Anh Vu

Real-time backend architecture using Node.js, Express and Google Cloud Platform

Metropolia University of Applied Sciences

Bachelor of Information Technology

Information Technology

Bachelor's Thesis

5 January 2021

Author Title Number of Pages Date	Anh Vu Real-time backend using NodeJS, Express and Google Cloud Platform 47 pages 5 January 2021
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Mobile Solutions
Instructors	Petri Vesikivi, Head of Mobile Solutions

Real-time applications, which assure the latency within the defined time limit, are becoming more popular due to the growth of Software as a service trend. Before the evolution of cloud computing, the only solution was to use native WebSockets which are difficult to set up and develop. Recently, Google Cloud Platform provides a developer-friendly, fast and responsive platform to make the process of developing real-time applications seamless.

The purpose of the thesis was to demonstrate and build a scalable, high-available and reliable backend architecture using Node.js and Google Cloud Platform. The thesis consists of a theoretical background including Node.js, monolithic and microservices architecture, serverless architecture and real-time database, which provide basic understanding of different architectures and technical solutions. The advantages and disadvantages of the architecture were also clearly analyzed and evaluated. Furthermore, a minimum viable product for a taxi booking app was created to demonstrate the architecture usage in a real use case.

To summarize, the thesis aimed to provide the insights of real-time backend architecture using Node.js and Google Cloud Platform. Moreover, the benefits of using this technology stacks were carefully examined in a case study. The thesis completed the first phase of the case study project, which focused on planning and designing the application's overall architecture. Although the outcome of the thesis satisfied all stakeholders, there are still many rooms for improvements in the future such as automated deployment and integration process.

Keywords	Node.js, Google Cloud Platform, Google Cloud Firestore, Google Cloud Functions, real-time application

Contents

List of Abbreviations

GCP	Google Cloud Platform, a cloud computing service powered by Google.
I/O	Input/output, the communication between a computer and other devices.
MVC	Model - View - Controller, a software design pattern which separates the business logic to three components
MVP	Model - View - Presentation, a derivative software design pattern of MVC which strictly forbids the interaction between model and view
MVVM	Model - View - ViewModel, a software design pattern which allows the view and model to interact directly
API	Application Programming Interface, a computing interface that allows the communications between multiple services.
CI/CD	Continuous integration and continuous delivery
orm	Object-relational mapping
npm	Node Package Manager

1 Introduction

The advancement of cloud computing service introduces a new way of developing low latency, high availability and scalability real-time software with minimal cost and required resources. People use real-time applications daily, such as online messaging, making video calls or booking a pizza. The definition of “real time” states that the system must respond in a time constraint (Cooling, 2019), hence, it requires a unique infrastructure to handle live data. Over the decades, the system architecture has been changed noticeably due to the need of minimizing cost and building complex, yet high quality systems (Cooling, 2019).

Before the cloud computing era, most real-time applications written in JavaScript used either Socket.io or WebSocket to maintain the live connection between client and server sides. Although Socket.io has been praised by the JavaScript developer community for many years, there are several drawbacks including callback-centric behavior, inconsistent message system and bizarre encodings (Roper, 2018). Besides, the lack of resources, including a substantial initial development budget and a team of experienced developers, may halt the success of Socket.io (Ilya, 2020). Firebase, which is a service of Google Cloud Platform, provides a ready-made real-time solution with a minimal initial cost, lower maintenance and operation cost (for low to normal traffic application), reliable performance and auto-scaled architecture. Furthermore, Firebase supports modern architectures such as microservices and serverless architecture, which significantly reduces the development, deployment and maintenance cost.

JavaScript has become the most popular programming language in 2019 (Chan, 2019). With the rapid growth of JavaScript popularity in the developer community, JavaScript projects such as Node.js have been welcomed and received positive feedback by the community. Node.js delivers the fast, effortless, yet seamless development experience and scalable and highly available application as a result of functional program, single-threaded and non-blocking I/O characteristics.

The choice of technologies in this thesis is based on the author's working experience in the industry; nevertheless, it is not a silver bullet, and the system architecture should be determined per project.

1.1 Project technical objective

The final goal of the project is to build a scalable, high-available, yet reliable real-time backend architecture using Node.js and Google Cloud Platform cloud service. The project implementation presents the technical insight of microservices and serverless architecture for real-time application. At the end of the case study practice, a fully operational real-time backend architecture for a taxi booking app is built.

1.2 Project summary & business objective

The main business objective of the thesis is to present a solution for developing a fully operational taxi booking app with minimal initial budget and high technical requirements. The taxi company’s goal is to connect multiple taxi drivers to an existing taxi booking app platform, which will increase the income of drivers and reduce the cost of customers. For the first phase, the desirable outcome is to build a Minimum Viable Product which is able to perform a full cycle from booking a taxi to ending a trip. Although there are some existing service providers in the market, the entrance and monthly fee is not reasonable, and the return on investment does not meet the expectation.

1.3 Structure of the thesis

The following table shows the thesis structure and the final objectives and content in each chapter.

Chapter	Title	Final objectives / contents
Chapter 1	Introduction	- Describing the background of the thesis topic. - Presenting the thesis's technical and business objectives, and the project summary. - Structure of the thesis
Chapter 2	Theoretical background	- Supplying the theoretical background for the project including Node.js, microservices architecture, serverless architecture and real-time database - Briefly describing the reasons of choosing Node.js and Google Cloud Platform
Chapter 3	Project	- Presenting project's details and implementation
Chapter 4	Conclusion	- Summarizing thesis's goals and outcomes - Evaluating the project and planning for future development
	References	- Listing all the sources in the thesis
	Appendices	- Listing information and code snippets reinforcing the content.

2 Theoretical background

2.1 Node.js

2.1.1 Overview

JavaScript V8 engine, which was the most powerful JavaScript engine at that time, was released by Lars Bak, a Google software engineer, in 2008 (Pasquali and Faaborg, 2017). As a result, Ryan Dahl developed Node.js, which is now the most popular JavaScript runtime environment running outside a web browser, in 2009 (OpenJS Foundation, 2020). For the first time in history, developers can build a backend application using JavaScript with high-level programming language features and high performance (Pasquali and Faaborg, 2017). Compared to other existing backend frameworks, Node.js provides an exclusive infrastructure due to event-driven, single-threaded and non-blocking I/O characteristics.

The main concept of Node.js is to grant the developer access to the JavaScript's event loop and to system resources (Wilson, 2018). Hence, the Node.js developers are responsible for creating and handling the callback functions, which respond to registered events. The figure 1 below shows how the event-driven architecture works in Node.js As can be seen, events and callbacks are encapsulated and managed by a single stack, which delegates the concurrency work to the system. At the beginning of the process, an application makes an event to the event demultiplexer. Then the event demultiplexer pushes several corresponding events to the event queue. The event loop iterates and executes the events of the event queue. The event is executed by its registered handler, and the control is given to the event loop when the handler finishes the execution. The event demultiplexer can receive an event request, which will perform other event execution again, while the handler is running. As an event-driven JavaScript runtime environment, Node.js is designed to create scalable applications (OpenJS Foundation, 2020).

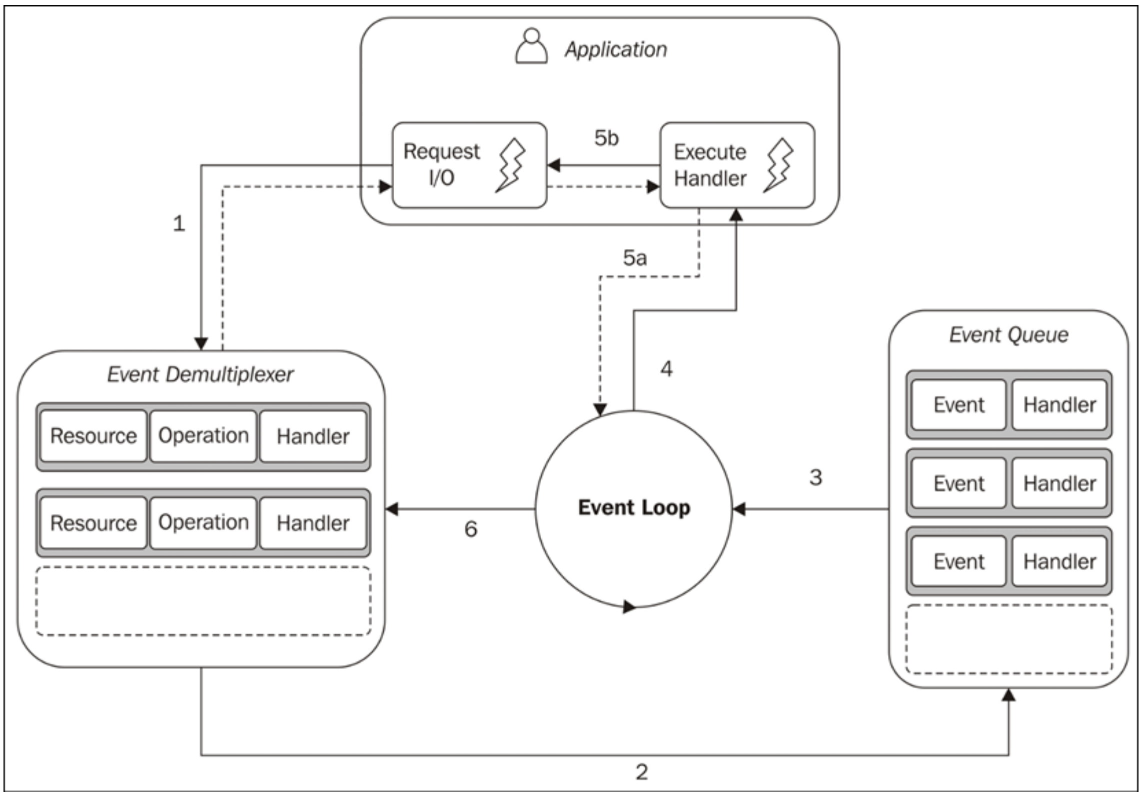


Figure 1. The reactor pattern (Casciaro and Mammino, 2020)

While other frameworks utilize parallelism by using multithreaded architecture, Node.js applies JavaScript's principle of single-threaded environment (Wilson, 2018). In comparison, there are plenty of advantages and disadvantages of single-threaded architecture. The author will discuss them in the next part.

Node.js uses a non-blocking I/O mechanism to access the system resources. In a non-blocking I/O system, the data can be always obtainable without waiting for the data read/write execution to complete. If the data is not ready to obtain, the system should return a predefined value which signals the data status (Casciaro and Mammino, 2020).

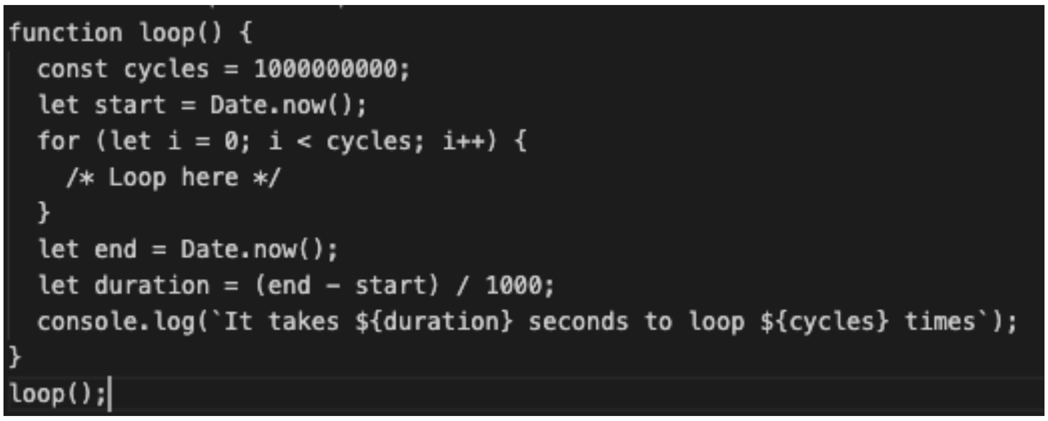
2.1.2 Benefits

The adoption of Node.js has increased sharply since it was initially released, and some popular backend frameworks, such as Express.js and Meteor.js, have been created for

Node.js environment. There are various reasons for the quick adoption from the community.

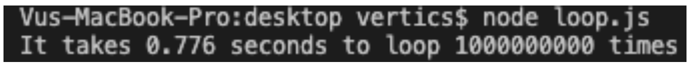
Firstly, Node.js applies JavaScript concept of functional programming, thus, it is effortless to modularize to maintain and reuse the code. According to npmjs, there are approximately 1.5 million JavaScript package modules in January 2021 (npm, Inc., 2020). With the significant number of packages, developers can reuse the existing packages, or create and maintain the new one with ease.

With the power of V8 engine, it is less than a second to loop through a billion times in JavaScript with a Macbook Pro 2019 with a 2.6 GHz Intel Core i8 processor. The script and the figure below show the code and the outcome of the billion-time loop experiment. Node.js is built on Chrome's V8 engine, therefore, developers do not need to worry about the performance in most cases.



```
function loop() {
  const cycles = 1000000000;
  let start = Date.now();
  for (let i = 0; i < cycles; i++) {
    /* Loop here */
  }
  let end = Date.now();
  let duration = (end - start) / 1000;
  console.log(`It takes ${duration} seconds to loop ${cycles} times`);
}
loop();
```

Script 1. Looping through 1 billion times using JavaScript



```
Vus-MacBook-Pro:desktop vertics$ node loop.js
It takes 0.776 seconds to loop 1000000000 times
```

Figure 2. The outcome of looping through 1 billion times

Node.js proves the concept of the single-threaded concept to be useful in developing high quality and performance applications (Pasquali and Faaborg, 2017). In comparison to other languages using the multithreaded mechanism, Node.js developers can leave the concurrency burden to the system, hence, the complexity and difficulty of multithreaded systems can be ignored (Meadow, 2018). Furthermore, Node.js also supports running parallel tasks in multiple processors by using the child_process module. Child processes mechanism allows developers to spawn, control and close the independent processes programmatically.

The non-blocking I/O concept allows other events to complete their cycles while other heavy load events, such as write or read files, operate. This mechanism reduces the response time and makes the application more reactive.

In conclusion, single-threaded, non-blocking I/O and functional programming features of Node.js benefit developers to build a fast, reliable, yet high performance application with seamless experience.

2.1.3 Drawbacks

By using Chrome’s V8 engine, the performance of Node.js is not an issue in most cases. However, Node.js performance is not as good as other existing backend frameworks when running CPU-intensive tasks such as data manipulation due to its single-threaded mechanism (Casciaro and Mammiano, 2020). Since the code runs on a single event-loop, the CPU-intensive tasks may block other tasks to operate. Although Node.js supports running multiple processes parallel, it is advised to consider using different frameworks. Besides, serverless architecture such as Google Cloud Functions or Amazon Lambda is recommended to use when dealing with CPU-intensive tasks.

JavaScript applies the functional programming paradigm, so object-oriented programming developers may find the new paradigm uncomfortable to work with. Besides, object-oriented programming languages apply strict type rules which require developers to take care about the type of variables. Strict type rule is used to verify the code validation, however, it requires more code lines to perform and it may lead to “lasagna code” because of many abstraction layers. Meanwhile, JavaScript uses dynamic type rules which cause some issues regarding the maintenance and development. There are various existing solutions in the market, and the most popular one is Typescript, which is released by Microsoft. It is recommended to use Typescript when developing Node.js applications.

In conclusion, developing with Node.js may come with some issues related to performance and development. In the scope of this thesis, Node.js is the choice for the backend.

2.1.4 Express

Express is the most well-known web framework running in Node.js runtime environment (MDN Contributors, 2020). Since Express was released, it has been receiving good feedback from many experts, and it has become one of the most popular choices for developing web servers. Express provides a layer to:

- create http request handler
- write server templates
- configure web server routes and port to listen to incoming requests
- add middleware to intercept the incoming requests.

Since Express is an unopinionated framework, it gives developers the freedom to choose the application architecture (MDN Contributors, 2020). There is no “right” or “wrong” way of structuring Express files, as long as it achieves the technical and business objectives. In the thesis project, Express will be used as a web framework to develop Node.js application.

2.1.5 MVC pattern

MVC which stands for model - view - controller is a popular software architecture pattern (Hibbard et al, 2020). The idea behind this pattern is to break down the business logic to three components, and each component has its own responsibility. The figure below represents MVC components, and how they connect to each other.

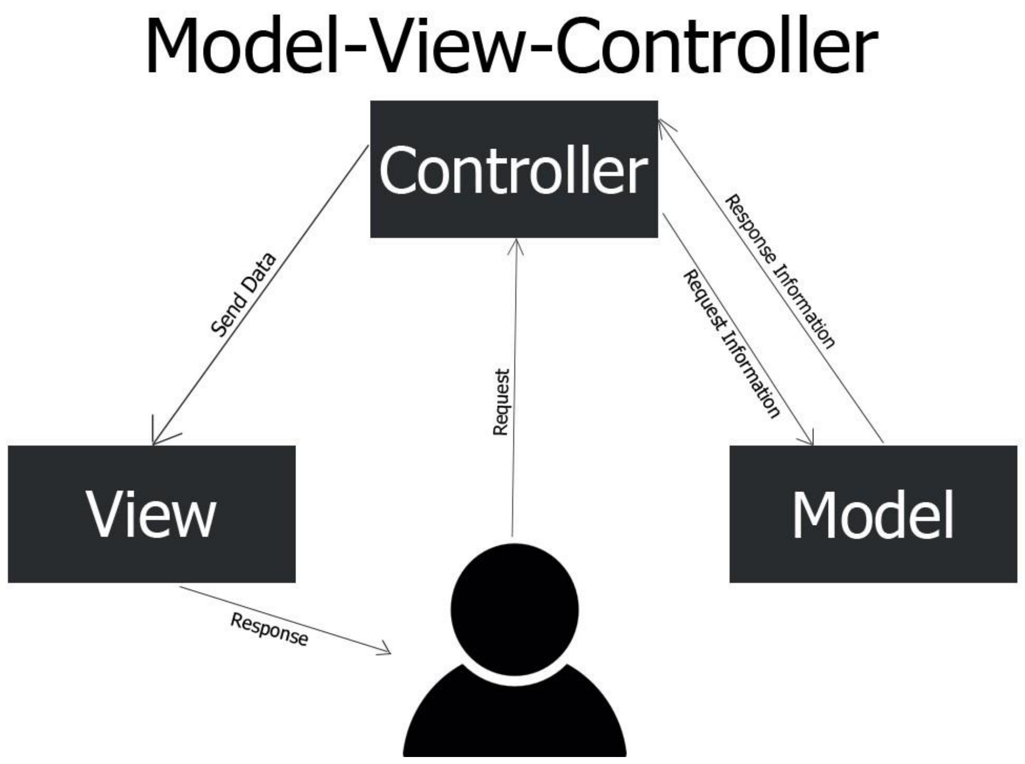


Figure 3. MVC Design Pattern (Spinelli, 2018)

Three components are responsible for:

- model: Representing the data structure of the application. It resembles the data-related functionality that users deal with.
- view: Displaying any elements and user interfaces which the user can see
- controller: Handling the logic of the application. It connects the view and model part by sending requests to model, handling response and sending the response back to view (Hibbard et al, 2020).

MVC design pattern benefits the software development by offering modularity (Chrome Developers, 2012). Modularity reduces the cost for maintenance, modification and feature extension. Furthermore, by separating the view, data and business logic, it is possible for a developer team to work in different components at the same time (Chrome Developers, 2012).

There are several extended versions of MVC such as MVP (Model - View - Presentation) or MVVM (Model - View - ViewModel). There is no silver bullet pattern that works for everything, so it is recommended to choose the pattern depending on the business objectives, working environment and developer experience. In this thesis, MVC is the choice for the case study's design pattern because it is recommended by Node.js/Express developer community.

2.2 Microservices architecture

2.2.1 Background

During the advancement of technology since the dot-com bubble, the software industry has been transforming rapidly; Consequently, the continuous deployment and delivery is a key factor in software engineering. The old and traditional deployment method is to make a single unit of deployment, which requires the new deployment process whenever a code line is added, removed or edited. Therefore, the deployment cost increases corresponding to the software complexity. Software engineers noticed the drawback of this pattern, and they built an alternative approach to adapt with new waves of transformation.

For many years, the traditional software architecture was to build a big and single monolith application which contains all tightly coupled components in a development ecosystem (Pacheco, 2018). Monolith application benefits the development team when they develop the software from scratch. Besides, monolith architecture promotes seamless testing, deploying and modifying experience, thus, the developers can focus on developing new features. Last but not least, monolith enables horizontal scaling by running multiple instances behind a load balancer (Richardson, 2018). However, when the application and your development team size expand steadily, it becomes "monolithic hell".

Monolithic hell is the state of software development when everything goes down, ranging from speed of development, deployment and testing to reliability, modifiability and scalability (Richardson, 2018). Since the components are tightly coupled, the bigger the development team is, the more frustrated the developers are. When the application grows, the frustration between different teams arises as there is no clear boundary between them. Furthermore, it is immensely challenging and costly to add or replace the current obsolete technology with the new and modern one when it comes to monolithic architecture as the whole application has to be written again. The figure 4 below represents a concrete monolithic hell example of an existing product. As can be observed, all teams in FTGO are working in a single, large and complex repository at the same time; thus, it leads to several problems regarding development, deployment and testing.

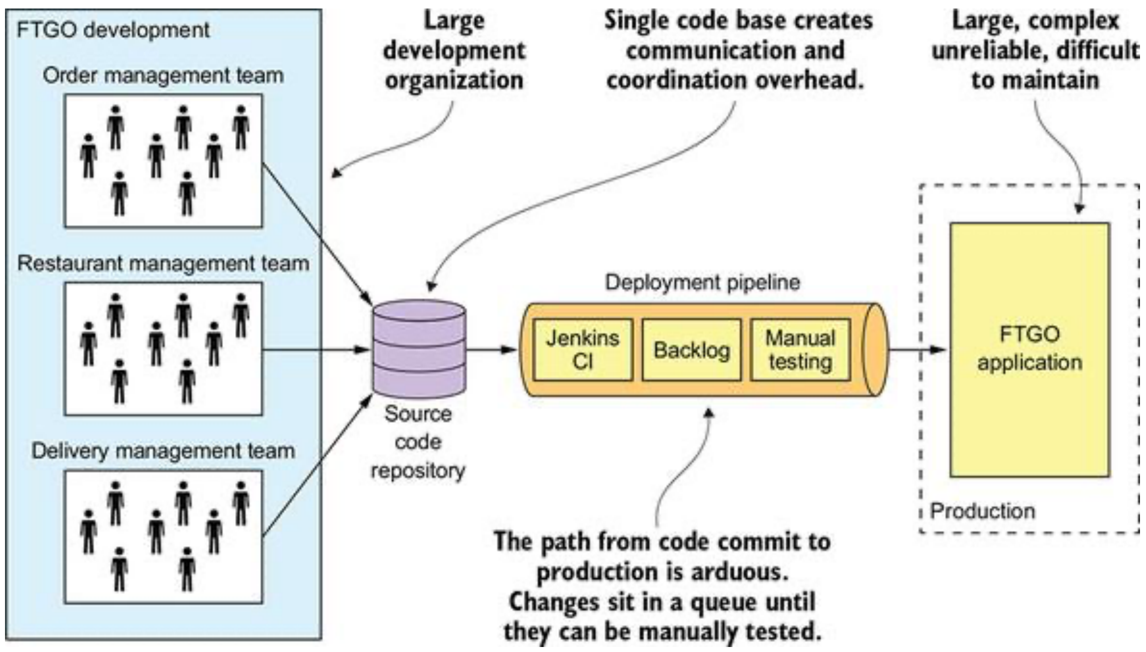

FTGO development
Order management team
Restaurant management team
Delivery management team
Large development organization
Single code base creates communication and coordination overhead.
Large, complex unreliable, difficult to maintain
Deployment pipeline
Jenkins CI
Backlog
Manual testing
Source code repository
FTGO application
Production
The path from code commit to production is arduous. Changes sit in a queue until they can be manually tested.

Figure 4. A monolithic hell case: FTGO application (Richardson, 2018)

2.2.2 Overview

Monolithic had dominated the software industry until Netflix and Amazon first adopted the microservices architecture (Brown, 2016). Microservices has taken over the monolithic domination, and it becomes the standard for developing big, multi-services and cross-team applications. In comparison with monolithic, multi-services provides an architecture that combines independent, bounded and interoperable components (McLarty et al, 2016). Adrian Cockcroft, a former software engineer of Netflix, added that microservices composes several loosely coupled service-oriented components that have boundaries (Richardson, 2018). The figure 5 below shows the microservices architecture of the new FTGO application after transformation from monolithic architecture. As can be seen, the old application has been divided into several service-oriented components which are loosely coupled. Each component works on its own system, and each backend service has an autonomous data model and database. Backend services interact with others by using lightweight protocols such as REST. API Gateway which is a layer that assures the reliability and maintainability of the API handles the incoming network request from client applications to forward the requests to the correct services. Therefore, it enables developer teams to independently develop, test, deploy and scale without affecting other services (Richardson, 2018).

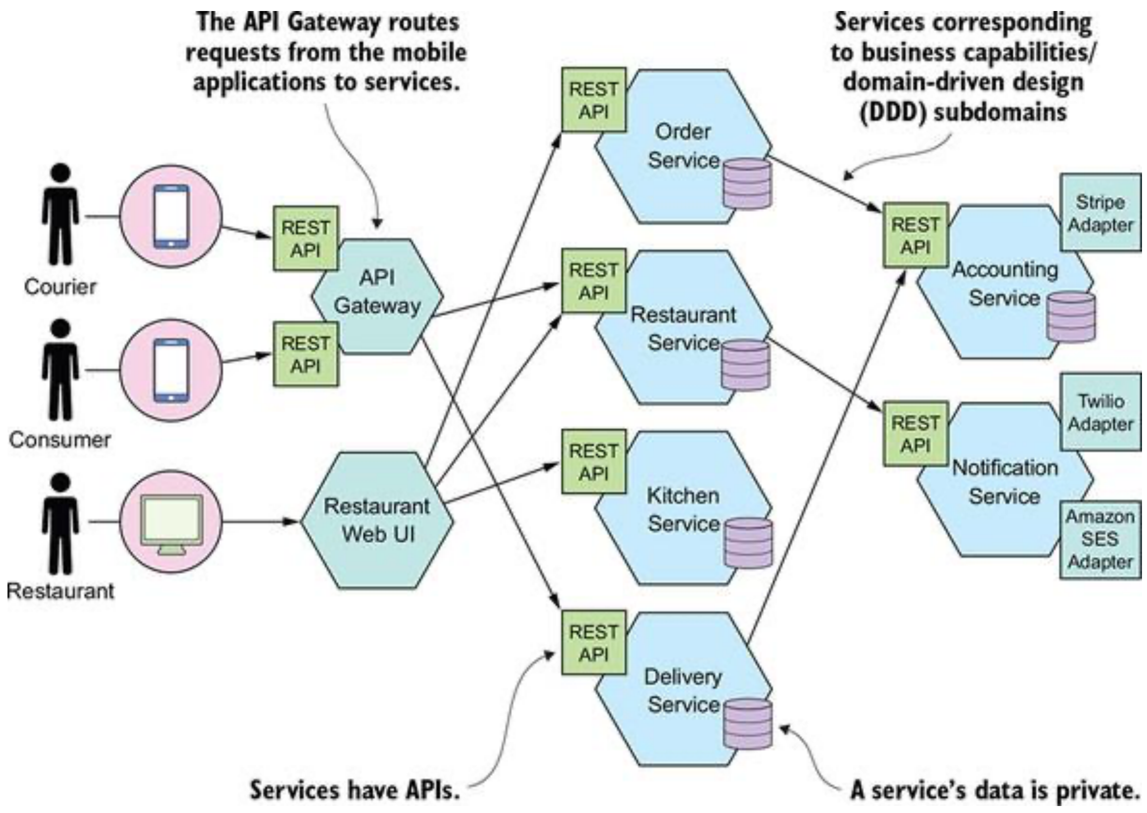


Figure 5. Microservice architecture of FTGO application (Richardson, 2018)

With microservices, developers are able to create a set of autonomous service-oriented components which are independently developed, tested and deployed. Besides, it is straightforward to build a horizontally and vertically scalable and robust application by using microservices. Unlike monolithic, microservices enable developers to experiment and adopt new technology for a single microservice depending on their experience, taste, business and technical objectives (Newman, 2021).

However, since there is no conventional method to break down the system into services, it requires deep experience and understanding of business and technology to do the job. If it is not done right, there is a chance that the system will be a distributed monolith application which requires to deploy multiple services simultaneously. Many developer teams hesitate to adopt microservices architecture because of the complex distributed systems. While monolithic has a single web server platform and a single database, the microservices component has its own independent web server and database. To coordinate multi-services for distribution, it requires intensive cross-team elaboration, software engineer experience and delivery skills. Last but not least, the cost of adopting microservices at the early phase is exceptionally expensive due to the need of skilled developers and resources to plan, architect and build a distribution pipeline (Newman, 2021).

2.2.3 Google Cloud Platform

As the microservices has been a megatrend in the software development industry, cloud service providers respond to it by providing a comprehensive set of tools to fully support microservices in their cloud service. As one of the leading cloud service providers, Google Cloud Platform offers consolidated solutions to migrate monolithic to microservices, ranging from hosting and database services to container management service, with scalability, high-availability, interoperability and security infrastructure (Google, 2020). Kubernetes, which is controversially the best production-grade container orchestration tool in the market, provides a full-scaled and secured way to deploy and host microservices, which reduces workload and stressful working hours for developer teams. The figure 6 below represents an example of microservices using Google Cloud Platform. As can be observed, the microservices is hosted in a Kubernetes Engine, and it can communicate to the backend systems by a private network connection powered by Cloud Interconnect or Cloud VPN. Meanwhile, API Gateway is handled by either Apigee or Google Cloud Load Balancing, which are more secure and scalable.

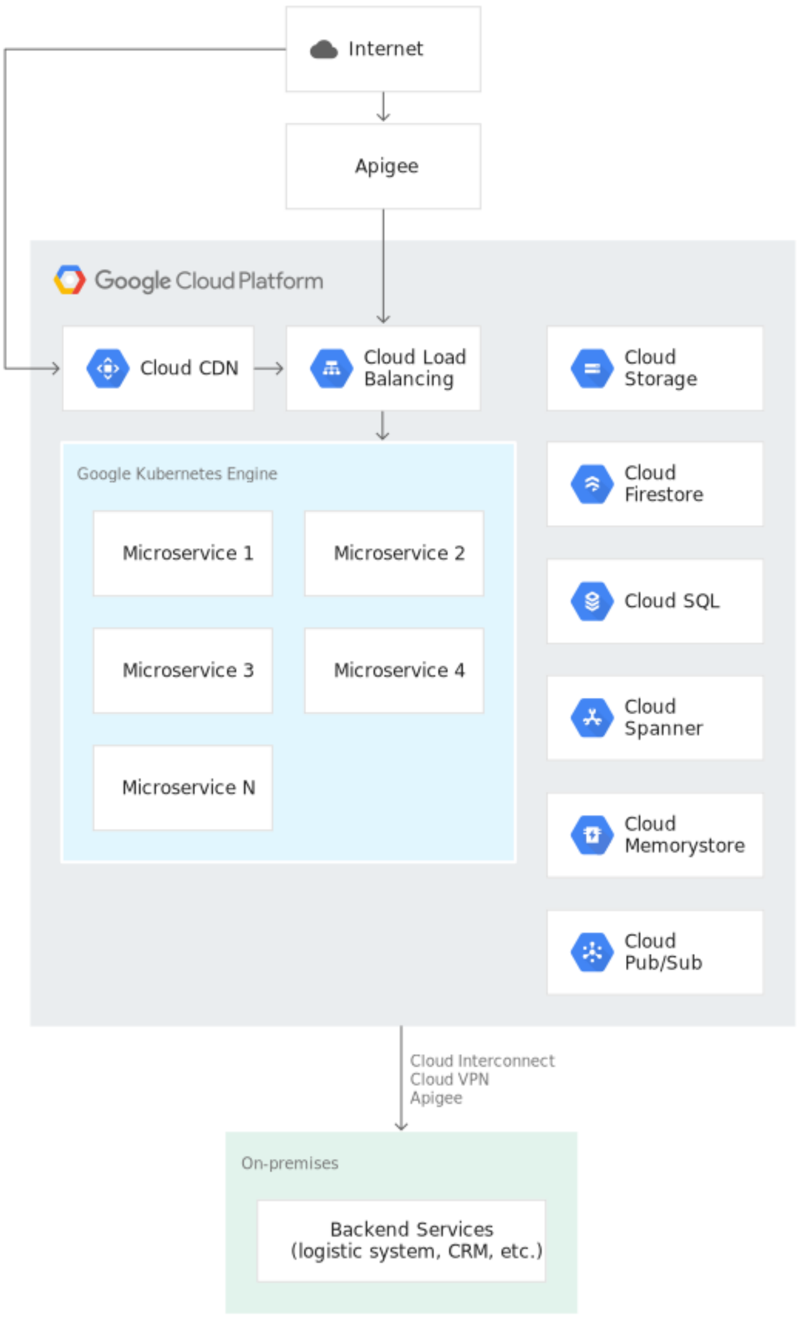


Figure 6. Google Cloud Platform microservices architecture example (Google, 2020)

For the scope of this thesis, the case study is designed by the above architecture. However, Kubernetes is not implemented and discussed further due to its complexity and the scale of the project.

2.3 Serverless architecture

2.3.1 Background

With the escalation of the Internet coverage, more than half of the global population has the access to the digital world (Clement, 2020). Consequently, the network traffic is increasing steadily. To ensure the availability of the system, software engineers have to change the infrastructure to not only scale the system horizontally and vertically but also keep the billing as minimal as possible. There is a trade-off between scalability and the service cost management because of the service cold start. Making a high available service results in better user-experience and expensive bills, while a less-available, yet reasonably priced service needs time, which ranges from a few milliseconds to minutes, to wake up the instance. Besides, it is arduous to predict the peak and the pit of network requests.

It was not until Google released App Engine, a serverless hosting service, in 2008 that people heard the word serverless. The original idea was to build a service that helps the developer to focus on doing the business logic instead of spending time to configure the system (Venema, 2018). However, App Engine had been unknown until it was previewed in 2011. Until this point, serverless was still a blur definition for the developer community. Other service providers such as Amazon, Microsoft and IBM released their own serverless services in the next few years, as a result, serverless has become one of the most popular architectures these days (Katzer, 2020). Functions as a service (FaaS) coming with the abstract service management has removed the barrier to the backend world for frontend developers (Dabit, 2020).

2.3.2 Overview

With the advancement of serverless architecture, the server management workload is delegated to the cloud providers. However, the workload is not totally removed from the teams; in fact, an operation team is in charge of handling the cloud service, while developers can pay attention to write the business logic (Katzer, 2020). The figure below shows an example of a monolith serverless architecture. As can be seen, the infrastructure and network communication is as same as a normal server-based application. However, serverless functions and datastore are scaled and configured by the cloud provider.

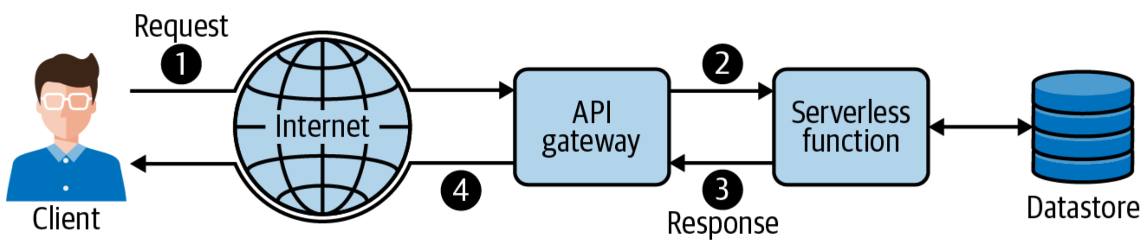


Figure 7. An example of monolith serverless (Katzer, 2020)

Serverless architecture allows developer teams to create a powerful scalable, secured and reliable server-based applications without server configuration and management; hence, the developers can focus on doing business logic related tasks and the company saves the resources for managing servers. Moreover, the cloud providers offer a pay by usage billing option for the serverless service. In comparison with traditional server hosting service, using serverless will lower the overall hosting cost if the application has some peak and pit period. For example, a taxi booking app usually receives peak network requests during weekend's nights, while it has low network traffic during weekdays. If the server is hosted in a traditional host service, then it will require a high memory and multicore processor computer to ensure the highly availability during a rush hour. However, the server does not use all the computer's potential most of the time, thus, wasting the company's money for excessive resources (Katzer, 2020). On the other hand, serverless provides a self-scalable solution and the pay per usage option, so it reduces the cost for running the service.

However, serverless is still in the early phase of development, and there are several optimizations that can be done in order to replace the traditional hosting service. The most common problem in serverless is the cold start time. The cold start time is the time period that requires to start an instance or service when it is in idle mode. The cold start time depends on the cloud providers, the technology and the application size. Although Amazon and Google have been trying to reduce the cold start time to under 1 second, it is much bigger than a desired network request response time, which is 100ms (Shilkov, 2021). Moreover, serverless has some native cloud provider problems, such as network, complex debugging system and compute time issues. With the advancement of cloud technology, it is believed that the big tech giants can maximize the serverless' capabilities in the near future. Besides, if the application's network traffic is nearly constant, it is inefficient to use the serverless pattern since the cost of running service is higher compared to traditional one (Katzer, 2020).

2.3.3 Google Cloud Functions

Every cloud service provider offers its own serverless service, so choosing the correct providers for the project is extremely tough, and it depends on multiple factors such as technology, team experience, budget, project and time constraints. In terms of technology, AWS is leading in cost-efficient and network performance while Google is better in terms of scalability and dependency management. However, they are both fast, reliable and less bug-prone, so technology should not be the main reason for choosing a serverless service (TechMagic, 2020). For the scope of the project, Google Cloud Functions will be chosen for serverless because of the author's limited experience and the integrity of the project.

Cloud Functions is a pay-as-you-go serverless service powered by Google. It aimed to reduce the resources for server management and cost reduction for running and maintaining services (Google, 2021). Besides, the automatic scaling feature and end-to-end development and debugging system makes the development process seamless. The figure 8 represents an example of using cloud functions to send the push notification through Firebase Cloud Messaging service. As can be observed, when a user follows others, the database is updated according to the action. Cloud Functions listens to the database changes, and it sends the push notifications to the devices by the Firebase Cloud Messaging.

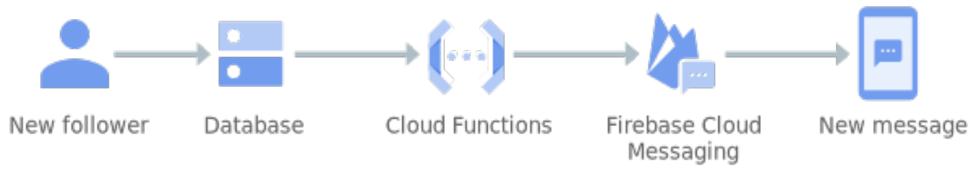


Figure 8. An example of Google Cloud Functions for sending the push notification (Google, 2021)

2.4 Cloud Firestore

2.4.1 Overview

Since the rapid growth of NoSQL databases, cloud providers have introduced multiple scalable database solutions such as DynamoDB, Realtime Database or Cloud Firestore. The ultimate objective is to offer a seamless experience for developers to create and maintain a flexible and scalable database.

Cloud Firestore enables the real time data synchronization between multiple client applications using real time listeners. Besides, the offline support ensures the application reliability despite network connection. With the revolution of NoSQL databases, Cloud Firestore provides a comprehensive hierarchical data model which stores data in key value pair format. Besides, The Cloud Firestore native querying is vivid, easy and efficient (Google, 2021). Furthermore, Google Firebase recently introduced a complete local testing system for Cloud Firestore and Cloud Functions, which improves the testability and maintainability. Although Cloud Firestore does not support complex queries such as searching documents by string, Google provides a straightforward integration with some popular search frameworks such as Algolia and Elastic Search, which provides the best search optimization solutions in the market. In addition, Cloud Firestore proposes smooth integration with Cloud Functions and App Engine, which is used in the case study. Google provides clear documentation for the Cloud Firestore, so it is easy to integrate for traditional frontend developers or backend developers without NoSQL database knowledge.

2.4.2 Comparison with Realtime Database

In addition to Cloud Firestore, Google offers Realtime Database, which is a cloud-based, real time database solution. In comparison with Cloud Firestore, Realtime Database provides less features and slower in terms of query response time. For example, Cloud Firestore uses indexed queries, while Realtime Database provides a deep query solution; hence, Cloud Firestore is faster than Realtime Database when the data set is big enough. In terms of the billing factor, Realtime Database offers the charge rate by bandwidth and storage, while Cloud Firestore changes the usage by number of database operations (Google, 2021). Therefore, Realtime Database is suitable for a small application which has unstructured data models and small dataset.

3 Project

3.1 Project summary

The project was built in order to create a platform which connects a taxi booking app to small traditional taxi companies. The current taxi booking app has millions of customers in Finland and served half million orders in 2019. The traditional taxi companies have been struggling with their businesses since the disruptive innovation in the taxi industry. In Finnish market, there are two ride-sharing services, Uber and Yango, which offer lower price, better service quality and location tracking service. Therefore, the digital transformation is vital for the survival of small taxi companies. There were some existing applications which had the same objectives in the market. However, they had problems related to performance and user experience issues. Firstly, drivers complained about missing orders, which were notified by push notifications, since the drivers might focus on driving. Besides, the current applications had several performance issues regarding taxi availability check, order operation, or location tracking issues. The project aimed to bring the seamless experience and high-quality application for drivers who work for the small taxi companies in Finland.

For the final product, the company owner should be able to manage their companies, taxis, drivers and prices, pay the invoices and subscription fees, handle the paperwork and check past and incoming orders in the web dashboard. The admin can do the owner’s workload and perform overall management. The mobile application should enable drivers to perform authentication, receive orders, handle order operations, and other user and order management work.

The case study implements the first phase of the application which represents a comprehensive backend architecture of the server-based application. Based on the initial scope meeting, the core features, such as taxi availability checking, order booking, order operations, overall company management, and payment system must be implemented. The customers also considered several possible features, which might be done in a later stage.

The development team included 3 developers and 1 project manager. The author was the lead developer for the project, and he was responsible for designing high-level software architecture and developing the backend part. Two other developers were in charge of developing the frontend part, including a mobile application and a web dashboard.

After the initial meeting, the technology requirement was partly decided. There would be 3 separated applications: A web application using ReactJS, a mobile application for drivers using React Native, and a server application using Node.js and Google Cloud Platform. Besides, the client asked for a scalable and highly available software applications with a tight budget, so using modern architecture, such as microservices and serverless architecture, and using automatically scaling platforms such as Cloud Functions and Cloud Firestore would match the initial requirements and speed up the development process. By understanding the business objectives and technical requirements, the first minimal viable product was successfully created.

3.2 Project challenges

The final product would be put into production to hundreds of users with bug free, scalable and highly available qualities. Besides, the sophisticated architecture and multiple third-party integrations should be planned ahead of the start of the project.

According to client’s requirements, the software should have low latency connection, minimal downtime and ability to receive multiple requests during the weekends’ nights. Besides, the client plans to create more features and integrations after the product release, so the application size can expand sharply. Therefore, it demands a scalable, highly available, yet flexible architecture.

Picking a technical stack is always a difficult task when starting the project from scratch. The decision is a combination of multiple factors including developer experience, budget, technical problems and architecture. Therefore, the technical stack should be carefully discussed in the initial meeting between developer teams.

The application compels multiple third-party integrations including client taxi booking platform, payment gateway and a Finnish bank and mobile authentication service. Integration needs the close collaboration between multiple teams in different organizations; hence, the time for communication should be taken into consideration when doing the estimation. Besides, it is challenging and costly to find the bug when it comes from the integration.
Finally, all operation issues should be clearly evaluated beforehand. Downtime or high network latency would result in revenue and customer loss for the client. Besides, setting up a system that can automatically deliver the software to the production should be in the priority list since it saves resources for testing, development and deployment process.

3.3 Technical solutions

To resolve the architecture issue, the developer team decides to use microservices architecture for the backend part. As a result, the backend consists of several autonomous loosely coupled service-oriented components which connect to each other through a private network. In comparison, microservices accelerates the development maintenance, and deployment process, and it also ensures the seamless work collaboration between multiple teams at the same time. Besides, the serverless architecture is used due to unpredictable number of network requests during the weekends and highly available requirements.

Based on the developer experience and technical requirements, JavaScript and Typescript were chosen as the main programming languages. JavaScript, which uses a function programming paradigm, has been popular for its dynamic, flexibility and modularity, while Typescript, which powers by Microsoft, is a “super type” version of JavaScript. Since the core products consist of client and server applications, a full stack programming language, such as JavaScript or Typescript, should be chosen. For the frontend side, ReactJS and React Native, which are Facebook’s projects, have been picked for the web dashboard and mobile application respectively. ReactJS is a well-known library for web application development, while React Native is used to create hybrid mobile applications. For the backend side, Node.js and its famous framework, Express, along with Typescript are used to ensure the seamless developer experience and technical requirement fulfillment. The reason to use Typescript for backend development is that JavaScript, which supports loosely dynamic types, tends to generate many unexpected bugs due to the nature of the backend system - heavily usage of data models. Besides, PostgreSQL is a choice for most databases due to its consistency, logical data structure and scalability. Since there is a need for real time connection between client and server sides, Cloud Firestore, which is a real-time NoSQL database, is used for the order service.

For the deployment tools, Kubernetes, which is a production-grade container orchestration tool, is the choice for handling microservices architecture because of its reliability, scalability and security. Due to the limitation of the topic and the complexity of Kubenetes, Kubernetes would not be discussed in the technical implementation.

3.4 Implementation

In this section, the complete backend architecture is thoroughly evaluated, discussed and implemented. At the end of the section, an example of the real-time backend architecture should be created.

3.4.1 Software architecture

The software architecture follows the technical solutions that were decided during the first technical meeting. Figure 9 below represents the overall architecture of the backend side.

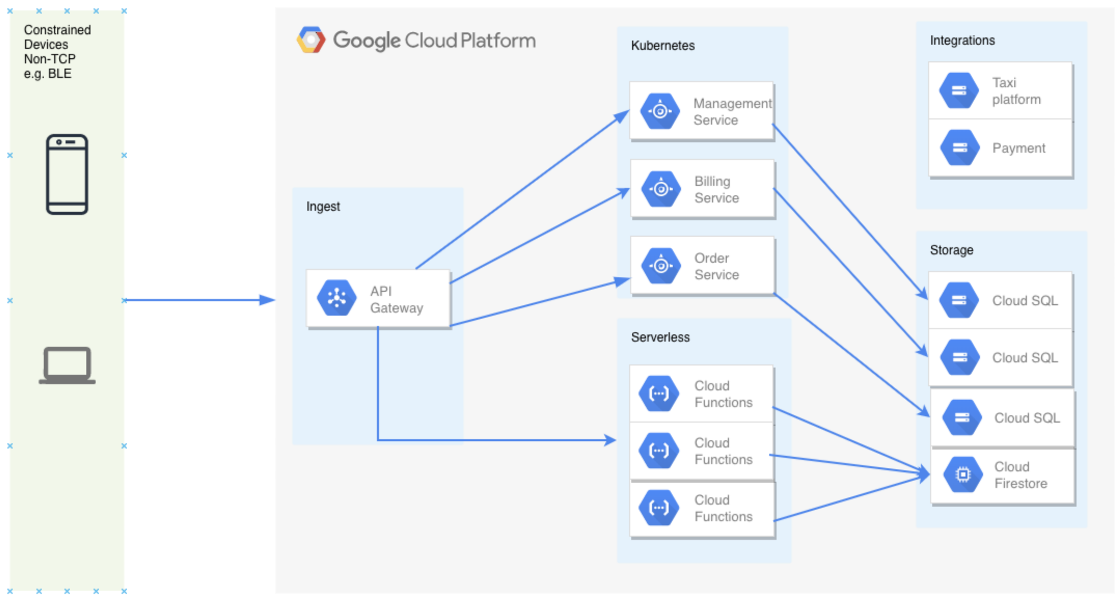


Figure 9. Overall architecture of the application.

As can be seen, the application contains 4 independent service-oriented components, which represents the core features. The microservice was divided by using domain driven design, which is a design methodology for a large-scale application. Each microservice has its own server and database, and it communicates to each other through a private network setting up by Google Virtual Private Cloud (VPC). Three services, which are managed by Kubernetes engines, are automatically scaled by using the load balancers and traffic ingress powered by Kubernetes. There are 2 components which are dedicated to handle the operations between the application and third-party services. The incoming network requests firstly come to an API Gateway, where they are redirected to the desired services. The API Gateway makes the API stable regardless of the future changes in the backend services. Besides, the API Gateway provides a server caching service, which is powered by Redis. The caching service reduces the response time and the workload of server instances. The serverless Cloud Functions is deployed and ran independently, hence, it is easier to perform the maintenance and future development. The real time connection between client and server is handled by the combination of Cloud Functions and Cloud Firestore, resulting in real time experience, straightforward integration and scalability. Therefore, the architecture meets all the technical requirements of scalability, high availability, low latency and seamless developer experience.

3.4.2 Services and tools

The rapid growth of cloud computing services has transformed the traditional software developers’ working methodologies. Before the cloud computing era, it was difficult to look for all-in-one solutions which met the technical and business requirements. Cloud services enable business and technical teams to work under the same dome by providing scalable, secured and economical packages. The project consumes multiple tools and services powered by Google Cloud Platform. In the next part, the tools and services which are not discussed in the theoretical background would be presented.

Google Cloud Platform

Google Cloud Platform (GCP) is a cloud computed platform powered by Google. GCP supports a great range of digital products, from automatically scalable hosting services, database services and infrastructure to data analytics and machine learning services. GCP provides great supporting tools for microservices and serverless architecture, such as Cloud Functions, Cloud Firestore and Realtime Database.

Google API Gateway

API Gateway provides a well-structured REST API, which can be consumed by client applications. API Gateway provides a stable endpoint system for client and third-party integrations regardless of the changes in the backend services (Google, 2021). Besides, it provides a secured communication throughout the network request cycle and prevents the cyber-attacks. API Gateway is a must-used tool for microservices architecture.

Redis

Redis is a memory-data storage which is used for server and database caching. Redis supports various data structures including strings, lists, sets, and bitmaps, to name a few. Besides, Redis has many useful built-in functions such as Lua scripting and transactions (Redis Labs, 2021). Redis works best in the case when the service receives multiple duplicated requests in a short period, resulting in high availability for the application.

PostgreSQL

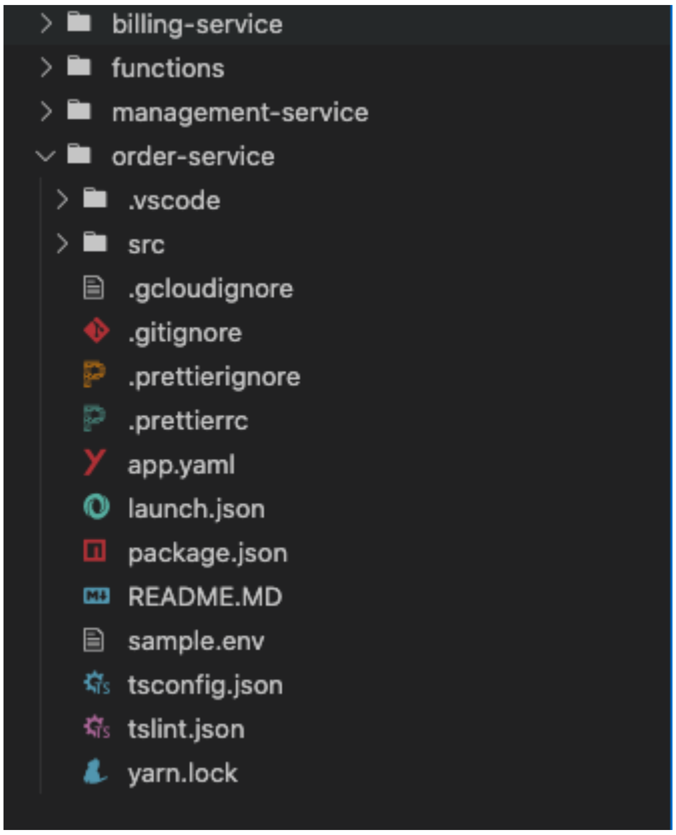
PostgreSQL is one of the most popular relational database systems in the market in terms of robustness, reliability, scalability and consistency. It supports abundant data types and data integrity. Besides, PostgreSQL is well-known for the query performance, concurrency and disaster recovery such as database replication and database recovery (PostgreSQL, 2021). In this project, PostgreSQL was hosted in CloudSQL, which supports vertical and horizontal database scaling.

3.4.3 Implementation

In this section, the step-by-step guide to build a complete architecture is described as images, scripts and descriptions.

Microservices initialization

Structuring microservices is always a controversial topic in the developer community. Some developers prefer placing each microservice in an independent repository, others believe that combining microservices into 1 repository gains more benefit. The structure, regardless of the choice, should benefit the teams in terms of development, testing and deployment process. For this project, the microservices are placed under a single repository, because it is easier to manage, maintain and deploy by a single pipeline.



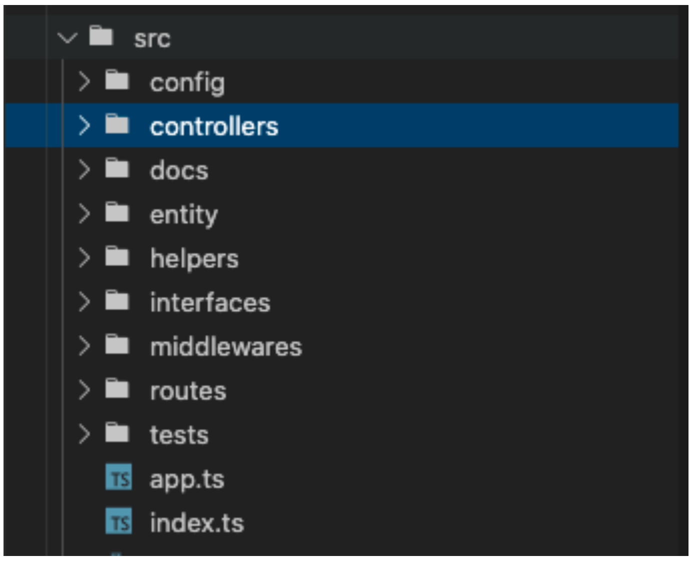
```
> billing-service
> functions
> management-service
v order-service
  > .vscode
  > src
    .gcloudignore
    .gitignore
    .prettierignore
    .prettierrc
    app.yaml
    launch.json
    package.json
    README.MD
    sample.env
    tsconfig.json
    tslint.json
    yarn.lock
```

Figure 10.Repository structure

The figure above shows how the directories are structured in the repository. Four directories represents four service-oriented components, and it is an independent loosely coupled backend service. The reusable modules, if any, are placed in the same root level with the independent services; hence, it is straightforward to inject and maintain the modules.

Setup service

It is recommended to use the express-generator package to generate an Express boilerplate application. For this project, the author used an existing company boilerplate, which provides great support for the MVC pattern. Figure 11 below illustrates the final structure of a single service. As can be observed, the structure abstracts the business logic, the API and the model parts. The entity folder handles the model parts, while the controllers and routes folders act as the controller and view respectively. Besides, middleware, which perform repetitive business logic, are kept in a separate folder.



```
src
  config
  controllers
  docs
  entity
  helpers
  interfaces
  middlewares
  routes
  tests
  app.ts
  index.ts
```

Figure 11.Service architecture

Setup docker

Docker is used to build an image of your application in order to run in multiple platforms. The script below shows the docker setup for a single service.

```
FROM node:10.13.0-alpine
# Set the working directory for the container
WORKDIR /usr/src/app
# Copy package.json
COPY package.json .
COPY .env .
RUN apk add --no-cache bash

RUN npm install
ADD . /usr/src/app
```

Script 1.   Docker configuration for Dockerfile

The script 2 illustrates the project setup using docker compose. Hence, the combination of Dockerfile and docker-compose.yaml results in a complete image of the backend service.

```
version: "3"
services:
  tictactoe:
    build: .
    volumes:
      - ./:/app
    env_file:
      - .env
    ports:
      - 8080:3000
    depends_on:
      postgres:
        condition: service_healthy
    command: bash -c "npm run dev"
  postgres:
    image: postgres:9.6-alpine
    restart: always
    ports:
      - 5432:5432
    environment:
      POSTGRES_DB: order_service
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
    volumes:
      - database-data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 5s
      timeout: 5s
      retries: 5
  pgadmin:
    image: chorss/docker-pgadmin4
    ports:
      - 5050:5050
volumes:
  database-data: {}

RUN npm install
ADD . /usr/src/app
```

Script 2. Docker configuration for docker-compose.yaml

Server configuration for deployment

The app.yaml file is read by Google App Engine during the deployment process in order to decide the instance and service configuration. The script 3 below illustrates a service script which uses the automatic scaling option. As can be observed, the service has at least 1 instance, and can scale up to 2 1024MB-memory instances. Besides, it can handle 20 concurrent requests or use up to 80% of memory before the second instance wakes up.

```
runtime: nodejs10
instance_class: F4
service: order-service
automatic_scaling:
  min_instances: 1
  max_instances: 2
  max_concurrent_requests: 20
  target_cpu_utilization: 0.8
  max_pending_latency: 100ms
```

Script 3. Script for app.yaml

Database configuration

The application uses the typeorm package to map the database entity to object using object-relational mapping (orm) paradigm. With the support of the orm pattern, writing database queries is fast and friendly for developers. Besides, the performance is almost as good as the native query. It is also easy to connect the server to the database. The script 4 below represents a json file where the database configuration is declared. This file should be included in the ".gitignore" file since it shows sensitive information.

```
[
  {
    "environment": "production",
    "type": "postgres",
    "name": "production",
    "host": "/cloudsql/project-name:region:production",
    "database": "production",
    "username": "postgres",
    "password": "postgres",
    "entities": [
      "build/entity/*.js"
    ],
    "migrations": [
      "build/migrations/*.js"
    ],
    "cli": {
      "entitiesDir": "build/entity",
      "migrationsDir": "build/migrations"
    },
    "logging": [
      "error"
    ],
    "synchronize": "false",
    "extra": {
      "socketPath": "/cloudsql/project-name:region:production"
    }
  }
]
```

Script 4. ormconfig.json configuration

Server configuration

Configuring the server in the correct manner ensures the reliability and stability of the application. The figure 5 below shows an example of setting up a Node.js server. As can be observed, several middleware was added in order to log events, allow cross-origin resources sharing (CORS), parse the requests and responses to json automatically, and handle the errors globally. Besides, the API routes and the documentation are defined in app.js. The backend application uses Node Package Manager (npm) to handle the dependencies.

```
dotenv.config()

/**
 * Create our app w/ express
 */
const app: express.Application = express()

/**
 * HELMET
 */
app.use(helmet())
/**
 * CORS
 */
app.use(cors())
app.options('*', cors())

/**
 * LOGGING
 */
app.use(morgan('combined', {stream: morganStream}))

/**
 * Body parsers and methods
 */
app.use(
         bodyParser.urlencoded({
             extended: true,
             limit: '50MB',
         }),
) // parse application/x-www-form-urlencoded
app.use(bodyParser.json({limit: '50MB'})) // parse application/json
app.use(methodOverride())

/**
 * Router
 */
```

```
app.use('/api/v1/companies', router.companyRouter)
/**
 * Swagger Documentation
 */
const swaggerSpec = YAML.load(path.join(__dirname + '/docs/swagger.yaml'))
app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(swaggerSpec))

app.use(errorHandler)

/**
 * Setting routes
 */

export default app
```

Script 5. Server configuration in App.js

The figure 6 below represents the port configuration for the server and how it recovers from unexpected bugs. As can be observed, the database connection was tested before the server initialization to prevent hidden bugs. Besides, the setup makes sure that the server's process is killed and restarted when the application gets either an uncaught expectation or an unhandled rejection. The application uses the forever npm package to restart the server automatically when it is programmatically killed. Therefore, it prevents the application from running in an unsafe server state.

```
const logger = initLogger(module)
createConnection()
         .then(() => {
             logger.info('Connect to db successfully!')
         })
         .catch(e => {
             logger.error('Db connection error', e.message)
         })

const server = app.listen(config.port, () => {
         logger.info(‘App is running on port ${config.port)’)
})

const exitHandler = () => {
         if (server) {
             server.close(() => {
                    logger.info('Server closed')
                    process.exit(1)
             })
         } else {
             process.exit(1)
```

```
          }
}

const unexpectedErrorHandler = error => {
          logger.error(error)
          exitHandler()
}

process.on('uncaughtException', unexpectedErrorHandler)
process.on('unhandledRejection', unexpectedErrorHandler)

process.on('SIGTERM', () => {
          logger.info('SIGTERM received')
          if (server) {
              server.close()
          }
})
```

Script 6. Server configuration in index.js

Besides, code sanitation should be put into priority list in order to make the code easy to develop, maintain and avoid the potential bugs. Tslint, prettier and husky are among the most popular npm packages for Typescript code checking tools. By using these tools, it assures the code quality, format and cross-platform support. Besides, since developers usually have their own code styles, using these tools force developers to follow the same code styles; hence, it reinforces the code consistency and coherence. The scripts 7 and 8 below illustrate a sample configuration for a Node.js server application. As can be seen, the files define many coding rules. For example, the “singleQuote” rule enforces the user to use a single quotation mark instead of the double quotation marks. However, rule overuse may cause the nightmare for the developers.

```
{
          "compilerOptions": {
              "sourceMap": true,
              "lib": ["es5", "es6", "es2017", "dom"],
              "moduleResolution": "node",
              "target": "es6",
              "outDir": "build",
              "module": "commonjs",
              "emitDecoratorMetadata": true,
              "experimentalDecorators": true,
              "skipLibCheck": true
          },
          "include": ["src/**/*.ts"],
          "exclude": ["node_modules"]
}
```

Script 7. Tslint configuration

Prettier is a well-known code formatter for JavaScript and Typescript developers. It aims to control the code quality and code format of the application.

```
{
  "arrowParens": "avoid",
  "bracketSpacing": false,
  "insertPragma": false,
  "printWidth": 80,
  "proseWrap": "preserve",
  "requirePragma": false,
  "semi": false,
  "singleQuote": true,
  "tabWidth": 2,
  "trailingComma": "all",
  "useTabs": true
}
```

Script 8. Prettier configuration

OpenAPI configuration

Documentation is the way backend developers can communicate with frontend developers indirectly. A clear documentation boosts the speed of development and maintenance process since it reduces the developer’s confusion, frustration and stress level. Besides, API documentation is used to publish the public API to the developer community. Swagger, which is controversially the best OpenAPI documentation tool, supports several programming languages integration. Google Cloud Platform also allows the developer to integrate Swagger with its API Gateway, which would be discussed in the next few pages. Swagger supports multiple formats such as json, YAML and many other languages. Since the introduction of OpenAPI 3.0, Swagger introduces the reusable components and a new way to define complex query params; hence, it saves time from rewriting the same code. The script 9 shows how a Swagger reusable schema is defined in YAML format.

```
components:
  # Reusable schemas, can be referenced as #/components/schemas/{schemas-name}
  schemas:
    User:
      type: object
      properties:
        id:
          type: number
          example: 1
        name:
          type: string
          minLength: 1
          example: John Smith
        email:
          type: string
          example: john@example.com
        city:
          type: string
          example: Espoo
        phoneNumber:
          type: string
          example: 0469132412
        address:
          type: string
          example: Espoo
        postalCode:
          type: string
          example: "02510"
        additionalInformation:
          type: string
          example: I want to work near home.
        birthDay:
          type: Date
          example: '2020-07-02'
```

Script 9. OpenAPI configuration example

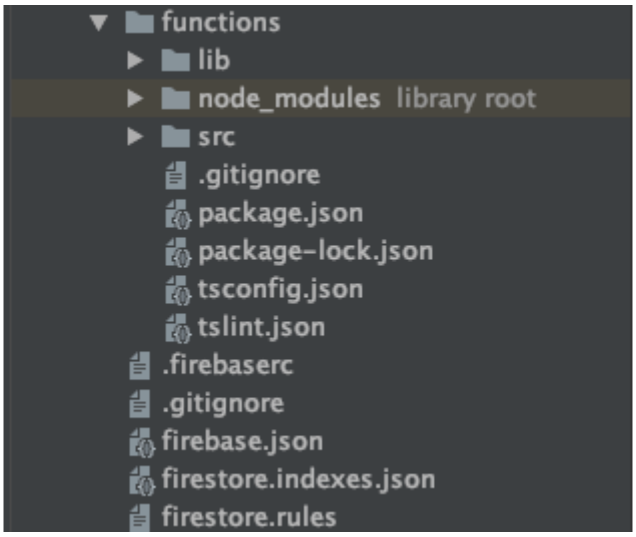
Setup serverless

To set up a serverless application, Node.js runtime environment and Google Cloud Shell must be installed. As discussed above, the Cloud Functions implementation is stored in the same repository with other microservice. The script below illustrates how to start a Cloud Functions project using Terminal. After running the script below, it introduces several steps to configure the serverless project, such as name, programming languages, code formatter and options to choose other GCP services.

```
cd functions
firebase init
```

Script 10. Script to init Google Cloud Functions with Typescript

The figure 12 shows a complete structure of a Google Cloud Functions project after initialization. As can be observed, there are several auto-generated scripts, which reduces time and effort to setup from scratch. Developers can start developing immediately by changing the index.ts, for Typescript projects, and deploy a new application using the script in the package.json file. Besides, developers can change the code formatter's rules, firestore's rules or firebase configuration by editing the files directly.



```
▼ functions
    ▶ lib
    ▶ node_modules  library root
    ▶ src
      .gitignore
      package.json
      package-lock.json
      tsconfig.json
      tslint.json
  .firebaserc
  .gitignore
  firebase.json
  firestore.indexes.json
  firestore.rules
```

Figure 12.Google Cloud Functions structure

By default, the auto-generated project does not include the local test environment. The script 11 below would install the correct environment and emulator for testing.

```
npm install --dev @firebase-functions-test @firebase/testing
```

Script 11. Script to install

Besides, Google allows the developer to create either HTTP Cloud Functions or Call Functions. HTTP Cloud Functions can be called as a normal HTTP network request, while Call Functions can be triggered in the client application.

Serverless implementation

With the rapid expansion of Google Cloud infrastructure, developers can optimize the high availability, scalability and network latency by deploying the services in a specific region or multi-regions. Multi-region service usually results in faster delay time, but more expensive billings. There is always a trade-off between several architecture options, and the architecture decision depends on not only technical requirements but also company resources and developer experience (Bass, Clements and Kazman, 2012). Therefore, the serverless implementation should be done in the same manner. The script below shows an example of creating a Cloud Functions which has interactions with Cloud Firestore. As can be observed, the cloud function would be hosted in europe-west1 region and has a timeout of 30 seconds. Besides, it uses the transaction for Cloud Firestore, which assures the concurrency of the database. Besides, the Cloud Functions can be configured to run a scheduled task. For the technical requirements, the Cloud Functions can receive 1000 concurrence requests at the same time, and all the functions run in their own instance; thus, it is highly available and scalable.

```
export const updateLobby = functions
    .runWith({
         timeoutSeconds: 30
     })
    .region('europe-west1')
    .https.onCall((data: LobbyRequest, context): Promise<string> => {
        return firestore.runTransaction(transaction => {
            const collectionRef = firestore.collection(data.collection)
            const documentRef = collectionRef.doc(data.orderId)
            return transaction.get(documentRef)
                .then(foundOrder => {
                    if (foundOrder.exists) {
                        const updatedData: OrderUpdatedData = {}
                        if (data.answererCoordinate) {
                            if (!isOrderLocation(data.answererCoordinate)) {
                                throw new functions.https.HttpsError('invalid-
argument', 'answererCoordinate format is incorrect')
                            }
                          updatedData.answererCoordinate         =         new
admin.firestore.GeoPoint(data.answererCoordinate.latitude,
data.answererCoordinate.longitude)
                        }
                        if (data.hostCoordinate) {
                            if (!isOrderLocation(data.hostCoordinate)) {
                                throw new functions.https.HttpsError('invalid-
argument', 'hostCoordinate format is incorrect')
                            }
                            updatedData.hostCoordinate          =          new
admin.firestore.GeoPoint(data.hostCoordinate.latitude,
data.hostCoordinate.longitude)
                        }
                        transaction.update(documentRef, updatedData)
                        return 'ok'
                    } else {
                        throw     new     functions.https.HttpsError('invalid-
argument', 'Could not find order by this id')
                    }
                })
                .catch(e => {
                    throw e
                })
        })
    })
```

Script 12.   An example of Google Cloud Functions

In addition, the Cloud Functions has a useful feature that benefits this case study: It can bind a listener to a specific Cloud Firestore collection or document. In other words, when there is a change in the Cloud Firestore, the Cloud Functions can be triggered. As can be observed, the function listens to every document in order-staging collection. If a document is updated, it will trigger the listener, and send the data to third-party. As a result, the client application can communicate with the client booking platform in a real time connection.

```
exports.updateOrderListener = functions
    .region('europe-west1')
    .firestore
    .document('/order-staging/{orderId}').onUpdate(async (change, context) => {
        try {
            return new Promise((resolve, reject) => {
                sendDataToFonecta(change.after.data())
                    .then(res => {
                        return resolve()
                    })
                    .catch(e => {
                        reject(e)
                    })
            })
        } catch (e) {
            throw e
        }
    })
```

Script 13.   An example of Cloud Functions listener

API Gateway configuration

For the API Gateway, it must be initialized and configured by the gcloud console. To initialize, the command line `gcloud beta api-gateway apis create` **`my-api`** `--project=`**`mvp-project`** should be run in the terminal or in the gcloud shell. The API Gateway can be configured by a yaml file, as shown in script 13.

```
swagger: '3.0'
info:
  title: my-api
  description: Sample API on API Gateway with a Google Cloud Functions backend
  version: 1.0.0
schemes:
  - https
produces:
  - application/json
paths:
  /hello:
    get:
      summary: Greet a user
      operationId: hello
      x-google-backend:
        address: https://europe-west1-mvp-project.cloudfunctions.net/helloGET
      responses:
        '200':
          description: A successful response
          schema:
            type: string
```

Script 13. An example of Cloud Functions listener

The script 14 below shows the script to deploy the API Gateway to the GCP using gcloud command line. As can be observed, the config, the api name and the project name along with the dedicated user account must be defined in the script.

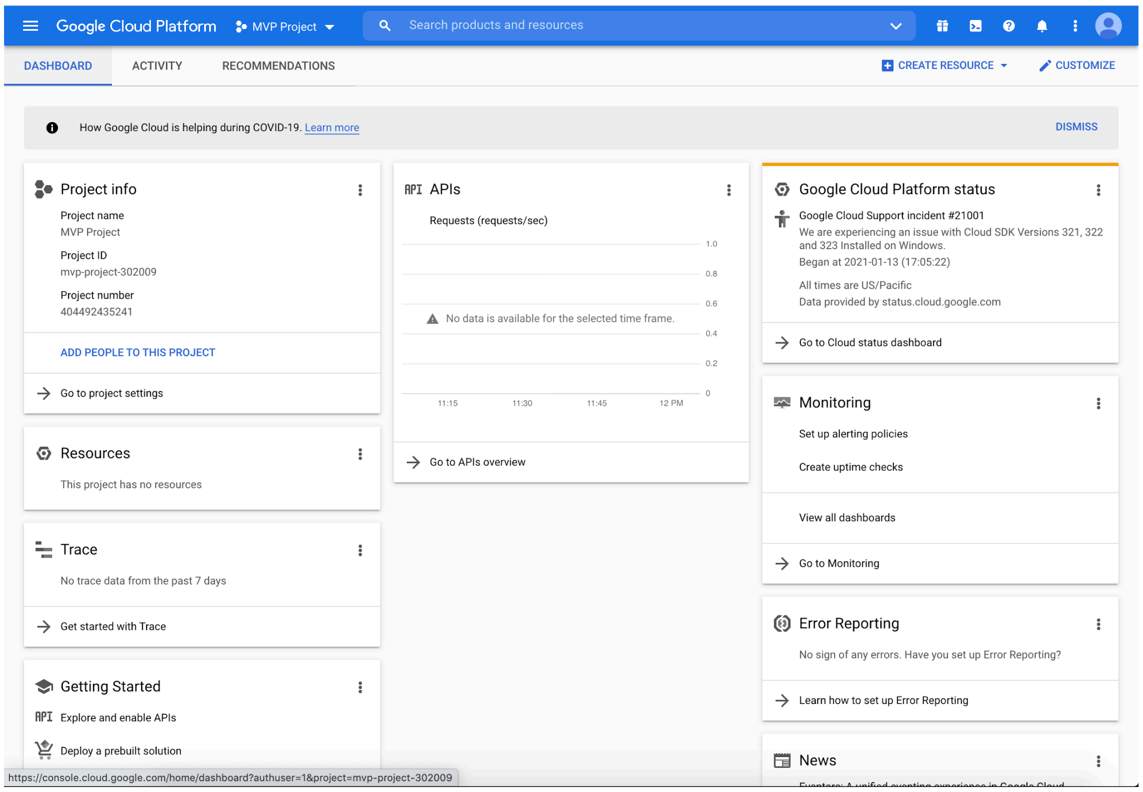
```
gcloud beta api-gateway api-configs create my-config \
  --api=my-api --openapi-spec=openapi2-functions.yaml \
  --project=mvp-project          --backend-auth-service-account=0000000000000-
compute@developer.gserviceaccount.com
```

Script 14. Script to init Google Cloud Functions with Typescript

Infrastructure setup

This part represents the project setup process in Google Cloud Console, starting from project creation to service host management and service configuration. Gcloud command-line interface is also used in some parts of the process.

The Google Cloud project can be created either in the Google Cloud Console or by the gcloud command lines: `gcloud projects create mvp-project`. After project creation, the project dashboard, as shown in figure 13, is visible and available to the admin users. In the project dashboard, admin can manage the users, projects, analytics, services and billings. The project dashboard informs general information about the project, which gives the overall insight for admin about the project's current status.



How Google Cloud is helping during COVID-19. Learn more

Project info

Project name
MVP Project

Project ID
mvp-project-302009

Project number
404492435241

ADD PEOPLE TO THIS PROJECT

Go to project settings

Resources

This project has no resources

Trace

No trace data from the past 7 days

Get started with Trace

Getting Started

Explore and enable APIs

Deploy a prebuilt solution

Requests (requests/sec)

No data is available for the selected time frame.

Go to APIs overview

Google Cloud Platform status

Google Cloud Support incident #21001
We are experiencing an issue with Cloud SDK Versions 321, 322 and 323 Installed on Windows.
Began at 2021-01-13 (17:05:22)

All times are US/Pacific
Data provided by status.cloud.google.com

Go to Cloud status dashboard

Monitoring

Set up alerting policies

Create uptime checks

View all dashboards

Go to Monitoring

Error Reporting

No sign of any errors. Have you set up Error Reporting?

Learn how to set up Error Reporting

News

https://console.cloud.google.com/home/dashboard?authuser=1&project=mvp-project-302009

Figure 13.Google Cloud Console project dashboard

Admin can add, edit or remove users from the application using the IAM & Admin setup. Besides, in the IAM dashboard, admin can restrict users to access and edit certain resources in the application, hence, it improves the security and accessibility of the application.

To enable App Engine, Cloud Functions and Cloud Firestore services, admin needs to add a billing account and enable the services manually in the Google Cloud Console. Then, the services can be seamlessly deployed to the services using gcloud command-line.

3.5 Evaluation

The development team successfully built a minimal viable product for the taxi booking app backend application, which is hosted in Google Cloud Platform. The backend application combines modern software architecture such as microservices and serverless and latest powerful technologies including Node.js, Express and Google Cloud services, resulting in high availability, scalability, interoperability, security and reliability. The outcome of the project meets the initial technical and business proposals which were decided at the first meeting with the client.

From the technical view, the application gave an insight about a different approach to the real time application development. The ultimate objectives of the solution are to reduce the company’s resources for maintenance, development and deployment process, bring the seamless experience to the developers, and deliver a scalable, highly available, secured and reliable backend application. With the advancement of cloud services, microservices and serverless are used to reduce the workload for the developers, avoid monolithic hell situations and save the company’s time and money. Meanwhile, Node.js, Express along with React and React Native provide a powerful framework to develop the full stack applications using a single programming language. Therefore, it removes the boundaries between traditional frontend and backend developers.

From the business view, the application fulfilled the requirements set from the initial meeting. The application is able to make a reliable real time connection between driver mobile applications and a third-party client taxi booking application, which is able to perform a full booking process. Moreover, company owners can manage their own drivers, taxis, orders, accountant work and price lists in a single application, while the admins can keep track of everything happening in the platform.

3.6 Future development

Although the first phase of the project was achieved, the current project architecture can be improved. With the revolution of continuous integration and continuous delivery (CI/CD), it removes mostly manual work for the deployment process, so it prevents human-errors and reduces the overall workload significantly. Hence, CI/CD should be added to the project. In addition, unit tests, integration tests and end-to-end tests should be added to the architecture, because it produces a bug-free application. In conclusion, the application architecture is still immature, so there are many rooms for improvements in the next phases.

4 Conclusion

By the detailed theoretical background and the concrete implementation for a case study, the aim of this study was to provide an approach to build a scalable, highly available, secured and reliable backend architecture for the real time application using Node.js, Express and Google Cloud Platform. With the revolution of the cloud computing, software development has changed to adapt with the new wave of technical innovations. Microservices and serverless have been adopted and proved as the new standard architecture in the market. Besides, cloud services have changed the way of developing, maintaining and deploying the services. Node.js and Express prove that the concept of single-threaded, non-blocking I/O works as effectively as other traditional multithreaded frameworks.

In conclusion, the thesis represents the insights for architecting a production-grade real time application using modern technologies. The project result satisfies the client's business goals and technical requirements. With the combination of modern architecture and Google Cloud Platform services, the project overall cost is cut by half, including the time and money spent to develop, maintain and deploy the services. However, there is still room for improvement in the later stage, ranging from testing to building a CI/CD pipeline to automate the deployment process. All in all, this thesis delivers an alternative, yet modern approach to develop a real time application, which tends to be more cost-efficient and developer friendly.

References

Roper. (2018). *Socket.io - The Good, the Bad, and the Ugly*. [Online]. Available at: https://dzone.com/articles/socketio-the-good-the-bad-and-the-ugly (Accessed: 31 December 2020)

Ilya. (2020). *Chat solutions: what to choose between Firebase, SendBird, Node.js + Socket.io?*. [Online]. Available at: https://medium.com/@forasoft/chat-solutions-what-to-choose-between-firebase-sendbird-node-js-socket-io-e0075e6ea408 (Accessed: 31 December 2020)

Chan. (2019). *The 10 most popular programming language, according to the Microsoft-owned Github*. [Online]. Available at: https://www.businessinsider.com/most-popular-programming-languages-github-2019-11 (Accessed: 6 February 2021)

Cooling. (2019). *The Complete Edition - Software Engineering for Real-Time Systems.* [Online]. Available at: https://learning.oreilly.com/library/view/the-complete-edition/9781839216589 (Accessed: 2 January 2021)

OpenJS Foundation. (2020). *About Node.js®.* [Online]. Available at: https://nodejs.org/en/about (Accessed: 31 December 2020)

OpenJS Foundation. (2020). *A brief history of Node.js.* [Online]. Available at: https://nodejs.dev/learn/a-brief-history-of-nodejs (Accessed: 31 December 2020)

Wilson. (2018). *Node.js 8 the Right way*. [Online]. Available at: https://learning.oreilly.com/library/view/nodejs-8-the/9781680505344 (Accessed: 31 December 2020)

Casciaro and Mammino. (2020). *Node.js Design Patterns - Third Edition.* [Online]. Available at: https://learning.oreilly.com/library/view/nodejs-design-patterns/9781839214110 (Accessed: 31 December 2020)

Pasquali and Faaborg. (2017). *Mastering Node.js - Second Edition*. [Online]. Available at: https://learning.oreilly.com/library/view/mastering-nodejs-/9781785888960 (Accessed: 31 December 2020)

Npm, Inc. (2020). *By the numbers.* [Online]. Available at: https://www.npmjs.com/. (Accessed: January 3 2021)

Meadow. (2018). *Single-threaded and Multi-threaded Processes*. [Online]. Available at: https://www.tutorialspoint.com/single-threaded-and-multi-threaded-processes (Accessed: 3 January 2021)

MDN contributors. (2020). *Express web framework (Node.js/JavaScript)*. [Online]. Available at: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs (Accessed: 3 January 2021)

MDN contributors. (2020). *Express/ Node introduction*. [Online]. Available at: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction (Accessed: 3 January 2021)

Hibbard et al (2020). *Your first week with Node.js, 2nd Edition*. [Online]. Available at: https://learning.oreilly.com/library/view/your-first-week/9781098122805/ (Accessed: 3 January 2021)

Chrome Developers. (2012). *MVC Architecture*. [Online]. Available at: https://developer.chrome.com/docs/apps/app_frameworks/ (Accessed: 3 January 2021)

Pacheco. (2018). *Microservice Patterns and Best Practices.* [Online]. Available at: https://learning.oreilly.com/library/view/microservice-patterns-and/9781788474030/ (Accessed: 4 January 2021)

Richardson. (2018). Microservices Patterns. [Online]. Available at: https://learning.oreilly.com/library/view/microservices-patterns/9781617294549 (Accessed: 4 January 2021)

McLarty et al. (2016). *Microservice Architecture*. [Online]. Available at: https://learning.oreilly.com/library/view/microservice-architecture/9781491956328 (Accessed: 4 January 2021)

Brown. (2016). *Beyond buzzwords: A brief history of microservices patterns.* [Online]. Available at: https://developer.ibm.com/articles/cl-evolution-microservices-patterns (Accessed: 5 January 2021)

Newman. (2021). Building Microservices, 2nd Edition. [Online]. Available at: https://learning.oreilly.com/library/view/building-microservices-2nd/9781492034018 (Accessed: 6 January 2021)

Google. (2020). *Migrating a monolithic application to microservices on Google Kubernetes Engine.* [Online]. Available at: https://cloud.google.com/solutions/migrating-a-monolithic-app-to-microservices-gke#designing_microservices (Accessed: 7 January 2021)

Clement. (2020). *Worldwide digital population as of October 2020.* [Online]. Available at: https://www.statista.com/statistics/617136/digital-population-worldwide (Accessed: 8 January 2021)

Venema. (2020). *Building Serverless Applications with Google Cloud Run*. [Online]. Available at: https://learning.oreilly.com/library/view/building-serverless-applications/9781492057086 (Accessed: 8 January 2021)

Katzer. (2020). *Learning Serverless.* [Online]. Available at: https://learning.oreilly.com/library/view/learning-serverless/9781492057000 (Accessed: 8 January 2021)

Dabit. (2020). *Fullstack serverless.* [Online]. Available at: https://learning.oreilly.com/library/view/full-stack-serverless/9781492059882 (Accessed: 8 January 2021)

Shilkov. (2021). *Comparison of Cold Starts in Serverless Functions across AWS, Azure, and GCP*. [Online]. Available at: https://mikhail.io/serverless/coldstarts/big3/ (Accessed: 8 January 2021)

TechMagic. (2020). *AWS Lambda vs Google Cloud Functions vs Azure Functions - What to Choose in 2020?.* Available at: https://medium.com/techmagic/aws-lambda-vs-google-cloud-functions-vs-azure-functions-what-to-choose-in-2020-6d5340b79d98 (Accessed: 8 January 2021)

Google. (2021). *Cloud Functions.* [Online]. Available at: https://cloud.google.com/functions (Accessed: 8 January 2021)

Google. (2020). *Cloud Firestore*. [Online]. Available at: https://firebase.google.com/docs/firestore (Accessed: 9 January 2021)

Google. (2020). *Choose a database: Cloud Firestore or Realtime Database*. [Online]. Available at: https://firebase.google.com/docs/firestore/rtdb-vs-firestore (Accessed: 9 January 2021)

Redis Labs. (2021). *Redis Home Page.* [Online]. Available at: https://redis.io/ (Accessed: 12 January 2021)

Google. (2021). *Redis Home Page.* [Online]. Available at: https://cloud.google.com/api-gateway/docs/about-api-gateway (Accessed: 12 January 2021)

PostgreSQL. (2021). *About.* [Online]. Available at: https://www.postgresql.org/about (Accessed: 12 January 2021)

Bass, Clements and Kazman. (2012). *Software Architecture in Practice, Third Edition*. [Online]. Available at: https://learning.oreilly.com/library/view/software-architecture-in/9780132942799/ (Accessed: 16 January 2021)