment

First and foremost, deployment becomes more flexible in a microservice system. Each service is capable of being tested, built and deployed without worrying about the other parts of the system [4]. Comparing to deploying the whole application, deploying a process for a microservice is relatively small and quick; therefore, frequency and building speed can be improved drastically. The release cycle of one service is completely separate from others, meaning it cannot be blocked if other the deployment of the service; therefore, the risk of failing is smaller.

3.2.2 Scalability

Scaling is another great advantage of building a microservice system. In most complex applications, there are always features that are extremely complicated and function as bottlenecks to the whole system. Usually, these features require the most attention from developers so it would not affect the whole system's scaling ability. With microservice approach, the resource used to scale a feature does not affect all the other uncritical services.[4.]

3.2.3 Easy adaptability of different technology stacks

Various and independent technology stacks can be used to optimize each service's performance and scalability, for example, graph-based stacks, protocol-buffer stacks, the NoSQL database, and the SQL database. Also, the usage of each technology stack is

Metropolia
University of Applied Sciences

totally dependent on that service, so all the unnecessary libraries or modules can be skipped.[4.]

3.2.4 Fault tolerance

The fault tolerance level a of microservice system is higher compared to a monolith system. Each service's failure does not cause the whole system to be broken.

3.2.5 Parallel development

Last but not least, parallel development is one of the biggest reasons why a microservice is a good option for some teams. Team A will build (and deploy) service A separately from Team B. There are fewer coordination issues and dependencies, and business functionality can be built quicker. This is an important point for business managers to think about when adopting a microservice when a system gets larger.

3.3 Disadvantages of microservice architecture

With its great advantages, microservice architecture also has to endures a great number of disadvantages and challenges.

3.3.1 Challenges for local environment set-up

Firstly, running all services locally is challenging for an engineer. For developers, it will get a lot tougher. In the event that a developer wishes to work on a journey or function that could encompass many services, the developer must run them all on one local device. This is much more difficult than running a single program easily. The automated tool will partly overcome this problem, but as the number of resources that make up a system grows, developers will encounter more problems while using an operating system. [5]

In a more complex applications, there are several dedicated teams just for maintaining a service. In the future, these teams will face an increasing number of difficulties. They

handle dozens, hundreds, or thousands of running services instead of, maybe, a few. There are more programs, more paths of contact, and more possible failure zones. [5.]

3.3.2 Unnecessarily complex DevOps operation

In a complex application system, which serves hundreds of business transaction. It would be beneficial if the operation and the development are handled separately, particularly given the popularity of DevOps as an activity (which I am a big proponent of). Does DevOps not mitigate this? The problem is that many companies also have different development and operational departments, and a company that does so is much more likely to fail with microservice adoption. It is also complicated for organizations which have embraced DevOps. It is already difficult (but vital to create good software) to be both a developer and an operator, but it is also very difficult to understand the dynamics of container orchestration systems, especially systems that grow at a rapid pace. [5.]

3.3.3 Expertise problems

When a microservice is achieved by professionals, the effects will be wonderful. However, it is important to consider an enterprise where, with a single monolithic structure, things cannot yet run smoothly. Increasing the number of processes, which increases the operating complexity, the likelihood of things getting worse is high. However, this is all possible with robust automation, control, orchestration and so on. The challenge, though, is rarely technology - seeking people who can use it successfully is the challenge. Currently, these talents are in extremely high demand and can be difficult to locate. [5.]

3.3.4 Strictness in following rules

Adopting a microservice is all about independency. However, in real world example, independency is often very difficult to achieve. This is where stuff can get incredibly complicated. If borders are currently not well established, what happens is that while services can technically be implemented in isolation, engineers feel that the only way to go is to deploy sets of services as a collective due to the inter-dependencies between services.

3.3.5 Overlooking the complexity of service communications

If a microservice system uses message queues for intra-service correspondence, the systems are basically glued together by a large database (the message queue or broker). Once again, while at first it does not seem like an obstacle, the scheme will eventually get complicated and hard to handle. Services that rely on this message may often need to be changed as the receiving service updates the specifics of the message it sends.

Services should be provided that could accommodate communications in several different formats, although this is difficult to maintain. There are occasions in which two separate versions of a service exist, attempting to process messages from the same queue when deploying new versions of services, or even messages sent by multiple versions of a sending service. This will relate to dynamic cases of the tip. It could be better to allow only some iterations of messaging to exist to prevent these edge cases, which ensures that deploying a series of versions of a set of services as a cohesive whole is required, meaning that messages from older versions are first drained properly. [5.]

3.3.6 Challenging versioning process

Versioning has to be very closely handled to minimize the problems described earlier. This can be very confusing very easily and can get to the point where development team or DevOps team do not understand which iterations of services are really going to work together properly.

It is extremely tough to control dependencies in software systems, whether it be node modules, Java modules, or C libraries. It is very difficult to manage the complexities of disputes between independent elements when being ingested by a single entity.

3.3.7 Giant monolith service

Individual resources and modules can be implemented in isolation, but it is necessary in most instances to run any kind of orchestration platform, such as Kubernetes. If service management is used, then a great deal of the sophistication of cluster management is managed automatically.

Metropolia
University of Applied Sciences

The net advantages are still present in certain situations, but it is important not to trivialize or neglect the extra difficulty of handling another massive, complicated system. Managed services may help, but these services are immature in many cases (Amazon EKS was only announced at the end of 2017). [5.]

3.3.8 Networking

There is a reasonably straightforward networking configuration with a more conventional model of services operating on known servers, with known addresses.

However, in a microservice, there will normally be multiple resources spread through many nodes by using microservices, which generally means there will be a much more dynamic networking structure. Load balancing between services will exist, DNS could be more commonly used to try to 'mask' the difficulty of this networking.

3.4 Micro-frontend applications

Although a microservice is often discussed within the server implementation framework, a front-end application can also utilize this type of approach, in other words so called, micro frontends.

In late 2016, ThoughtWorks Technology Radar first came up with the word Micro Frontends. It applies to the front-end ecosystem and the principles of microservices. The new trend is to create a feature-rich and efficient browser framework that sits on top of a microservice architecture, a single page application. Whereas a microservice architecture in a server can quickly be adopted, the frontend monolith architecture is mostly used by a variety of applications and it can get overwhelmed when the codebase becomes larger and more complex. Below figure shows the example of how a normal monolith frontends application would look like for an e-commerce site.[6.]

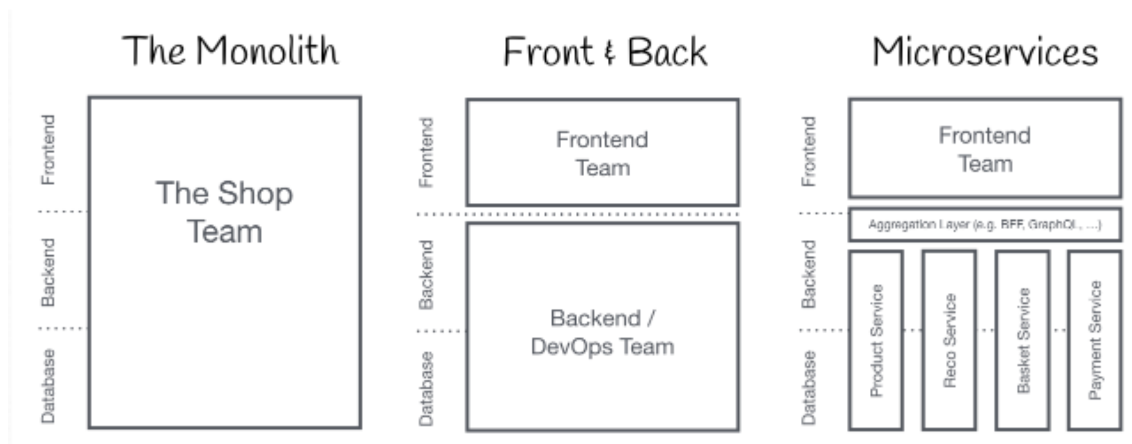


Figure 10. Monolith Frontends. Copied from Geers (2017) [6].

Usually, the micro frontends architecture divides the user interface into several pages or components, given that each page or each component is complex enough so that one dedicated team can develop it. For the example that is shown in below figures, there can be one search team, one checkout team, one product page team and one user page team in a complex e-commerce application. [7.]

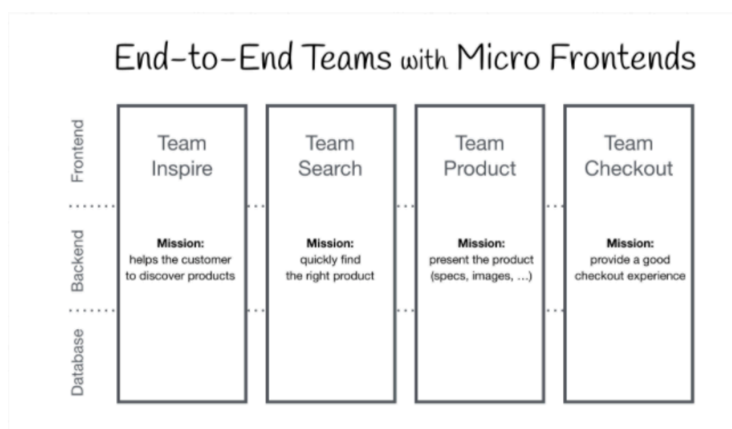


Figure 11. End-to-end teams with Micro Frontends. Copied from Geers (2017) [6].

3.4.1 Advantages of micro-frontend

A micro frontends architecture has many of the same benefits as a normal microservice. The biggest strength of a micro frontend application lies within its easiness to do incremental upgrades. This is usually the beginning point of a micro-frontend journey with many organizations. The old, big, frontend monolith is kept back by code written under

deployment pressure, and it is getting to the point where it is tempting to make a complete rewrite.

Another benefit of the micro frontend is that the source code would be much smaller than the source code of a single monolithic frontend. The smaller codebases appear to be simpler and easier to deal with. In particular, the difficulty resulting from unintended and improper coupling between components that should not be conscious of each other is avoided. We find it more difficult for such unintended pairing to occur by drawing tighter lines along with the boundary contexts of the program. [7.]

Independent deploy ability of micro frontends is important, as with microservices. This reduces the complexity of any deployment issued, which in turn reduces the risk associated with it. Each micro frontend should have its own continuous delivery pipeline, regardless of how or where the frontend code is hosted, which builds, checks and deploys it all the way to output. With very little consideration given to the current state of other codebases or pipelines, engineer teams should be capable of deploying each micro frontend. It does not matter if the old monolith is in a quarterly release cycle or if the next team has forced a semi-finished or broken function into the master branch of the application. It should be able to do so if a given micro frontend is ready to go to production, and that determination should be up to the team that develops it and manages it.

As a higher-order advantage in decoupling both codebases and delivery cycles, there is still a long way to go from imagination to production and beyond in having fully autonomous departments that can own a software segment. In order to achieve this, our teams would concentrate on lateral portions of company functionality but not on technological capability. A simple way to do that is to design the product depending on the customers' insight, so that each micro frontend service encapsulates a single page and is owned by a single team end-to-end. This gives the teams a greater synergy in comparison to teams that are built around technological or 'horizontal' issues, such as styling, formatting or validation. [7.]

3.4.2 Disadvantages of micro-frontend

With many benefits stated above, the micro frontends still has many drawbacks that prevent most engineer teams to fully adopt microservice in the user interface side.

Metropolia
University of Applied Sciences

First and foremost, the payload size in a micro-frontend application is huge because of duplication. Independent JavaScript packages can cause common dependencies to be duplicated, raising the number of bytes sending to clients over the network. For starters, if each micro frontend contains its own copy of React, then customers need to download many times of React. There is a clear correlation between page output and user engagement/conversion. [7.]

Environment differences in the micro frontend architecture are more pronounced than that in the server microservice. There are a lot of issues related to developing in an environment that is different from production environment. If the development-time container behaves differently than the production one, then deploying to production might be problematic as there might be an unexpected error that leads to an application behaving differently. Global styling is one major bottleneck that needs to be well-thought between each micro frontend services environment [7].

The last drawback is related to operation complexity. Micro frontends, being a more modular architecture, would eventually lead to more issues to deal with, such as more repositories, more resources, more pipelines to build/deploy, more servers, and more territories. [7.]

3.5 Dockers and Kubernetes for microservices

3.5.1 What is container?

In conventional software development, code is built in one computing environment like a local machine, and it often runs with bugs and errors when on top of other environments like other local machines or server environment.

In modern software architecture, the term “container” is usually used by most engineers. Basically, a container is a common software unit that packages code and all its dependencies, so that the program runs from one operating environment to another easily and efficiently. [8.]

Metropolia
University of Applied Sciences

Containerization means bundling an application along with all the associated configuration files, libraries and dependencies needed for it to run in an efficient and bug-free manner across multiple computing environments. [8.]

The two popular containerization environments are Kubernetes and Docker.

3.5.2 Why Docker?

Docker is software containerization platform. Its job is to bundle each microservice into a "Docker container" that can be run and deployed independently in all type of environmental, ranging from a local machine to a server. [8.]

Taking a simple microservice e-commerce application into consideration, the usage of Docker is illustrated in Figure 12 below.

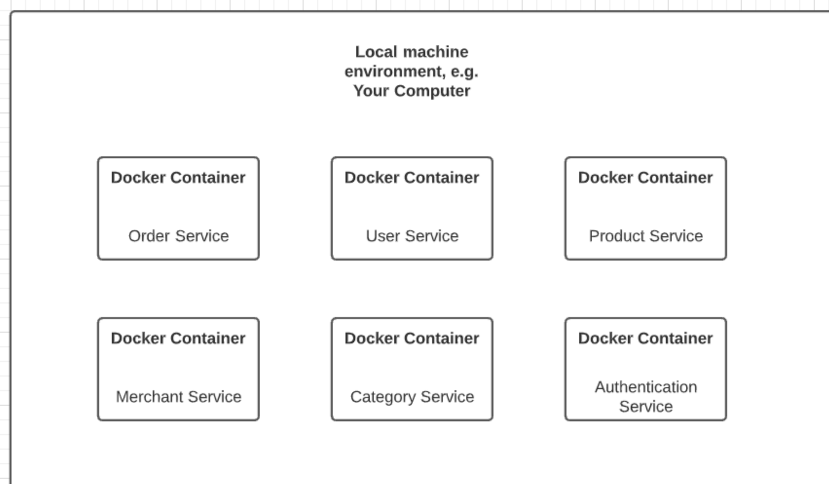


Figure 12. Example of usage of Docker containers in simple microservice e-commerce application

The conventional way of running the application demonstrated in Figure 12 would be that the correct environment for each service needs to be properly installed and precise knowledge of how to start each service is required. Therefore, there are several assumptions made in order to run the whole application locally: the environment for each service

Metropolia
University of Applied Sciences

is working properly and coherently with each other and each service is initialized perfectly.

In a real-world situation, both of the assumptions above are rarely achieved as different services might need different versions of the runtime environment (for example, a different Node version). Docker solves both of the issues by wrapping up everything that is needed for a program regardless of any type of runtime environment or language and all the initial start-up processes. [8.]

Some of the advantages of adopting Docker in a microservice application are listed below:

- The Docker community is huge.
- Docker is extremely lightweight compared to running each Virtual Machine for each service.
- Docker works well with Kubernetes (which helps continuous integration and continuous deployment).
- Docker is supported by most modern cloud platform tools such as Amazon Web Service, Google Cloud Platform, Microsoft Azure, Ansible and Kubernetes.
- Docker unifies all environment into one single Docker container; therefore, the risk of failure coming from environment differences is reduced.

[8.]

3.5.3 Why Kubernetes?

In the previous section, Docker was shown to be the only approach to run a microservice application. However, it this will lead to another problem: How to manage different Docker containers and run them according to different needs? This is where Kubernetes comes in handy.

Kubernetes is a tool for running a bunch of different containers (or Docker containers). Based on some given configuration, Kubernetes decides what container to run and interact with. [14.]

Example usage of Kubernetes in the previous e-commerce application is shown in the below figure.

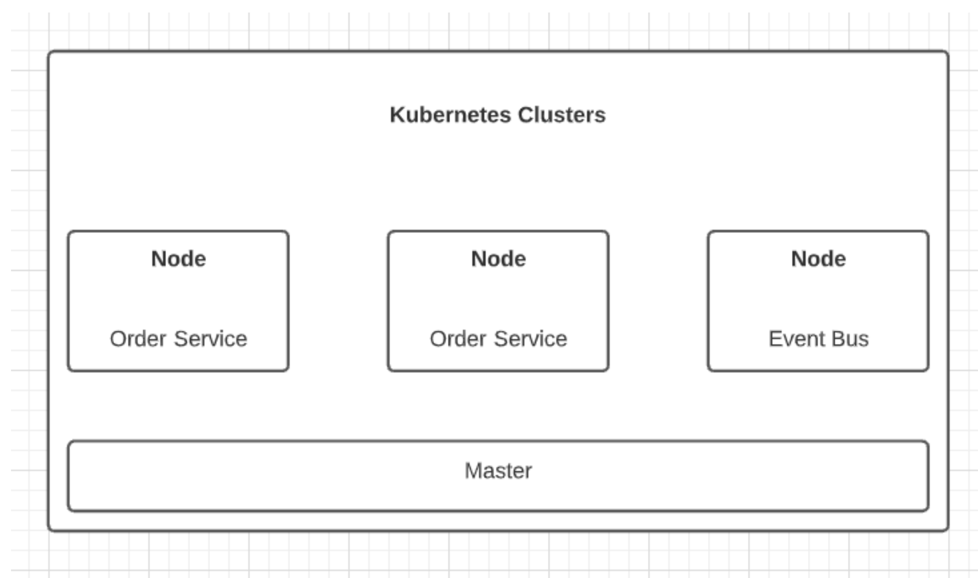


Figure 13. Example of Kubernetes Cluster running a basic e-commerce application

Figure 13 shows that Kubernetes runs everything in a cluster, which is called “Kubernetes Cluster”. The cluster is managed by a configuration defined by the developer and then run in “Master”. Each “Node” represents each virtual machine entity, in this case, the Docker container. The above Kubernetes cluster has been configured so that it should run two copies of Order Service and it should be accessible from the network. Therefore, the Kubernetes cluster takes care of all the communication between Event Bus and Order Service. The developer should not do any direct requests from Order Service to Event Bus and let Kubernetes act as a medium.

However, the complexity of using Kubernetes should be taken into consideration prior to using it. For a simple application, the usage of Kubernetes can be unnecessary and reduce productivity. Since most ongoing projects do not start Kubernetes from the beginning, making a transition from other tools to Kubernetes can be cumbersome and require excessive resources.[14]

3.6 Monorepo for microservices

3.6.1 Monorepo and Polyrepo

The usage of the version control system (VCS) is often well-discussed between engineer teams in order to find out what works best for each particular project. In terms of modern applications, the topic of discussion is mostly whether one team should use the monorepo approach or the polyrepo approach.

The polyrepo approach means that each service or application within an organization has its own repository. For example, a microservice e-commerce application can have an order service repository, a user service repository, a frontend repository, and a mobile repository. Therefore, an organization could end up having multiple repositories.

Vice versa, the monorepo approach represents a version control system in which all codebases for all services are kept under one single repository.

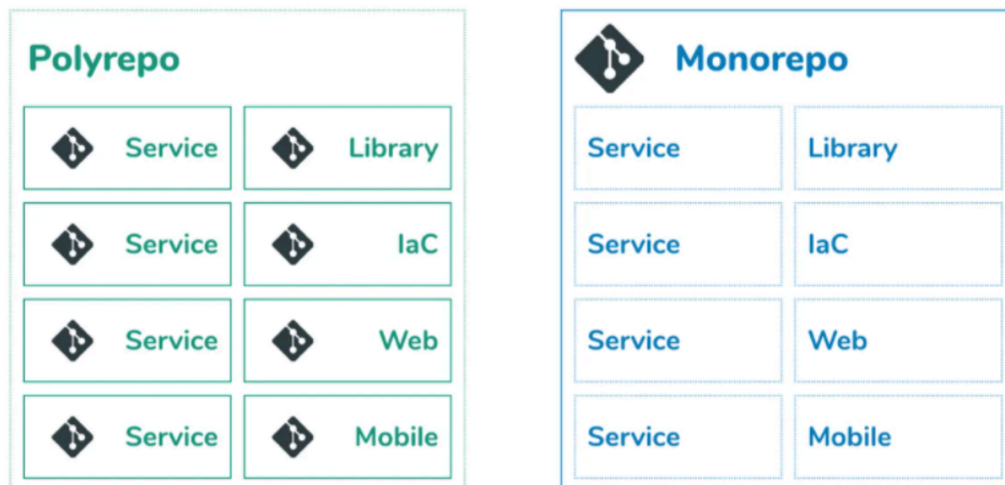


Figure 14. An illustration of Monorepo and Polyrepo. Copied from Shanaghy (2019) [9]

Both approaches should have their own advantages and disadvantages which need to be considered in detail.

Metropolia
University of Applied Sciences

3.6.2 Monorepo usage in microservice application

At first glance, polyrepo seems to be an easy and understandable approach to a microservice application as there are different independent services. It has many benefits with this approach, including team development, versioning, and less conflict between engineering teams. However, if a microservice application is maintained by a smaller team, this approach might be cumbersome. Especially the serverless architecture is widely popular at the moment. [9.]

Monorepo is extremely beneficial when it comes to bug fixing and testing across multiple services. All changes are made within one single repository, and a simple CI/CD script detecting what services are affected by the changes can make deployment extremely fast. This would help reducing the risk of having multiple pull requests and ending up having a partially merged bug fix. [8]

Another great advantage lies within its quick onboarding process for a new team member. Getting all the product code in one repository is a way to help individuals get started quickly, whether there are new colleagues or a new local environment.

3.7 Implementing a simple microservice application

3.7.1 Setting up

This project was going to be written in NodeJS and ReactJS which is a basic application illustrating the usage of microservices.

The application includes one frontend application and several backend services. The goal of this application is to create a simple blog listing application that should have a posting feature, a comment feature, and a modification feature to decide whether comments meet the standard requirements. Figure 15 shows the final results of the project.

Create Post

Title

Submit

Posts

Hoang Vo Thesis

- This is the first simple application for chapter 3.7 in Thesis

New Comment

Submit

Hoang Thesis 2

- This is the second post

New Comment

Submit

Hoang Thesis 3

New Comment

Submit

Figure 15. Final result for a simple blogging application

In order to create this application using the microservice approach, the project should contain a separate client service which has the frontend codebase, the posts service, the comments service, the moderation service, the query service and the event-bus service.

As the project is simple and straightforward, the usage of Dockers, Kubernetes and any Database Management System (DBMS) was excluded to avoid unnecessary complex transactions.

All backend services have the same “package.json” configuration as their job is mainly to send requests to event-bus.

```json
{
  "name": "query",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  ▷ Debug
  "scripts": {
    "start": "nodemon index.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "axios": "^0.19.2",
    "cors": "^2.8.5",
    "express": "^4.17.1",
    "nodemon": "^2.0.2"
  }
}
```

Listing 1. List of all dependencies used in backend services

As shown in Listing 1, the backend framework used in this project was ExpressJS version 4.17.1 which runs on the NodeJS environment. The “axios” library is for making an asynchronous request, “CORS” is for allowing a request made across different services, and “nodemon” is for re-running the service every time code has changed.

3.7.2 Basic data flow and implementation

Each service had its own database, and every transaction was handled via Event Bus. By using this approach, the integrity of all the transactions was secured. Figure 16 demonstrates how the application was built.

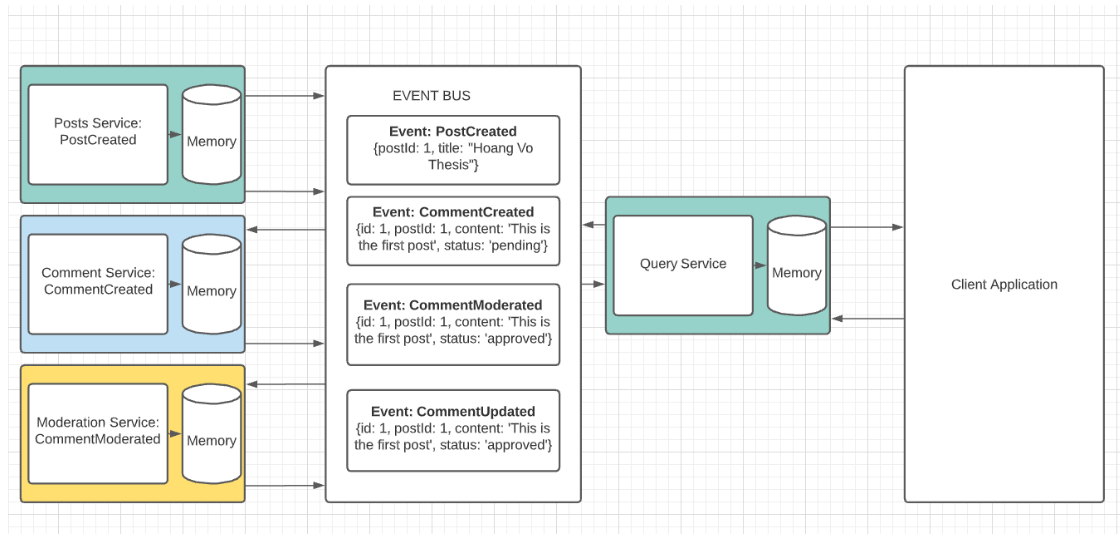


Figure 16. Data flow in blogging application

Figure 16 shows all six services of this application. Client applications will communicate purely with the Query Service to get all the needed information. the Query Service will listen to all the required events in Event Bus and store that information in its memory so that it can return the result to Client as soon as some requests come. The Post Service handles all the Post related action, which is PostCreated. In the same sense, Comment-Service handles all the Comment related action, including CommentCreated and CommentUpdated. It also listens to the "CommentModerated" event from Event Bus to react accordingly. Moderation Services will particularly listen to the CommentCreated event and decide whether comments are "approved" or "rejected". Then it emits an event called "CommentUpdated" to the Event Bus.

The codebase for the Post Service is demonstrated in Listing 3 below.

```
const express = require('express');
const bodyParser = require('body-parser');
const { randomBytes } = require('crypto');
const cors = require('cors');
const axios = require('axios');

const app = express();
app.use(bodyParser.json());
app.use(cors());

const posts = {};

app.get('/posts', (req, res) => {
  res.send(posts);
});

app.post('/posts', async (req, res) => {
  const id = randomBytes(4).toString('hex');
  const { title } = req.body;

  posts[id] = {
    id,
    title
  };

  await axios.post('http://localhost:4005/events', {
    type: 'PostCreated',
    data: {
      id,
      title
    }
  });

  res.status(201).send(posts[id]);
});

app.post('/events', (req, res) => {
  console.log('Received Event', req.body.type);

  res.send({});
});

app.listen(4000, () => {
  console.log('Listening on 4000');
});
```

Listing 2. PostService implementation

As shown in Listing 2, the PostService's job is to take a request from a client whenever the user makes a post. Then it stores that information in its memory and emits an event to EventBus to announce all Event Bus subscribers that "PostCreated" has been dispatched. After that, it returns a success response to the client.

Metropolia
University of Applied Sciences

```
const express = require('express');
const bodyParser = require('body-parser');
const { randomBytes } = require('crypto');
const cors = require('cors');
const axios = require('axios');

const app = express();
app.use(bodyParser.json());
app.use(cors());

const commentsByPostId = {};

app.get('/posts/:id/comments', (req, res) => {
  res.send(commentsByPostId[req.params.id] || []);
});

app.post('/posts/:id/comments', async (req, res) => {
  const commentId = randomBytes(4).toString('hex');
  const { content } = req.body;

  const comments = commentsByPostId[req.params.id] || [];

  comments.push({ id: commentId, content, status: 'pending' });

  commentsByPostId[req.params.id] = comments;

  await axios.post('http://localhost:4005/events', {
    type: 'CommentCreated',
    data: {
      id: commentId,
      content,
      postId: req.params.id,
      status: 'pending'
    }
  });

  res.status(201).send(comments);
});

app.post('/events', async (req, res) => {
  console.log('Event Received:', req.body.type);

  const { type, data } = req.body;

  if (type === 'CommentModerated') {
    const { postId, id, status, content } = data;
    const comments = commentsByPostId[postId];

    const comment = comments.find(comment => {
      return comment.id === id;
    });
    comment.status = status;

    await axios.post('http://localhost:4005/events', {
      type: 'CommentUpdated',
      data: {
        id,
        status,
        postId,
        content
      }
    });
  }

  res.send({});
});

app.listen(4001, () => {
  console.log('Listening on 4001');
});
```

Listing 3. CommentService implementation

Metropolia University of Applied Sciences

The implementation of CommentService is much more complicated compared to PostService. CommentService needs to do two jobs asynchronously. Its first job is to take a comment request from the client whenever the customer makes a comment as shown in line 17. After comments are created, it dispatches an event called “CommentCreated” to EventBus. Its second job is to listen out for an event called “CommentModerated” from Event Bus, shown in line 40. This event happens after ModerationService has either rejected/approved a “comment”. Then it dispatches the result to EventBus with the event “CommentUpdated”.

```
1   const express = require('express');
2   const bodyParser = require('body-parser');
3   const axios = require('axios');
4
5   const app = express();
6   app.use(bodyParser.json());
7
8   app.post('/events', async (req, res) => {
9     const { type, data } = req.body;
10
11    if (type === 'CommentCreated') {
12      const status = data.content.includes('orange') ? 'rejected' : 'approved';
13
14      await axios.post('http://localhost:4005/events', {
15        type: 'CommentModerated',
16        data: {
17          id: data.id,
18          postId: data.postId,
19          status,
20          content: data.content
21        }
22      });
23    }
24
25    res.send({});
26  });
27
28  app.listen(4003, () => {
29    console.log('Listening on 4003');
30  });
31
```

Listing 4. ModerationService implementation

The job of the Moderation service is straightforward. As shown in line 8, it listens to an event called “CommentCreated” from Event Bus and decides whether the comment is approved/rejected based on the appearance of the word “orange”. Then it dispatches an event called “CommentModerated”.

```
const express = require('express');
const bodyParser = require('body-parser');
const cors = require('cors');
const axios = require('axios');

const app = express();
app.use(bodyParser.json());
app.use(cors());

const posts = {};

const handleEvent = (type, data) => {
  if (type === 'PostCreated') {
    const { id, title } = data;

    posts[id] = { id, title, comments: [] };
  }

  if (type === 'CommentCreated') {
    const { id, content, postId, status } = data;

    const post = posts[postId];
    post.comments.push({ id, content, status });
  }

  if (type === 'CommentUpdated') {
    const { id, content, postId, status } = data;

    const post = posts[postId];
    const comment = post.comments.find(comment => {
      return comment.id === id;
    });

    comment.status = status;
    comment.content = content;
  }
};

app.get('/posts', (req, res) => {
  res.send(posts);
});

app.post('/events', (req, res) => {
  const { type, data } = req.body;

  handleEvent(type, data);

  res.send({});
});

app.listen(4002, async () => {
  console.log('Listening on 4002');

  const res = await axios.get('http://localhost:4005/events');

  for (let event of res.data) {
    console.log('Processing event:', event.type);

    handleEvent(event.type, event.data);
  }
});

```

Listing 5. QueryService implementation

As shown in Listing 5, that the job of QueryService is to only listen to some events in EventBus and store the information in its memory. Every time it receives a request from

the client to fetch some information, it reaches out to memory and gets that information, returning the data to clients.

To sum up, the above example shows how the microservice works in a nutshell and how to effectively use Event Bus for asynchronous communication

4 When to use monolith approach or microservice approach?

Prior to any particular greenfield projects, all members of an engineering team usually discuss whether they should go for a monolith or microservice approach. This chapter includes some helpful tips.

4.1 Team size

Team size is the biggest factor when it comes to considering adopting a microservice. If the team size is small enough, the monolith approach might be a better solution as handling different services with a small team can be a difficult task. [2.]

4.2 State

If the application is mostly stateless, then neither microservice nor monolith is a good option. Serverless might be a good answer in case of stateless application. However, if the application is stateful, then microservice is a good fit with considering it might get complicated quickly. [5.]

4.3 Dependency

If the system has a monolith dependency, meaning most services rely on one or two particular systems, then the monolith architecture might be a good approach. Adopting a microservice in this particular case might result in the service crossing its boundaries and creating direct dependency between microservices. However, if the system has no monolith dependency, then it is the best scenario, allowing the microservice to thrive.

4.4 Expertise

Adopting the microservice often requires DevOps expertise to manage all the deployment pipelines. A bad DevOps pipeline can lead to a disaster, which could mean that all services are not manageable. If the there is no DevOps expertise in the team, it could be helpful to try a microservice first before going all in.[4.]

5 Introduction to gRPC

As in many RPC programs, gRPC is based on the concept of defining a service, determining the methods with their parameters and returning types that can be called remotely. The server implements this interface on the server side and runs a gRPC server to deal with client calls. The client has a stub on the client side (referred to in some languages as just a client) that provides the same strategies as the server. [10.]

In a number of environments - from servers within Google to local machine environment- gRPC clients and servers can run and speak to each other and can be written in any of the licensed languages of gRPC. Therefore, developers can quickly build a gRPC server in Java with clients in Go, Python, or Ruby, for instance. In addition, the new Google APIs will provide their interfaces in gRPC formats, enabling engineers to quickly integrate Google features into applications. [10.]

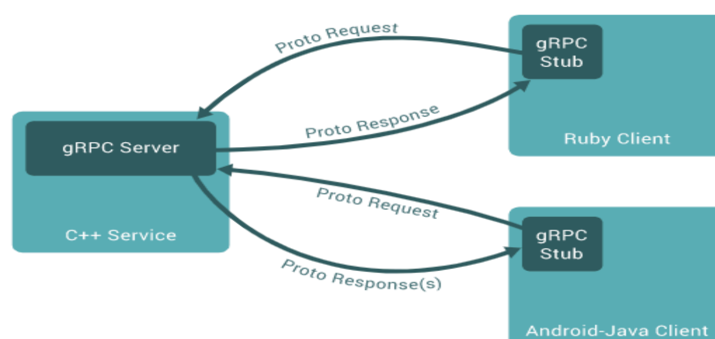


Figure 17. gRPC flow. Copied from gRPC [10].

5.1 What is protocol buffer?

Protocol buffer is a cornerstone of every gRPC applications. It is an open-source technique to serialize structured data. [10] In a different manner, it is an alternative to JSON or XML for serializing data transfer. The language of the protocol buffer language called Protobuf.

Protobuf is language neutral, meaning all languages can use Protobuf to transfer data. Unlike JSON or XML, Protobuf is optimized and run in binary format. The configured data is written in files called ".proto", and the configurations are called "messages". [10.]

```
example.proto ×
1   syntax = "proto3";
2
3   message Greeting {
4     string first_name = 1;
5   }
6
7   message GreetRequest {
8     Greeting greeting = 1;
9   }
10
11  message GreetResponse {
12    string result = 1;
13  }
14
15  service GreetService {
16    rpc Greet(GreetRequest) returns (GreetResponse) {};
17  }
```

Listing 6. Example of one protocol buffer file

There are two main benefits of using Protobuf compared to JSON/XML. The first is its extremely fast performance as the Protobuf transfer compresses data in binary format. Protobuf also provides a method to separate context and data, meaning data can be repeatedly sent without duplicating a context (or field name/ property name) like in JSON or XML. The only downside of using Protobuf is that the transferred data is not human-readable data as in JSON or XML. [10.]

5.2 What is HTTP/2?

HTTP/2 is the newer standard for internet communications that directly addresses several drawbacks of the traditional HTTP/1.1. It is now mostly compatible on all modern browsers. [12.]

5.2.1 Disadvantages of HTTP/1.1

The HTTP/1.1 protocol was originally released in 1977 and it was adopted by the World Wide Web. For each request from the client, HTTP/1.1 opens up a new TCP connection to a server. Modern browsers have several workarounds with this limitation by allowing multiple parallel TCP connections, but it has not been efficient enough for large applications. [12.]

HTTP/1.1 does not compress headers and it only works with request /response mechanism. Modern web applications load up to a hundred assets per page on average, each asset carrying a heavy plaintext header. [12.]

All of these inefficiencies add latency and increase network packet size to both the client and the HTTP/1.1 server. [12.]

5.2.2 Advantage of HTTP/2

HTTP/2 was published in 2015 and it provided a complete solution for all the drawbacks of HTTP/1.1 [12].

Multiplexing is a technique through which HTTP/2 allows multiple requests between the client and server for each TCP connection. Furthermore, HTTP/2 introduces HPACK to compress large plaintext headers, and it also provides a mechanism called server push. With the HTTP/2 protocol, servers can push streams of multiple messages for one request from the client; therefore, servers do not need to wait for the client's request. [12.]

While HTTP/1.1 text makes it easy to debug, it is not efficient over the network. HTTP/2 utilizes this disadvantage and uses a binary system instead, which also makes it a great match with the protocol buffer as the protocol buffer is a binary protocol.

5.3 gRPC in a nutshell

5.3.1 Different types of gRPC APIs

There are four types of gRPC APIs: Unary API, Server Streaming API, Client Streaming API and Bi-Directional Streaming API [10].

Unary RPC calls are the basic Request / Response that have the same design as the REST API. The client sends one message to the server and receives one response from the server. A Unary API call is suited when the data that needs to be transferred is small and concrete. It is advisable to start with the Unary API and switch to other streaming APIs when performance is an issue. [13.]

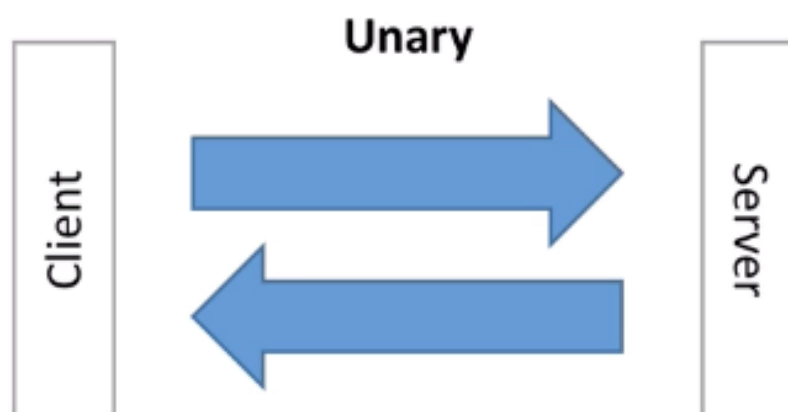


Figure 18. Unary API call architecture. Data gathered from gRPC. [13]

The Server streaming RPI API is the new kind of API enabled by the introduction of HTTP/2. The client sends one message to the server and receives an infinite number of responses from the server until either party stops the action. This API is suited when the server needs to send a large chunk of data or in a situation in which the server needs to push data to the client without having the client requests, e.g., live chat and feeds. [13.]

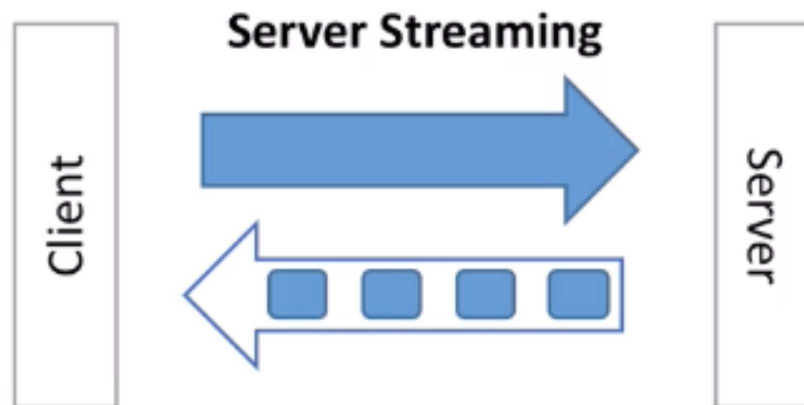


Figure 19. Sever Streaming API architecture. Data gathered from gRPC [13].

The Client streaming RPC API works the other way round compared to the Server streaming RPC API. The client sends an infinite number of messages to the server and receives one response from the server at any time. There is no guarantee that the server sends a response after it receives all the messages from client side. Server decides whenever it needs to send something. This API should be used when the client needs to send a large amount of data to the servers without expecting any response. It can also be used when server processing is expensive and when responding to every request from the client is unnecessary. [13.]

Metropolia
University of Applied Sciences

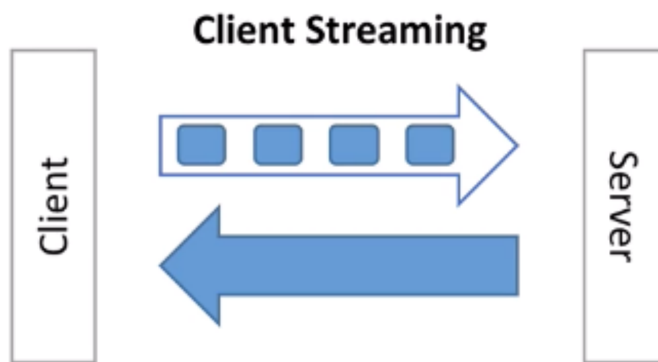


Figure 20. Client streaming API architecture. Data gathered from gRPC [13].

The last RPC API is a Bi-Directional RPC API. The client sends many messages to the server, receiving many responses from the server. The number of messages does not need to match. Both the client side and the server side decide whenever a message needs to be sent. [13.] This type of API is suited when both sides need to send big data asynchronously.

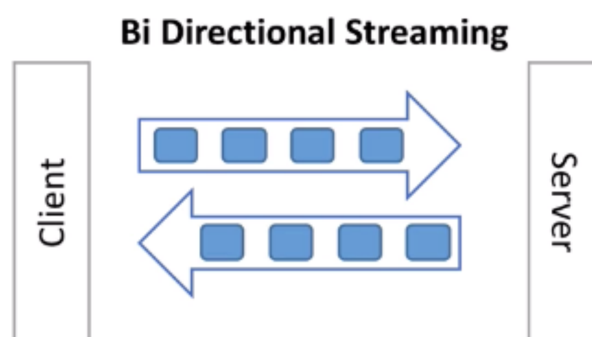


Figure 21. Bi-Directional Streaming RPC API architecture. Data gathered from gRPC [13].

5.3.2 Scalability in gRPC

gRPC, on top of being an effective technique performance-wise, is also extremely scalable thanks to the default asynchronous characteristic in the gRPC server. In a nutshell, it means that there would not be any request that is halted, and that each server can serve millions of requests in parallel.

Furthermore, gRPC clients can be asynchronous or synchronous which depends on each system's configuration to optimize the performance and the scalability. gRPC clients can also perform load balancing directly. [13.] Hence, systems can scale horizontally.

5.3.3 Security in gRPC (SSL)

By default, gRPC strongly advocates for the user to use SSL (encryption over the wire) in the API [10]. Hence, encryption and security will be a first priority . All communications are not intercepted between the gRPC server and the client. Each language provides an API to load gRPC with the required certificates and provides encryption capability out of the box.

In general, gRPC is built on the basis of security from the ground up.

5.4 gRPC versus REST

For a long time, the REST API has been a cornerstone in web programming. However, gRPC has recently begun to encroach on its domain. A GitHub user called husobee stated in his blog that the gRPC API is 25 times faster than the same approach using the REST API [11]. There are some really legitimate explanations for this.

5.4.1 Protobuf versus JSON

The payload format is one of the most important variations between REST and gRPC. JSON is commonly used in REST messages. While this is not a strict rule, the whole REST ecosystem, including tooling, best practices, and tutorials, is based on JSON.

gRPC, on the other hand, embraces Protobuf messages. Protobuf is a very effective and packed format in terms of efficiency whereas JSON is a textual file. JSON can be compressed, but it then misses the main benefit, which lies in its easiness to debug readable text-like format.

5.4.2 HTTP/2 versus HTTP/1.1

In section 5.2.1 and 5.2.2, the advantages and disadvantages of HTTP/2 and HTTP/1.1 are discussed. gRPC is built on top of HTTP/2 and REST is built on top of HTTP/1.1, which shows the clear advantages in favor of gRPC in terms of performance and security.

5.4.3 Streaming versus Request-Response

REST only supports the request and response model, and it does not utilize the full ability of HTTP 1.x; however, gRPC takes full advantage of HTTP/2 and allows streaming from both the server and client.

5.4.4 API oriented versus CRUD oriented

REST is based on CRUD resources, so the REST model design is based on the create, retrieve, update and delete actions while gRPC offers a free design with no constraints. The API focuses on its main purpose without playing around with the predefined CRUD model.

6 Building large-scale application using microservice and gRPC

6.1 Project overview

This project focuses on building a complex application that requires the microservices architecture. The application is a basic task management application, in which it has four completely different services. Each service uses its own technology stacks, programming languages and database system. gRPC is used as a transportation layer instead

of traditional HTTP/1 and REST API to demonstrate the capability of gRPC in a real-world scenario. Each service also has its own CI/CD pipelines and a development environment that uses Docker and Kubernetes. Table 1 below shows the descriptions and technology used in each service in the applications.

Table 1. Description and technology of each service.

Service	Description	Technology
Project Service	This service handles project creation and project modification.	Python, MySQL
Task Service	This service handles task creation, task modification.	Ruby, PostgreSQL
User Service	This service handles all user management actions.	NodeJS, MongoDB
API Service	It handles all API routing in the applications.	Golang

6.2 Project's architecture

This project is mainly focus on the basic usage of gRPC and microservices architecture. Therefore, many unnecessaries usage of technologies are used to demonstrate the communication between each service. The below figures show the basic architecture of this project.

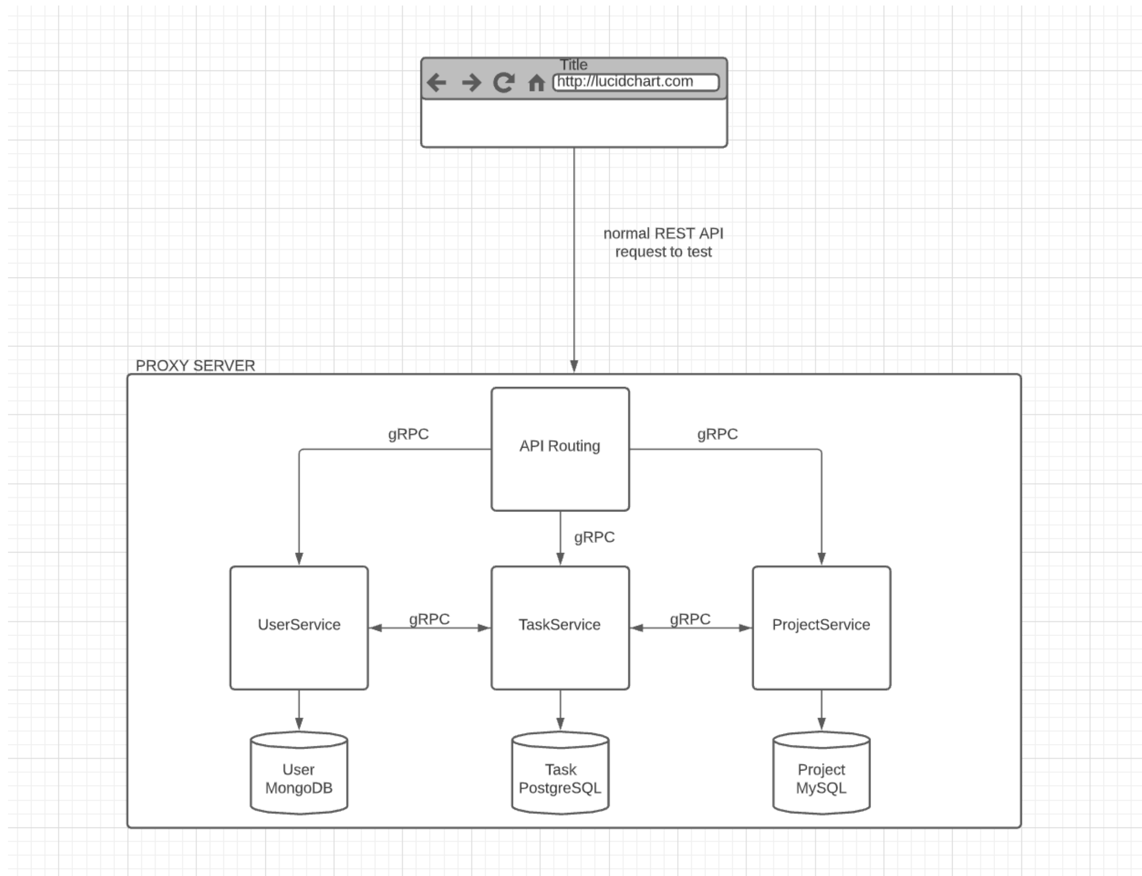


Figure 22. Project overall architecture

The above figure shows how the project was carried out. Three services are communicating via gRPC between each other. Each service has its own development environment. In order to test the service, the whole project is wrapped into one proxy server, which is served as a gateway for all types of client-side application requests. The figure below shows one part of the Docker-compose file.

docker-compose.yml

```
version: '3.3'

services:
    user:
        build:
            context: ./userService
        image: microservice/demo/user
        restart: "no"
        environment:
            - DB_URI=mongodb://mongo:27017/
            - DB_NAME=Microservice-demo-user
        ports:
            - 8080:50051
        depends_on:
            - mongo

    project:
        build:
            context: ./projectService
        image: microservice/demo/project
        restart: "no"
        environment:
            - DB_URI=mysql+mysqldb://root:password@mysql:3306/microservice-project
        ports:
            - 8081:50052
        depends_on:
            - mysql
            - user

    task:
        build:
            context: ./taskService
        image: microservice/demo/task
        restart: "no"
        entrypoint: [./init]
        environment:
            - USER_ADDRESS=user:50051
            - PROJECT_ADDRESS=project:50052
            - DB_URI=postgres://postgres:password@postgresql:5432/microservice-task
        ports:
            - 8082:50053
        depends_on:
            - postgresql
            - user
            - project

    api:
        build:
            context: ./apiService
        image: microservice/demo/api
        restart: "no"
        environment:
            - USER_ADDRESS=user:50051
            - PROJECT_ADDRESS=project:50052
            - TASK_ADDRESS=task:50053
        ports:
            - 8083:50059
        command: serve
        depends_on:
            - user
            - project
            - task

```

Figure 23. Project Docker build process

In the above figure, there is one docker-compose file to run all the development for each service. The api-gateway service is used to create a proxy layer to route all services

together and serve as one unit for testing purposes or for the REST API client-side application. The below figure shows the end result of the build process.

```
AWS: finnair-ifec  hoangvo@Hoangs-MBP  ~  docker ps
CONTAINER ID   IMAGE                      COMMAND                  CREATED          STATUS          PORTS                               NAMES
a47c5cbde353   microservice/demo/api      "apiService proxy"       55 minutes ago   Up 50 minutes   0.0.0.0:9090->9090/tcp, 50059/tcp   task-app-microservices_api-gateway_1
58ccbde543d8   microservice/demo/api      "apiService serve"       55 minutes ago   Up 50 minutes   0.0.0.0:8083->50059/tcp             task-app-microservices_api_1
de8c6c2aeaa4   microservice/demo/task     "./init"                 55 minutes ago   Up 50 minutes   0.0.0.0:8082->50053/tcp             task-app-microservices_task_1
0fc2ee851b83   microservice/demo/project  "python service.py"      55 minutes ago   Up 50 minutes   0.0.0.0:8081->50052/tcp             task-app-microservices_project_1
7a4c66d8cf36   microservice/demo/user     "docker-entrypoint.s…"   55 minutes ago   Up 50 minutes   0.0.0.0:8080->50051/tcp             task-app-microservices_user_1
e7972c424101   mysql:5.7                  "docker-entrypoint.s…"   55 minutes ago   Up 50 minutes   0.0.0.0:3306->3306/tcp, 33060/tcp   task-app-microservices_mysql_1
b7cae271b172   postgres                   "docker-entrypoint.s…"   55 minutes ago   Up 50 minutes   0.0.0.0:5432->5432/tcp              task-app-microservices_postgresql_1
587d32ce40dc   mongo                      "docker-entrypoint.s…"   55 minutes ago   Up 50 minutes   0.0.0.0:27017->27017/tcp            task-app-microservices_mongo_1
```

Figure 24. Project built successfully

7 Conclusion

The goal of this thesis was to demonstrates the effectiveness and limitations of adopting the microservices architecture and gRPC in a real-world scenario. This was achieved by successfully deploying all services for all the projects.

The drawbacks of using the microservices architecture were also visible in the e-commerce projects. As the scope of the project was small with one developer handling all services, adopting microservices was unnecessary and became troublesome in code maintenance.

The front-end part of the gRPC projects were not completed and, therefore, the walkaround of using the REST API and HTTP/1.1 as a routing method was used. It was more difficult to properly test the latency of the gRPC endpoint on top of HTTP/2. This also showed the problems in using gRPC, and an end-to-end ecosystem was required in order for the system with gRPC to run smoothly.

To summarize, the projects were able to cover all the important aspects of the microservices architecture and gRPC, especially the data communication between services. This thesis also gives a brief theory introduction to the microservice architecture and the usage of gRPC and HTTP/2 in modern, complex applications. Even though these modern techniques bring many improvements, abusing them can endanger an entire application. One should carefully consider all perspectives before beginning actual coding.

All in all, microservices and gRPC have proved their capability in improving the performance and speed of applications. The technologies have received a considerable

amount of support from the community and large tech-cooperation, and they will become more mature over time. Therefore, their adoptability will become more straightforward and easier for all teams considering using it.

References

1 Bro, Automation. What Are REST APIs? Online. August 29th, 2020. <https://hack-ernoon.com/a-quick-introduction-to-rest-api-bp1l3uaw>. Accessed February 1st, 2021.

2 Haq, Siraj ul. Introduction to Monolith Architecture and MicroServices Architecture. Online. May 2nd, 2018. <https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63> Accessed February 1st, 2021.

3 Kathpalia, Harish. gRPC – A Modern Framework for Microservices Communication. Online. September 15th, 2020. <https://www.capitalone.com/tech/software-engineering/grpc-framework-for-microservices-communication/> Accessed February 1st, 2021.

4 Hauer, Philipp. Microservices in a Nutshell. Online. April 11th, 2015. <https://phauer.com/2015/microservices-nutshell-pros-cons/ >. Accessed February 1st, 2021.

5 Dwmkerr. The Death of Microservices Madness in 2018. Online. January 12th 2018. <https://dwmkerr.com/the-death-of-microservice-madness-in-2018/ >. Accessed February 1st =, 2021.

6 Geers, Michael. What Are Micro Frontends? Online. 2017. <https://micro-frontends.org/ > . Accessed February 1st, 2021.

7 Jackson, Cam. Micro Frontends. Online. June 19th, 2019. <https://martin-fowler.com/articles/micro-frontends.html#Benefits >. Accessed February 1st, 2021.

8 Syed, Abraar & Rao, Karthic. Docker & the Rise of Microservices. Online. <https://timber.io/blog/docker-and-the-rise-of-micro-services/#:~:text=With%20Docker%2C%20you%20can%20make,ship%20and%20deploy%20your%20application >. Accessed February 1st, 2021.

9 Shanaghy, Eoin. How to End Microservice Pain and Embrace the Monorepo. Online. July 3rd, 2019. <https://www.fourtheorem.com/blog/mon-orepo#:~:text=Polyrepo%20is%20when%20multi-ple%20source,kept%20in%20a%20single%20repository >. Accessed February 1st, 2021.

10 gRPC. Introduction to gRPC. Online. <https://grpc.io/docs/what-is-grpc/introduction/ >. Accessed February 1st, 2021.

11 Husobee. REST v. gRPC. Online. May 28th, 2016. <https://husobee.github.io/go-lang/rest/grpc/2016/05/28/golang-rest-v-grpc.html >. Accessed March 2nd, 2021.

12 Grigorik, Ilya & Surma. Introduction to Http/2. Online. September 3rd, 2019. <https://developers.google.com/web/fundamentals/performance/http2 >. Accessed March 2nd, 2021.

13 gRPC. gRPC Core Concepts. Online. <https://grpc.io/docs/what-is-grpc/core-concepts/ >. Accessed March 10th, 2021.

14 Zero, Enqueue. Kubernetes in a Nutshell. Online. https://enqueuezero.com/kubernetes-in-a-nutshell.html . Accessed March 10th, 2021.