IMPLEMENTING MICROSERVICE ARCHITECTURE TO INDONESIA E-GOVERNMENT APPLICATION: STUDY CASE CIVIL REGISTRATION WEB APPLICATION

by

Rizal Hamdan Arigusti (韩充赞)

A Thesis

Submitted to the Faculty of Information Engineering Nanjing Xiaozhuang University

In Partial Fulfillment of the Requirements for the degree of

Bachelor of Software Engineering

School of Information Engineering

Fangshan, Nanjing

May 2020

To: Dean Xiangzun Zhao
School of Information Engineering

This thesis, written by Rizal Hamdan Arigusti, and entitled Implementing Microservice Architecture in Indonesia E-Government Application: Study Case Civil Registration Web Application, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this thesis and recommend that it be approved.

李青
Qing Li

吴海勇
Haiyong Wu

宋万里
Wan Li Song

Date of Defense: May 19, 2020
The thesis of Rizal Hamdan Arigusti is approved.

赵向军
Dean Xiangjun Zhao
School of Information Engineering

Nanjing Xiaozhuang University, 2020

Dedication

To my beloved family.

ACKNOWLEDGMENTS

Praise and deep gratitude to Allah SWT for the abundance of grace, and guidance of Him given to the author that made this thesis can be completed properly. Greetings and Shalawat may always be devoted to the Prophet Muhammad SAW. For the grace of Allah, the author finally able to complete the thesis entitled **Implementing Microservice Architecture To Indonesia E-Government Application: (**Study Case Civil Registration Web Application). This thesis is a requirement for achieving a Bachelor of Software Engineering at the School of Information Engineering, Nanjing Xiaozhuang University.

I would like to express my wholehearted thanks to my parents and my sister for their love and support in my entire life and particularly through the process of pursuing the bachelor degree in Islamic University of Indonesia and Nanjing Xiaozhuang University.

On this ocassion the author would like to say thank you profusely for all the help that has been given, either directly or indirectly during the preparation of this final thesis to complete. In particular gratitudes that are due to:

1. Professor Qing Li (李青) and Dr. Raden Teduh Dirgahayu S.T., M.Sc. as my lecturers and advisors who has provided guidance and encouragement in the preparation of this thesis.
2. Professor Xiangzun Zhao as Dean of the School of Information Engineering, Nanjing Xiaozhuang University.
3. Hendrik, S.T., M.Eng. as Head of Informatics Department, Islamic University of Indonesia.
4. Mrs. Sophie Mou who take care of the author when the author was in China.
5. Colleagues, friends, seniors, and juniors when the author was in Nanjing Xiaozhuang University and Islamic University of Indonesia.

6. Last but not least, those who cannot be mentioned one by one, who have helped and support me to finish this thesis.

The author realizes that this thesis has not been perfect, both in terms of material or presentation. The suggestions and constructive criticisms are expected in the completion of this thesis.

Recently the author hope that this thesis can provide things that are useful and add insight to the reader, and especially for the author as well.

Nanjing, May 2020

RIZAL HAMDAN ARIGUSTI

STUDENT ID : L18139906

DECLARATION OF ORIGINALITY

I, the undersigned below:

Name : Rizal Hamdan Arigusti
Student ID : 16523013

Hereby declared that the thesis I wrote with the title : IMPLEMENTING MICROSERVICE ARCHITECTURE TO INDONESIA E-GOVERNMENT APPLICATION: STUDY CASE CIVIL REGISTRATION WEB APPLICATION

1. Is truly a research written and conducted purely by myself, not copying from other published researches, and also not a result of plagiarism.
2. I will allow Nanjing Xiaozhuang University and Universitas Islam Indonesia to manage and keep the copy of this thesis, to be used as they deem necessary.

I made this statement of declaration with fully responsibility, and I'm willing to accept any consequences according to the rules and regulations should the statement above proved to be wrong in any way.

Nanjing, May 2020

METERAI TEMPEL
TGL 20
82416AHF504128493
6000
ENAM RIBU RUPIAH

Rizal Hamdan Arigusti

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Developing e-government applications becomes one of the main concerns of the Indonesian Government. However, there are some issues during the implementation of these applications. These issues are low-level availability, the unreliability of e-government services, and inflexibility scaling. This research conducted to propose a method to fix these issues. This method was implementing microservice architecture to one of the e-government applications that is Civil Registration Web Application.

To implement the microservice architecture, the author did system requirements analysis and business domain analysis. Then the author designed the microservice architecture by considering what services needed to run and how every service communicates with each other. After that, the author developed all the services in the microservices architecture and run them on the Kubernetes cluster. The result showed the application can run smoothly with its microservice architecture. The result also showed that all services are more available and scalable when they were running on the Kubernetes cluster.

Keywords: E-Government, Microservice, Kubernetes, Civil Registration Web Application

CHAPTER 1. INTRODUCTION

1.1 Background

E-government is an implementation of ICT (Information and Communication Technology) in public services to make them more accessible, accountable, and effective (Prahono and Elidjen, 2015). It can give effectiveness and efficiency in delivering public services and will make some stakeholders feel satisfied (Sabani, Deng and Thai, 2019). As a result, countries across the globe are developing and improving their e-government applications.

Following the trend, developing and improving e-government applications also has become one of the main concerns of the Indonesian Government. From 2014 to 2019, the Indonesia government committed to spending US$6.78 billion in the e-government development program (Sabani, Deng and Thai, 2019). However, researchers found performance problems during the implementation of e-government in Indonesia. The Low-level availability, the unreliability of e-government services, and the quality of application security become the main problems of Indonesia's e-government (Sabani, Deng and Thai, 2019). Factors that cause these problems are a lack of infrastructure and a lack of human resource quality (Prahono and Elidjen, 2015).

From a technical perspective, some techniques can address these e-government issues. From the non-functional requirement aspects, a method like changing the application architecture can be one of the solutions. Currently, two common application architectures used by many developers around the world are monolith architecture and microservice architecture.

In monolith architecture, developers bundle every application tier (user interface, business logic, and data access) and third-party modules into one unit. This architecture gives simplicity in development and deployment, especially in small-scale applications. However, this architecture has some limitations. One of the limitations is coming from the availability aspect. Since an application is a bundled of modules, if there is some memory leak or bug, it can shut down the entire application (Kharenko, 2019).

Microservice architecture comes to address these limitations. Fowler and Lewis describe microservice is an approach to developing a single application as a suite of small services (Fowler and Lewis, 2019). Newman also describes microservice as small, autonomous services that work together (Newman, 2015). In microservice, every service is independently deployable. It makes continuous delivery and deployment easier (Richardson, 2019). It also makes the codebase more straightforward to understand and modify by developers since each service is small (Richardson, 2019).

Implementing microservice architecture into e-government can solve or at least minimize availability issues in the application. In this architecture, independently deploying every service becomes one of its benefits. Deployment of a service does not require the availability of other microservices. Once running, if one required microservices is not available, the application still works even though partly. This benefit becomes the source of other benefits such as reliability and fault isolation (Soldani, Tamburri and Van Den Heuvel, 2018). So microservice architecture can improve the availability of our e-government applications.

Another benefit of Microservices is it allows each service to be independently scaled to meet demand for the application feature it supports. This enables teams to right-size infrastructure needs, accurately measure the cost of a feature, and maintain availability if a service experiences a spike in demand ("What are Microservices? | AWS", 2020). This benefit can make the e-government applications become more flexible to scale.

These microservice architecture's benefits motivate the author to develop an e-government application in which its architecture is using Microservice Architecture. Civil Registration Web Application is the e-government application that will be the study case in this research. The author chose the Civil Registration Web Application to be a study case because Civil Registration is one of the most vital government services.

1.2 Problem Identification

The author identified three questions that are needed to answer after this research has completed. These three questions are:

1. How to design and implement microservice architecture for a civil registration web application?
2. How to develop a civil registration web application with microservice architecture?
3. How to improve the availability of the civil registration web application and make it more flexible to scale?

1.3 Objectives

Objectives of this research are listed below:

1. Improving the availability of the e-government web application and make it more flexible to scale by implementing the microservice architecture.
2. Developing a civil registration web application as a study case for the microservices architecture.

1.4 Research Scope

The scope of this study are listed below:

1. This research was more focusing on the back-end development and implementation of some microservice tools and technologies to the system. The system analysis was also only done by doing observation and literature review.
2. This research was not including the implementation of the system to the real government institution.
3. This research did not cover all the business processes provided by the civil registration institutions in Indonesia.

1.5 Research Benefit

One of the benefits of microservice architecture is every service in the system is loosely coupled. It means every service has a low dependency on other services. If one service is shutting down, it will not affect other services. Another benefit of microservice architecture is every service can be flexible to scale. If the demand for one service is high, developers only need to scale up one service. there is no need to scale up the entire application. So from these two benefits, the implementation of microservice architecture hopefully can address or at least can minimize the availability issues of an e-government application in Indonesia and also makes it more flexible to scale.

1.6 Methodology

There are some software development methodology models that currently used by developers around the world. These models include Waterfall, Agile SCRUM, Spiral, etc. In this research, the author chose Waterfall as the software development methodology. The author used this model because all functional requirements of the civil registration web application were relatively stable. The waterfall model also divides the whole development process into several different phases (Analysis, Design, Development, and Testing) that executed sequentially.

Based on the waterfall model, the author divided the whole research and development process into four phases that executed sequentially. These phases are:

1. *Analysis* – in this phase, the author did requirement analysis and business domain analysis based on previous literature and existing systems. The result of requirement analysis and business domain analysis became a guide for the author to implement some features and also to design microservice architecture for the system.
2. *System Design* – in this phase, the author designed the architecture of civil registration web application based on the microservice architecture principles.
3. *Implementation* – in this phase, the author wrote some codes for every service in the microservice architecture and run it in the Kubernetes cluster. The purpose of running every service in Kubernetes is to manage every service and make them more available and scalable.
4. *Testing* – in this phase, the author used some software testing techniques in order to check whether the system can fulfill all the requirements that had defined before or not.

CHAPTER 2. THEORITICAL BASIS

2.1 Fundamental Theory

2.1.1 E-Government

E-Government is the use of information and communication technology (ICT) by government institutions to achieve better communication between government to government (G2G), government to business (G2B), and government to customers (G2C) (Prahono and Elidjen, 2015). The implementation of e-government can improve accessibility, accountability, effectiveness, efficiency, and transparency during government activities. By using e-government, some institutions can deliver their public services effectively and efficiently (Sabani, Deng and Thai, 2019).

UNDP divides the development level of e-government in a country into five stages (Siau & Long, 2005). These five stages are listed below.

1. *emerging stage* – e-government only provides static information about their institution's profile and services in an online system. There is no interaction and transaction between public citizens and the institution's online system.
2. *enhanced* stage – e-government has started to become more dynamic and regularly updated their information.
3. *interactive* stage – e-government become more interactive and more sophisticated. For instance, citizens can download some forms and complete the form manually.
4. *transactional* stage – in this stage, e-government can provide two-way communication and secure transaction between the public and the online system. For instance, online civil registration portal, online taxes portal, etc. Most of the developing countries currently are in this stage (Sabani, Deng and Thai, 2019).

5. *seamless* stage – this is the final stage where all of the e-governments are integrated. In this stage, citizens can access all public services in a one-stop portal.

There are some instances of e-government applications in Indonesia. One of them is the civil registration web application named *Sistem Informasi Administrasi Kependudukan* which becomes the study case of this paper. *Sistem Informasi Administrasi Kependudukan* is an information system that implements information and communication technology for facilitating civil information management (Undang Undang Republik Indonesia No.24 Tahun 2013, 2013). Some outputs of this system are civil identification number, family card, civils card, birth certificate, death certificate, etc ("Sistem informasi administrasi kependudukan", 2020).

2.1.2 Microservice Architecture

There are so many solutions to improve the quality of software. From the non-functional requirement perspective (availability, reliability, maintainability, and scalability), developers can use mirroring/DRC/cloud techniques to improve the software's non-functional quality. Developers also can use a certain algorithm to improve system performance (Jayanto, 2017).

Microservices architecture also becomes one of these solutions to improve the non-functional quality of software (Jayanto, 2017). According to Google trends, microservices has become one of the growing concepts since 2014 (Balalaie, Heydarnoori and Jamshidi, 2016). Many global big companies, such as Google, Amazon, and Netflix have adapted microservices architecture into their product (Stenroos, 2019). As a comparison and for better understanding, we also need to understand one of the traditional architecture which is still used by some companies. This architecture is monolith architecture..

Monolith architecture is a traditional architecture where entire application modules are bundled and built into one unit. This architecture is designed for running solely on one single instance of computation (Götz et al., 2018).

Because of monolith architecture build all the modules and program into one unit bundled application, this kind of architecture is easy to develop and easy to deploy. However, some drawbacks can make this architecture is not suitable for big-scale enterprise application. The first drawback of this architecture is it lacks flexibility. For instance, if numbers of users who send requests to the application are high and make the application can't handle it with one instance, developers can't scale up horizontally. It only supports to scale up vertically instead. Some other drawbacks like dependency hell, difficult to maintain, changing one module require to reboot the entire application, and locking developers to use one language and one framework (Dragoni et al., 2017).

Different from monolith architecture, microservice architecture is a cloud-native architecture that consists of multiple small services. Each service is independent to deploy and also potentially built on different platforms and technology stack. This architecture is running on its process while communicating through a lightweight communication protocol like RESTful or RPC-based APIs (Balalaie, Heydarnoori and Jamshidi, 2016).

Microservice architecture has some benefits. One of the key benefits of microservice architecture is it supports independently scaling up of each service, so microservice provides the possibility to improve scalability and flexibility to application development (Wan, Guan, Wang, Bai and Choi, 2018). Another benefit of microservice is if there is some failure in one component, it will not affect other components in the system (Jayanto, 2017).

2.2 Microservice Tools

2.2.1 Docker Container

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another ("What is a Container? | Docker", 2020). It also provides operating-system-level virtualization under Linux kernel, so it can isolate and control resources for a set of processes. Virtualization in the container is different from the virtual machine. While a virtual machine emulates the physical hardware, the container only virtualizes the operating system level so that it is lightweight with less overhead (Amaral et al., 2015).

A container needs a tool named container engine to run its processes. Docker is one of the container engines that currently used by many IT companies. It was launched in 2013 as an open-source project and it can run in the various operating system such as Linux, Windows, and Mac. Some cloud computing services like Amazon Web Service and Microsoft Azure also provide Docker in some of their products.

Docker container is an excellent match for implementing microservice architecture (Amaral et al., 2015). Every service's code and dependencies are wrapped into one or more instances of container and run them on various platform. Since a container is lightweight, so every service can boot in very fast.

2.2.2 Kubernetes

Kubernetes (also known as k8s or "kube") is an open source container orchestration platform that automates many of the manual processes involved in deploying, managing, and scaling containerized applications ("What is Kubernetes?", 2020).

("Kubernetes vs. Docker: What Does It Really Mean? | Sumo Logic", 2020) explained that Kubernetes is made up many components which these components all talk to each other through the API server. Every components operates its own function and then exposes metrics, that can be collected for monitoring later on. All of these components can be divided into three main parts.

- *The Control Plane/The Master* – is the orchestrator. In Control Plane parts, There are multiple components that help facilitate the container orchestration. For instance, Etcd for storage, the API server for communication between components, the scheduler which decides which nodes pods should run on, and the controller manager, responsible for checking the current state against the desired state .
- *Nodes* – are where containers actually get deployed to run. Nodes are the physical infrastructure that your application runs on, the server of VMs in your environment.
- *Pods* – are the lowest level resource in the Kubernetes cluster. A pod is made up of one or more containers. When defining the cluster, limits are set for pods which define what resources, CPU and memory, they need to run. The scheduler uses this definition to decide on which nodes to place the pods.

2.3 Related Works

According to our best knowledge, some research focused on developing civil registration web application. One example is research by Dedi, Iqbal, and Fahroji. They developed a civil registration web application for the local area institution office in Indonesia (Iqbal, Fahroji & Dedi, 2019). They also claimed that most of the functional requirements of their application were working. However, there was no implementation of microservice architecture in their works, so our works in this paper could be an improvement for their works.

Different from Dedi, Iqbal, and Fahroji, Jayanto implemented microservice architecture for his research. In his research (Jayanto, 2017), he designed and developed an online public complaint system which is also categorized as an e-government application. He implemented microservice architecture by using Java Spring Boot Framework and he also claimed that most of the functional requirements in his application were working. Another work that implemented microservice architecture is research by Sani, Fillah, Tjahyanto, and Suryotrisongko. They implemented microservice architecture on the E-Incubator application. According to their research, E-Incubator is an online incubation and investment application. Same as Jayanto's works, they also claimed that most of the functional requirements in their application were working. They also added that microservice architecture could make their application gave a faster response to the users' requests. However, Both Jayanto and Sani (with his co-authors) works didn't use any container technology for their developed application. In this research, we were not only implementing microservice architecture but also using container technology to develop our application.

CHAPTER 3. ANALYSIS AND DESIGN

This chapter explained some results of the analysis phase and design phase during the civil registration web application development. In the analysis phase, the author did some system requirement analysis including its actors, functional requirements, and non-functional requirements. The author also did business domain analysis to guide the author in designing microservice architecture during the analysis phase. In the design phase itself, all of the analysis results became a guide for the author to did a system design process.

3.1 System Requirement Analysis

Before implementing some codes into the system, the author needed to do system requirement analysis. System requirement analysis is used for getting requirements of the system in more detail. These requirements include users, functional features, and non-functional features of the civil registration web application.

The author did system requirement analysis by doing observation of some similar systems or applications. The first observation that the author did is accessing lampid.surabaya.go.id which is a civil registration web application that is already developed by the Surabaya government. The next observation that the author did is reviewing some literature that discuss how to develop civil registration web application. For instance, the author reviewed a paper titled by *Sistem Informasi Administrasi Kependudukan Berbasis Web di Kelurahan Sangiang Jaya*. From this paper, the author got some basic functional requirements that should be included in the civil registration web application. The author also did observations by reviewing some of Indonesia's constitutions. Last but not least, the author did a literature review to some publications related to microservice architecture so the author can know how to implement a good microservice architecture to the civil registration web application. From observations, the author got the requirements of the system.

3.1.1 Actor Identification

Actors are users that will interact with the civil registration web application. From the observation process, the author identified two kinds of actors for the civil registration web application. These actors are:

1. *Administrator* – People who are employed by government institutions to insert and verify citizens` data.
2. *Citizens* – The main user of the system. This actor will insert populations and vital events (birth, marriage, divorce, and dead) data into the system based on their experience.

3.1.2 Functional Requirement

There were some functional requirements identified by the author after the observation process had done. All of these functional requirements will be explained in Table 3.1 below.

Table 3.1 Functional Requirements for the Application

Code	**Functional Requirement Description**
FR-01	System should support administrator to insert citizens identification card data and system will automatically create the citizens` account
FR-02	System should support administrator to verify marriage certificate data that are input by citizens
FR-03	System should support administrator to verify and update family card data
FR-04	System should support administrator to verify birth certificate data that are input by citizens
FR-05	System should support administrator to login their account

Table 3.1 Functional Requirements for the Application Continue

Code	**Functional Requirement Description**
FR-06	System should support administrator to register other administrators` account
FR-07	System should support citizens to login their account that has created by administrator
FR-08	System should support citizen to see profile data for their identification card
FR-09	System should support citizen to register marriage certificate
FR-10	System should support citizen to see data in their family card.
FR-11	System should support citizen to register birth certificate

Based on all the functional requirements in Table 3.1, the author identified eleven use cases that are described in Table 3.2.

Table 3.2 Identified Use Case Table

Code	**Use Case**	**Functional Requirements**	**Actor**
UC-01	Insert citizens identification card data	FR-01	Administrator
UC-02	Verify citizen’s marriage certificate data	FR-02	Administrator
UC-03	Verify and update citizen’s family card data	FR-03	Administrator
UC-04	Verify citizen’s birth certificate data	FR-04	Administrator
UC-05	Register other administrators’ account	FR-06	Administrator
UC-06	Login as Administrator	FR-05	Administrator
UC-07	Login as Citizen	FR-07	Citizens

Table 3.2 Identified Use Case Table Continue

Code	**Description**	**Functional Requirements**	**Actor**
UC-08	See profile data in identification card	FR-08	Citizens
UC-09	Register and see marriage certificate	FR-09	Citizens
UC-10	See family card data	FR-10	Citizens
UC-11	Register and see birth certificate	FR-11	Citizens

From the actors and the use cases that had identified, the author modeled a use-case diagram. Figure 3.1 shows the use-case diagram that had modeled by the author.

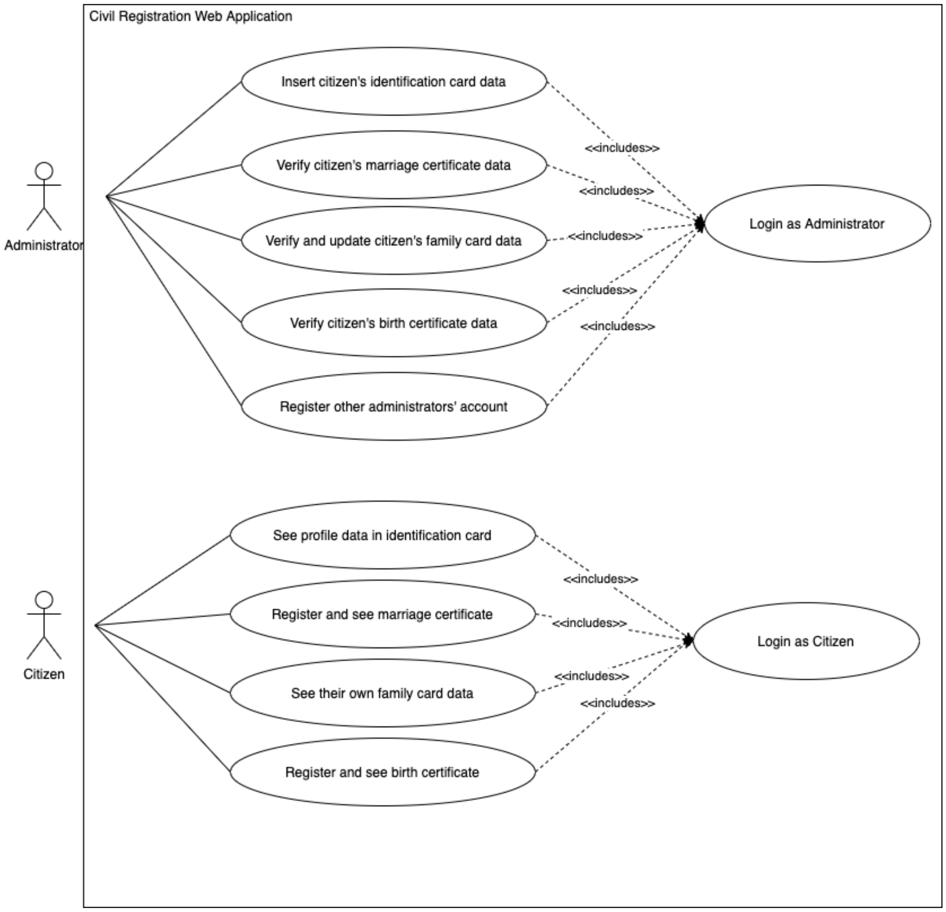


Figure 3.1 Use Case Diagram

3.1.3 Non-functional Requirement

In this research, the author was more focusing on implementing microservice architecture into the developed application. The author implemented the microservice architecture to make the application more available and flexible to scale. So to achieve this purpose, the application should have some non-functional requirements. These non-functional requirements are:

Table 3.3 Non-functional Requirements for the Application

Code	Description
NFR-01	System should have multiple services that are connected each other with some network protocol
NFR-02	Every service in the system should be independently developed and deployed.
NFR-03	Every service in the system should be flexible to scale.
NFR-04	Every service in the system should have high runtime availability

3.2 Business Domain Analysis

In this research, the author used microservice architecture to develop the civil registration web application. This application consisted of multiple small services that can connect through a network communication protocol. Every service was also independently developed, deployed, and scaled so it might fulfill non-functional requirements that had discussed before.

The author designed a microservice architecture based on Domain-Driven Design concept and also based on the use cases that had discussed before. In the Domain-Driven Design concept, the author needed to identify the business domain that will be supported by the web applications and also all of its subdomain.

The author identified that civil registration became the business domain that supported by the developed web application. Then based on the use cases, the author identified there were some subdomain that were consisted in this business domain. Based on the UC-01, UC-07, and UC-08, the application will have citizen registration process and this data registration process will be used for other use cases such UC-02, UC-03, etc. So, the author identified that this **citizen registration** becomes one of the subdomain in the civil registration business. Then, UC-02 and UC-09 also have their own registration process that becomes the subdomain in the civil registration business. This subdomain is **marriage registration**. UC-03 and UC-10 also shows that the application will have another subdomain named **family registration**. The author then identified another subdomain name **birth registration** based on the UC-04 and UC-11. Last identified subdomain is **admin registration** that the author identified based on the UC-05 and UC-06. Then finally based on the Domain Driven Design concept, every subdomain will have at least one independent service that will handle their business process.

3.3 System Design

After the author analyzed system requirements, the author did system design for the application. In system design itself, the author was more focusing on how to design the microservice architecture. So, the author did an architectural design for the system. This architectural design used for guiding authors when authors implemented some codes to create civil registration web application.

3.3.1 Architectural Design

In architectural design, the author designed a system architecture for the application. The author designed the system architecture based on the result of business domain analysis and system requirement analysis. The system architecture that the author designed was also implementing the microservice architecture where an application consists of multiple small services that communicate through a computer network protocol like HTTP or AMQP.

From all the subdomains that were identified in business domain analysis, the author decided that every subdomain has its service in the web application. These services will be connected to each other and serve the request from the client. Every service also had its database whether it would be a relational database or NoSQL database.

Based on the system requirement analysis results, there is an actor named Administrator. Administrators are responsible for inserting or verifying some citizens' data. So, the system should provide an account for the Administrator. From this requirement, authors decided to create a service for managing administrator account and it also will have its database to save the administrator's data.

Based on the system requirement analysis results, there was also an authentication rule for the system. For instance, only an authenticated administrator can insert or verify citizens' data. So, the author decided that the application should have its authentication service, and this authentication service will connect to the Administrator database..

The author also chose one of the microservice patterns to design the microservice architecture. This pattern is the API Gateway pattern. In the API Gateway pattern, the author need to make an API Gateway as one of the services, then this API Gateway became a medium between client to every service and also became a medium between one service to another service. API Gateway also can be used for checking whether a request from a client is the authenticated and authorized one or not.

In designing a microservice architecture, the author needed to choose what kind of inter-process communication that will be used for the application. this inter-process communication was used for one service to connect other services. In this research, the author chose to use Request/Response Communication with REST technology. This Request/Response was used for communication between API Gateway to every service and between API Gateway to every Client User Interface.

In this research, the author also chose to use Message-Based Communication with message queue/broker technology. This Message-Based Communication was used for publish/subscribe communication among services. So it could make all data in the application became more consistent and also reduce the Request/Response Communication among services.

In the publish/subscribe communication, there are two message event that will be implemented in the application. The first message event is marriage-event. This event is happened when there is a marriage record that the administrator verify. After that, the marriage service will publish a message event to the message broker then citizen service and family service will subscribe it. Citizen service subscribe this event to update the marriage status data in the one citizen record and family service subscribe this event to create a new family record that will automatically add the people who married in the family members data of the record. Since the family service will create a new record based on this event, there is no need to make a POST request to the family service REST API.

Another message event that the author will implement is birth-event. This event is happened when there is a birth record that the administrator verify. When this event is happening, birth service will publish it to the message broker then the author will make family service and citizen service subscribe it. Family service subscribe this event to add the new baby into the family members data in the one family record and citizen service subscribe this event to create a new citizen record of the new baby.

After the author decided what service that needed to create, what pattern that was used, and what inter-process communications were used, the author designed the microservices architecture diagram that can be seen on Figure 3.2.

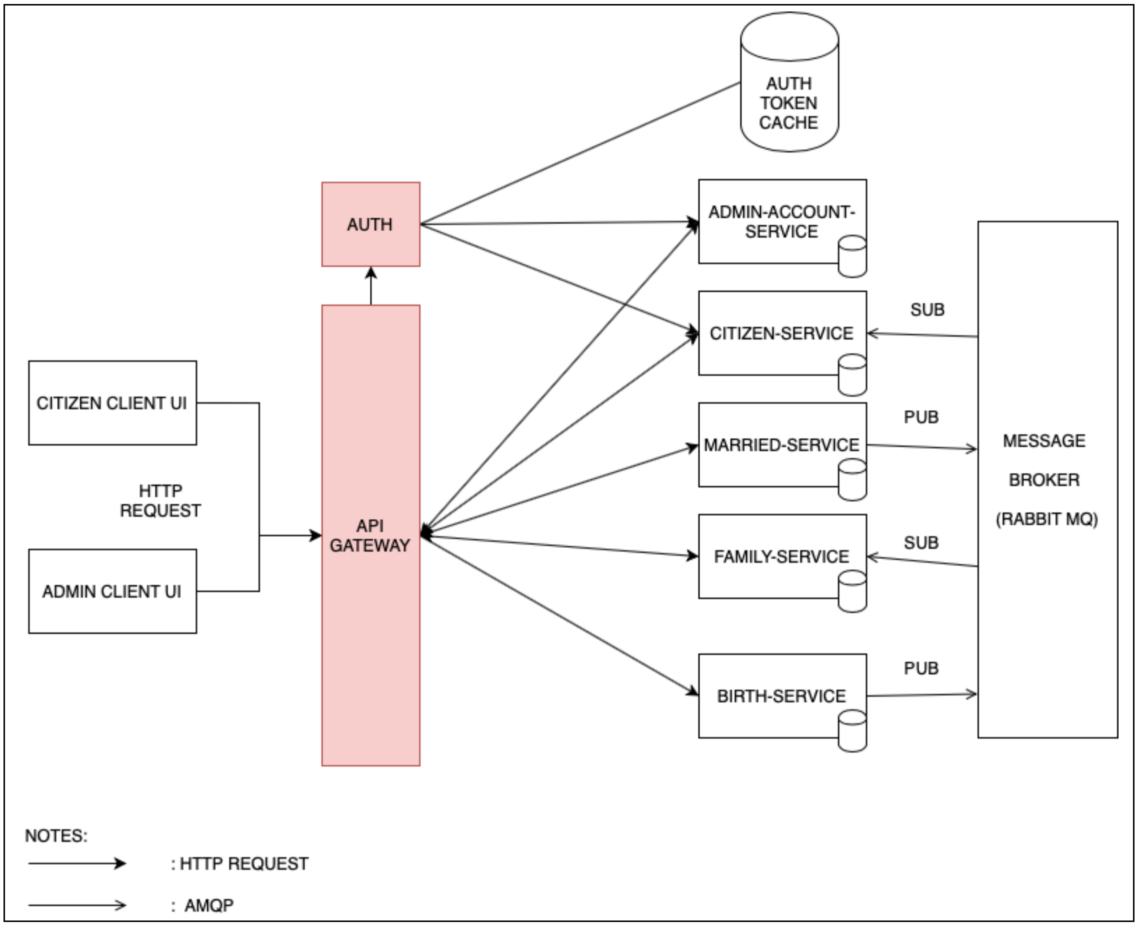


Figure 3.2 Microservice Architecture Diagram for the Application

CHAPTER 4. IMPLEMENTATION AND TESTING

This chapter explained how the author implemented and tested the civil registration web application. There were some technologies, tools, and libraries used by the author. The author used three programming languages to create some back-end services, which one service only created with one programming language. For the database, the author used MySQL, MongoDB, and Redis. Every database connected to one or more service(s). Then the author used Kubernetes as a container orchestrator with Docker as its container engine. Last but not least, the author also used some external libraries to support some functionalities in every service. The author explains the implementation phase results in REST API routing rules that exposed on every backend service, some screenshots of UI from frontend services, and some screenshots of Kubernetes objects in the dashboard. In the testing phase, the author tested the whole application based on the use-case that had defined. The author also tested the Kubernetes cluster with some test cases to make sure the availability and scalability of the application.

4.1 Back-end Service Implementation

Based on the microservice architecture diagram, there are six back-end services and one API Gateway that needed to implement. Every service was implemented by the author to expose a REST API and connect them to the API Gateway. Then the client will make an HTTP request to the application through the API Gateway whenever they want to access the application.

The author used different programming languages, framework, and databases for every back-end service that were developed. The author did this to achieve one of the benefit of microservice architecture that is flexibility in using technology stack. For more detail, these programming languages, frameworks, and databases usage are showed in Table 4.1 below.

Table 4.1 Programming Language, Framework, and Database Usage

Service	**Programming Language**	**Framework**	**Connected Database**
API Gateway	Node JS	Express JS	Redis
Citizen-service	Golang	-	MySQL
Marriage-service	Golang	-	MongoDB
Family-service	Golang	-	MongoDB
Birth-service	Golang	-	MongoDB
Admin-account-service	Python	Flask	MongoDB
Auth-service	Python	Flask	MongoDB, Redis

4.1.1 Implementation of Citizen-service

Citizen-service was a service responsible for managing citizens' profile data such as identification number (NIK), full name, date of birth, etc. This service exposed a REST API and connected it to the API Gateway. Whenever the client wants to access this service, the client needs to make an HTTP request to the API Gateway by specifying the URL and the HTTP Method (GET, POST, PUT, DELETE). Then, the API Gateway will check whether the request is authenticated or not. If the request is authenticated, API Gateway will send the request to the Citizen-service then the service will send a response to the client through the API Gateway.

The implementation result of this service is a REST API that can be accessed by the client by specifying the URL and HTTP Method. All the URL and HTTP Methods that were exposed in this service are shown in Table 4.2 below.

Table 4.2 Citizen-service REST API Exposed URL

No	**URL**	**Method**	**Usage**
1	http://ip.address/api/v1/citizens	GET	Getting all the citizens record from database and send them through a HTPP response.
2	http://ip.address/api/v1/citizens/{NIK*}	GET	Getting one citizen record that the value of NIK column is {NIK*} then send it through a HTTP response.
3	http://ip.address/api/v1/citizens	POST	Inserting one citizen record to the database and send HTTP OK response status if it is success.
4	http://ip.address/api/v1/citizens/auth	POST	Authenticating citizen account. If authentication is success, service will send JWT token to the client.
5	http://ip.address/api/v1/citizens/{NIK*}	PUT	Updating one citizen record that its NIK value is {NIK*} then send HTTP OK response status if it is success.
6	http://ip.address/api/v1/citizens/verify/{NIK*}	PUT	Changing verified status value of one record into “True”. This one record should be a record that its NIK value is {NIK*}.

Table 4.2 Citizen-service REST API Exposed URL Continue

No	URL	Method	Usage
7	http://ip.address/api/v1/citizens/{NIK*}	DELETE	Deleting one citizen record in database that its NIK value is {NIK*}.

Based on the microservice architecture diagram that had defined before, Citizen-service connected to its database. The author decided that Citizen-service will use the MySQL database and only have one table which is the citizen table. In the citizen table, some data needed to record such as NIK, full name, sex, etc.

The author used one of ORMs that can connect the Citizen-service to the MySQL database. This ORM is *gorm*. With *gorm*, the author only needed to define all the citizen table columns in a struct data type then do migrations to the MySQL database. Querying to the MySQL database is also easier when the author used *gorm*. The author only needed to call a function that had already provided by *gorm*.

Because the author decided that every service should connect its database and only responsible for one kind of data. So, the author connected the Citizen-service to the RabbitMQ message brokers. From RabbitMQ, Citizen-service subscribed to some event channels to make the citizens' profile data in the database more consistent.

4.1.2 Implementation of Marriage-service

Marriage-service was a service responsible for managing citizens' marriage data such as married certificate number, husband name, wife name, date of marriage, etc. Same as Citizen-service, Marriage service also exposed a REST API and connected to the API Gateway. Based on the microservice architecture, whenever clients need some data from Marriage-service, they need to make a request to the API Gateway. If the request can pass the authentication checking process in API Gateway, it will be forwarded to the Marriage-service. Then last, Marriage-service will send its response to the client through the API Gateway.

The result of Marriage-service implementation is a REST API that can be accessed by the client by specifying the URL and HTTP Method. All the URL and HTTP Methods that were exposed in this service are shown in Table 4.3 below.

Table 4.3 Marriage-service REST API Exposed URL

No	**URL**	**Method**	**Usage**
1	http://ip.address/api/v1/married	GET	Getting all the marriage record from database and send them through a HTPP response.
2	http://ip.address/api/v1/married/{number*}	GET	Getting one marriage record that the value of married certificate number or registration number field is {number*} then send it through a HTTP response.
3	http://ip.address/api/v1/married	POST	Inserting one marriage record to the database and send HTTP OK response status if it is success.

Table 4.3 Marriage-service REST API Exposed URL Continue

No	URL	Method	Usage
4	http://ip.address/api/v1/married/verif/{number*}	PUT	Changing verified status value of one record into "True". This one record should be a record that its married registration number value is {number*}.
5	http://ip.address/api/v1/married/{number*}	DELETE	Deleting one married record in database that its married certificate number value is {number*}.

Based on the microservice architecture diagram, Marriage-service will have its own database. The author chose MongoDB as the database for this service. This database has one collection that is *married-regis*. In *married-regis*, some data needed to record such as married certificate number, registration number, husband name, wife name, etc.

The author used one of Golang external libraries as the MongoDB driver which its name is *go-mongo-driver*. *go-mongo-driver* not only used for connecting the Marriage-service to the MongoDB database but also used for saving, updating, and deleting one or more record(s) in MongoDB collection. The author also defined the married-regis collection schema by using go mongo-driver.

The author also made the Marriage-service able to publish an event to the RabbitMQ. Marriage-service will publish this event when the Administrator verified a marriage record in the database. Other services such as Citizen-service and Family-service subscribed to this event in RabbitMQ then they will do their business logic whenever the event is published.

4.1.3 Implementation of Family-service

Family-service was a service responsible for managing citizens' family data such as the family identification number, head of household, family members, etc. Family-service also exposed a REST API and connected it to the API Gateway. Based on the microservice architecture, whenever clients need some data from Family-service, they need to make an HTTP request to the API Gateway. If the request can pass the authentication checking process in API Gateway, it will be forwarded to the Family-service. Then Family-service will send its response to the client through the API Gateway.

The result of Family-service implementation is a REST API that can be accessed by the client by specifying the URL and HTTP Method. All the URL and HTTP Methods that were exposed in this service are shown in Table 4.4 below.

Table 4.4 Family-service REST API Exposed URL

No	**URL**	**Method**	**Usage**
1	http://ip.address/api/v1/family	GET	Getting all the family record from database and send them through a HTPP response.
2	http://ip.address/api/v1/family/{number*}	GET	Getting one family record that the value of married certificate number or registration number field is {number*} then send it through a HTTP response.

Table 4.4 Family-service REST API Exposed URL Continue

No	**URL**	**Method**	**Usage**
3	http://ip.address/api/v1/family/verify/{number*}	PUT	Changing verified status value of one record into "True". This one record should be a record that its married registration number value is {number*}.
4	http://ip.address/api/v1/family/add/{number*}	PUT	Adding one family member to one record of family in database. this one family record is a family that its family card number is {number*}
5	http://ip.address/api/v1/family/update-location/{number*}	PUT	Updating location data of one family record. this one family record is a family that its family card number is {number*}
6	http://ip.address/api/v1/family/{number*}	DELETE	Deleting one married record in database that its married certificate number value is {number*}.

Based on the microservice architecture diagram, Family-service will have its database. The database that will be used in this service is MongoDB. This database has one collection which is *family-regis*. In *family-regis*, some data needed to record such as family card number, head of household, family members, address of the family, etc.

The author used one of Golang external libraries as the MongoDB driver which its name is *go-mongo-driver*. *go-mongo-driver* not only used for connecting the Family-service to the MongoDB database but also used for saving, updating, and deleting one or more record(s) in MongoDB *collection.* The author also defined the *family-regis collection* schema by using *go-mongo-driver*.

Because the author decided that every service should connect its database and only responsible for one kind of data. So, the author connected the Family-service to the RabbitMQ message brokers. From RabbitMQ, Family-service subscribed to some event channels to make the family data in the database more consistent.

4.1.4 Implementation of Birth-service

Birth-service was a service responsible for managing citizens' birth data such as the date of birth, time of birth, parents, etc. Same as three previous services, Birth-service also exposed a REST API and connected to the API Gateway. Based on the microservice architecture, whenever clients need some data from Family-service, they need to make an HTTP request to the API Gateway. If the request can pass the authentication checking process in API Gateway, it will be forwarded to the Birth-service. Then Birth-service will send its response to the client through the API Gateway.

As mentioned before, Birth-service was exposing a REST API to serve some clients' requests. So, there are some HTTP URLs and Methods that were exposed by Birth-service. All of these URLs and Methods are shown in Table 4.5 below.

Table 4.5 Birth-service REST API Exposed URL

No	URL	Method	Usage
1	http://ip.address/api/v1/birth	GET	Getting all the birth record from database and send them through a HTPP response.
2	http://ip.address/api/v1/birth/{number*}	GET	Getting one birth record that the value of birth registration number is {number*} then send it through a HTTP response.
3	http://ip.address/api/v1/birth	POST	Inserting one birth record to the Birth-service database and send HTTP OK response status to the client if the inserting process is success
4	http://ip.address/api/v1/birth/{number*}	PUT	updating one birth record from database that its birth registration number is {number*}
5	http://ip.address/api/v1/birth/{number*}	DELETE	Deleting one birth record in database that its birth registration number is {number*}.

Based on the microservice architecture diagram, Birth-service hast its database. The database that is used in this service is MongoDB. This database has one collection which is *birth-regis*. In *birth-*

regis, some data needed to record such as birth registration number, full name, parents, date of birth, etc.

The author used one of Golang external libraries as the MongoDB driver that its name is *go-mongo-driver*. *go-mongo-driver* not only used for connecting the Birth-service to the MongoDB database but also used for saving, updating, and deleting one or more record(s) in MongoDB collection. The author also defined the *birth-regis* collection schema by using *go-mongo-driver*.

The author also made the Birth-service able to publish an event to the RabbitMQ. This event is published by Birth-service when a birth record is a success to save in the database. Other services like Citizen-service and Family-service subscribed to this event in RabbitMQ then they will do their business logic when the event is published.

4.1.5 Implementation of Admin-service

Admin-service was a service responsible for managing Administrator data such as the administrator's username, password, full name, etc. Same as other backend services, Admin-service also exposed a REST API and connected to the API Gateway. Based on the microservice architecture, whenever clients need some data from Admin-service, they need to make an HTTP a request to the API Gateway. If the request can pass the authentication checking process in API Gateway, it will be forwarded to the Admin-service. Then Admin-service will send its response to the client through the API Gateway.

Admin-service exposed some HTTP URLs and Methods, so clients or other services can make some requests to this service. All of these URLs and Methods are shown in Table 4.6.

Table 4.6 Admin-service REST API Exposed URL

No	**URL**	**Method**	**Usage**
1	http://ip.address/api/v1/admin/{username*}	GET	Getting one administrator record that the value of its username is {username*} then send it through a HTTP response.
2	http://ip.address/api/v1/admin	GET	Getting all admin record to the Admin-service database.
3	http://ip.address/api/v1/admin	POST	Inserting one admin record to the Admin-service database and send HTTP OK response status to the client if the inserting process is success
4	http://ip.address/api/v1/admin/{username*}	DELETE	Deleting one admin record in database that its birth username is {username*}.

Based on the microservice architecture diagram, Birth-service has its database. The database that is used in this service is MongoDB. This database has one collection which is *admin-regis*. In *admin-regis*, some data needed to record such as administrator's full name, password, username, location, etc.

The author used one of Python external libraries as the MongoDB ORM which its name is *mongoengine. mongoengine* not only used for connecting the Admin-service to the MongoDB database but also used for saving, updating, and deleting one or more record(s) in MongoDB *collection.* The author also defined the *admin-regis collection* schema by extending a Python class to one of the *mongoengine* class.

4.1.6 Implementation of Auth-service

Auth-service was a service responsible for handling authentication requests from the client. Auth-service exposed a REST API and connected it to the API Gateway. Based on the microservice architecture, whenever clients want to authenticate, they need to make an HTTP request to the API Gateway. Then API Gateway directly forward the request to this service.

In Auth-service, there is only one exposed URL in its REST API. This URL is http://ip.address/api/v1/auth and only allowed the POST request method. The POST request that is made by the client need to have username and password fields in its body. Then Auth-service will validate username and password fields based on the *admin-regis* record in the MongoDB database. If the username and password are valid, then Auth-service creates a JSON Web Token and saves them to the Redis Key-Value Store. After that, Auth-service returns a response that contains the token that had created before in its response body. Whenever a client needs to make an HTTP request to some backend services, he/she will put the token on HTTP Header so API Gateway can check the request whether it is from the authenticated one or not.

4.1.7 Implementation of API Gateway

Based on the architectural design, the author chose to use the API Gateway pattern for microservice architecture in the civil registration web application. So, by implementing this pattern, the author needed to make an API Gateway as a backend service. The use of API Gateway actually can be various. But in this research, the author implemented the API Gateway only for medium routing and authentication checking.

For medium routing functionality in API Gateway, the author created a REST API. The author implemented the REST API in API Gateway using the Node JS programming language and its framework Express JS. The author then wrote some codes to make the routing rules (URLs and HTTP Methods) in this service. These routing rules are used by API Gateway to specifying the request forward destination. All of these routing rules are shown in Table 4.7 below.

Table 4.7 API Gateway Routing Rules

Exposed URL	**Method**	**Destination Service**	**Destination Exposed URL (Method)**
http://ip.address/citizens	GET	Citizen-service	http://ip.address/api/v1/citizens
http://ip.address/citizens/{NIK*}	GET	Citizen-service	http://ip.address/api/v1/citizens/{NIK*}
http://ip.address/citizens	POST	Citizen-service	http://ip.address/api/v1/citizens
http://ip.address/citizens/auth	POST	Citizen-service	http://ip.address/api/v1/citizens/auth
http://ip.address/citizens/{NIK*}	PUT	Citizen-service	http://ip.address/api/v1/citizens/{NIK*}
http://ip.address/citizens/verify/{NIK*}	PUT	Citizen-service	http://ip.address/api/v1/citizens/verify/{NIK*}
http://ip.address/citizens/{NIK*}	DELETE	Citizen-service	http://ip.address/api/v1/citizens/{NIK*}
http://ip.address/married	GET	Marriage-service	http://ip.address/api/v1/married

Table 4.8 API Gateway Routing Rules Continue

Exposed URL	**Method**	**Destination Service**	**Destination Exposed URL (Method)**
http://ip.address/married/{number*}	GET	Marriage-service	http://ip.address/api/v1/married/{number*}
http://ip.address/married	POST	Marriage-service	http://ip.address/api/v1/married
http://ip.address/married/verif/{number*}	PUT	Marriage-service	http://ip.address/api/v1/married/verif/{number*}
http://ip.address/married/{number*}	DELETE	Marriage-service	http://ip.address/api/v1/married/{number*}
http://ip.address/family	GET	Family-service	http://ip.address/api/v1/family
http://ip.address/family/{number*}	GET	Family-service	http://ip.address/api/v1/family/{number*}
http://ip.address/family/verify/{number*}	PUT	Family-service	http://ip.address/api/v1/family/verify/{number*}
http://ip.address/family/add/{number*21}	PUT	Family-service	http://ip.address/api/v1/family/add/{number*}
http://ip.address/family/update-location/{number*}	PUT	Family-service	http://ip.address/api/v1/family/update-location/{number*}
http://ip.address/family/{number*}	DELETE	Family-service	http://ip.address/api/v1/family/{number*}
http://ip.address/birth	GET	Birth-service	http://ip.address/api/v1/birth

Table 4.8 API Gateway Routing Rules Continue

Exposed URL	**Method**	**Destination Service**	**Destination Exposed URL (Method)**
http://ip.address/birth/{number*}	GET	Birth-service	http://ip.address/api/v1/birth/{number*}
http://ip.address/birth	POST	Birth-service	http://ip.address/api/v1/birth
http://ip.address/birth/{number*}	PUT	Birth-service	http://ip.address/api/v1/birth/{number*}
http://ip.address/birth/{number*}	DELETE	Birth-service	http://ip.address/api/v1/birth/{number*}
http://ip.address/admin/{username*}	GET	Admin-service	http://ip.address/api/v1/admin/{username*}
http://ip.address/admin	GET	Admin-service	http://ip.address/api/v1/admin
http://ip.address/admin	POST	Admin-service	http://ip.address/api/v1/admin
http://ip.address/admin/{username*}	DELETE	Admin-service	http://ip.address/api/v1/admin/{username*}
http://ip.address/auth	POST	Auth-service	http://ip.address/api/v1/auth

For authentication checking in API Gateway, the author created a middleware with Node JS in API Gateway service. Some backend services in the application will be secured with this middleware. Whenever a client makes an HTTP request to the API Gateway, this middleware took place to check did the client provides an authentication token or not. If the client provides the token, this middleware will check again whether this token valid or not. This token will be valid if the token exists in the Redis Key-Value Store. Then this middleware will allow the API Gateway to forward the client's request to other backend services.

4.2 Front-end Services Implementation

4.2.1 Administrator UI Implementation

The author implemented Administrator UI to provide some features for Administrators that were decided in functional requirements analysis. The author implemented this UI by using Argon Dashboard Template. This template was created by creative-tim.com and can be found on their GitHub repository.

Figure 4.1 shows the administrator login page. This page can be accessed via http://admin.ip.address/login.html. From this page, administrators can fill in their username and password that had registered in Admin-service. If the username and the password are valid, the application will show the administrator dashboard page that is shown in Figure 4.2.

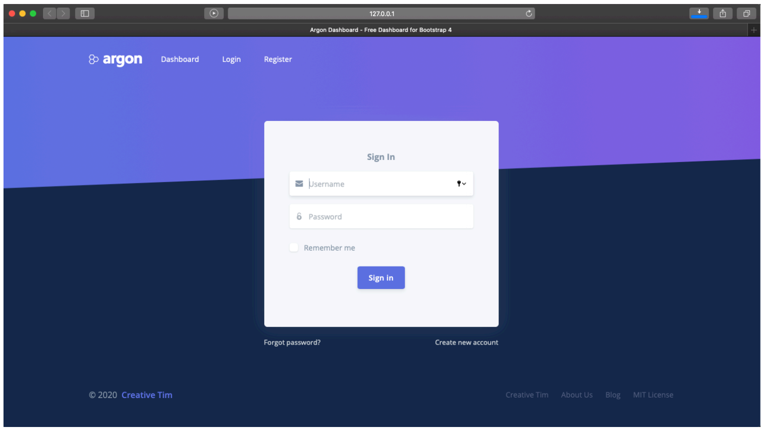
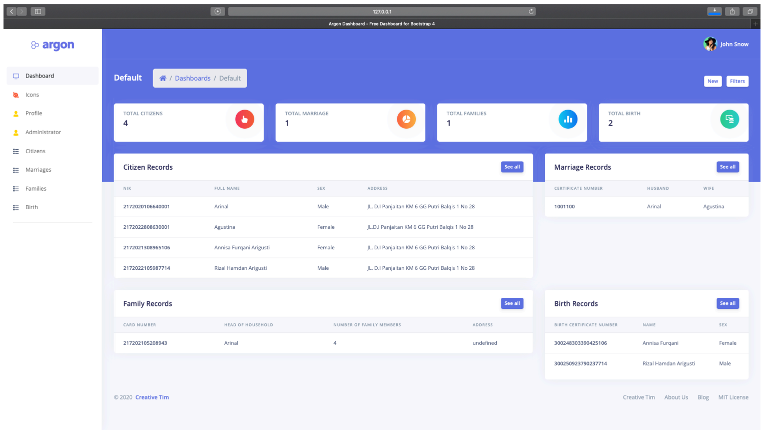


Figure 4.1 Administrator Login Page

Figure 4.2 shows the administrator dashboard page. This page can be accessed by the administrator via http://admin.ip.address/index.html and only after they have authenticated. Whenever a browser loads this page, the application will fetch some requests to the API-Gateway.



TOTAL CITIZENS 4

TOTAL MARRIAGE 1

TOTAL FAMILIES 1

TOTAL BIRTH 2

Citizen Records

NIK	FULL NAME	SEX	ADDRESS
2172020106640001	Arinal	Male	JL. D.I Panjaitan KM 6 GG Putri Balqis 1 No 28
2172022808630001	Agustina	Female	JL.D.I Panjaitan KM 6 GG Putri Balqis 1 No 28
2172021308965106	Annisa Furqani Arigusti	Female	JL. D.I Panjaitan KM 6 GG Putri Balqis 1 No 28
2172022105987714	Rizal Hamdan Arigusti	Male	JL. D.I Panjaitan KM 6 GG Putri Balqis 1 No 28

Marriage Records

CERTIFICATE NUMBER	HUSBAND	WIFE
1001100	Arinal	Agustina

Family Records

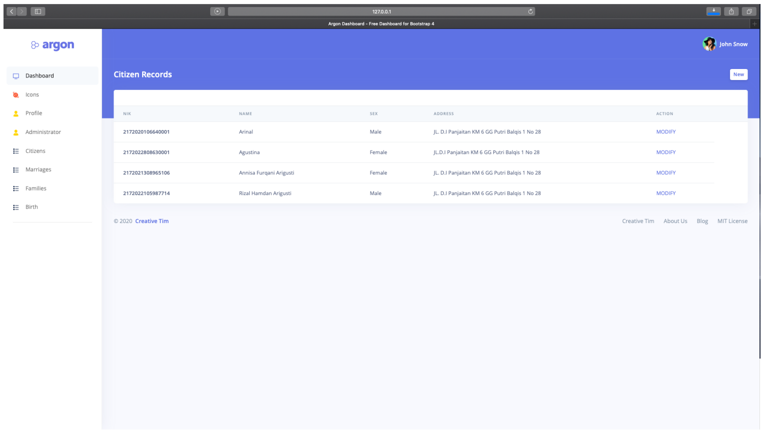
CARD NUMBER	HEAD OF HOUSEHOLD	NUMBER OF FAMILY MEMBERS	ADDRESS
217202105208943	Arinal	4	undefined

Birth Records

BIRTH CERTIFICATE NUMBER	NAME	SEX
300248303390425106	Annisa Furqani	Female
300250923790237714	Rizal Hamdan Arigusti	Male

Figure 4.2 Administrator Dashboard Page

Figure 4.3 shows the citizen table page. this page can be accessed by the administrator via http://admin.ip.address/citizens.html and only after they have authenticated. Whenever browser load this page, the application will fetch get all citizen requests to the API-Gateway. If there is no problem in backend services, the application will receive the data from the API-Gateway response and list some citizen data on this page.



Citizen Records

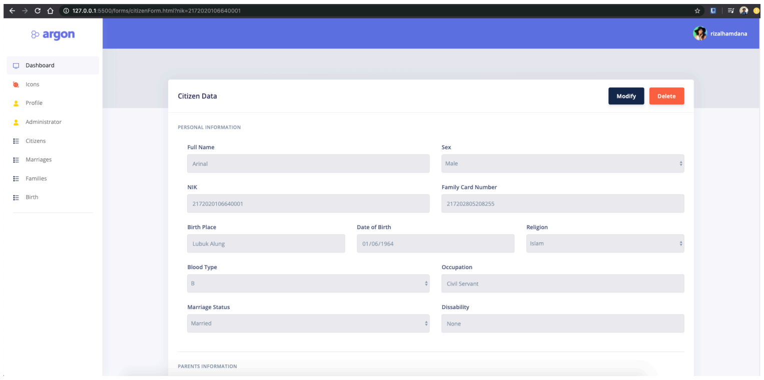
NIK	NAME	SEX	ADDRESS	ACTION
2172020106640001	Arinal	Male	JL. D.I Panjaitan KM 6 GG Putri Balqis 1 No 28	MODIFY
2172022808630001	Agustina	Female	JL.D.I Panjaitan KM 6 GG Putri Balqis 1 No 28	MODIFY
2172021308965106	Annisa Furqani Arigusti	Female	JL. D.I Panjaitan KM 6 GG Putri Balqis 1 No 28	MODIFY
2172022105987714	Rizal Hamdan Arigusti	Male	JL. D.I Panjaitan KM 6 GG Putri Balqis 1 No 28	MODIFY

© 2020 Creative Tim

Figure 4.3 Citizen Table Page

Figure 4.4 shows the citizen form page. This page can be accessed by clicking modify link on one row in the citizen table page. Then the application will show this page to the administrator. When a browser loads this page, the application will send a get one citizen request to the API-Gateway. Then the API-Gateway will send a response with some data from Citizen-service's. After the application receives the response, the application will map all the data to some text fields that are provided on this page.

In the citizen form page, the administrator can delete one record of citizen profile data that is showed on this page. The administrator only needs to click the delete button and the application will send a delete one citizen record to the API-Gateway. On this page, the administrator also can modify and update one citizen record. By clicking the modify button, then filled in some text field and submit, the application will send an update one citizen request to the API-Gateway.



Citizen Data

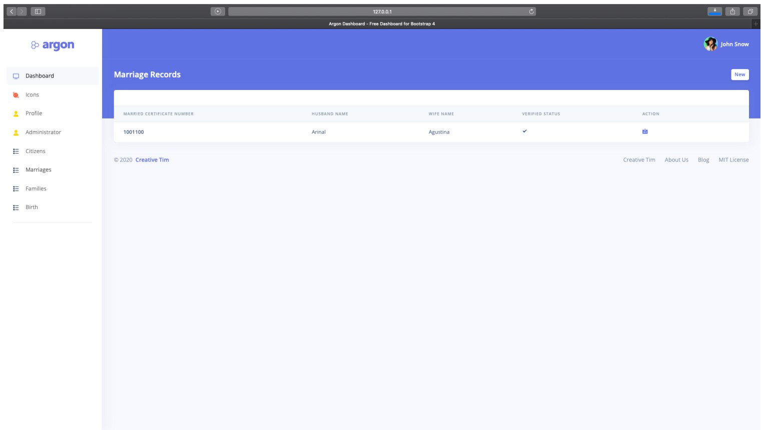
PERSONAL INFORMATION

Full Name: Arinal
Sex: Male
NIK: 2172020106640001
Family Card Number: 217202805208255
Birth Place: Lubuk Alung
Date of Birth: 01/06/1964
Religion: Islam
Blood Type: B
Occupation: Civil Servant
Marriage Status: Married
Dissability: None

PARENTS INFORMATION

Figure 4.4 Citizen Form Page

Figure 4.5 shows the marriage table page. This page can be accessed by the administrator via http://admin.ip.address/marriages.html and only after they have authenticated. Whenever the browser loads this page, the application will fetch get all marriages requests to the API-Gateway. If there is no problem with backend services, the application will receive the data from the API-Gateway response and list some marriage data on this page.



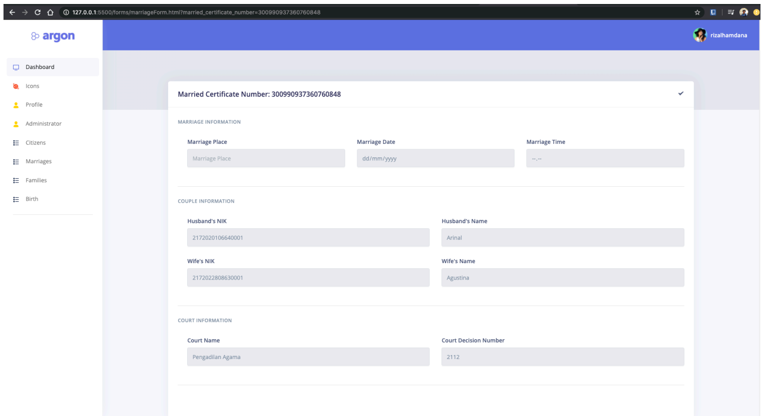
Marriage Records

MARRIED CERTIFICATE NUMBER	HUSBAND NAME	WIFE NAME	VERIFIED STATUS	ACTION
1001100	Arinal	Agustina	✓	

Figure 4.5 Marriage Table Page

Figure 4.6 shows the marriage form page. The administrator can access this page by clicking the badge icon button on the marriage table page. When the browser load this page, the application sends a get one marriage request to the API-Gateway. After the application receives a response, the application will map all the data to some text fields that are provided on this page.

In the marriage form page, the administrator can also verify one record of marriage certificate data that is showed on this page. The administrator only needs to click the verify button and the application will send a verify one marriage record to the API-Gateway.



Married Certificate Number: 300990937360760848

MARRIAGE INFORMATION

Marriage Place: Marriage Place
Marriage Date: dd/mm/yyyy
Marriage Time: --:--

COUPLE INFORMATION

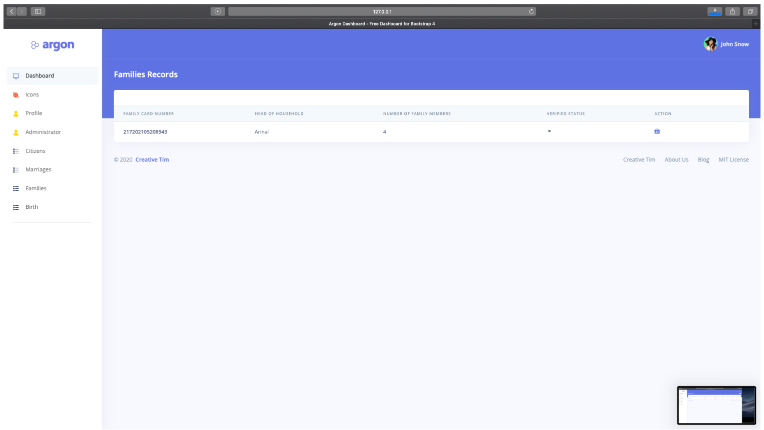
Husband's NIK: 2172020106640001
Husband's Name: Arinal
Wife's NIK: 2172022808630001
Wife's Name: Agustina

COURT INFORMATION

Court Name: Pengadilan Agama
Court Decision Number: 2112

Figure 4.6 Marriage Form Page

Figure 4.7 shows the family table page. This page can be accessed by the administrator via http://admin.ip.address/families.html and only after they have authenticated. Whenever browser loads this page, the application will fetch *get all families* request to the API-Gateway. If there is no problem with backend services, the application will receive the data from the API-Gateway response and list some family data on this page.



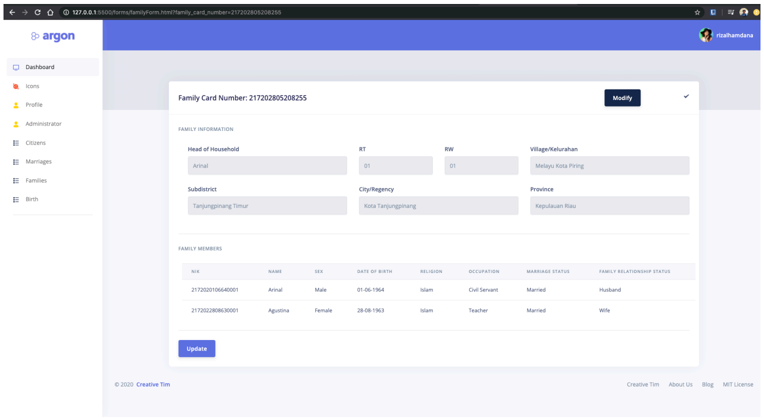
Families Records

FAMILY CARD NUMBER	HEAD OF HOUSEHOLD	NUMBER OF FAMILY MEMBERS	VERIFIED STATUS	ACTION
217202105208943	Arinal	4	×	▣

Figure 4.7 Family Table Page

Figure 4.8 shows the family form page. The administrator can access this page by clicking the badge icon button on the family table page. When the browser load this page, the application sends a get one family request to the API-Gateway. After the application receives a response, the application will map all the data to some text fields and a table that is provided on this page.

In the family form page, the administrator can also verify one record of family card data that is showed on this page. The administrator only needs to click the verify button and the application will send a verify one family record to the API-Gateway. On this page, the administrator also can modify and update one family record. By clicking the modify button, then filled in some text field and submit, the application will send an update one family request to the API-Gateway.



Family Card Number: 217202805208255

FAMILY INFORMATION

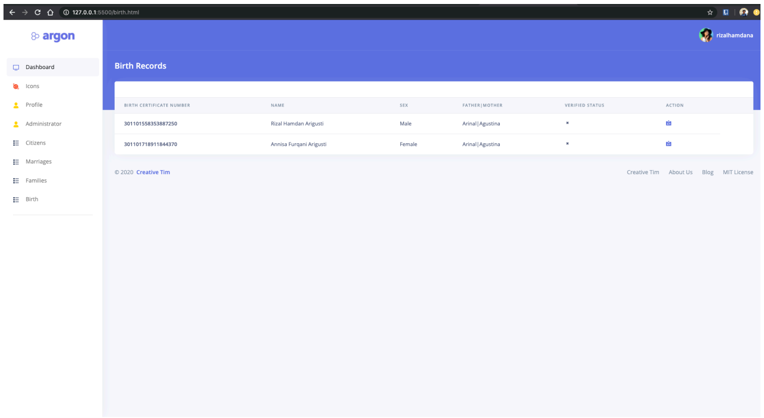
Head of Household	RT	RW	Village/Kelurahan
Arinal	01	01	Melayu Kota Piring
Subdistrict	City/Regency		Province
Tanjungpinang Timur	Kota Tanjungpinang		Kepulauan Riau

FAMILY MEMBERS

NIK	NAME	SEX	DATE OF BIRTH	RELIGION	OCCUPATION	MARRIAGE STATUS	FAMILY RELATIONSHIP STATUS
2172020106640001	Arinal	Male	01-06-1964	Islam	Civil Servant	Married	Husband
2172022808630001	Agustina	Female	28-08-1963	Islam	Teacher	Married	Wife

Figure 4.8 Family Form Page

Figure 4.9 shows the birth table page. This page can be accessed by the administrator via http://admin.ip.address/families.html and only after they have authenticated. Whenever the browser loads this page, the application will fetch get all birth requests to the API-Gateway. If there is no problem with backend services, the application will receive the data from the API-Gateway response and list some birth data on this page.



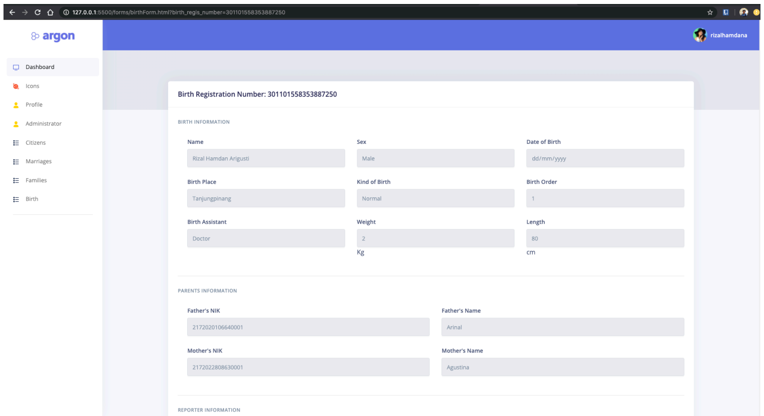
Birth Records

BIRTH CERTIFICATE NUMBER	NAME	SEX	FATHER|MOTHER	VERIFIED STATUS	ACTION
301101558353887250	Rizal Hamdan Arigusti	Male	Arinal|Agustina	×	
301101718911844370	Annisa Furqani Arigusti	Female	Arinal|Agustina	×	

Figure 4.9 Birth Table Page

Figure 4.10 shows the birth form page. The administrator can access this page by clicking the badge icon button on the birth table page. When the browser load this page, the application sends a get one birth request to the API-Gateway. After the application receives a response, the application will map all the data to some text fields that are provided on this page.

In the birth form page, the administrator can also verify one record of birth certificate data that is showed on this page. The administrator only needs to click the verify button and the application will send a verify one birth record request to the API-Gateway.



Birth Registration Number: 301101558353887250

BIRTH INFORMATION

Name: Rizal Hamdan Arigusti
Sex: Male
Date of Birth: dd/mm/yyyy
Birth Place: Tanjungpinang
Kind of Birth: Normal
Birth Order: 1
Birth Assistant: Doctor
Weight: 2 Kg
Length: 80 cm

PARENTS INFORMATION

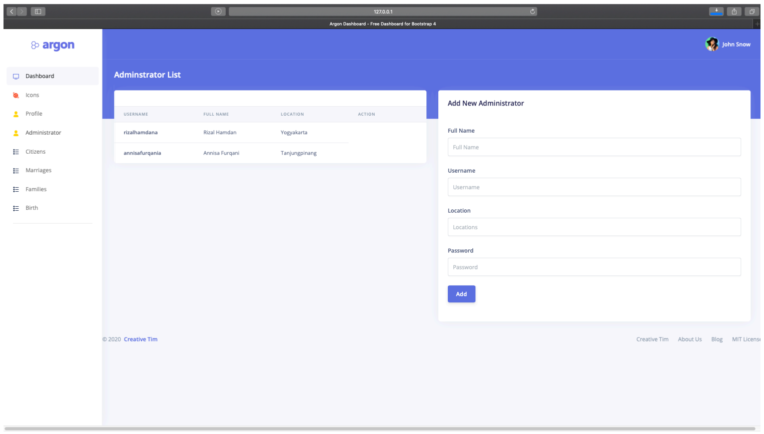
Father's NIK: 2172020106640001
Father's Name: Arinal
Mother's NIK: 2172022808630001
Mother's Name: Agustina

REPORTER INFORMATION

Figure 4.10 Birth Form Page

Figure 4.9 shows the admin registration page. This page can be accessed by the administrator via http://admin.ip.address/admin.html and only after they have authenticated. Whenever the browser loads this page, the application will fetch get all admin requests to the API-Gateway. If there is no problem with backend services, the application will receive the data from the API-Gateway response and list some admin data on the admin table.

There is also an admin registration form on the admin registration page. The administrators use this to register other administrators' accounts. They only need to fill the form then click submit. After that, the application will send insert one admin request to the API-Gateway.



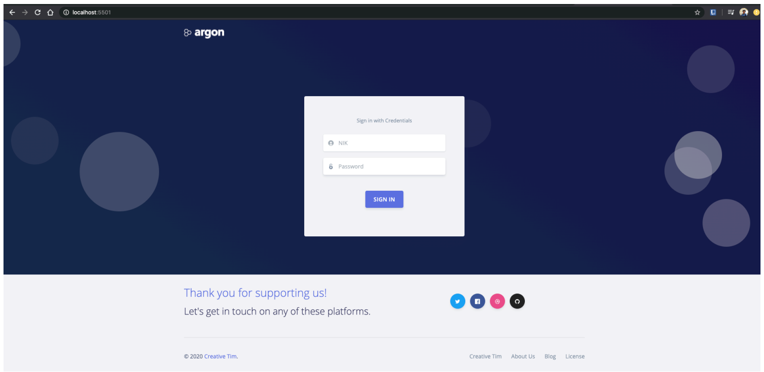
USERNAME	FULL NAME	LOCATION	ACTION
rizalhamdana	Rizal Hamdan	Yogyakarta	
annisafurqania	Annisa Furqani	Tanjungpinang	

Figure 4.11 Admin Registration Page

4.2.2 Citizens UI Implementation

The author implemented Citizens UI to provide some features for citizens that were decided in functional requirements analysis. The author implemented this UI by using Argon Design System Template. This template was created by creative-tim.com and can be found on their Github repository.

Figure 4.12 shows the citizen login page. This page can be accessed via http://citizen.ip.address/index.html. From this page, citizens can fill in their NIK and password that had registered in Citizen-service. If the NIK and the password are valid, the application will show the citizen profile page that is shown on Figure 4.13.



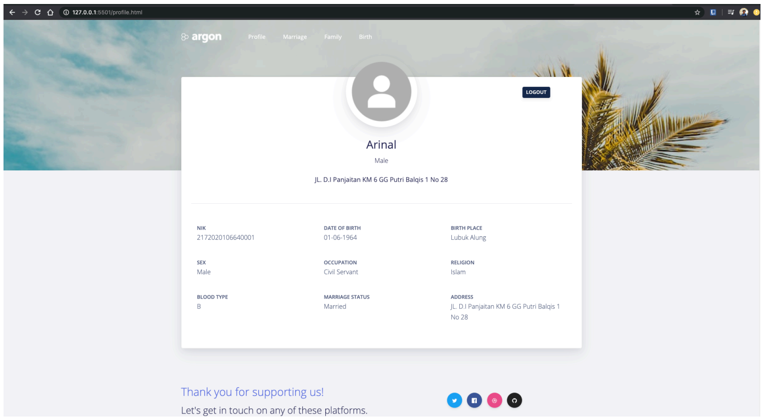
Thank you for supporting us!

Let's get in touch on any of these platforms.

© 2020 Creative Tim.

Figure 4.12 Citizen Login Page

Figure 4.14 shows the citizen profile page. This page can be accessed by the citizens via http://citizen.ip.address/index.html and only after they have authenticated. Whenever the browser loads this page, the application will fetch get one citizen request to the API-Gateway based on the authentication token. If there is no problem with backend services, the application will receive a response from the API-Gateway and show the profile data on this page.



Arinal

Male

JL. D.I Panjaitan KM 6 GG Putri Balqis 1 No 28

NIK: 2172020106640001

DATE OF BIRTH: 01-06-1964

BIRTH PLACE: Lubuk Alung

SEX: Male

OCCUPATION: Civil Servant

RELIGION: Islam

BLOOD TYPE: B

MARRIAGE STATUS: Married

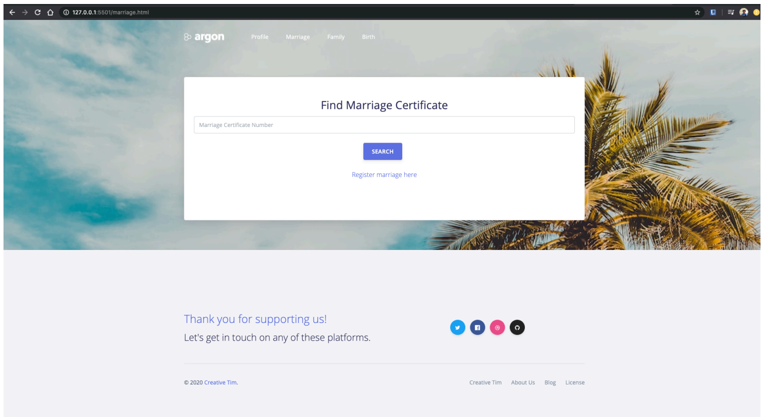
ADDRESS: JL. D.I Panjaitan KM 6 GG Putri Balqis 1 No 28

Thank you for supporting us!

Let's get in touch on any of these platforms.

Figure 4.13 Citizen Profile Page

Figure 4.14 shows the find marriage certificate page. Whenever a citizen fills in the search text field and clicks the search button, the application will fetch get one marriage request to the API-Gateway based on the value of the filled-in text field. If the back end services can find the data in the database, they will send a response to this page. The result can be seen in Figure 4.15.

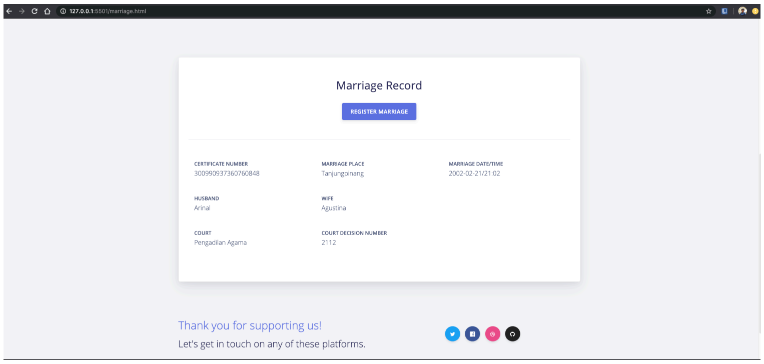


Thank you for supporting us!

Let's get in touch on any of these platforms.

© 2020 Creative Tim.

Figure 4.14 Find Marriage Certificate Page



CERTIFICATE NUMBER
300990937360760848

MARRIAGE PLACE
Tanjungpinang

MARRIAGE DATE/TIME
2002-02-21/21:02

HUSBAND
Arinal

WIFE
Agustina

COURT
Pengadilan Agama

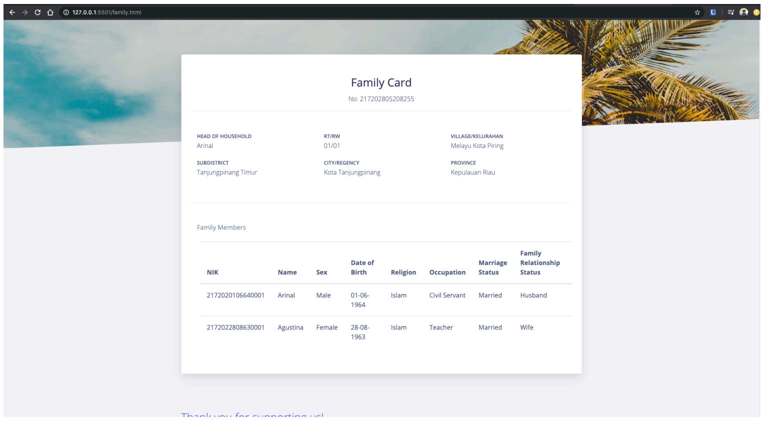
COURT DECISION NUMBER
2112

Thank you for supporting us!

Let's get in touch on any of these platforms.

Figure 4.15 Find Marriage Certificate Result

Figure 4.16 shows the citizen's family card page. This page can be accessed by the citizens via http://citizen.ip.address/family.html and only after they have authenticated. Whenever the browser loads this page, the application will fetch get one family request to the API-Gateway based on the authenticated user's family card number. If the back end services can find the data in the database, the application will receive a response from the API-Gateway and show the family card data.



Family Card

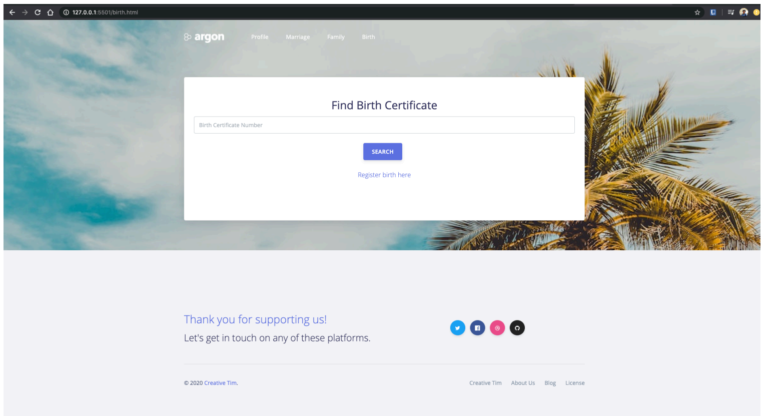
No: 217202805208255

HEAD OF HOUSEHOLD Arinal	**RT/RW** 01/01	**VILLAGE/KELURAHAN** Melayu Kota Piring
SUBDISTRICT Tanjungpinang Timur	**CITY/REGENCY** Kota Tanjungpinang	**PROVINCE** Kepulauan Riau

Family Members

NIK	Name	Sex	Date of Birth	Religion	Occupation	Marriage Status	Family Relationship Status
2172020106640001	Arinal	Male	01-06-1964	Islam	Civil Servant	Married	Husband
2172022808630001	Agustina	Female	28-08-1963	Islam	Teacher	Married	Wife

Figure 4.16 Citizen’s Family Card Page

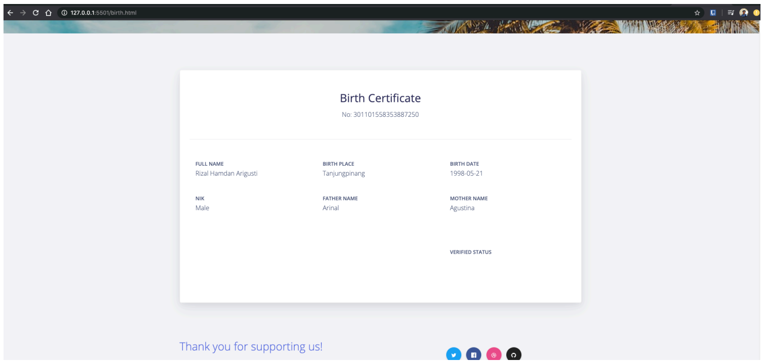


Thank you for supporting us!

Let's get in touch on any of these platforms.

Figure 4.17 Find Birth Certificate Page

Figure 4.17 shows the find birth certificate page. Whenever a citizen fills in the search text field and clicks the search button, the application will fetch get one birth request to the API-Gateway based on the value of the filled-in text field. The result of finding a birth certificate can be seen in Figure 4.18.



Birth Certificate

No: 301101558353887250

FULL NAME: Rizal Hamdan Arigusti
BIRTH PLACE: Tanjungpinang
BIRTH DATE: 1998-05-21
NIK: Male
FATHER NAME: Arinal
MOTHER NAME: Agustina
VERIFIED STATUS:

Thank you for supporting us!

Figure 4.18 Find Birth Certificate Result

Figure 4.19 shows the register birth page. Citizens can register a birth certificate on this page by filling in all provided the text fields. After filling in, citizens click the submit button and the application will send an insert one birth request to the API-Gateway.

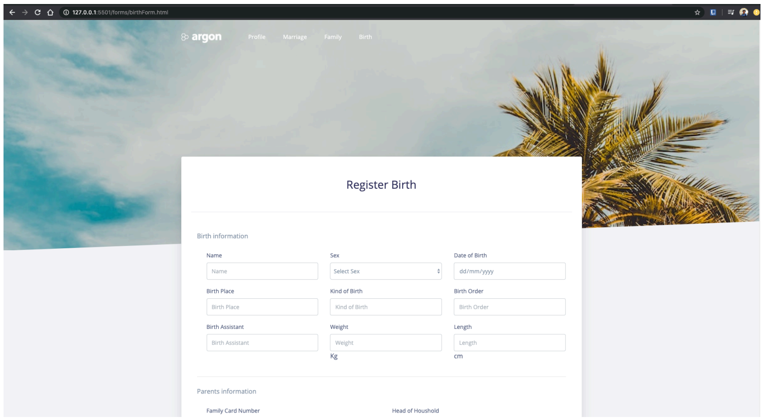
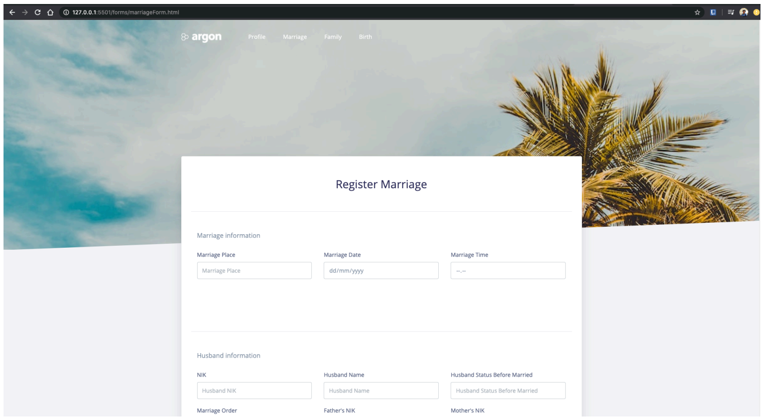


Figure 4.19 Register Birth Page

Figure 4.20 shows the register marriage page. Citizens can register a marriage certificate on this page by filling in all provided the text fields. After filling in, citizens click the submit button and the application will send an insert one marriage request to the API-Gateway.



Register Marriage

Marriage information

Marriage Place: Marriage Place
Marriage Date: dd/mm/yyyy
Marriage Time: --.--

Husband information

NIK: Husband NIK
Husband Name: Husband Name
Husband Status Before Married: Husband Status Before Married
Marriage Order
Father's NIK
Mother's NIK

Figure 4.20 Register Marriage Page

4.3 Implementation of Kubernetes Cluster

For fulfilling non-functional requirements that had defined before, the author decided that all the backend services will be run on the Kubernetes cluster. In the Kubernetes cluster, the author run every backend service in a containerized application and also made some configurations so every service can connect to each other.

4.3.1 Building Docker Image for Backend Services

Before running on the Kubernetes cluster, every backend service needed to build into a Docker image. The author built every backend service’s docker image by creating a Dockerfile and run a build command on the Docker application. All of the Dockerfile that were made by the author can be seen in Appendix A. All of the Docker images that were built are shown in Table 4.8.

Table 4.8 Backend Services’ Docker Images

No	**Backend Service**	**Docker Image**
1	Citizen-service	rizalhamdana/citizen-service

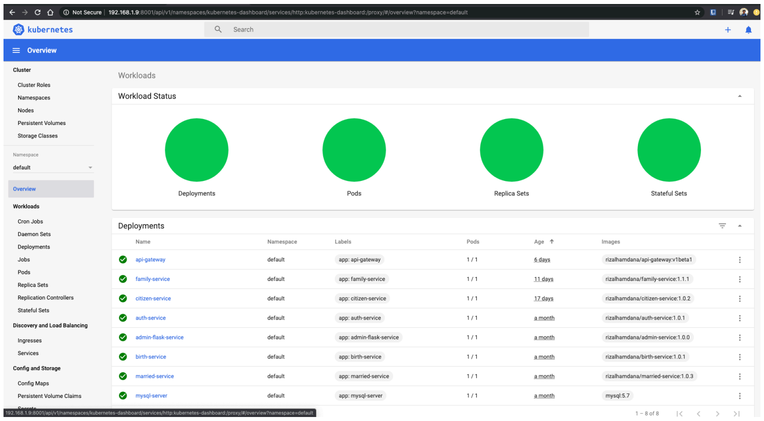
Table 4.8 Backend Services' Docker Images

No	Backend Service	Docker Image
2	Marriage-service	rizalhamdana/married-service
3	Family-service	rizalhamdana/family-service
4	Birth-service	rizalhamdana/birth-service
5	Admin-service	rizalhamdana/admin-service
6	Auth-service	rizalhamdana/auth-service
7	API Gateway	rizalhamdana/api-gateway

4.3.2 Running Application on Kubernetes Cluster

The author created some Kubernetes objects to run all the backend services on the cluster. These objects included workloads and services. The author created all of these objects by defining some Kubernetes configuration files and applying them to the cluster with *kubectl* command. Some of these Kubernetes configuration files are shown in Appendix B.

The author used an add-on from Kubernetes named *minikube* dashboard to monitor all of the objects in the cluster. Figure 4.21 shows the *minikube* dashboard's overview page for monitoring the Kubernetes cluster.

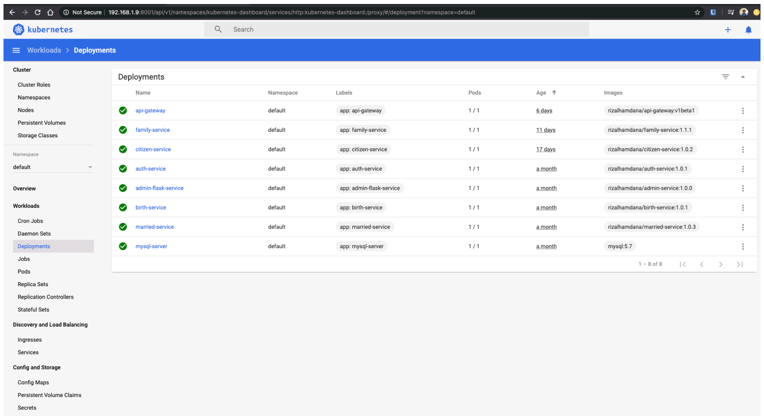


Deployments

	Name	Namespace	Labels	Pods	Age	Images
✓	api-gateway	default	app: api-gateway	1 / 1	6 days	rizalhamdana/api-gateway:v1beta1
✓	family-service	default	app: family-service	1 / 1	11 days	rizalhamdana/family-service:1.1.1
✓	citizen-service	default	app: citizen-service	1 / 1	17 days	rizalhamdana/citizen-service:1.0.2
✓	auth-service	default	app: auth-service	1 / 1	a month	rizalhamdana/auth-service:1.0.1
✓	admin-flask-service	default	app: admin-flask-service	1 / 1	a month	rizalhamdana/admin-service:1.0.0
✓	birth-service	default	app: birth-service	1 / 1	a month	rizalhamdana/birth-service:1.0.1
✓	married-service	default	app: married-service	1 / 1	a month	rizalhamdana/married-service:1.0.3
✓	mysql-server	default	app: mysql-server	1 / 1	a month	mysql:5.7

Figure 4.21 Minikube Dashboard Overview Page

Figure 4.22 shows all the Deployments that are created on the Kubernetes cluster. Deployments run multiple replicas of application (backend services) and automatically replace any instances that fail or become unresponsive ("Deployment | Kubernetes Engine Documentation | Google Cloud", 2020). In this way, Deployments help ensure that one or more instances of the application (backend services) are available to serve user requests.

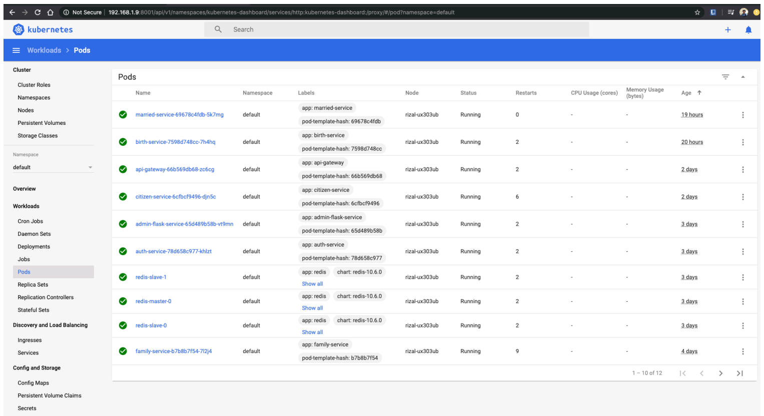


Deployments

	Name	Namespace	Labels	Pods	Age	Images
✓	api-gateway	default	app: api-gateway	1 / 1	6 days	rizalhamdana/api-gateway:v1beta1
✓	family-service	default	app: family-service	1 / 1	11 days	rizalhamdana/family-service:1.1.1
✓	citizen-service	default	app: citizen-service	1 / 1	17 days	rizalhamdana/citizen-service:1.0.2
✓	auth-service	default	app: auth-service	1 / 1	a month	rizalhamdana/auth-service:1.0.1
✓	admin-flask-service	default	app: admin-flask-service	1 / 1	a month	rizalhamdana/admin-service:1.0.0
✓	birth-service	default	app: birth-service	1 / 1	a month	rizalhamdana/birth-service:1.0.1
✓	married-service	default	app: married-service	1 / 1	a month	rizalhamdana/married-service:1.0.3
✓	mysql-server	default	app: mysql-server	1 / 1	a month	mysql:5.7

Figure 4.22 Kubernetes Deployments Component List

Figure 4.23 shows some Pods that are running on the Kubernetes cluster. Based on ("Pod | Kubernetes Engine Documentation | Google Cloud", 2020) Pods are the smallest, most basic deployable objects in Kubernetes. A Pod represents a single instance of a running process in the cluster.



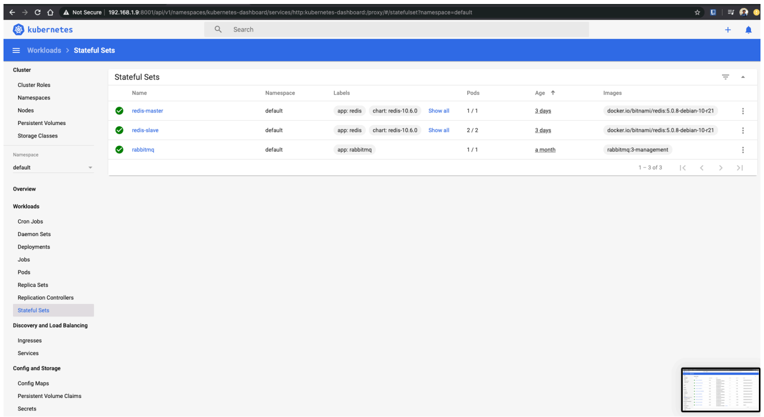
Pods

Name	Namespace	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Age
married-service-69678c4fdb-5k7mg	default	app: married-service pod-template-hash: 69678c4fdb	rizal-ux303ub	Running	0	-	-	19 hours
birth-service-7598d748cc-7h4hq	default	app: birth-service pod-template-hash: 7598d748cc	rizal-ux303ub	Running	2	-	-	20 hours
api-gateway-66b569db68-zc6cg	default	app: api-gateway pod-template-hash: 66b569db68	rizal-ux303ub	Running	2	-	-	2 days
citizen-service-6cfbcf9496-djn5c	default	app: citizen-service pod-template-hash: 6cfbcf9496	rizal-ux303ub	Running	6	-	-	2 days
admin-flask-service-65d489b58b-vt9mn	default	app: admin-flask-service pod-template-hash: 65d489b58b	rizal-ux303ub	Running	2	-	-	3 days
auth-service-78d658c977-khlzt	default	app: auth-service pod-template-hash: 78d658c977	rizal-ux303ub	Running	2	-	-	3 days
redis-slave-1	default	app: redis chart: redis-10.6.0 Show all	rizal-ux303ub	Running	2	-	-	3 days
redis-master-0	default	app: redis chart: redis-10.6.0 Show all	rizal-ux303ub	Running	2	-	-	3 days
redis-slave-0	default	app: redis chart: redis-10.6.0 Show all	rizal-ux303ub	Running	2	-	-	3 days
family-service-b7b8b7f54-7l2j4	default	app: family-service pod-template-hash: b7b8b7f54	rizal-ux303ub	Running	9	-	-	4 days

1 – 10 of 12

Figure 4.23 Kubernetes Pods Component List

Figure 4.24 shows all the StatefulSets on the Kubernetes cluster. StatefulSets represent a set of Pods with unique, persistent identities and stable hostnames. The state information and other resilient data for any given StatefulSet Pod is maintained in persistent disk storage associated with the StatefulSet ("StatefulSet | Kubernetes Engine Documentation | Google Cloud", 2020).



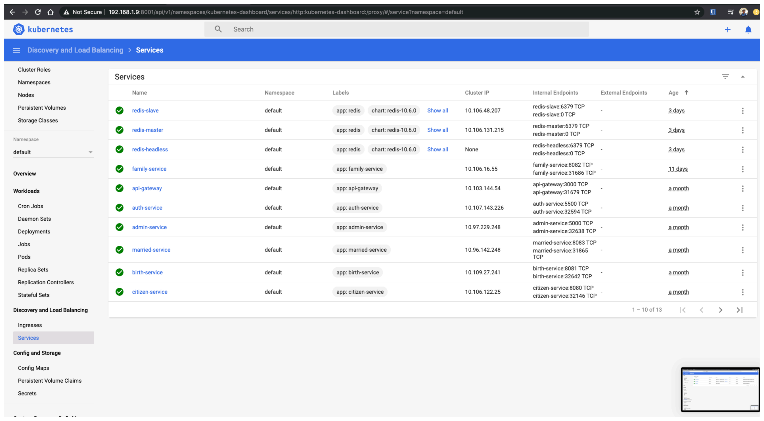
Stateful Sets

Name	Namespace	Labels	Pods	Age	Images
redis-master	default	app: redis chart: redis-10.6.0 Show all	1 / 1	3 days	docker.io/bitnami/redis:5.0.8-debian-10-r21
redis-slave	default	app: redis chart: redis-10.6.0 Show all	2 / 2	3 days	docker.io/bitnami/redis:5.0.8-debian-10-r21
rabbitmq	default	app: rabbitmq	1 / 1	a month	rabbitmq:3-management

1 – 3 of 3

Figure 4.24 Kubernetes Stateful Sets Component List

Figure 4.26 below shows all the Services objects that are created in the Kubernetes cluster. Services is an abstract way to expose an application running on a set of Pods as a network service ("Service", 2020).



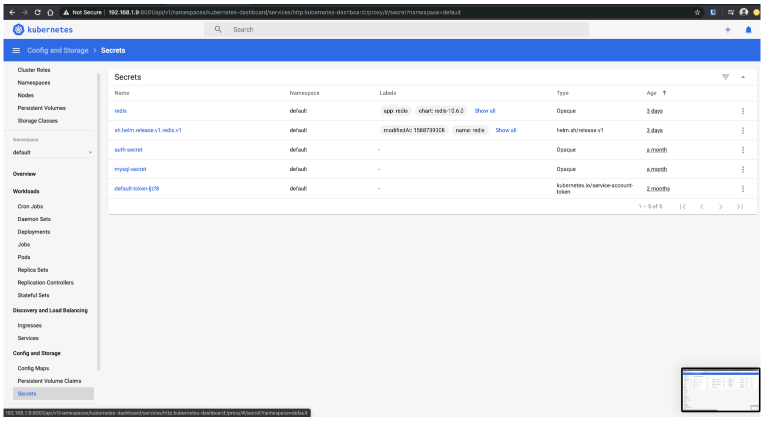
Services

Name	Namespace	Labels	Cluster IP	Internal Endpoints	External Endpoints	Age
redis-slave	default	app: redis chart: redis-10.6.0 Show all	10.106.48.207	redis-slave:6379 TCP redis-slave:0 TCP	-	3 days
redis-master	default	app: redis chart: redis-10.6.0 Show all	10.106.131.215	redis-master:6379 TCP redis-master:0 TCP	-	3 days
redis-headless	default	app: redis chart: redis-10.6.0 Show all	None	redis-headless:6379 TCP redis-headless:0 TCP	-	3 days
family-service	default	app: family-service	10.106.16.55	family-service:8082 TCP family-service:31686 TCP	-	11 days
api-gateway	default	app: api-gateway	10.103.144.54	api-gateway:3000 TCP api-gateway:31679 TCP	-	a month
auth-service	default	app: auth-service	10.107.143.226	auth-service:5500 TCP auth-service:32594 TCP	-	a month
admin-service	default	app: admin-service	10.97.229.248	admin-service:5000 TCP admin-service:32638 TCP	-	a month
married-service	default	app: married-service	10.96.142.248	married-service:8083 TCP married-service:31865 TCP	-	a month
birth-service	default	app: birth-service	10.109.27.241	birth-service:8081 TCP birth-service:32642 TCP	-	a month
citizen-service	default	app: citizen-service	10.106.122.25	citizen-service:8080 TCP citizen-service:32146 TCP	-	a month

1 – 10 of 13

Figure 4.25 Kubernetes Services Component List

Figure 4.27 shows a list of Secrets objects in the Kubernetes cluster. Secrets are secure objects which store sensitive data, such as passwords, OAuth tokens, and SSH keys, in the Kubernetes cluster ("Secret | Kubernetes Engine Documentation | Google Cloud", 2020).



Secrets

Name	Namespace	Labels	Type	Age
redis	default	app: redis chart: redis-10.6.0 Show all	Opaque	3 days
sh.helm.release.v1.redis.v1	default	modifiedAt: 1588739308 name: redis Show all	helm.sh/release.v1	3 days
auth-secret	default	-	Opaque	a month
mysql-secret	default	-	Opaque	a month
default-token-ljzf8	default	-	kubernetes.io/service-account-token	2 months

1 – 5 of 5

Figure 4.26 Kubernetes Secrets Component List

4.4 Application Testing

For testing the application, the author did a testing technique named black-box testing. The author also divided this testing into two parts. The first part is testing the functional requirements and the second part is testing the Kubernetes cluster.

4.4.1 Functional Requirement Testing

In the functional requirement testing, the author made some test cases to check whether the system can fully support all the functional requirements or not and also to check does the system can validate the data that are input by the users. Table 4.9 shows the result of this testing.

Table 4.9 Functional Requirement Testing Result

Code	**Test Case**	**Expected Result**	**Success/Not Success**
TC-01	Admin fills in recorded username and password in each text field on Admin Login Page and click Submit button.	Authentication success and system will showed the Admin Dashboard Page.	Success
TC-02	Admin fill in invalid username and password in each text field on Admin Login Page and click Submit button	System showed invalid credentials message on Admin Login Page.	Success
TC-03	Admin fills in all the text field in Citizen Form Page to insert new citizen record.	System showed a success message on Admin Form Page.	Success
TC-04	Admin does not fill in one or more text field(s) in Citizen Form Page to insert new citizen record.	System showed a message that one or more text field(s) should be filled in.	Success

Table 4.9 Functional Requirement Testing Result Continue

Code	**Test Case**	**Expected Result**	**Success/Not Success**
TC-05	Admin clicks verify button for one marriage record on Marriage Form Page.	System showed a message that one record is verified and then showed “check” icon on the verified status column for one record in Marriage Table Page.	Success
TC-06	Admin clicks verify button for one marriage record on Marriage Form Page.	System automatically created a family record that its family members are husband and wife in one marriage record	Success
TC-07	Admin fills in all the text fields in Family Form Page in order to updating a family record.	System shows a updating success message on Family Form Page	Success
TC-10	Admin clicks verify button for one birth record on Birth Form Page.	System showed a message that one record is verified and then showed “check” icon on the verified status column for one record in Birth Table Page.	Success
TC-11	Citizen fills in recorded NIK and password in each text field on Citizen Login Page and click Submit button.	Authentication success and system will show the Citizen Profile Page.	Success
TC-12	Citizen fills in all the text fields on Marriage Registration Form Page with valid data in order to register new marriage record	System show register success message on Marriage Registration Page	Success

Table 4.9 Functional Requirement Testing Result Continue

Code	**Test Case**	**Expected Result**	**Success/Not Success**
TC-13	Citizen does not fill in one or more text field(s) on Marriage Registration Form Page.	System showed a message that one or more text field(s) should be filled in.	Success
TC-14	Citizen fills in husband’s and/or wife’s NIK text field(s) on Marriage Registration Form Page with invalid data in order to register new marriage record.	System showed a message that filled in husband’s and/or wife’s NIK is invalid.	Success
TC-15	Citizen fills in all the text field on Birth Registration Form Page with valid data in order to register new birth record	System show register success message on Birth Registration Page	Success
TC-16	Citizen does not fill in one or more text field(s) on Birth Registration Form Page.	System showed a message that one or more text field(s) should be filled in.	Success
TC-17	Citizen fills in father’s and/or mother’s NIK text field(s) on Birth Registration Form Page with invalid data	System showed a message that filled in father’s and/or mother’s NIK is invalid.	Success
TC-18	Citizen fills in reporter’s and/or witness’s NIK text field(s) on Birth Registration Form Page with invalid data	System showed a message that filled in reporter’s and/or witness’s NIK is invalid.	Success

4.4.2 Kubernetes Cluster Testing

The author did Kubernetes cluster testing to make sure that every backend service in the cluster is more available and flexible to scale. Table 4.10 below shows the result of this testing.

Table 4.10 Kubernetes Cluster Testing Result

Code	**Test Case**	**Expected Result**	**Success/Not Success**
KTC-01	Scaling up citizen-service deployment component to three replicas.	Kubernetes automatically create three replicas of citizen-service pods.	Success
KTC-02	Scaling up married-service deployment component to three replicas.	Kubernetes automatically create three replicas of married-service pods.	Success
KTC-03	Scaling up family-service deployment component to three replicas.	Kubernetes automatically create three replicas of family-service pods.	Success
KTC-04	Scaling up birth-service deployment component to three replicas.	Kubernetes automatically create three replicas of birth-service pods.	Success
KTC-05	Scaling up admin-service deployment component to three replicas.	Kubernetes automatically create three replicas of admin-service pods.	Success
KTC-06	Scaling up auth-service deployment component to three replicas.	Kubernetes automatically create three replicas of auth-service pods.	Success

Table 4.10 Kubernetes Cluster Testing Result Continue

Code	**Test Case**	**Expected Result**	**Success/Not Success**
KTC-07	Scaling up api-gateway deployment component to three replicas.	Kubernetes automatically run three replicas of api-gateway pods.	Success
KTC-08	Scaling down citizen-service deployment component to one replicas.	Kubernetes automatically run one replicas of citizen-service pods.	Success
KTC-10	Scaling down family-service deployment component to one replicas.	Kubernetes automatically run one replicas of family-service pods.	Success
KTC-11	Scaling down birth-service deployment component to one replicas.	Kubernetes automatically run one replicas of birth-service pods.	Success
KTC-12	Scaling down admin-service deployment component to one replicas.	Kubernetes automatically run one replicas of admin-service pods.	Success
KTC-13	Scaling down auth-service deployment component to one replicas.	Kubernetes automatically run one replicas of auth-service pods.	Success
KTC-14	Scaling down api-gateway deployment component to one replicas.	Kubernetes automatically one run replicas of api-gateway pods.	Success

Table 4.10 Kubernetes Cluster Testing Result Continue

Code	**Test Case**	**Expected Result**	**Success/Not Success**
KTC-15	Deleting one backend service pod.	Kubernetes automatically recreate one pod for that one backend service and other backend service pods are still running in the Kubernetes cluster.	Success

Testing result in Table 4.10 shows some insights to the author. From KTC-01 until KTC-14 results, they show that every service can scale flexibly. The developers later can scale up or scale down the number of pods that run for every service following the demands to the one or more service(s). KTC-15 testing result shows the entire application can become more available since if there is one service that shut down, it will not affect other services. KTC-15 result also shows that Kubernetes can automatically recreate the pod of shut down service so the service can be available to serve the request from the client.

CHAPTER 5. CLOSING

5.1 Conclusion

From the implementation and testing results, there were some conclusions that the author can obtain. These conclusions are:

1. In this research, the author successfully designed and implemented microservice architecture to the Civil Registration Web Application. The author designed the microservice architecture by analysing and identifying some business subdomains in the application. From the identified subdomains, the author implemented a REST API service for every subdomain and connected them to database and message broker. After that the author chose to use HTTP and AMQP as the communication protocol between services so every service can work together to provide the application to the users.
2. The author successfully developed the Civil Registration Web Application. The author developed the application by following the Waterfall Software Development Methodology. The author first analysed the system requirements by doing observation to some related previous literature. After that, the author designed the microservice architecture for the application by considering the business subdomains, system requirements and communication protocol for every service. Then the author developed every services with some programming languages and connected them to database and message broker. Last but not least, the author did black-box testing to test the functional requirements that had defined before.
3. The application can be more available and flexible to scale. the author achieved this by making all the services can be run on the Kubernetes cluster. The Kubernetes cluster then will responsible to maintain the availability of the application. With Kubernetes Deployment object, every service also can be scaled up and down so it can fulfill the request demands from the users.

5.2 Recommendations

From the implementation and testing result, the author realized that the application that was developed in this research still has some limitations. So in the future, this application still needs further development. Some recommendations from the author so this application can be better. These recommendations are:

1. The application that was developed only runs on the single-node Kubernetes cluster which is the author laptop. So in the future, this application needs to deploy to the real server or cloud services like AWS, Alibaba Cloud, Microsoft Azure, or GCP.
2. The application currently only covers four business processes or services of civil registration in Indonesia. These four services are citizen registration, marriage registration, family registration, and birth registration. In the future, the functional requirements need to be added more so it can cover all the business processes or services of the civil registration in Indonesia. For instances, in the future the application can support the death registration, divorce registration, etc.
3. The application currently run the MySQL database and Redis in the Kubernetes cluster with pods object. This situation is not good since literature said that pods are transient and have big chance to restart or failover. So, in the future, the MySQL database and Redis need to run on different servers or cloud platforms.
4. In this research, the microservice architecture was only tested with black-box testing to the Kubernetes cluster. This technique and strategy are not really appropriate. So, in the future, the microservice architecture needs to be evaluated and tested with different techniques and strategies, for instance using Microservice Architecture Analysis Tool (MAAT) or Chaos Engineering Testing.

BIBLIOGRAPHY

Fowler, M. and Lewis, J. (2019). *Microservices*. [online] martinfowler.com. Available at: https://martinfowler.com/articles/microservices.html [Accessed 24 Nov. 2019].

Kharenko, A. (2019). *Monolithic vs. Microservices Architecture*. [online] Medium. Available at: https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59 [Accessed 24 Nov. 2019].

Newman, S. (2015). *Building Microservices*. 1st ed. O'Reilly Media, Inc., p.2.

Prahono, A. and Elidjen (2015). Evaluating the Role e-Government on Public Administration Reform: Case of Official City Government Websites in Indonesia. *Procedia Computer Science*, 59, pp.27-33.

Richardson, C. (2019). *Microservices Pattern: Microservice Architecture pattern*. [online] microservices.io. Available at: https://microservices.io/patterns/microservices.html [Accessed 25 Nov. 2019].

Sabani, A., Deng, H. and Thai, V. (2019). Evaluating the Development of E-Government in Indonesia. In: *2nd International Conference on Software Engineering and Information Management*. New York: ACM, pp.254-258.

Soldani, J., Tamburri, D. and Van Den Heuvel, W. (2018). The pains and gains of microservices: A Systematic grey literature review. *Journal of Systems and Software*, 146, pp.215-232.

Amaral, M., Polo, J., Carrera, D., Mohomed, I., Unuvar, M., & Steinder, M. (2015). Performance Evaluation of Microservices Architectures Using Containers. *2015 IEEE 14Th International Symposium On Network Computing And Applications*. doi: 10.1109/nca.2015.49

Balalaie, A., Heydarnoori, A., & Jamshidi, P. (2016). Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software*, *33*(3), 42-52. doi: 10.1109/ms.2016.64

Dragoni, N., Giallorenzo, S., Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: Yesterday, Today, and Tomorrow. *Present And Ulterior Software Engineering*, 195-216. doi: 10.1007/978-3-319-67425-4_12

Götz, B., Schel, D., Bauer, D., Henkel, C., Einberger, P., & Bauernhansl, T. (2018). Challenges of Production Microservices. *Procedia CIRP*, *67*, 167-172. doi: 10.1016/j.procir.2017.12.194

Iqbal, M., Fahroji, W., & Dedi. (2019). Sistem Informasi Administrasi Kependudukan Berbasis Web di Kelurahan Sangiang Jaya. In *SEMNASTIK* (pp. 306-313). Semarang: Universitas Dian Nuswantoro.

Jayanto, D. (2017). *Rancang Bangun Back-End "SIAP": Sistem Informasi Aspirasi Dan Pengaduan Masyarakat Berbasis Web Dengan Menggunakan Metode Microservice Springboot* (Undergraduate). Institut Teknologi Sepuluh Nopember.

Sani, N., Fillah, W., Tjahyanto, A., & Suryotrisongko, H. (2019). Development of Microservice Based Application E-Inkubator: Incubation and Investment Service Provider for SMEs. *Procedia Computer Science*, *161*, 1064-1071. doi: 10.1016/j.procs.2019.11.217

Siau, K., & Long, Y. (2005). Synthesizing e-government stage models – a meta-synthesis based on meta-ethnography approach. *Industrial Management & Data Systems*, *105*(4), 443-458. doi: 10.1108/02635570510592352

Sistem informasi administrasi kependudukan. (2020). Retrieved 12 January 2020, from https://id.wikipedia.org/wiki/Sistem_informasi_administrasi_kependudukan

Stenroos, K. (2019). *Microservices in Software Development* (Undergraduate). Metropolia University of Applied Sciences.

Undang Undang Republik Indonesia No.24 Tahun 2013. (2013).

Wan, X., Guan, X., Wang, T., Bai, G., & Choi, B. (2018). Application deployment using Microservice and Docker containers: Framework and optimization. *Journal Of Network And Computer Applications*, *119*, 97-109. doi: 10.1016/j.jnca.2018.07.003

What is a Container? | Docker. (2020). Retrieved 12 January 2020, from https://www.docker.com/resources/what-container

Deployment | Kubernetes Engine Documentation | Google Cloud. (2020). Retrieved 11 May 2020, from https://cloud.google.com/kubernetes-engine/docs/concepts/deployment

Kubernetes vs. Docker: What Does It Really Mean? | Sumo Logic. (2020). Retrieved 9 May 2020, from https://www.sumologic.com/blog/kubernetes-vs-docker/

Pod | Kubernetes Engine Documentation | Google Cloud. (2020). Retrieved 11 May 2020, from https://cloud.google.com/kubernetes-engine/docs/concepts/pod

Secret | Kubernetes Engine Documentation | Google Cloud. (2020). Retrieved 11 May 2020, from https://cloud.google.com/kubernetes-engine/docs/concepts/secret

Service. (2020). Retrieved 11 May 2020, from https://kubernetes.io/docs/concepts/services-networking/service/

StatefulSet | Kubernetes Engine Documentation | Google Cloud. (2020). Retrieved 11 May 2020, from https://cloud.google.com/kubernetes-engine/docs/concepts/statefulset

What is Kubernetes?. (2020). Retrieved 9 May 2020, from https://www.redhat.com/en/topics/containers/what-is-kubernetes

What are Microservices? | AWS. (2020). Retrieved 27 May 2020, from https://aws.amazon.com/microservices/

APPENDIX A. DOCKERFILES

CITIZEN-SERVICE DOCKERFILE

```
FROM golang:1.14.0-alpine3.11 as builder

LABEL maintainer="Rizal Hamdan <ari.gusti12@gmail.com>"

RUN apk update && apk add --no-cache git

WORKDIR /app

COPY go.mod go.sum ./

RUN go mod download

COPY . .

RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main ./src/main.go


FROM alpine:latest
RUN apk --no-cache add ca-certificates

WORKDIR /root/

COPY --from=builder /app/main .
COPY --from=builder /app/.env .

# Expose port 8080 to the outside world
EXPOSE 8080

#Command to run the executable
CMD ["./main"]
```

MARRIAGE-SERVICE DOCKERFILE

```
FROM golang:1.14.0-alpine3.11 as builder

LABEL maintainer="Rizal Hamdan <ari.gusti12@gmail.com>"

RUN apk update && apk add --no-cache git

WORKDIR /app

COPY go.mod go.sum ./

RUN go mod download

COPY . .

RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main ./src/main.go


FROM alpine:latest
RUN apk --no-cache add ca-certificates

WORKDIR /root/

# Copy the Pre-built binary file from the previous stage. Observe we also copied the .env file
COPY --from=builder /app/main .
COPY --from=builder /app/.env .

EXPOSE 8083

CMD ["./main"]
```

FAMILY-SERVICE DOCKERFILE

```
FROM golang:1.14.0-alpine3.11 as builder

LABEL maintainer="Rizal Hamdan <ari.gusti12@gmail.com>"

RUN apk update && apk add --no-cache git

WORKDIR /app

COPY go.mod go.sum ./

RUN go mod download

COPY . .

RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main ./src/main.go


FROM alpine:latest
RUN apk --no-cache add ca-certificates

WORKDIR /root/

# Copy the Pre-built binary file from the previous stage. Observe we also copied
the .env file
COPY --from=builder /app/main .
COPY --from=builder /app/.env .

EXPOSE 8082

CMD ["./main"]
```

BIRTH-SERVICE DOCKERFILE

```
FROM golang:1.14.0-alpine3.11 as builder

LABEL maintainer="Rizal Hamdan <ari.gusti12@gmail.com>"

RUN apk update && apk add --no-cache git

WORKDIR /app

COPY go.mod go.sum ./

RUN go mod download

COPY . .

RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main ./src/main.go


FROM alpine:latest
RUN apk --no-cache add ca-certificates

WORKDIR /root/

# Copy the Pre-built binary file from the previous stage. Observe we also copied the .env file
COPY --from=builder /app/main .
COPY --from=builder /app/.env .

EXPOSE 8081

CMD ["./main"]
```

ADMIN-SERVICE DOCKERFILE

```
FROM python:3.7-alpine3.11

COPY . /app

WORKDIR /app

RUN apk update && apk add gcc python3-dev musl-dev

RUN pip install -r requirements.txt

EXPOSE 5000

CMD [ "python", "./app.py" ]
```

AUTH-SERVICE DOCKERFILE

```
FROM python:3.7-alpine3.11

COPY . /app

WORKDIR /app

RUN apk update && apk add gcc python3-dev musl-dev

RUN pip install -r requirements.txt

EXPOSE 5500

CMD [ "python", "./app.py" ]
```

API-GATEWAY DOCKERFILE

```
FROM node:10

WORKDIR /usr/src/app

COPY package*.json ./

RUN npm install

COPY . .

EXPOSE 3000
CMD [ "node", "index.js" ]
```

APPENDIX B. KUBERNETES CONFIGURATION FILES

CITIZEN-SERVICE DEPLOYMENT RESOURCE CONFIGURATION

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: citizen-service
  labels:
    app: citizen-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: citizen-service
  template:
    metadata:
      labels:
        app: citizen-service
    spec:
          containers:
        - name: citizen-service
          image: rizalhamdana/citizen-service:1.0.2
          imagePullPolicy: IfNotPresent
          ports:
            - name: http
              containerPort: 8080
          envFrom:
            - secretRef:
                name: mysql-secret
```

MARRIAGE-SERVICE DEPLOYMENT RESOURCE CONFIGURATION

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: married-service
  labels:
    app: married-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: married-service
  template:
    metadata:
      labels:
        app: married-service
    spec:
      containers:
        - name: married-service
          image: rizalhamdana/married-service:1.0.2
          imagePullPolicy: IfNotPresent
          ports:
            - name: http
              containerPort: 8083
```

FAMILY-SERVICE DEPLOYMENT RESOURCE CONFIGURATION

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: family-service
  labels:
    app: family-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: family-service
  template:
    metadata:
      labels:
        app: family-service
    spec:
      containers:
        - name: family-service
          image: rizalhamdana/family-service:1.1.1
          imagePullPolicy: IfNotPresent
          ports:
            - name: http
              containerPort: 8082
```

BIRTH-SERVICE DEPLOYMENT RESOURCE CONFIGURATION

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: birth-service
  labels:
    app: birth-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: birth-service
  template:
    metadata:
      labels:
        app: birth-service
    spec:
      containers:
        - name: family-service
          image: rizalhamdana/birth-service:1.0.0
          imagePullPolicy: IfNotPresent
          ports:
            - name: http
              containerPort: 8083
```

ADMIN-SERVICE DEPLOYMENT RESOURCE CONFIGURATION

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: admin-flask-service
  labels:
    app: admin-flask-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: admin-flask-service
  template:
    metadata:
      labels:
        app: admin-flask-service
    spec:
      containers:
        - name: admin-flask-service
          image: rizalhamdana/admin-service:1.0.0
          imagePullPolicy: IfNotPresent
          ports:
            - name: http
              containerPort: 5000
```

AUTH-SERVICE DEPLOYMENT RESOURCE CONFIGURATION

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: auth-service
  labels:
    app: auth-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: auth-service
  template: e
    metadata:
      labels:
        app: auth-service
    spec:
      containers:
        - name: auth-service
          image: rizalhamdana/auth-service:1.0.
          imagePullPolicy: IfNotPresent
          ports:
            - name: http
              containerPort: 5000
          envFrom:
            - secretRef:
                name: auth-secret
            - secretRef:
                name: redis
```

AUTH-SERVICE SECRET RESOURCE CONFIGURATION

```
apiVersion: v1
kind: Secret
metadata:
  name: auth-secret
type: Opaque
stringData: # We dont need to worry about converting to base64
  JWT_PRIVATE_KEY: NullPointerException
  JWT_ALGORITHM: HS256
```

CITIZEN-SERVICE SERVICE RESOURCE CONFIGURATION

```
apiVersion: v1
kind: Service
metadata:
  name:
  labels:
    app: citizen-service
spec:
  type: NodePort
  selector:
    app: citizen-service `
  ports:
    - name: citizen-service
      protocol: TCP
      port: 8080
      targetPort: 8080
```

MARRIAGE-SERVICE SERVICE RESOURCE CONFIGURATION

```
apiVersion: v1
kind: Service
metadata:
  name: married-service
  labels:
    app: married-service
spec:
  type: NodePort
  selector:
    app: married-service
  ports:
    - name: married-service
      protocol: TCP
      port: 8083
      targetPort: 8083
```

FAMILY-SERVICE SERVICE RESOURCE CONFIGURATION

```
apiVersion: v1
kind: Service
metadata:
  name: family-service
  labels:
    app: family-service
spec:
  type: NodePort
  selector:
    app: family-service
  ports:
    - name: family-service
      protocol: TCP
      port: 8082
      targetPort: 8082
```

BIRTH-SERVICE SERVICE RESOURCE CONFIGURATION

```
apiVersion: v1
kind: Service
metadata:
  name: birth-service
  labels:
    app: birth-service
spec:
  type: NodePort
  selector:
    app: birth-service
  ports:
    - name: birth-service
      protocol: TCP
      port: 8081
      targetPort: 8081
```

ADMIN-SERVICE SERVICE RESOURCE CONFIGURATION

```
apiVersion: v1
kind: Service
metadata:
  name: admin -service
  labels:
    app: admin-service
spec:
  type: NodePort
  selector:
    app: admin-flask-service
  ports:
    - name: admin-service
      protocol: TCP
      port: 5000
      targetPort: 5000
```

AUTH-SERVICE SERVICE RESOURCE CONFIGURATION

```
apiVersion: v1
kind: Service
metadata:
  name: auth-service
  labels:
    app: auth-service
spec:
  type: NodePort
  selector:
    app: auth-service
  ports:
    - name: auth-service
      protocol: TCP
      port: 5500
      targetPort: 5500
```

API-GATEWAY SERVICE RESOURCE CONFIGURATION

```
apiVersion: v1
kind: Service
metadata:
  name: api-gateway
  labels:
    app: api-gateway
spec:
  type: NodePort
  selector:
    app: api-gateway
  ports:
    - name: api-gateway
      protocol: TCP
      port: 3000
      targetPort: 3000
```