ITS
Institut
Teknologi
Sepuluh Nopember

TUGAS AKHIR - KI141502

IMPLEMENTASI *PUBLISH/SUBSCRIBE* PADA RANCANG BANGUN SISTEM MONITORING PERANGKAT JARINGAN DI ITS

AFIF RIDHO KAMAL PUTRA
NRP 05111440000173

Dosen Pembimbing I
Royyana Muslim Ijtihadie S.Kom, M.Kom., Ph.D

Dosen Pembimbing II
Bagus Jati Santoso, S.Kom., Ph.D

JURUSAN DEPARTEMEN INFORMATIKA
Fakultas Teknologi Informasi dan Komunikasi
Institut Teknologi Sepuluh Nopember
Surabaya, 2018

(Halaman ini sengaja dikosongkan)

TUGAS AKHIR - KI141502

IMPLEMENTASI *PUBLISH/SUBSCRIBE* PADA RANCANG BANGUN SISTEM MONITORING PERANGKAT JARINGAN DI ITS

AFIF RIDHO KAMAL PUTRA
NRP 05111440000173

Dosen Pembimbing I
Royyana Muslim Ijtihadie S.Kom, M.Kom., Ph.D

Dosen Pembimbing II
Bagus Jati Santoso, S.Kom., Ph.D

JURUSAN DEPARTEMEN INFORMATIKA
Fakultas Teknologi Informasi dan Komunikasi
Institut Teknologi Sepuluh Nopember
Surabaya, 2018

(Halaman ini sengaja dikosongkan)

UNDERGRADUATE THESIS - KI141502

IMPLEMENTATION OF PUBLISH/SUBSCRIBE ON DESIGN OF NETWORK MONITORING DEVICE SYSTEM IN ITS

AFIF RIDHO KAMAL PUTRA
NRP 05111440000173

Supervisor I
Royyana Muslim Ijtihadie S.Kom, M.Kom., Ph.D

Supervisor II
Bagus Jati Santoso, S.Kom., Ph.D

Department of INFORMATICS
Faculty of Information Technology and Communication
Institut Teknologi Sepuluh Nopember
Surabaya, 2018

(Halaman ini sengaja dikosongkan)

LEMBAR PENGESAHAN

IMPLEMENTASI *PUBLISH/SUBSCRIBE* PADA RANCANG BANGUN SISTEM MONITORING PERANGKAT JARINGAN DI ITS

TUGAS AKHIR

Diajukan Guna Memenuhi Salah Satu Syarat
Memperoleh Gelar Sarjana Komputer
pada
Bidang Studi Arsitektur dan Jaringan Komputer
Program Studi S1 Jurusan Departemen Informatika
Fakultas Teknologi Informasi dan Komunikasi
Institut Teknologi Sepuluh Nopember

Oleh :
AFIF RIDHO KAMAL PUTRA
NRP: 05111440000173

Disetujui oleh Dosen Pembimbing Tugas Akhir :

Royyana Muslim Ijtihadie S.Kom, M.Kom., Ph.D .....
NIP: 198407082010122004 (Pembimbing 1)

Bagus Jati Santoso, S.Kom., Ph.D .....
NIP: 051100116 (Pembimbing 2)

SURABAYA
Juni 2018

(Halaman ini sengaja dikosongkan)

IMPLEMENTASI *PUBLISH/SUBSCRIBE* PADA RANCANG BANGUN SISTEM MONITORING PERANGKAT JARINGAN DI ITS

Nama	:	**AFIF RIDHO KAMAL PUTRA**
NRP	:	**05111440000173**
Jurusan	:	**Departemen Informatika FTIK**
Pembimbing I	:	**Royyana Muslim Ijtihadie S.Kom, M.Kom., Ph.D**
Pembimbing II	:	**Bagus Jati Santoso, S.Kom., Ph.D**

Abstrak

Saat ini, dengan didukung oleh konsep SaaS (Software as a Service), aplikasi web berkembang dengan pesat. Para penyedia layanan aplikasi web berlomba-lomba memberikan pelayanan yang terbaik, seperti menjaga QoS (Quality of Service) sesuai dengan perjanjian yang tertuang dalam SLA (Service Level Agreement). Hal tersebut dikarenakan permintaan akses ke suatu aplikasi web biasanya meningkat dengan seiring berjalannya waktu. Keramaian akses sesaat menjadi hal yang umum dalam aplikasi web saat ini. Saat hal tersebut terjadi, aplikasi web akan di akses lebih banyak dari kebiasaan. Jika aplikasi web tersebut tidak menyediakan kemampuan untuk menangani hal tersebut, bisa menyebabkan aplikasi web tidak dapat berjalan dengan semestinya yang sangat merugikan pengguna.

Elastic cloud merupakan salah satu bagian dari komputasi awan yang sedang populer, dimana banyak riset dan penelitian yang berfokus di bidang ini. Elastic cloud bisa digunakan untuk menyelesaikan permasalah di atas. Lalu sebuah perangkat lunak bernama Docker dapat dapat diterapkan untuk mendukung elastic cloud.

Dalam tugas akhir ini akan dibuat sebuah rancangan sistem yang memungkinkan aplikasi web berjalan di atas Docker. Sistem ini bisa beradaptasi sesuai dengan kebutuhan dari aplikasi yang sedang berjalan. Jika aplikasi membutuhkan sumber daya tambahan, sistem akan menyediakan sumber daya berupa suatu container baru secara otomatis dan juga akan mengurangi penggunaan sumber daya jika aplikasi sedang tidak membutuhkannya. Dari hasil uji coba, sistem dapat menangani sampai dengan 57.750 request dengan error request yang terjadi sebesar 7.83%.

***Kata-Kunci**: aplikasi web, autoscale, docker, elastic cloud*

IMPLEMENTATION OF PUBLISH/SUBSCRIBE ON DESIGN OF NETWORK MONITORING DEVICE SYSTEM IN ITS

Name	:	**AFIF RIDHO KAMAL PUTRA**
NRP	:	**05111440000173**
Major	:	**Informatics FTIK**
Supervisor I	:	**Royyana Muslim Ijtihadie S.Kom, M.Kom., Ph.D**
Supervisor II	:	**Bagus Jati Santoso, S.Kom., Ph.D**

Abstract

Nowdays, with the concept of SaaS (Software as a Service), web applications have developed a lot. Web service providers are competing to provide the best service, such as QoS (Quality of Service) requirements specified in the SLA (Service Level Agreement). The load of web applications usually very drastically along with time. Flash crowds are also very common in today's web applications world. When flash crowds happens, the web application will be accessed more than usual. If the web applications does not provide the ability to do so, it can make the web application not work properly which is very disadvantegeous to the users.

Elastic cloud is one of the most popular part of cloud computing, with much researchs in this subject. Elastic clouds can be used to solve the above problems. Then a Docker can be applied to support the elastic cloud.

In this final task will be made an application system that allows web applications running on top of Docker. This system can adjust according to the needs of the running applications. If the application requires additional resources, the system will automatically supply the resources of a new container and will

also reduce resource usage if the application is not needing it. From the test results, the system can handle up to 57,750 requests and error ratio of 7.83%.

***Keywords**: autoscale, docker, elastic cloud, web application*

KATA PENGANTAR

بِسْمِ ٱللَّهِ ٱلرَّحْمَٰنِ ٱلرَّحِيمِ

Alhamdulillahirabbil'alamin, segala puji bagi Allah SWT, yang telah melimpahkan rahmat dan hidayah-Nya sehingga penulis dapat menyelesaikan Tugas Akhir yang berjudul **IMPLEMENTASI PENGENDALI ELASTISITAS SUMBER DAYA BERBASIS DOCKER UNTUK APLIKASI WEB**. Pengerjaan Tugas Akhir ini merupakan suatu kesempatan yang sangat baik bagi penulis. Dengan pengerjaan Tugas Akhir ini, penulis bisa belajar lebih banyak untuk memperdalam dan meningkatkan apa yang telah didapatkan penulis selama menempuh perkuliahan di Teknik Informatika ITS. Dengan Tugas Akhir ini penulis juga dapat menghasilkan suatu implementasi dari apa yang telah penulis pelajari. Selesainya Tugas Akhir ini tidak lepas dari bantuan dan dukungan beberapa pihak. Sehingga pada kesempatan ini penulis mengucapkan syukur dan terima kasih kepada:

1. Allah SWT atas anugerahnya yang tidak terkira kepada penulis dan Nabi Muhammad SAW.
2. Ibu Henning Titi Ciptaningtyas, S.Kom., M.Kom selaku pembimbing I yang telah membantu, membimbing, dan memotivasi penulis mulai dari pengerjaan proposal hingga terselesaikannya Tugas Akhir ini.
3. Bapak Bagus Jati Santoso, S.Kom., Ph.D selaku pembimbing II yang juga telah membantu, membimbing, dan memotivasi penulis mulai dari pengerjaan proposal hingga terselesaikannya Tugas Akhir ini.
4. Darlis Herumurti, S.Kom., M.Kom., selaku Kepala Jurusan Teknik Informatika ITS pada masa pengerjaan Tugas Akhir, Bapak Radityo Anggoro, S.Kom., M.Sc., selaku koordinator TA, dan segenap dosen Teknik Informatika yang telah memberikan ilmu dan pengalamannya.

5. Serta semua pihak yang telah turut membantu penulis dalam menyelesaikan Tugas Akhir ini.

Penulis menyadari bahwa Tugas Akhir ini masih memiliki banyak kekurangan. Sehingga dengan kerendahan hati, penulis mengharapkan kritik dan saran dari pembaca untuk perbaikan ke depannya.

Surabaya, Juni 2017

Muhammad Fahrul Razi

DAFTAR ISI

(Halaman ini sengaja dikosongkan)

DAFTAR TABEL

(Halaman ini sengaja dikosongkan)

DAFTAR GAMBAR

(Halaman ini sengaja dikosongkan)

DAFTAR KODE SUMBER

(Halaman ini sengaja dikosongkan)

BAB I

PENDAHULUAN

Pada bab ini akan dipaparkan mengenai garis besar Tugas Akhir yang meliputi latar belakang, tujuan, rumusan dan batasan permasalahan, metodologi pembuatan Tugas Akhir, dan sistematika penulisan.

1.1 Latar Belakang

Saat ini, dengan didukung oleh konsep SaaS (*Software as a Service*), aplikasi web berkembang dengan pesat. Banyak perusahaan, seperti Google, Amazon, dan Microsoft yang berhasil mencapai kesuksesan dari aplikasi web. Para penyedia layanan aplikasi web juga berlomba-lomba memberikan pelayanan yang terbaik, seperti menjaga QoS (*Quality of Service*) sesuai dengan perjanjian yang tertuang dalam SLA (*Service Level Agreement*) [1]. Hal tersebut dikarenakan permintaan akses ke suatu aplikasi web biasanya meningkat dengan seiring berjalannya waktu. Keramaian akses sesaat menjadi hal yang umum dalam aplikasi web saat ini. Saat hal tersebut terjadi, aplikasi web akan diakses lebih banyak dari keadaan biasanya. Jika aplikasi web tersebut tidak menyediakan kemampuan untuk menangani hal tersebut, bisa menyebabkan aplikasi web tidak dapat berjalan dengan semestinya yang sangat merugikan pengguna. Biasanya, pengembang akan melakukan pengaturan sumber daya *server* secara manual agar bisa menangani permasalahan di atas, tapi akan memakan banyak biaya dan waktu. Tapi jika tidak ditangani, akibatnya aplikasi web tidak bisa berjalan saat pengalami puncak permintaan dari pengguna. Saat ini banyak tersedia layanan komputasi awan, yaitu sebuah model komputasi yang mana pengguna akan membayar sesuai dengan sumber daya yang digunakan. Dengan bantuan dari komputasi awan, penggembang bisa melakukan *scale up* dan *scale down* sumber daya *server* dari aplikasi web

secara manual atau memanfaatkan API yang disediakan oleh *platform* yang bisa diakses dalam rentang waktu jam bahkan menit. Perkembangan dari komputasi awan melahirkan teknologi yang dikenal dengan *autonomos elastic cloud*, sebuah sistem yang secara dinamis akan menambahkan sumber daya sesuai dengan jumlah permintaan. Saat permintaan akses ke suatu aplikasi web meningkat, *elastic cloud* secara otomatis akan menambahkan sumber daya untuk aplikasi dan juga secara otomatis akan mengurangi sumber daya dari aplikasi saat permintaan aksesnya menurun.

Elastic cloud merupakan salah satu bagian dari komputasi awan yang sedang populer, dimana banyak riset dan penelitian yang berfokus di bidang ini. Saat ini, biasanya *elastic cloud* berbasis pada *virtual machines* (VMs). VM sendiri dianggap terlalu berat untuk menjalankan sebuah aplikasi web, karena biasanya yang dibutuhakan oleh suatu aplikasi web hanya *web server* (Apache, Nginx), bahasa pemrograman yang digunakan, basis data, dan komponen lainnya, tidak keseluruhan sistem operasi yang terjadi jika menggunakan VM. Dalam hal ini, menggunakan VM untuk mengembangkan aplikasi web hanya akan membuang-buang sumber daya dan menurunkan performa dari aplikasi. Selain itu, penerapan *elastic cloud* yang berjalan di atas VM tidak bisa meningkatkan sumber daya dengan cepat yang bisa merusak QoS.

Sebuah perangkat lunak bernama *docker* dapat menyelesaikan permasalahan dari VM. *Docker* adalah sebuah perangkat lunak yang berfungsi sebagai wadah untuk membungkus dan memasukkan sebuah perangkat lunak ke dalam sebuah lingkungan beserta semua hal yang dibutuhkan oleh perangkat lunak tersebut. Selain membungkus aplikasi, *docker* menjadikan aplikasi yang berjalan di atasnya menjadi terisolasi sehingga menghilangkan kemungkinan terjadinya kebocoran suatu proses aplikasi yang bisa menyebabkan kerusakan pada

host. *Docker container* berjalan di atas *host* dan menggunakan *kernel* yang sama dengan *host* yang mana memungkinkan *container* dapat dibangun dengan cepat dan membuat penggunaan sumber daya menjadi lebih efisien.

Dalam tugas akhir ini akan dibuat sebuah rancangan sistem yang memungkinkan untuk menjalankan aplikasi web berbasis *docker*. Sistem ini bisa beradaptasi sesuai dengan kebutuhan dari aplikasi yang sedang berjalan. Jika aplikasi membutuhkan sumber daya tambahan, sistem akan menyediakan sumber daya berupa suatu *container* baru secara otomatis dan juga akan mengurangi penggunaan sumber daya jika aplikasi sedang tidak membutuhkannya. Proses skalabilitas ini termasuk skalabilitas secara horizontal, yaitu dengan menambah *instance*, dalam kasus ini berupa *docker container*, dari aplikasi web. Sistem ini juga menyediakan sebuah *server docker repository* untuk menaruh aplikasi web dalam format *docker*. Pengembang yang ingin memasang atau memperbarui aplikasinya di sistem ini akan melakukan *push* aplikasi web dalam format *docker* ke *server repository* dan sistem secara otomatis akan membangun atau memperbarui aplikasi tersebut di *server master host*.

1.2 Rumusan Masalah

Rumusan masalah yang diangkat dalam tugas akhir ini adalah sebagai berikut :

1. Bagaimana cara membuat sistem yang dapat melakukan skalabilitas secara otomatis terhadap aplikasi web berbasis *docker* dengan menggunakan *Proactive Model* dan *Reactive Model*?
2. Bagaimana cara membuat sistem yang dapat mendistribusikan akses pengguna ke aplikasi web berbasis *docker* secara efisien?
3. Bagaimana cara membuat sistem yang dapat melakukan

pembaruan untuk sebuah aplikasi web yang sudah berjalan tanpa terjadi *downtime*?

1.3 Batasan Masalah

Dari permasalahan yang telah diuraikan di atas, terdapat beberapa batasan masalah pada tugas akhir ini, yaitu:

1. Semua *container* dari aplikasi web akan berjalan hanya pada satu *server master*.
2. Perhitungan algoritma skala akan menggunakan *Proactive Model* dan *Reactive Model* untuk menentukan jumlah *container* yang dibentuk atau dihapus.
3. Aplikasi web yang diuji coba hanya akan melakukan komputasi tanpa terhubung dengan layanan luar, seperti koneksi ke suatu basis data dan layanan REST API.

1.4 Tujuan

Tujuan dari pembuatan tugas akhir ini adalah membuat sistem yang dapat melakukan skalabilitas secara otomatis terhadap aplikasi web berbasis *docker* dengan menggunakan *Proactive Model* dan *Reactive Model* untuk menentukan sumber daya yang diperlukan oleh aplikasi. Selain itu, sistem ini juga memiliki fitur *hot-upgrade*, yaitu dapat melakukan pembaruan terhadap aplikasi yang sudah terpasang tanpa terjadi *downtime*.

1.5 Manfaat

Tugas akhir ini diharapkan dapat memberikan kemudahan seorang pengembang aplikasi berbasis web dengan tidak perlu melakukan konfigurasi *server* secara langsung untuk melakukan skalabilitas aplikasinya. Pengembang tidak perlu mengawasi aplikasinya saat terjadi perubahan permintaan yang tiba-tiba

melonjak tinggi kemudian mengaturnya supaya bisa mengatasi permintaan tersebut. Sistem akan secara otomatis melakukan hal tersebut. Untuk menggunakan sistem ini, pengembang hanya perlu menyimpan aplikasinya di sebuah *server docker repository* dan sistem akan mengelolanya lebih lanjut.

(Halaman ini sengaja dikosongkan)

BAB II

TINJAUAN PUSTAKA

2.1 *Publish/subscribe*

Publish/subscribe muncul sebahai paradigma komunikasi yang populer untuk sistem terdistribusi dalam skala yang besar. dalam *publish/subscribe consumer* akan berlangganan ke suatu *event* yang diinginkan. terlepas dari kegiatan *consumer*, ada *event producer* yang akan menerbitkan suatu *event*. jika event yang diterbitkan oleh produser cocok dengan *event* yang dilanggani oleh *consumer*, *event* tersebut akan dikirim kepada *consumer* secara *asynchronus*. Interaksi ini difasilitasi oleh *middleware publish/subscribe*. *middleware publish/subscribe* dapat dipusatkan menjadi sebuah *node* tunggal yang berperan sebagai *broker* dari sebuah *event* atau dipisahkan menjadi kumpulan beberapa *node broker* dari sebuah *event*.

pada dasarnya, *publish/subscribe* dibagi dari dua jenis yaitu: *topic-based* dan *content-based*. Pada *topic-based publish-subscribe*, *event* diterbitkan melalui sebuah topik dan *consumer event* akan berlangganan topik tersebut untuk mendapatkan data dari suatu *event*. berlangganan pada kasus *topic-based* tidak didukung pemilahan data dari suatu *event*. contohnya, *consumer* akan menerima semua data dari suatu *event* yang diterbikan pada suatu topik. pada *content-based publish-subscribe*, berlangganan pada kasus ini didukung oleh fitur pemilahan yang diterapkan pada suatu *event* yang diterbitkan. data yang dipilah oleh *consumer* pada suatu *event* yang berlangganan akan dikirimkan ke *consumer*. [2]

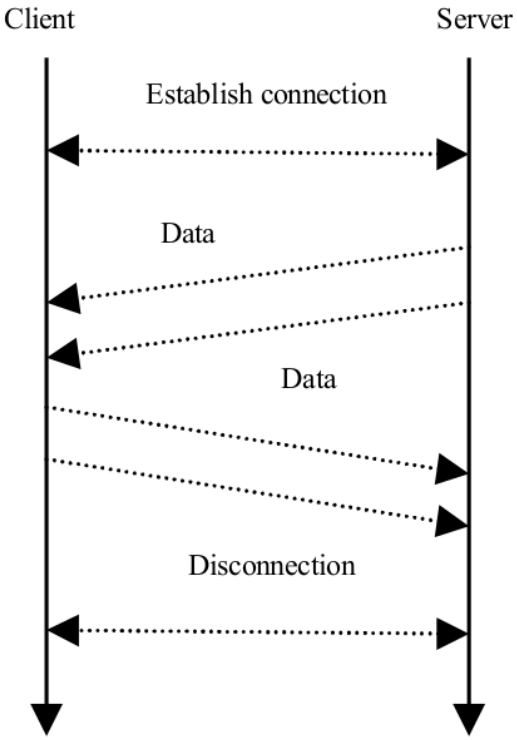
2.2 *Websocket*

Websocket adalah protokol berbasis TCP yang menyediakan channel komunikasi full-duplex antara server dan client melalui koneksi TCP tunggal. dibandingkan dengan skema komunikasi web real-time tradisional, protokol websocket menghemat

banyak sumber daya bandwidth pada jaringan, sumber daya server, dan performa real-time yang sangat jauh lebih baik dibanding websocket tradisional. Websocket adalah protocol berbasis TCP yang independen. Websocket hanya berhubungan dengan HTTP yang memiliki handshake yang diterjemahkan oleh HTTP server sebagai pengembangan dari sebuah request. Websocket terdiri dari dua bagian yaitu: handshake dan data transfer.

Untuk membuat koneksi Websocket, client harus mengirimkan request HTTP kepada server. setelah itu protokol akan diupgrade menjadi protokol Websocket. setelah itu server akan mengenali tipe request berdasarkan header pada HTTP. Protokol akan diupgrade menjadi Websocket apalbila diminta oleh Websocket, dan kedua kubu (client dan server) akan memulai komunikasi full-duplex, yang berarti client dan server dapat bertukar data kapanpun sampai salah satu dari client atau server menutup koneksi tersebut. Model komunikasi Websocket dapat dilihat pada gambar 2.1

Websocket memiliki kemampuan yang lebih baik dalam berkomunikasi dibandingkan dengan skema komunikasi tradisional, dimana komunikasi terjadi secara realtime. sekali koneksi sudah berhasil dibuat, server dan client melakukan aliran data dua arah, dimana aktivitas tersebut meningkatkan mempuan server untuk mengirim data. Bandingkan dengan protokol HTTP, dimana informasi yang dikirimkan lebih ringkas dan mengurangi transmisi dari data yang redundan. Dengan skala user yang besar dan kebutuhan komunikasi realtime yang tinggi, menurunkan beban pada jaringan akan menjadi keuntungan dibanding komunikasi realtime secara tradisional.



Gambar 2.1: Model Komunikasi *Websocket*

[3].

2.3 *SNMP*

(Halaman ini sengaja dikosongkan)

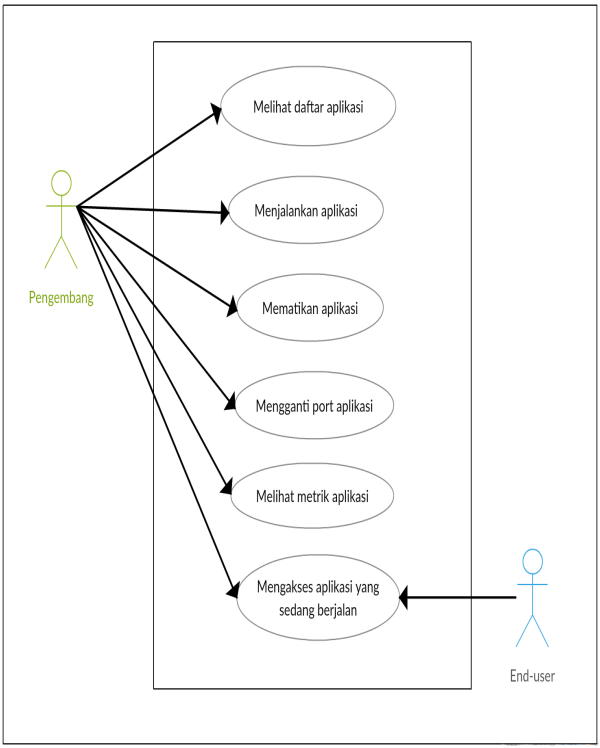
BAB III

DESAIN DAN PERANCANGAN

Pada bab ini dibahas mengenai analisis dan perancangan sistem.

3.1 Kasus Penggunaan

Terdapat dua aktor dalam sistem ini, yaitu pengembang (administrator) dan *end-user* (pengguna) dari aplikasi web yang dikelola oleh sistem. Diagram kasus penggunaan digambarkan pada Gambar 3.1.



Gambar 3.1: Diagram Kasus Penggunaan

Diagram kasus penggunaan pada Gambar 3.1 dideskripsikan masing-masing pada Tabel 3.1.

Tabel 3.1: Daftar Kode Kasus Penggunaan

Kode Kasus Penggunaan	**Nama Kasus Penggunaan**	**Keterangan**
UC-0001	Melihat daftar aplikasi *web*.	Pengembang dapat melihat daftar aplikasi web yang ada di *docker registry*.
UC-0002	Menjalankan aplikasi *web*.	Pengembang dapat menjalankan aplikasi web yang ada di *docker registry* jika aplikasi dalam keadaan mati.
UC-0003	Mematikan aplikasi *web*.	Pengembang dapat mematikan aplikasi web yang ada di *docker registry* jika aplikasi sedang berjalan.
UC-0004	Mengganti port aplikasi *web*.	Pengembang harus dapat mengganti port yang disediakan oleh aplikasi agar bisa diakses dari luar.
UC-0005	Melihat metrik sumber daya aplikasi *web*.	Pengembang dapat melihat metrik dari sebuah aplikasi, yaitu jumlah *container* dan jumlah request ke aplikasi.

Tabel 3.1: Daftar Kode Kasus Penggunaan

Kode Kasus Penggunaan	Nama Kasus Penggunaan	Keterangan
UC-0006	Mengakses aplikasi yang sedang berjalan.	Pengembang dan *end-user* dapat mengakses aplikasi yang sudah berjalan sesuai dengan domain yang diberikan oleh sistem.

3.2 Arsitektur Sistem

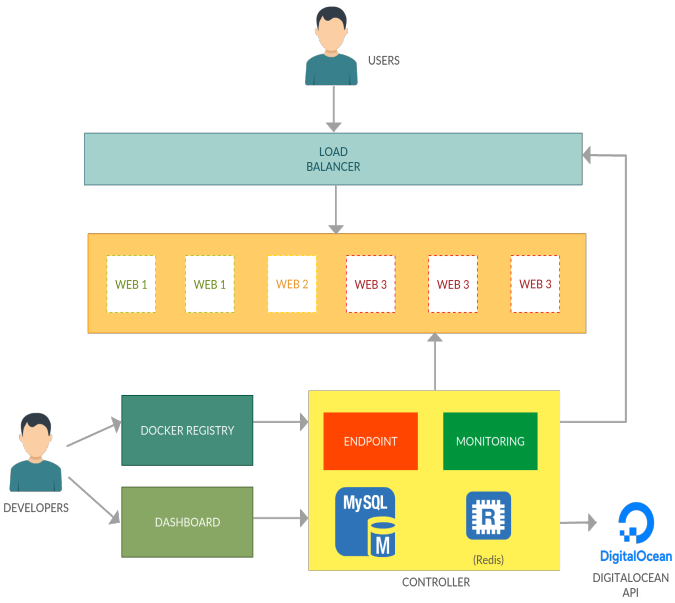
Pada sub-bab ini, dibahas mengenai tahap analisis dan kebutuhan bisnis dan desain dari sistem yang akan dibangun.

3.2.1 Desain Umum Sistem

Sistem yang akan dibuat yaitu sistem yang dapat melakukan skalabilitas secara otomatis terhadap aplikasi web berbasis *docker* dengan menggunakan *Proactive Model* dan *Reactive Model*.

Sistem ini akan digunakan oleh pengguna, yaitu *end-user* dari aplikasi yang mana hanya bisa melakukan akses terhadap suatu aplikasi yang sudah berjalan. Selain itu juga digunakan oleh pengembang, yaitu orang mengelola aplikasi. Pengguna dari sistem ini hanya bisa melakukan akses atau permintaan kepada load balancer untuk mengakes aplikasi tertentu. Sedangkan pengembang dapat menambahkan dan memperbarui aplikasi web ke *Private Docker Repository*. Sistem akan secara otomatis akan memperbarui aplikasinya sesuai dengan yang aplikasi terakhir yang dimasukkan oleh pengembang. Penjelasan secara umum arsitektur sistem akan diuraikan pada Gambar 3.2. Secara garis besar, ada empat *server* yang akan digunakan

membangun sistem ini, yaitu *load balancer*, *master host*, *controller*, dan *docker registry*.



Gambar 3.2: Desain Umum Sistem

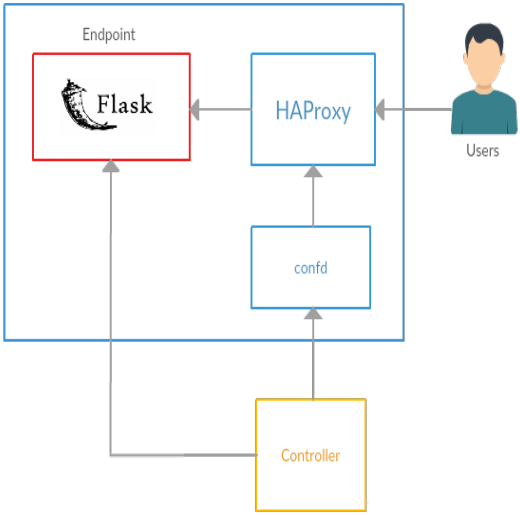
3.2.2 Desain Load Balancer

Load balancer digunakan sebagai pembagi beban aplikasi dan merekam permintaan dari pengguna. *Load balancer* sendiri akan dibangun menggunakan perangkat lunak Haproxy.

Akses ke load balancer akan atur oleh DNS dari DigitalOcean. Pengguna bisa mengakses aplikasi web melalui domain yang disediakan. Saat melakukan akses domain, DNS DigialOcean akan mengarahkan permintaan ke server load balancer. Dari *server load balancer*, HAProxy akan membaca domain mana yang diinginkan oleh pengguna. Setelah

mengetahui domain yang dituju, HAProxy akan mengarahkan permintaan tersebut menuju *container* dari aplikasinya.

Haproxy akan menggunakan UNIX *socket interface* sebagai perantara untuk mengelola log. Dengan memanfaatkan log tersebut, *load balancer* menyediakan API tentang status dirinya yang dibutuhkan oleh *server controller*. Log yang terdapat pada *server* ini akan disajikan dengan memberikan sebuah *endpoint*. *Endpoint* akan dibangun dengan menggunakan kerangka kerja Flask. Secara umum, arsitektur dari *server load balancer* dapat dilihat pada Gambar 3.3



Gambar 3.3: Desain Load Balancer

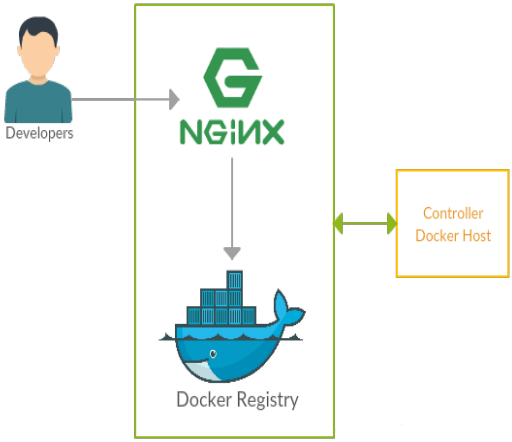
Agar load balancer bisa mengetahui aplikasi yang sedang aktif, konfigurasi dari HAProxy akan dikelola oleh perangkat lunak bernama confd. Perangkat lunak ini akan membaca dari *server controller* tentang konfigurasi yang paling baru. Jika

terjadi perubahan data, maka confd akan menyesuaikan konfigurasi dari HAProxy agar sesuai dengan aplikasi yang tersedia.

3.2.3 Desain Private Docker Registry

Private docker registry, yang selanjutnya hanya akan disebut *docker registry*, digunakan untuk menyimpan *docker image* aplikasi web yang dikelola oleh pengembang. *Docker registry* akan dibangun di atas *server* yang sudah memiliki *docker engine* yang mana akan menjalankan dua *container*, yaitu *container docker registry* dan *nginx*. *Container docker registry* akan dibangun di atas *image docker registry* yang secara resmi disediakan oleh Docker. Kemudian ada *container* nginx yang akan digunakan untuk menghubungkan *container docker registry* dengan jaringan luar. *Nginx* digunakan agar *container docker registry* bisa terlindungi dengan memanfaatkan fitur *auth* yang dimilikinya. Selain itu juga proses pengaturan SSL dan domain relatif lebih mudah dan banyak referensi yang bisa digunakan dibandingkan dengan langsung memasangnya pada *container docker registry*. Untuk membangun rancangan sistem tersebut, menggunakan *docker compose*, yaitu sebuah perangkat lunak yang digunakan untuk mendesain rancangan sistem yang menggunakan *docker* sebagai basisnya dan mengelola *container* yang berjalan. Rancangan umum dari *docker registry* seperti yang digambarkan pada Gambar 3.4.

Untuk menambah pengamanan dari *docker registry* ini, aksessnya akan melalui protokol HTTPS. SSL yang akan dipakai disediakan oleh Let's Encrypt, sebuah lembaga yang menyediakan SSL secara gratis kepada umum. Selain itu, untuk mempermudah akses ke *docker registry*, akan disediakan URL yang bisa digunakan oleh pengembang, yaitu `https://registry.nota-no.life`.



Gambar 3.4: Desain Docker Registry

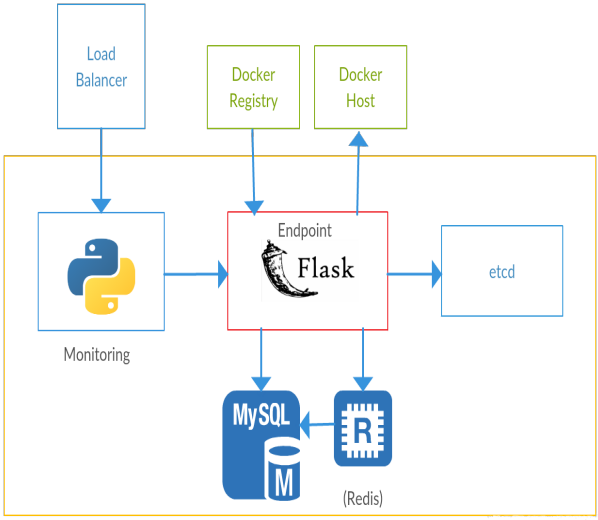
3.2.4 Desain *Server Controller*

Server controller akan digunakan untuk memantau keseluruhan sistem. Terdapat dua subsistem utama pada *server* ini, yaitu bagian yang menangani *endpoint* dan bagian yang melakukan *monitoring* terhadap sistem. Secara umum, arsitektur rancangan dari *server controller* dapat dilihat pada Gambar 3.5.

Teknologi yang akan digunakan pada *server* ini yang pertama adalah Flask, untuk membuat *endpoint*. *Endpoint* tersebut akan terhubung dengan MySQL, sebagai tempat penyimpanan data dari sistem, seperti data image pada *docker registry*, *container* yang sedang berjalan, dan domain yang didaftarkan pada DNS DigitalOcean. Lalu ada Redis, digunakan sebagai *task queue* untuk pemrosesan data yang diberikan oleh *endpoint*. Lalu terakhir, *endpoint* akan terhubung dengan etcd, sebagai wadah untuk menyimpan konfigurasi *load balancer*.

Selanjutnya terdapat *script monitoring* menggunakan bahasa pemrograman Python. Fungsinya adalah untuk mengolah data

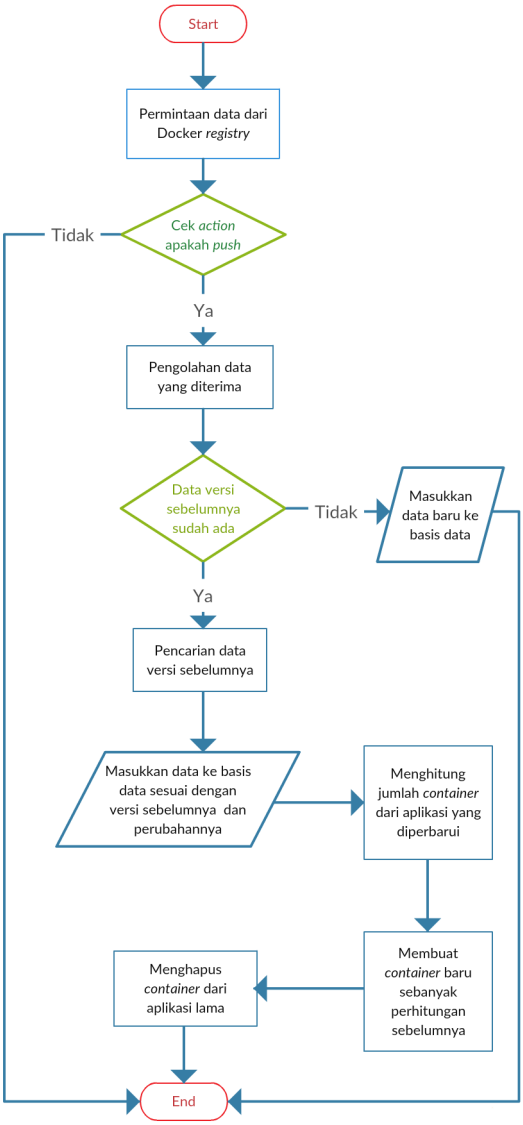
yang ada pada sistem, seperti data pada *load balancer* dan *master host*. *Script* ini juga yang akan menentukan keputusan untuk menambahkan atau mengurangi sumber daya yang ada pada sistem.



Gambar 3.5: Desain Controller

3.2.4.1 Perancangan Endpoint

Endpoint yang dibuat di *server* ini akan digunakan untuk berkomunikasi dengan *host* lain. Pertama, terdapat sebuah *endpoint* untuk menangkap notifikasi yang diberikan oleh *docker registry* jika terjadi suatu kejadian. Penjelasan cara kerja dari *endpoint* ini ditunjukkan pada Gambar 3.6. Selain itu juga disediakan *endpoint* untuk kebutuhan dasbor, seperti untuk mendaftar aplikasi yang tersedia, informasi secara rinci dari aplikasi yang ada, dan status dari aplikasi.



Gambar 3.6: Diagram Alur Pengelolaan *Notification* Docker *Registry*

3.2.4.2 Perancangan Sistem Monitoring

Sistem monitoring merupakan subsistem yang ada *server controller*. Sistem ini bertugas untuk mengolah data dan memantau sistem secara keseluruhan. Data yang akan diolah berasal dari *master host* dan *load balancer*. Dari *server master host* akan didapatkan data penggunaan CPU dan *memory* dari *container* aplikasi-aplikasi yang sedang berjalan. Kemudian dari *server load balancer* akan didapatkan data jumlah *request* ke aplikasi. Dari data-data tersebut, proses perhitungan dengan menggunakan *Proactive Model* dan *Reactive Model* akan dilakukan.

Proactive Model akan menggunakan data dari jumlah *request* yang ada pada server *load balancer* untuk melakukan prediksi berapa jumlah *request* kedepannya dengan menggunakan perhitungan berdasarkan ARIMA.

Reactive Model akan menggunakan data CPU dan *memory* untuk menentukan apakah sumber daya dari aplikasi sudah melebihi batas yang ditentukan atau tidak. Jika sudah melebihi batas atas, maka akan ditentukan berapa jumlah *container* yang diperlukan untuk mengatasi hal tersebut.

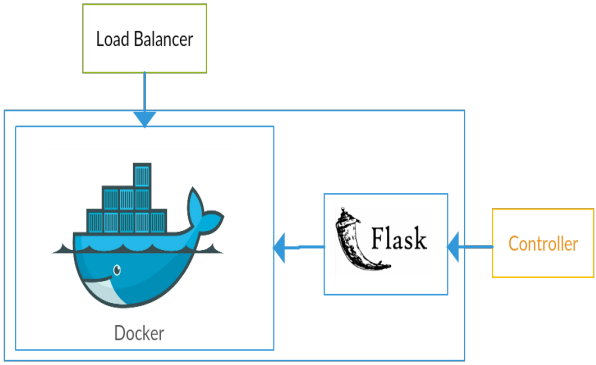
Dari perhitungan di atas, maka sistem ini akan melakukan penambahan atau pengurangan *container* berdasarkan kebutuhan. Sistem ini akan memberitahu *master host* untuk penyesuaian sumber daya dan juga melakukan pembaruan konfigurasi dari aplikasi. Pembaruan tersebut berguna agar *server load balancer* bisa mengetahui keaadaan terbaru dari aplikasi dan *container* yang berjalan di atasnya.

3.2.4.3 Penggunaan Task Queue

Pada *server controller* ini, akan banyak proses yang berjalan dalam jangka waktu yang panjang karena melakukan banyak eksekusi perintah di dalamnya. Jika proses tersebut berada di

dalam fungsi yang dipanggil melalui protokol HTTP, maka umpan balik yang diberikan akan menunggu semua proses yang ada di dalamnya selesai. Hal tersebut akan membuat klien yang melakukan permintaan perlu menunggu dan merupakan hal yang tidak efisien. Untuk mengatasi hal tersebut, proses yang memerlukan banyak perintah, akan dimasukkan ke dalam sebuah *queue* atau yang bisa disebut sebagai *task queue*. Untuk task queue nya akan menggunakan Redis sebagai wadah untuk menampung perintah atau fungsi yang akan dikerjakan. Lalu, untuk menjalankan perintah atau fungsi yang sudah masuk ke dalam Redis, akan menggunakan *worker* yang disediakan oleh pustaka Python bernama RQ.

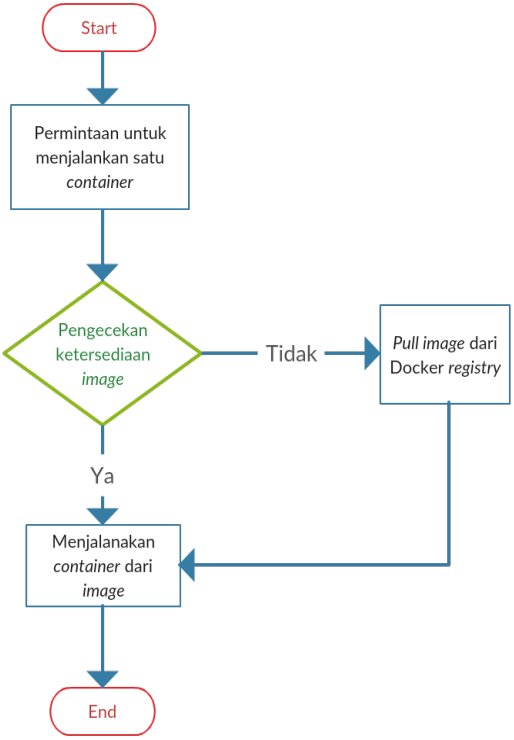
3.2.5 Desain Master Host



Gambar 3.7: Desain Master Host

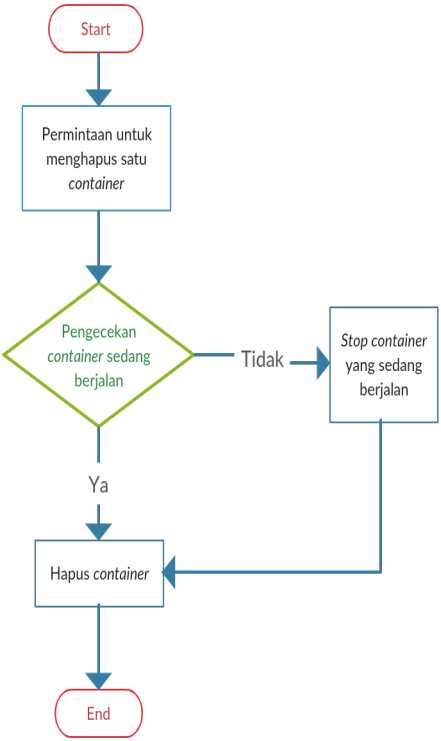
Master host merupakan sebuah *server* yang akan menjalankan apalikasi-aplikasi yang diinginkan oleh pengembang. *Host* ini memiliki *docker engine* yang berguna untuk menjalankan *container* dari aplikasi-aplikasi. Pada host ini akan dipasang `dockerpy` sebagai API untuk mengambil

informasi-informasi dari *container*, seperti penggunaan *memory* dan CPU. Data-data digunakan oleh *server controller* untuk mengetahui informasi *container* yang ada dan juga digunakan untuk memberitahu *host* apakah harus menambah atau mengurangi *container* dari sebuah aplikasi. *server controller* dapat mengakses data tersebut melalui layanan yang dibuat menggunakan perangkat kerja Flask. Dengan menggunakan perangkat kerja tersebut, layanan yang diberikan akan berjalan pada protokol HTTP.



Gambar 3.8: Diagram Alur Menjalankan *Container*

Host akan mengambil data aplikasi yang berupa *docker image* yang akan di jadikan *container* dari *docker registry*. Secara umum, perancangan dari sistem ini dapat dilihat pada Gambar 3.7. Proses untuk menambahkan *container* baru, diagram alurnya dapat dilihat pada Gambar 3.8. Lalu, proses untuk menghapus *container* yang sedang berjalan, diagram alurnya ditunjukkan pada Gambar 3.9.



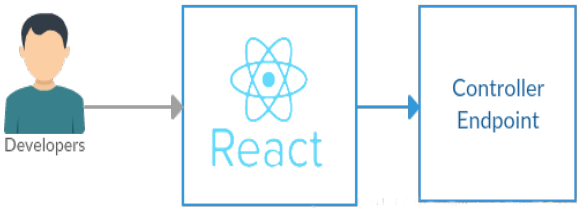
Gambar 3.9: Diagram Alur Menghentikan *Container*

Terakhir, *end-user* bisa melakukan akses terhadap aplikasi melewati *load balancer*. *Load balancer* yang akan mengarahkan

pengguna untuk mengakses *container* yang ada dari aplikasi.

3.2.6 Desain Dasbor

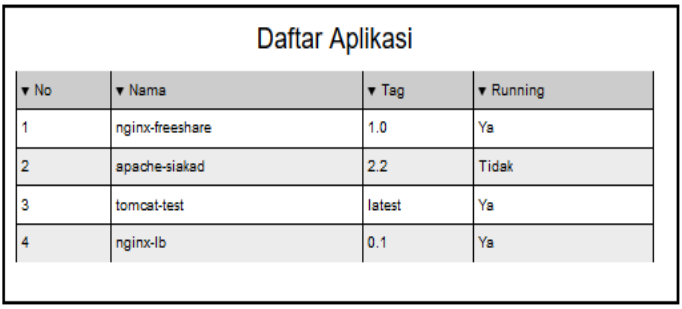
Dasbor adalah halaman yang digunakan sistem administrator untuk mengelola aplikasi dan menampilkan status metrik dari aplikasi. Dasbor adalah aplikasi berbasis web yang dibangun menggunakan React sebagai tampilan depan halaman (*frontend*) dan Flask sebagai *backend*. Secara umum, arsitektur dari dasbor seperti pada Gambar 3.10.



Gambar 3.10: Desain Dasbor

Halaman depan akan menggunakan Material UI untuk mendapatan tampilan yang sederhana dan nyaman digunakan. Dasbor digunakan oleh sistem administrator untuk berinteraksi dengan sistem. Dasbor memiliki menu-menu yang memudahkan pengelolaan sistem. Menu-menu terdapat pada dasbor antara lain:

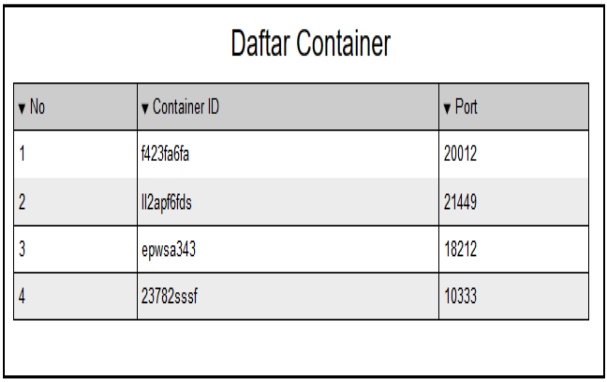
- Daftar Aplikasi
  Halaman utama dari dasbor akan menampilkan daftar aplikasi yang ada pada *docker registry*. Secara langsung, pengembang bisa langsung melihat status dari aplikasi, apakah sedang berjalan atau mati. Antar muka rancangannya ditunjukkan pada Gambar 3.11.



Gambar 3.11: Desain Antar Muka Dasbor Beranda

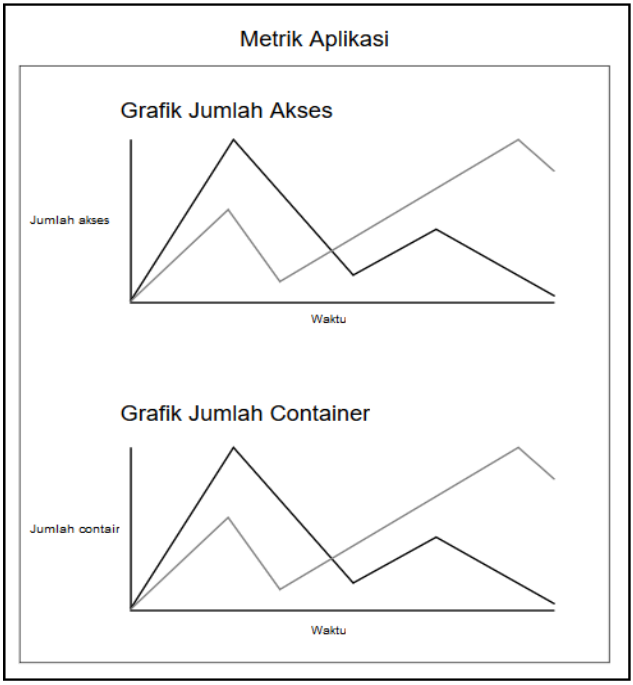
Gambar 3.12: Desain Antar Muka Informasi Aplikasi

- Informasi Aplikasi
  Jika pengembang memilih salah satu dari aplikasi yang ada pada beranda, maka akan diarahkan ke halaman informasi dari aplikasi. Pada halaman ini, pengembang dapat menentukan port dari aplikasi, menjalankan aplikasi, menghentikan aplikasi, dan melihat versi aplikasi sebelumnya. Rancangan antar muka untuk halaman ini seperti yang digambarkan pada Gambar 3.12.



Gambar 3.13: Desain Antar Muka Daftar Container

- Daftar Container
  Pengembang juga disediakan halaman yang memperlihatkan data *container* yang sedang berjalan dari suatu aplikasi. Pada data ini diinformasikan tentang ID dan port dari *container* yang sedang berjalan. Antar muka untuk halaman ini seperti pada Gambar 3.13.
- Metrik Aplikasi
  Pada halaman ini, pengembang dapat melihat keadaan aplikasi, yaitu status jumlah akses dan jumlah *container* pada waktu tersebut. Rancangan halaman ini ditunjukkan seperti pada Gambar 3.14.



Gambar 3.14: Desain Antar Muka Metrik Aplikasi

(Halaman ini sengaja dikosongkan)

BAB IV

IMPLEMENTASI

Bab ini membahas implementasi sistem Pengendali Elastisitas secara rinci. Pembahasan dilakukan secara rinci untuk setiap komponen yang ada, yaitu: *docker registry*, *master host*, *controller*, *load balancer*, dan dasbor.

4.1 Lingkungan Implementasi

Lingkungan implementasi dan pengembangan dilakukan menggunakan. Perangkat lunak yang digunakan dalam pengembangan adalah sebagai berikut:

- Sistem Operasi Linux Ubuntu Server 14.04 LTS
- Docker CE
- Python 2.7
- Redis
- MySQL
- Flask
- JMeter 3.2

4.2 Implementasi Docker *Registry*

Docker registry dibangun pada *server* dengan IP `139.59.97.244` dan dapat diakses dari domain `https://registry.nota-no.life`. *Docker registry* dapat digunakan setelah melakukan pemasangan *server docker registry* seperti yang dijelaskan pada Lampiran A.

4.2.1 Pengaturan *Notification*

Setelah melakukan pemasangan, selanjutnya adalah menambahkan konfigurasi untuk memberitahu `server controller` jika terjadi suatu kejadian pada *docker registry*, seperti *push* dan *pull image*. Pada pengembangan sistem ini, kejadian yang diperlukan adalah *push*, yaitu saat pengembang

memasukkan aplikasi baru atau memperbarui aplikasi yang sudah ada di Docker *registry*. Jika pengembang melakukan *push* suatu aplikasi ke *docker registry*, baik itu merupakan aplikasi pertama yang dimasukkan atau memperbarui aplikasi yang sudah ada, maka *docker registry* akan memberitahukan kejadian tersebut kepada `server controller`. Untuk melakukan hal tersebut, buat folder `registry` di dalam folder `docker-registry`. Kemudian di dalamnya buat berkas dengan nama `config.yml` seperti Kode Sumber IV.1.

```
version: 0.1
log:
  fields:
    service: registry
storage:
  cache:
    blobdescriptor: inmemory
  filesystem:
    rootdirectory: /var/lib/registry
http:
  addr: :5000
  headers:
    X-Content-Type-Options: [nosniff]
health:
  storagedriver:
    enabled: true
    interval: 10s
    threshold: 3
notifications:
  endpoints:
    - name: alistener
      url: http://controller.nota-no.life/
          docker-registry-endpoint
      timeout: 60000ms
```

```
        threshold: 5
        backoff: 1s
```

Kode Sumber IV.1: Isi config.yml

Pada Kode Sumber IV.1 dapat dilihat bahwa *notification* akan dikirimkan ke *server host* melalui *endpoint* `http://controller.nota-no.life/docker-registry-endpoint`. *Timeout* yang digunakan adalah 60000 ms (*milisecond*) atau 1 menit. *Timeout* berguna jika `server controller` tidak memberikan balasan dalam waktu 1 menit, maka Docker *registry* akan mencoba untuk mengirim ulang pemberitahuannya sampai dipastikan bahwa `server controller` sudah menerima pesannya dengan baik.

4.2.2 Melakukan Akses Terhadap *Docker Registry*

Docker registry dapat diakses dari komputer yang memiliki mesin *docker* di dalamnya. Sebelum dapat mengakses *docker registry*, perlu menambahkan CA (*Certification Authority*) dari Let's Encrypt pada komputer yang digunakan. Untuk melakukan proses tersebut buka folder `/usr/local/share/ca-certificates` dan buat berkas dengan nama `docker-regsitry.crt` di dalamnya. Masukkan Kode Sumber B.1 untuk mengisi berkasnya. Setelah itu simpan berkas dan perbarui CA dengan menjalankan `sudo update-ca-certificates`. Langkah terakhir adalah melakukan restart terhadap *service docker* yang sedang berjalan dengna menggunakan perintah `sudo service docker restart`. Setelah proses di atas, selanjutnya adalah melakukan *login* dengan menggunakan perintah `docker login https://registry.nota-no.life`. Masukkan *username* dan *password* yang sesuai, dan jika berhasil masuk akan muncul tulisan `Login Succeeded` yang menandakan sudah berhasil melakukan akses terhadap *docker registry*.

4.2.3 Menambahkan dan memperbarui aplikasi

Setelah berhasil terhubung dengan *docker registry*, selanjutnya dapat mencoba untuk melakukan interaksi dengan menambahkan aplikasi baru ke dalamnya. Untuk melakukan percobaan, penulis melakukannya dengan perangkat lunak nginx dalam format *docker* yang disediakan oleh Docker Hub. Untuk melakukan unduh, jalankan perintah berikut pada Kode Sumber IV.2.

```
docker pull nginx
```

Kode Sumber IV.2: Perintah *Pull* Nginx

Setelah berhasil diunduh, selanjutnya jalankan perangkat lunak nginx dengan menggunakan perintah yang tertera pada Kode sumber IV.3.

```
docker run ––name tesnginx nginx
```

Kode Sumber IV.3: Perintah Menjalankan *Image* Nginx

Parameter `--name` berguna untuk memberikan nama pada *container* agar mudah dikenali dimana lokasi aplikasi saat dijalankan. Pada kasus ini *container* diberi nama dengan `tesnginx`. Setelah menjalankannya, *container* yang terbentuk dapat digunakan lebih lanjut, misalnya dengan mengubah data yang ada didalamnya, menambahkan fitur baru, atau hanya sekedar mengganti nama dari aplikasi. Setelah melakukan modifikasi terhadap *container*, jika ingin membuat *images* baru dari *container* tersebut, maka hal pertama yang harus dilakukan adalah menghentikan *container* yang sedang berjalan dengan menggunakan perintah `docker stop tesnginx` untuk kasus yang digunakan oleh penulis. Setelah itu lakukan *commit* dengan menjalankan perintah seperti pada Kode Sumber IV.4.

```
docker commit tesnginx registry.nota-no.
    life/ tesnginx:1.0
```

Kode Sumber IV.4: Perintah *Commit Container* Nginx

Parameter keempat pada perintah di atas adalah `registry.nota-no.life/tesnginx:1.0` yang merupakan penamaan *image* yang terbentuk. Parameter tersebut memiliki tiga bagian dengan pola seperti `[URL]/[nama]:[versi]`. Artinya membuat *image* dengan URL *repository* ada pada `registry.nota-no.life`. Kemudian nama dari *image*-nya sendiri adalah `tesnginx` dan versinya adalah `1.0`. Setelah melakukan commit, maka *image* baru akan terbentuk. Langkah terakhir adalah melakukan *push image* ke *docker registry* yang tersedia dengan menggunakan perintah seperti Kode Sumber IV.5.

```
docker push registry.nota-no.life/tesnginx
    :1.0
```

Kode Sumber IV.5: Perintah *Push Image* Terbaru Nginx

Untuk memperbarui aplikasi yang sudah diungguh pada *docker registry*, proses yang dilakukan sama dengan menambahkan aplikasi baru.

4.3 Implementasi Master Host

Master Host merupakan *server* yang digunakan untuk menjalankan semua *container* dari aplikasi. *Server* ini memiliki IP publik, yaitu `128.199.182.29`. *Server* ini menyediakan sebuah *endpoint* dengan port 5000 yang digunakan oleh *server* pengendali untuk melakukan komunikasi. *Endpoint* dibangun dengan menggunakan perangkat kerja Flask. Lalu, interaksi

dengan *docker deamon* menggunakan pustaka docker-py. Penggunaan pustaka tersebut agar interaksi dapat dilakukan dengan mudah.

Berikut adalah penjelasan *endpoint* yang disediakan oleh *server* ini:

- `/start_container`
  Rute ini memiliki metode `POST` dan berguna untuk menjalankan atau memulai suatu *container* dari suatu aplikasi. *Endpoint* ini akan dipanggil oleh *server* Controller jika ada permintaan dari pengguna untuk menjalankan aplikasi dan membuat *container* baru untuk memenuhi kebutuhan aplikasi. Saat akan membuat *container* baru, akan dilakukan pengecekan apakah *image* dari aplikasi yang akan dijalankan sudah ada atau belum. Jika *image* tidak ada, maka terlebih dahulu akan melakukan *pull* dari *server docker registry*.

- `/delete_container/<container_id>`
  Rute ini memiliki metode `GET` dan berguna untuk menghentikan aplikasi dan menghapus satu *container* dari aplikasi yang sedang berjalan. Rute ini digunakan saat pengguna ingin menghentikan aplikasinya atau aplikasi kelebihan jumlah *container*, sehingga harus mengurangi *container* yang sedang berjalan. Untuk menghapus container yang sedang berjalan, maka *container* pertama kali harus dihentikan prosesnya, kemudian menghapusnya.

- `/container_info`
  Rute ini digunakan untuk mendapatkan data dari satu atau lebih *container*. Data yang akan dihasilkan yaitu penggunaan *memory* dan CPU dari *container* yang sedang berjalan.

4.4 Implementasi Server Controller

Server controller merupakan *server* yang akan mengelola keseluruhan data dari sistem. Pada *server* ini semua keputusan yang dilakukan oleh sistem dilakukan, seperti menambahkan dan menghapus *container*, menjalankan, memperbarui, dan menghapus aplikasi, dan memperbarui konfigurasi *load balancer*. *Server controller* memilik IP 128.199.250.137 dan domain *http://controller.nota-no.life* yang digunakan oleh dasbor. Selain itu, *server* dapat diakses melalui port 5000.

4.4.1 *Endpoint Docker Regsitry*

Server docker registry akan mengirimkan suatu kejadian jika terjadi perubahan pada datanya. Oleh karena itu, *server* Controller memiliki rute `/docker-registry-endpoint` dengan metode `POST` yang akan menangkap pesannya.

Data yang diterima berupa data dalam format JSON. Data yang akan diproses semuanya berada di dalam *key* `events`. Di dalam data `events`, selanjutnya *key* yang akan digunakan adalah *key* `action` dan `target`.

Key `action` akan memberitahu kejadian apa yang sedang terjadi. Umumnya isi yang sering muncul adalah *string* `push` dan `pull`. *string* `push` menunjukkan adanya kejadian saat pengembang memasukkan aplikasi baru atau memperbarui aplikasi yang sudah ada di *docker registry*. Lalu *string* `pull` adalah nilai yang menunjukkan kejadian saat ada suatu host yang mengambil sebuah *image* dari *docker registry*. Dalam pengembangan sistem ini, proses hanya akan dilanjutkan jika bernilai `push` yaitu saat ada yang melakukan perubahan data *image* pada *docker registry* dan mengabaikan jika ada *host* yang melakukan unduh *image*.

Key `target` memiliki beberapa data di dalamnya, dan *key* yang digunakan adalah `host`, `repository`, `tag`, dan

`mediatype`. *Key* `host` menunjukkan letak alamat, dalam bentuk URL atau IP, dari *docker registry*. *Key* `repository` merupakan nama dari *image* atau aplikasi yang ada di dalam *event*. *Key* `tag` menunjukkan versi atau jenis dari *image*. Misalnya bernilai `1.0`, maka menunjukkan bahwa *image* tersebut berada dalam versi 1.0. Lalu ada juga `tag` yang berisi nilai `demo`, berarti kemungkinannya *image* tersebut untuk keperluan percobaan.

Lalu terakhir ada *key* `mediatype`. Dalam mengirimkan *notification*, *docker registry* tidak hanya melakukannya sekali dalam satu *event*, tapi bisa saja beberapa kali, sesuai dengan *event* yang terjadi. Misal *event* `push`, bisa saja untuk melakukan *event* tersebut diperlukan lebih dari satu proses Untuk menanganinya. Masing-masing proses tersebutlah yang akan dikirimkan. Agar tidak terjadi tabrakan pemroses *event* yang sama, diperlukan pengecekan `mediatype`. Proses hanya akan dilanjutkan jika *key* tersebut bernilai `application/vnd.docker.distribution.manifest.v2+json`.

Selanjutnya adalah pengecekan apakah *image* sudah ada di dalam basis data. Jika data *image* belum ada, maka data yang baru tersebut akan dimasukkan ke dalam basis data. Proses akan berakhir sampai disana jika itu merupakan *image* baru. Untuk menjalankan aplikasinya bisa melalui dasbor. Lalu jika *image* yang diproses sudah ada ada di dalam basis data, maka masukkan data baru ke dalam basis data. Biasanya penambahan data baru ini yang berbeda hanya data `tag`-nya saja. Selanjutnya adalah mengecek apakah *image* tersebut sedang berjalan atau dengan kata lain ada *container* yang sedang aktif menggunakan *image* itu. Jika ada maka buat *container* dari *image* yang baru sejumlah *container* dari image lama. Setelah *container* semua *container* baru terbentuk, baru hapus *container* dari *image* lama dan perbarui pengaturan *load balancer*.

Semua pemrosesan data yang dikirim oleh *docker registry* dilakukan dengan memasukkannya ke dalam Redis. Kemudian

ada *worker* yang akan membaca data yang masuk ke Redis dan menjalankan fungsinya sehingga pemrosesan di atas akan dilakukan secara *asynchronous*.Hal tersebut dilakukan agar *docker registry* bisa mendapatkan balasan secepat mungkin dari *server* Controller. Jika tidak dimasukkan ke dalam Redis, maka pemrosesan yang dilakukan di atas harus diselesaikan terlebih dahulu sebelum bisa memberikan balasan.

4.4.2 Skema Basis Data Controller Menggunakan MySQL

Untuk mengelola data yang ada pada sistem, dibutuhkan basis data sebagai tempat penyimpanannya, yaitu MySQL. MySQL *server* yang digunakan adalah versi 5.5.55. Data yang disimpan antara lain adalah data dari *image* yang ada di *docker registry*, data *container* yang sedang berjalan pada *server master*, dan data domain untuk masing-masing *image* atau aplikasi. MySQL *server* memiliki definisi tabel `images`, `containers`, dan `domains`. Table `images` digunakan untuk menyimpan data *image* yang ada di *server docker registry*. Berikut definisi tabel `images` pada Table 4.1.

Tabel 4.1: Tabel images

No	Kolom	Tipe	Keterangan
1	id	int	Sebagai primary key pada tabel, nilai awal adalah `AUTO_INCREMENT`.
2	host	varchar(50)	Menunjukkan URL atau IP dari *docker registry* untuk mengunduh *image*.
3	repository	varchar(50)	Nama aplikasi.
4	tag	varchar(50)	Versi atau label yang diberikan kepada sebuah *image*.

Tabel 4.1: Tabel images

No	Kolom	Tipe	Keterangan
5	domain	varchar(50)	Domain yang diberikan oleh sistem untuk *image* yang bersangkutan.
6	port	int	Port dari *image* yang dibuka untuk *host* saat dijalankan.
7	version	int	Penomoran urutan *image* yang masuk ke dalam sistem.
8	isRunning	int	Status apakah image sedang berjalan (1), proses untuk dijalankan atau dimatikan (2), dan juga mati (0).

Tabel `containers` digunakan untuk menyimpan data *containers* yang sedang berjalan pada *server master host*. Penyimpanan ini diperlukan agar mempercepat dan mempermudah saat mengolah data *container* karena tidak perlu memintanya secara langsung dari *server master*. Definisi tabel `containers` seperti pada Tabel 4.2.

Tabel 4.2: Tabel containers

No	Kolom	Tipe	Keterangan
1	id	int	Sebagai primary key pada tabel, nilai awal adalah `AUTO_INCREMENT`.
2	image_id	int	*Foreign key dari image* yang merujuk ke table images.

Tabel 4.2: Tabel containers

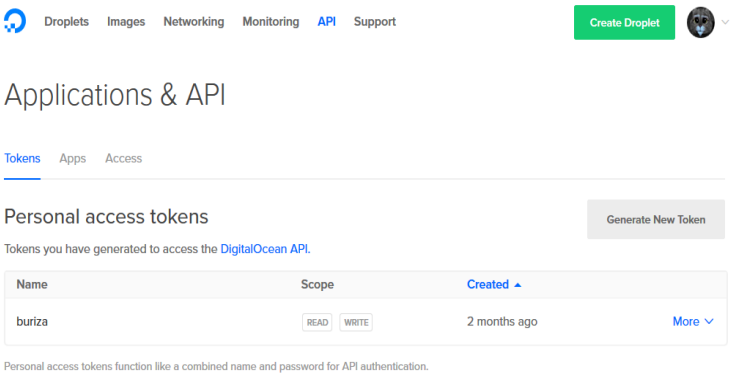
No	Kolom	Tipe	Keterangan
3	container_id	varchar(50)	ID dari containers yang sedang berjalan. ID didapatakan dari *server master* saat pembuatan *container* berhasil dilakukan.
4	port	int	Merupakan *port* dari *container* yang diberikan oleh *host*.
5	status	varchar(50)	Status apakah *container* dalam keadaan sedang berjalan atau mati.

Tabel `domains` digunakan untuk menyimpan ID record dari domain yang didaftarkan ke DNS DigitalOcean. ID record digunakan untuk menghapus domain yang terdaftar jika sudah tidak dibutuhkan lagi. Definisi tabel `domains` seperti pada Tabel 4.3

Tabel 4.3: Tabel domains

No	Kolom	Tipe	Keterangan
1	id	int	Sebagai primary key pada tabel, nilai awal adalah `AUTO_INCREMENT`.
2	domain	varchar(50)	Menyimpan subdomain.
3	record_id	varchar(50)	ID dari domain yang diberikan oleh DigitalOcean saat memasukkan domain baru.

4.4.3 Menambahkan dan Menghapus Domain



Gambar 4.1: DigitalOcean Control Panel

Domain yang digunakan dikelola oleh DigitalOcean. Untuk melakukan interaksi, seperti menambahkan record domain baru, pada pengembangan Tugas Akhir ini menggunakan API dalam bahasa pemrograman `curl` yang disediakan. Hal pertama yang harus dilakukan adalah mendapatkan token untuk dapat mengakses API. Token bisa didapatkan dari *control panel* Digital Ocean, seperti pada Gambar 4.1.

Setelah mendapatkan token, langkah selanjutnya adalah melakukan akses menggunakan `Curl` API. Namun, dalam implementasi pada Tugas Akhir ini, `curl` diganti dengan pustaka `requests` pada Python dan metode yang digunakan adalah `POST`.

Dalam melakukan permintaan ke API, hal pertama yang perlu disiapkah adalah `header`. Pada bagian ini, diperlukan dua *key*. Yang pertama yaitu `Content-Type` yang berisi nilai `application/json`. Lalu yang kedua adalah `Authorization` dengan nilai sesuai dengan token yang didapatkan sebelumnya.

Untuk datanya sendiri diperlukan tiga *key*, yaitu `type`, `name`,

dan `data`. Untuk key `type` digunakan untuk menentukan jenis *record* DNS dan nilai yang digunakan adalah `A`. Nilai `A` menunjukkan suatu domain atau subdomain, dalam kasus ini subdomain, ke suatu IP. Selanjutnya *key* `name` digunakan untuk menentukan nama subdomain yang didaftarkan. Yang terakhir adalah *key* `data`, menunjukkan IP yang adakn ditunjuk oleh domain yang didaftarkan. Untuk nilainya sendiri menunjuk IP *server master*, yaitu `128.199.160.188`. Setelah semua siap, permintaan ke API bisa dilakukan. melalui URL `https://api.digitalocean.com/v2/domains/<namaDomain>/records` Jika berhasil, maka akan mengembalikan status `201 Created` dan ID dari *record* yang didaftarkan.

Setelah suatu *record* terdaftar, ada saatnya untuk menghapusnya jika tidak ada aplikasi yang menggunakannya. Untuk melakukan proses tersebut, metode yang digunakan adalah `DELETE`. Untuk `header` sama seperti menambahkan *record*. Pada proses penghapusan *record* tidak ada data yang dikirim. Permintaan dilakukan melalu URL `https://api.digitalocean.com/v2/domains/<namaDomain>/records/<id_record>`. Jika berhasil akan mengembalikan status `204 No Content`.

4.4.4 Implementasi Task Queue Menggunakan Redis

Pada *server* ini, banyak proses yang terjadi terdiri dari banyak subprosess lainnya. Proses yang terjadi semuanya berasal dari Flask yang merupakan sebuah perangkat kerja Web yang mana permintaan terhadap layanan melalui protokol HTTP. Jika proses yang panjang dibiarkan saja berjalan tanpa ada kendali lebih lanjut, maka balasan yang akan diberikan kepada klien yang meminta layanan akan menunggu seluruh proses berakhir. Bisa saja saat menjalankan proses tersebut terdapat kesalahan atau hubungan antara klien dan *server* terputus yang

menyebabkan proses berhenti ditengah jalan. Untuk mengatasi permasalah tersebut, digunakan prinsip *task queue* menggunakan Redis. Jadi suatu proses akan dilemparkan ke dalam Redis, dan program akan 'melupakan' proses yang sudah diberikan kepada Redis sehingga melanjutkan menjalankan perintah yang diberikan, tidak peduli proses sebelumnya sudah dieksekusi atau belum. Setelah masuk ke dalam Redis, terdapat *worker* yang akan menjalankan proses tersebut. Dengan cara tersebut, proses yang panjang akan bekerja dibelakang dan balasan atau koneksi antara klien dan *server* tidak berlangsung lama, yang mengurangi kemungkinan terjadi kesalahan didalamnya.

Untuk melakukan hal tesebut, Redis harus terpasang, seperti yang dijelaskan pada A. Untuk terhubung dengan Redis, menggunakan pustaka RQ dan Redis untuk Pyhton seperti yang dijelaskan pada A. Redis yang sudah terpasang secara umum akan berjalan pada port `6379`. Selanjutnya adalah membuat koneksi ke Redis dan menyiapkan *queue* untuk menampung proses yang diberikan. Implementasinya seperti yang ditunjukkan pada Kode Sumber IV.6

```
con = redis.Redis('localhost', port=6379,
    db=0)
q = Queue(connection=con)
```

Kode Sumber IV.6: Koneksi Redis

4.4.5 Penyimpanan Konfigurasi Load Balancer pada etcd

File konfigurasi HAProxy pada *server load balancer* akan berubah secara otomatis karena confd yang berjalan akan membaca perubahan data pada etcd yang ada di *server controller*. etcd sendiri berjalan pada port `5050`. Pada *server controller* ini, disimpan data *image*, *container*, dan *domain* dari aplikasi yang sedang berjalan. Penyimpanan data dilakukan

dalam bentuk JSON. Untuk satu aplikasi, format yang digunakan seperti pada Kode Sumber IV.7.

```
{
  "image_name": image_name,
  "domain": domain,
  "containers": listContainers
}
```

Kode Sumber IV.7: Format Penyimpanan Data *Image* pada etcd

Lalu, sebuah data di dalam *key* `containers` memiliki format seperti Kode Sumber IV.8. Satu data menyatakan satu *container* yang sedang berjalan. Jika ada pada suatu waktu ada tiga *container* yang berjalan, maka data pada *key* `containers` ada tiga buah.

```
{
  "name": name,
  "ip": ip,
  "port": port
}
```

Kode Sumber IV.8: Format Penyimpanan Data *Container* pada etcd

Untuk menyimpan dan menghapus data menggunakan pustaka `python-etcd`. Sedangkan untuk memastikan apakah data sudah masuk, bisa menggunakan `curl`. Perintah yang digunakan untuk menampilkan data adalah `curl http://localhost:5050/v2/keys/images/`.

4.4.6 Implementasi Program Monitoring HAProxy

Pada *server controller* ini, *log* yang dihasilkan dari HAProxy akan diproses. Data *log* yang dihasilkan berupa data dalam format CSV. *Server load balancer* akan mengirimkan semua data *log* tersebut, tapi data yang diolah selanjutnya hanya

menggunakan tiga kolom saja, yaitu pada indeks ke 0, 1, dan 7. Berikut adalah penjelasan untuk masing-masing kolom yang digunakan:

1. 0 `pxname`
   Kolom `pxname` atau *proxy name* menunjukkan nama aplikasi yang sedang berjalan. Dengan kata lain, kolom ini menunjukkan nama aplikasi atau *image* yang ada di *server master*.
2. 1 `svname`
   Kolom `svname` atau *service name* menunjukkan *container* yang dituju atau digunakan oleh suatu aplikasi. Permintaan pengguna ke suatu aplikasi akan diarahkan ke *container* menggunakan data ini.
3. 7 `stot`
   Kolom `stot` menunjukkan penggunakan akses ke service tersebut. Nilainya merupakan kumulatif semua *request* yang diterima. Jika konfigurasi dari HAProxy diubah, maka nilainya akan kembali ke nol.

Hasil data yang didapatkan di atas akan dimodelkan menggunakan ARIMA untuk mendapatkan prediksi *request* ke depannya. Untuk model ARIMA sendiri dibangun berdasarkan *dataset log request* ke website World Cup 1998 selama 92 hari [4]. Data yang di dapatkan pertama kali diolah dari dari data biner ke dalam format csv. Setelah itu mengecilkan data dengan mengelompokkannya untuk jarak waktu 1 jam untuk proses pembuatan model. Data *request* kemudian dikecilkan dengan rasio mengambil 1000 *request* asli untuk dijadikan 1 *request* untuk model.

4.4.7 Implementasi Program Monitoring Server Master

Monitoring pada *server* Master Host bertujuan untuk mengetahui jumlah penggunaan sumber daya CPU dan *memory* dari *container* yang sedang berjalan. Data status *container* yang

akan diberikan oleh Master Host merupakan data mentah dalam format JSON. Pada subsistem ini data tersebut akan diolah untuk mengetahui secara pasti berapa penggunaan CPU dan *memory*. Hasil pengolahan data tersebut digunakan untuk perhitungan pada *Reactive Model*.

Data penggunaan CPU didapatkan dengan melakukan perhitungan penggunaan CPU dalam selang waktu tertentu. Variable yang dibutuhkan untuk melakukan perhitungan didapatkan dari *server* Master Host. Setelah mendapatkan data dalam format JSON, selanjutnya adalah menghitungnya menggunakan pseudocode yang tertera pada Kode Sumber IV.9.

```
function calCPUPercent:
    cpuPercent <- 0.0
    cpuDelta <- float64(v.CPUStats.CPUUsage
        .TotalUsage) - float64(previousCPU)
    systemDelta <- float64(v.CPUStats.
        SystemUsage) - float64(
        previousSystem)
    if systemDelta > 0.0 AND cpuDelta > 0.0
        cpuPercent = (cpuDelta /
            systemDelta) * float64(len(v.
            CPUStats.CPUUsage.PercpuUsage))
            * 100.0
    return cpuPercent
```

Kode Sumber IV.9: Pseudocode Menghitung Penggunaan CPU

Untuk mendapatkan penggunaan *memory* dari sebuah *container*, dapat menggunakan *key* dengan nama *memory_stats*. Nilai yang diberikan berupa penggunaan *memory* dalam satuan *byte*. Penggunaan satu buah *container* dibatasi sampai 512 MB. Lalu batas atas dari penggunaannya adalah 80 %. Jika ada *container* yang melebihi batas atas tersebut, maka akan ditambahkan

sebagai bagian dari *container* yang melebihi batas penggunaan.

Setelah mendapatkan jumlah container yang melebihi batas penggunaan CPU dan memory, selanjutnya adalah menentukan jumlah container yang harus dibentuk menggunakan Reactive Model. Perhitungannya menggunakan rumus seperti yang tertera pada Kode Sumber IV.11. Variable `totalExceed` menunjukkan jumlah *container* yang sudah melebihi batas penggunaan. Lalu variable `THRESHOLD_RESOURCE` merupakan batas atas dari penggunaan sumber daya yang bernilai 0.8.

```
NReactive = math.ceil(totalExceed * (1 –
    THRESHOLD_RESOURCE) / THRESHOLD_RESOURCE
    )
```

Kode Sumber IV.10: Perhitungan *Reactive Model*

4.4.8 Implementasi Pengendali Elastasitas

Dengan menggunakan data yang dihasilkan dari sistem yang melakukan monitoring terhadap *server Load Balancer* dan Master Host, maka pada sistem ini hasil perhitungan dari kedua *server* tersebut akan diputuskan. Pada perhitungan ini, di tentukan bahwa jumlah maksimal permintaan yang bisa diterima oleh satu *container* adalah 500 buah, yang dinyatakan dengan variable f.

Dari perhitungan fungsi di atas, bisa diketahui berapa jumlah *container* yang diperlukan aplikasi. Jika jumlah *container* yang dibutuhkan lebih banyak dari jumlah *container* yang sedang berjalan, maka *container* baru akan dibuat. Namun jika jumlah *container* yang sedang berjalan lebih banyak dari yang dibutuhkan, maka *container* akan dikurangi. Ada perbedaan yang terjadi saat menambahkan dan mengurangi *container*. Pada fungsi di atas, terdapat variable `DELAY_SCALE_IN`, yaitu pengurangan *container* hanya akan terjadi jika pengecekan sudah

diakukan sebanyak nilai tersebut. Jadi, pada kasus ini, pengurangan *container* akan terjadi jika sudah terjadi pengecekan sebanyak 10 kali yang masing-masingnya konsisten bahwa jumlah *container* yang berjalan lebih banyak dari yang diperlukan. Namun, jika diperlukan penambahan *container*, maka akan dilakukan saat itu juga tanpa ada *delay*.

```
function totalScaling():
    DELAY_SCALE_IN = 10
    f = 500
    if f != 0:
        nProactive = math.ceil(
            predictedRequest / f)
    else:
        nProactive = 1

    if nProactive == 0:
        nProactive = 1

    if nReactive > 0:
        return 0, nReactive + max(nInstance
            , nProactive)
    elif nProactive >= nInstance:
        return 0, nProactive
    elif lastTimes >= DELAY_SCALE_IN:
        return 0, nProactive
    else:
        result = lastTimes + 1
        return result, -1
```

Kode Sumber IV.11: Perhitungan *Reactive Model*

4.4.9 Implementasi *Endpoint* Dasbor

Pada *server controller* ini terdapat endpoint yang digunakan oleh dasbor untuk menampilkan data. Endpoint tersebut dapat diakses melalui `http://controller.nota-no.life/api/`. Berikut adalah penjelasan rute yang disediakan:

1. `/get_images`
   Rute ini memiliki metode `GET`, digunakan untuk mengambil semua daftar aplikasi atau *image* yang ada pada *docker registry*.
2. `/get_image_info/<image_id>`
   Rute ini memiliki metode `GET`, digunakan untuk mengambil informasi lebih lanjut dari suatu aplikasi.
3. `/get_containers/<image_id>`
   Rute ini memiliki metode `GET`, digunakan untuk mengambil daftar semua *container* yang sedang berjalan untuk suatu aplikasi.
4. `/start_image/<image_id>`
   Rute ini memiliki metode `GET`, digunakan untuk menjalankan aplikasi atau *image* yang ada pada *docker registry*.
5. `/stop_image/<image_id>`
   Rute ini memiliki metode `GET`, digunakan untuk menghentikan aplikasi atau *image* yang yang sedang berjalan. Dengan menggunakan rute ini, semua *container* yang sedang berjalan akan dimatikan dan domain yang terdaftar akan dihapus.
6. `/update_images`
   Rute ini memiliki metode `POST`, digunakan untuk memperbarui data port dari aplikasi atau image saat dijalankan. Pengaturan ini harus benar agar aplikasi dapat berjalan dengan keadaan normal.
7. `/get_metrics/<image_id>`
   Rute ini memiliki metode `GET`, digunakan untuk

mendapatkan data tentang jumlah *request* dan jumlah *container* yang digunakan dari sebuah aplikasi yang sedang berjalan.

4.5 Implementasi Load Balancer

Load balancer adalah sebuah *server* yang digunakan untuk *end user* sebagai pintu masuk untuk melakukan akses terhadap aplikasi. *Server* ini menentukan *container* mana yang akan diakses oleh pengguna berdasarkan domain yang digunakan. *Server load balancer* memiliki IP `128.199.160.188` dan menggunakan HAProxy sebagai perangkat lunak *load balancer*-nya. Proses pemasangan HAProxy dapat dilihat pada A. Selain itu, untuk konfigurasi dari HAProxy menggunakan `confd`. Dengan menggunakan `confd`, konfigurasi dari HAProxy dapat menjadi dinamis dan menyesuaikan dengan kebutuhan. Proses pemasangan confd dapat dilihat pada A.

4.5.1 Pengaturan Teknik *Balancing*

Pada berkas *template* konfigurasi HAProxy A.4, diatur teknik *balancing* yang digunakan pada bagian *backend* adalah *Round-Robin*. *Round-Robin* sendiri adalah teknik dimana mendistribusikan permintaan pengunjung menuju suatu *backend* dengan cara membaginya secara bergantian sesuai dengan kemampuan masing-masing *backend*. Teknik tersebut digunakan karena merupakan salah satu teknik yang paling mudah untuk diimplementasikan dan cocok diterapkan untuk mengalirkan permintaan dari pengguna ke *container*. Dikarenakan aplikasi dijalankan pada *container* yang memiliki sumber daya yang sama, maka proses pendistribusian menggunakan *Round-Robin* juga efektif. HAProxy akan mendistribusikan permintaannya dari satu *continer* ke *container* lain secara bergantian yang mana hal tersebut akan mempermudah untuk mengelolanya karena

dapat mengetahui *container* mana yang akan digunakan oleh HAProxy saat ini dan juga kedepannya.

4.5.2 Pengaturan Domain

Domain yang digunakan selama proses pengembangan dikelola oleh DigitalOcean. Penulis mengatur agar domain yang diakses oleh pengguna agar diarahkan menuju *server load balancer*. Permintaan yang diteruskan dari DNS DigitalOcean akan diproses oleh HAProxy dan akan meneruskan permintaan menuju aplikasi sesuai dengan domain yang digunakan. Untuk menambahkan dan menghapus domain di DNS DigitalOcean, dapat dilakukan dengan mengakses antarmuka yang disediakan dan juga bisa menggunakan API. Untuk proses tersebut, dilakukan oleh *server controller* menggunakan API berbasis bahasa pemrograman *Curl*.

4.5.3 Pengaturan *Endpoint Log*

Pada berkas *template* konfigurasi HAProxy A.4 terdapat pengaturan *log* yang akan dihasilkan oleh HAProxy, yaitu pada baris `sock mode 660 level admin`. Artinya, *log* yang dihasilkan dapat diakses melalui soket. Untuk mengakses *log* HAProxy yang sekarang, dapat menggunakan perintah `echo 'show stat' | nc -U /run/haproxy/admin.sock`.
Perintah tersebut akan menghasilkan sebuah data dalam format CSV yang merepresentasikan keadaan HAProxy saat itu.

Log yang dihasilkan oleh HAProxy akan digunakan oleh `server controller`. Agar `server controller` bisa mendapatkannya, dibuatkan sebuah *endpoint* menggunakan Flask yang mana akan memberikan hasil mentahan dari *log* HAProxy. *Endpoint* tersebut diakses melalui port 5000 dengan rute yang disediakan yaitu `/get_stats` dengan metode `GET`.

4.5.4 Pangaturan *Hot-Upgrade*

Pengaturan *hot-upgrade* merupakan teknik agar saat terjadi pergantian konfigurasi dari HAProxy minim atau bahkan tidak terjadi *packet drop* atau layanan menjadi *down*. Untuk melakukan hal tersebut, pada pembuatan sistem kali ini dengan cara melakukan manipulasi terhadap paket yang diterima oleh *server*. Saat pertama kali akan terhubung dengan *server*, saat melakukan koneksi dengan menggunakan TCP, pertama kali klien akan mengirimkan SYN. SYN sendiri adalah *flag* yang digunakan pada saat pertama kali akan membuat koneksi dengan komputer yang dituju atau *server* yang dituju.

Salah satu cara kerja dari *flag* SYN adalah klien akan mengirimkan ulang jika gagal mendapatkan balasan *flag* ACK. Jadi, sebelum melakukan konfigurasi ulang terhadap HAProxy, terlebih dahalu mengatur agar semua paket bertipe SYN yang masuk pada port 80 akan di *drop* dengan cara menjalankan perintah seperti pada Kode Sumber IV.12. Port 80 sendiri adalah port yang digunakan oleh HAProxy untuk menerima permintaan dari luar.

```
iptables -I INPUT -p tcp --dport 80 --syn -
    j DROP
```

Kode Sumber IV.12: Menambahkan Rule Input pada `iptables`

Pengaturan *drop* paket tersebut menggunakan `iptables`. Setelah proses tersebut berhasil, selanjutnya adalah melakukan konfigurasi ulang HAProxy, yaitu dengan melakukan *restart* layanan. Setelah HAProxy berjalan kembali dengan pengaturan yang baru, maka pengaturan *drop* paket SYN dihapus dengan menggunakan perintah seperti yang tertera pada Kode Sumber IV.13 dan *request* yang sebelumnya tidak mendapatkan balasan akan mendapatkannya setelah proses ini.

```
iptables –D INPUT –p tcp ––dport 80 ––syn –
    j DROP
```

Kode Sumber IV.13: Menghapus Rule Input pada `iptables`

Proses tersebut memungkinkan tidak terjadi drop saat ada pengguna yang melakukan akses terhadap aplikasi. Efek yang terjadi adalah terjadinya *delay* pada *request* selama terjadi proses konfigurasi ulang HAProxy.

4.6 Implementasi Dasbor

Dasbor diimplementasikan menggunankan perangkat kerja React bagian *frontend* dan Flask untuk *backend*nya. Dasbor digunakan untuk mempermudah pengembang dalam mengelola aplikasi. Dasbor memiliki menu-menu sebagai berikut:

- Daftar Aplikasi
- Infromasi Aplikasi
- Daftar *Container*
- Matrik Aplikasi

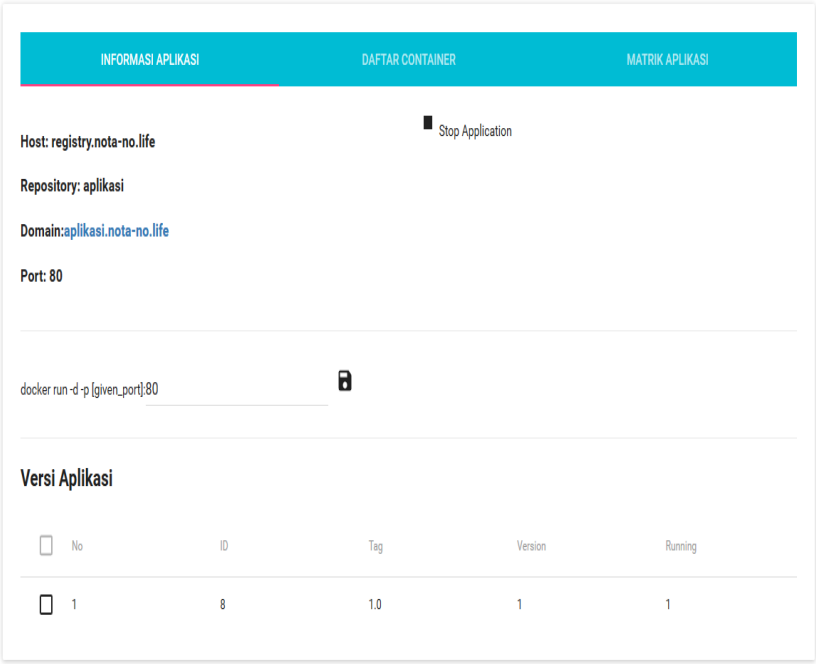
Masing-masing menu berikutnya akan dijelaskan secara rinci.

4.6.1 Daftar Aplikasi

Daftar aplikasi, juga merupakan beranda dari dasbor, adalah menu yang digunakan untuk melihat daftar aplikasi atau *image* yang ada pada *server docker registry*. Pada halaman ini, bisa dilihat nama beserta versi terakhir dari aplikasi. Lalu juga terdapat status apakah aplikasi sedang berjalan atau tidak. Antar muka kelola daftar aplikasi ditunjukkan pada Gambar 4.2.

Gambar 4.2: Dasbor Daftar Aplikasi

4.6.2 Informasi Aplikasi

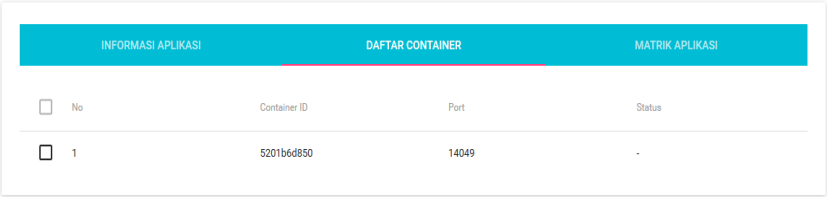


Gambar 4.3: Dasbor Informasi Aplikasi

Halaman ini menunjukkan infromasi lengkap dari sebuah aplikasi. Pada halaman ini bisa dilihat nama aplikasi, port yang digunakan, domain dari aplikasi, dan versi dari aplikasi. Pada halaman ini, pengguna bisa mengatur port dari aplikasi agar dapat berjalan dengan baik. Dari halaman ini juga aplikasi pertama kali akan dijalankan. Jadi pada halaman ini terdapat kontrol untuk menajalankan dan mematikan aplikasi. Antar muka informasi ditunjukkan pada Gambar 4.3.

4.6.3 Daftar *Container*

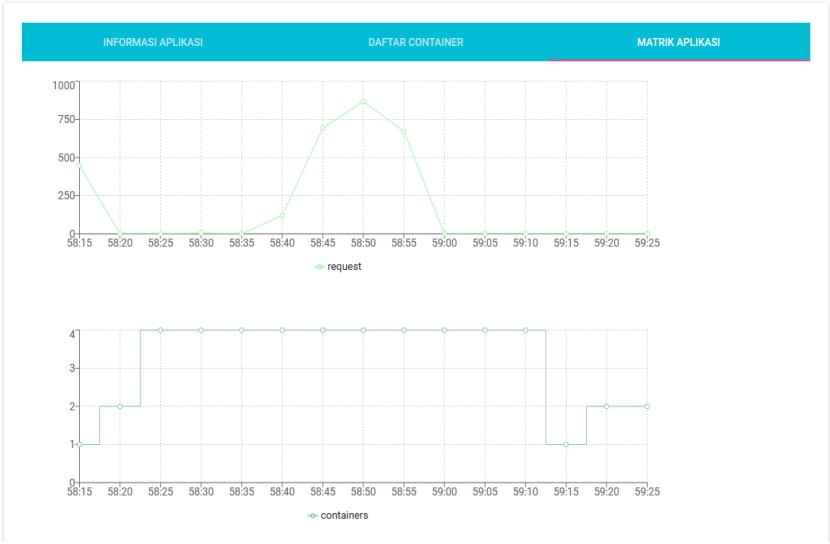
Pada halaman ini, pengguna dapat melihat daftar *container* yang sedang berjalan untuk sebuah aplikasi. Informasi yang diberikan berupa ID dan port dari container. Antar muka halaman daftar *container* ditunjukkan pada Gambar 4.4.



Gambar 4.4: Dasbor Daftar *Container*

4.6.4 Metrik Aplikasi

Halaman metrik aplikasi digunakan untuk memantau keadaan sekarang dari aplikasi. Metrik yang diberikan adalah jumlah *request* pengguna ke aplikasi dan jumlah *container* dari aplikasi. Data akan diperbarui sekitar lima detik sekali. Antar muka metrik aplikasi ditunjukkan pada Gambar 4.5.



Gambar 4.5: Dasbor Matrik Aplikasi

(Halaman ini sengaja dikosongkan)

BAB V

PENGUJIAN DAN EVALUASI

5.1 Lingkungan Uji Coba

Lingkungan pengujian menggunakan komponen-komponen yang terdiri dari: satu *server load balancer*, satu *server master host*, satu *server controller*, satu *server docker registry*, dan enam komputer penguji. Semua *server* menggunakan Virtual Private Server dari DigitalOcean. Lalu, untuk komputer penguji menggunakan lima buah desktop dan satu buah VPS sebagai *docker* klien yang digunakan untuk membuat *docker image*. Pengujian dilakukan di Laboratoriom Pemrograman Jurusan Teknik Informatika ITS.

Spesifikasi untuk setiap komponen yang digunakan ditunjukkan pada Tabel 5.1.

Tabel 5.1: Spesifikasi Komponen

No	Komponen	Perangkat Keras	Perangkat Lunak
1	Load balancer	2 core processor, 4GB RAM, 20GB SSD	Ubuntu 14.04.5 LTS, HAProxy, Python 2.7
2	Master host	8 core processor, 16GB RAM, 20GB SSD	Ubuntu 14.04.5 LTS, Docker 17.03.0-ce, Python 2.7
3	Controller	2 core processor, 4GB RAM, 20GB SSD	Ubuntu 14.04.5 LTS, Redis, MySQL, Python 2.7
4	Docker registry	1 core processor, 512MB RAM, 20GB SSD	Ubuntu 14.04.5 LTS, Docker 17.03.0-ce, Python 2.7
5	Komputer penguji	Processor Core2Duo E7300, 2GB RAM	Windows 8, JMeter 3.2

Tabel 5.1: Spesifikasi Komponen

No	Komponen	Perangkat Keras	Perangkat Lunak
6	Docker klien	1 core processor, 1GB RAM, 20GB SSD	Ubuntu 16.04 LTS, Docker 17.03.0-ce

Untuk akses ke masing-masing komponen, digunakan IP publik yang disediakan untuk masing-masing komponen tersebut. Selain menggunakan IP, ada sebagian *server* yang bisa diakses melalui domain. Detailnya ditunjukkan pada Tabel 5.2.

Tabel 5.2: IP dan Domain Server

No	Server	IP dan Domain
1	Load balancer	128.199.160.188
2	Master host	128.199.182.29
3	Controller	128.199.250.137 http://controller.nota-no.life
4	Docker registry	139.59.97.244 https://registry.nota-no.life

5.2 Skenario Uji Coba

Uji coba akan dilakukan untuk mengetahui keberhasilan sistem yang telah dibangun. Skenario pengujian dibedakan menjadi 2 bagian, yaitu:

- **Uji Fungsionalitas**
  Pengujian ini didasrkan pada fungsionalitas yang disajikan sistem.
- **Uji Performa**
  Pengujian ini untuk menguji ketahanan sistem terhadap sejumlah permintaan ke aplikasi secara bersamaan. Pengujian dilakukan dengan melakukan *benchmark* pada

sistem.

5.2.1 Skenario Uji Coba Fungsionalitas

Uji fungsionalitas dibagi menjadi 2, yaitu uji mengelola aplikasi berbasis *docker* dan uji fungsionalitas menu aplikasi Dasbor.

5.2.1.1 Uji Mengelola Aplikasi Berbasis Docker

Pengujian ini dilakukan untuk mengetahui apakah sistem sudah bisa menyimpan dan mengelola data *docker image* dari aplikasi yang dimasukkan oleh pengembang. Pengujian menggunakan VPS yang berperan sebagai *docker* klien. Pengujian dilakukan dengan memasukkan data *docker image* ke *server docker registry* yang sudah disediakan. Dengan menggunakan sebuah komputer lain yang digunakan untuk membuat aplikasi web berbasis *docker*, dari sana aplikasi tersebut akan ditaruh ke *server docker registry*.

Alamat dari Docker *registry* yang digunakan adalah `https://registry.nota-no.life`. Setelah berhasil melakukan login pada *server docker registry*, selanjutnya adalah memasukkan *image* baru tersebut. *Image* yang dibuat adalah sebuah aplikasi web berbasis PHP 7 dengan *web server* Apache. Aplikasi web tersebut menyediakan sebuah halaman yang berisi sebuah teks yang dibuat melalui pemanggilan fungsi PHP.

Pertama kali *image* tersebut dimasukkan ke *server docker registry*, kemudian data dari *image* tersebut akan disimpan di *server controller*. Setelah data tersimpan, selanjutnya adalah menjalankan aplikasi melalui dasbor yang disediakan. Sebelum menjalankan aplikasi, terlebih dahulu mengatur *port* dari aplikasi yang akan berjalan. Setelah mengatur *port* dengan benar, selanjutnya adalah menjalankan aplikasinya. Jika aplikasi berhasil dijalankan, maka aplikasi dapat diakses melalui domain

yang disediakan.

Setelah aplikasi berjalan, selanjutnya adalah memperbarui aplikasi dengan mengganti teks yang ditampilkan. Untuk itu, pada komputer yang digunakan untuk membuat *image* sebelumnya, maka dibuatkan image baru dari aplikasi dengan versi terbaru. Image versi terbaru ini kemudian di *push* ke *docker registry*. Setelah selesai melakukan *push*, harapannya aplikasi yang sebelumnya sudah berjalan, akan diperbarui secara otomatis oleh sistem.

Terakhir, pengujian yang dilakukan adalah menghentikan aplikasi yang sudah berjalan. Fungsi untuk menghentikan aplikasi yang sedang berjalan ini terdapat pada dasbor. Jika proses ini berhasil, maka domain yang sebelumnya digunakan untuk mengakses aplikasi akan hilang dan pengguna tidak bisa lagi melakukan akses terhadap aplikasi.

Daftar uji fungsionalitas menambahkan dan memperbarui aplikasi dijelaskan pada Tabel 5.3.

Tabel 5.3: Skenario Uji Mengelola Aplikasi Berbasis Docker

No	Uji Coba	Hasil Harapan
1	Pengguna melakukan *login* ke *server docker registry*	Pengguna berhasil melakukan *login* dengan menggunakan *username* dan *password* yang sudah ditentukan.
2	Pengguna menambahkan *image* baru dari sebuah aplikasi ke *server docker registry*.	Pengguna berhasil menambahkan *image* baru dan data tersimpan pada *server controller*.

Tabel 5.3: Skenario Uji Mengelola Aplikasi Berbasis Docker

No	Uji Coba	Hasil Harapan
3	Pengguna bisa mengatur *port* dari aplikasi menggunakan dasbor yang disediakan	Data *port* dari aplikasi yang tersimpan bisa diganti sesuai dengan kebutuhan pengguna.
4	Pengguna bisa menjalankan aplikasi melalui fitur yang ada pada dasbor	Aplikasi berhasil berjalan dan pengguna mendapatkan domain yang digunakan untuk mengakses aplikasi.
5	Pengguna memperbarui aplikasi yang sedang berjalan dengan melakukan *push* ke *server docker registry*.	Aplikasi yang sedang berjalan akan diperbarui secara otomatis tanpa perlu perintah dari pengguna.
6	Penggua menghentikan aplikasi yang sedang berjalan.	Aplikasi berhasil dihentikan dan pengguna tidak bisa lagi melakukan akses aplikasi.

5.2.1.2 Uji Fungsionalitas Menu Aplikasi Dasbor

Aplikasi Dasbor digunakan untuk mengelola dan memantau aplikasi. Aplikasi Dasbor terdiri dari 4 bagian utama, yaitu halaman beranda, informasi aplikasi, informasi *container*, dan metrik dari aplikasi. Rancangan pengujian dan hasil yang diharapkan ditunjukkan dengan Tabel 5.4.

Tabel 5.4: Skenario Uji Fungsionalitas Aplikasi Dasbor

No	Menu	Uji Coba	Hasil Harapan
1	Kelola aplikasi	Menambahkan aplikasi baru atau memperbarui aplikasi	Dasbor dapat menampilkan daftar aplikasi terbaru yang dimasukkan atau diperbarui oleh pengembang.
		Menjalankan aplikasi yang sudah masuk ke dalam sistem	Aplikasi dapat berjalan dan pengguna mendapatkan domain untuk mengakses aplikasi.
		Menghentikan aplikasi yang sedang berjalan	Aplikasi yang sedang berjalan dapat dihentikan dan pengguna tidak bisa lagi melakukan akses terhadap aplikasi.
		Mengganti *port* aplikasi agar dapat berjalan dengan baik	Pengguna dapat mengganti *port* aplikasi agar aplikasi dapat berjalan dengan benar.

Tabel 5.4: Skenario Uji Fungsionalitas Aplikasi Dasbor

No	Menu	Uji Coba	Hasil Harapan
2	Lihat informasi aplikasi	Memilih salah satu aplikasi yang ada	Pengguna dapat melihat infromasi secara lengkap tentang aplikasi.
3	Lihat informasi *container*	Memilih salah satu aplikasi yang ada	Pengguna dapat melihat infromasi secara lengkap tentang *container* yang sedang berjalan untuk aplikasi tersebut.
4	Lihat metrik aplikasi	Memilih salah satu aplikasi yang ada	Pengguna dapat melihat grafik penggunaan CPU dan *memory* dari aplikasi .

5.2.2 Skenario Uji Coba Performa

Uji performa dilakukan dengan menggunakan lima buah desktop untuk melakukan akses secara bersamaan ke aplikasi menggunakan aplikasi JMeter. Desktop akan mencoba mengaskses halaman dari aplikasi web yang sudah berjalan, dengan domain aplikasi.nota-no.life. Halaman yang akan diakses berisi sebuah teks yang dihasilkan dari pemanggilan fungsi PHP.

Percobaan dilakukan dengan lima skenario jumlah *concurrent user* yang berbeda, yaitu sebanyak 800, 1600, 2400, 3200, dan 4000 pengguna dalam rentang waktu inisialisasi $\pm$ 15 detik. Waktu tersebut menunjukkan masing-masing pengguna akan mengirimkan request selama $\pm$ 15 detik, namun tidak termasuk waktu menunggu balasan dari *server*, yang artinya

keseluruhan permintaan tersebut akan lebih dari waktu tersebut dan bergantung pada kemampuan *server* untuk memberikan respon. Pengujian *request* ini bertujuan untuk mengukur kemampuan dari *proactive model*. Untuk masing-masingnya, dicoba sebanyak empat perhitungan *proactive model* yang berbeda menggunakan ARIMA yang berbeda, yaitu ARIMA(1,1,0), ARIMA(2,1,0), ARIMA(3,1,0), ARIMA(4,1,0). *Proactive model* sendiri berguna untuk mengetahui jumlah *request* kedepannya agar sistem bisa menyediakan sumber daya berdasarkan predeksi tersebut.

Selain itu, untuk memperkirakan sumber daya yang dibutuhkan sistem kedepannya, digunakan *reactive model*. *Model* tersebut akan menghitung jumlah *container* yang sumber daya CPU dan *memory*-nya sudah melebihi batas yang ditentukan. Sistem akan membentuk *container* baru berdasarkan perhitungan *reactive model* tersebut jika ada *container* yang penggunaannya sudah melebihi batas atas dan mengurangi *container* jika ada *container* yang tidak digunakan. Percobaan akan dilakukan sebanyak enam kali dan berikutnya akan dijelaskan data apa yang diuji untuk masing-masingnya.

5.2.2.1 Uji Performa Kecepatan Menangani *Request*

Pengujian dilakukan dengan mengukur jumlah waktu yang diperlukan untuk menyelesaikan *request* yang dilakukan oleh komputer penguji. Waktu yang diukur adalah perbedaan jarak antara *request* pertama dan yang terakhir dilakukan oleh klien yang mendapatkan balasan dari *server*.

5.2.2.2 Uji Performa Penggunaan CPU

Pengujian dilakukan dengan menghitung penggunaan CPU yang terjadi pada *server master host*. Penggunaan CPU di sini

adalah penggunaan dari *container* aplikasi yang sedang berjalan. Perhitungan dilakukan dengan mengambil nilai rata-rata penggunaan CPU dari masing-masing *container* selama proses pengujian dilakukan. Nilai yang didaptkan berupa total persen penggunaan CPU oleh *container* dibandingkan dengan keseluruhan kemampuan CPU.

5.2.2.3 Uji Performa Penggunaan *Memory*

Pengujian dilakukan dengan menghitung penggunaan *memory* yang terjadi pada *server master host*. Penggunaan *memory* di sini adalah penggunaan dari *container* aplikasi yang sedang berjalan. Perhitungan dilakukan dengan mengambil nilai rata-rata pengguanaan *memory* dari masing-masing aplikasi selama proses pengujian dilakukan.

5.2.2.4 Uji Performa Keberhasilan *Request*

Pengujian dilakukan dengan menghitung jumlah *request* yang gagal dilakukan selama skenario dijalankan. Dari semua jumlah *request* yang dikirimkan selama pengujian, akan didapatkan persen *request* yang gagal dilakukan.

5.3 Hasil Uji Coba dan Evaluasi

Berikut dijelaskan hasil uji coba dan evaluasi berdasarkan skenario yang telah dijelaskan pada subbab 5.2.

5.3.1 Uji Fungsionalitas

Berikut dijelaskan hasil pengujian fungsionalitas pada sistem yang dibangun.

5.3.1.1 Uji Mengelola Aplikasi Berbasis Docker

Pengujian dilakukan sesuai dengan skenario yang dijelaskan pada subbab 5.2.1.1 dan pada Tabel 5.3. Hasil pengujian seperti tertera pada Tabel 5.5.

Tabel 5.5: Hasil Uji Coba Mengelola Aplikasi Berbasis Docker

No	Uji Coba	Hasil
1	Pengguna melakukan *login* ke *server docker registry*	OK.
2	Pengguna menambahkan *image* baru dari sebuah aplikasi ke *server docker registry*.	OK.
3	Pengguna bisa mengatur *port* dari aplikasi menggunakan dasbor yang disediakan	OK.
4	Pengguna bisa menjalankan aplikasi melalui fitur yang ada pada dasbor.	OK.
5	Pengguna memperbarui aplikasi yang sedang berjalan dengan melakukan *push* ke *server docker registry*.	OK.
6	Penggua menghentikan aplikasi yang sedang berjalan.	OK.

Sesuai dengan skenario uji coba yang diberikan pada Tabel 5.3, hasil uji coba menunjukkan semua skenario berhasil ditangani.

5.3.1.2 Uji Fungsionalitas Menu Aplikasi Dasbor

Sesuai dengan skenario pengujian yang dilakukan pada aplikasi dasbor. Pengujian dilakukan dengan menguji setiap

menu pada aplikasi dasbor. Hasil uji coba dapat dilihat pada Table 5.6. Semua skenario yang direncanakan berhasil ditangani.

Tabel 5.6: Hasil Uji Fungsionalitas Aplikasi Dasbor

No	Menu	Uji Coba	Hasil
1	Kelola aplikasi	Menambahkan aplikasi baru atau memperbarui aplikasi	Dasbor berhasil menampilkan daftar aplikasi terbaru yang dimasukkan atau diperbarui oleh pengembang.
		Menjalankan aplikasi yang sudah masuk ke dalam sistem	Aplikasi berhasil berjalan dan pengguna mendapatkan domain untuk mengakses aplikasi.
		Menghentikan aplikasi yang sedang berjalan	Aplikasi yang sedang berjalan berhasil dihentikan dan pengguna tidak bisa lagi melakukan akses terhadap aplikasi.

Tabel 5.6: Hasil Uji Fungsionalitas Aplikasi Dasbor

No	Menu	Uji Coba	Hasil
		Mengganti *port* aplikasi agar dapat berjalan dengan baik	Pengguna berhasil mengganti *port* aplikasi agar aplikasi dapat berjalan dengan benar.
2	Lihat informasi aplikasi	Memilih salah satu aplikasi yang ada	Pengguna berhasil melihat infromasi secara lengkap tentang aplikasi.
3	Lihat informasi *container*	Memilih salah satu aplikasi yang ada	Pengguna berhasil melihat infromasi secara lengkap tentang *container* yang sedang berjalan untuk aplikasi tersebut.
4	Lihat metrik aplikasi	Memilih salah satu aplikasi yang ada	Pengguna berhasil melihat grafik penggunaan CPU dan *memory* dari aplikasi .

5.3.2 Hasil Uji Performa

Seperti yang sudah dijelaskan pada subbab 5.2 pengujian performa dilakukan dengan melakukan akses ke aplikasi dengan sejumlah pengguna secara bersama-sama. Pengujian dilaukan dengan memberikan *request* secara berkelanjutan dengan jumlah pengguna terdiri dari lima bagian, yaitu 800, 1600, 2400, 3200, dan 4000 pengguna. Untuk jumlah *request* yang dihasilkan dari masing-masing pengguna selama rentang waktu request ± 15 detik dapat dilihat pada Tabel 5.7. Jumlah tersebut akan diolah oleh *reactive model*. Lalu jumlah penggunaan CPU dan memory selama menangani *request* tersebut akan digunakan oleh *proactive model* untuk menambahkan atau mengurangi *container* yang ada.

Tabel 5.7: Jumlah *Request* ke Aplikasi

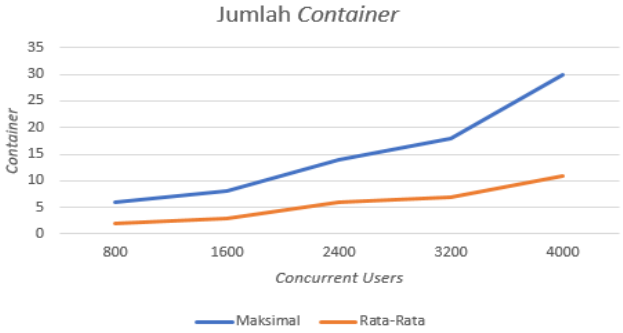
Concurrent Users	**Jumlah *Request***
800	± 16.925
1.600	± 26.650
2.400	± 34.943
3.200	± 50.092
4.000	± 57.750

Pada Tabel 5.8 dapat dilihat jumlah *container* yang terbentuk selama proses *request* dari *user* yang dilakukan selama enam kali. Nilai yang ditampilkan berupa nilai rata-rata selama percobaan dibulatkan ke atas. Sistem dapat menyediakan *container* sesuai dengan jumlah *request* yang diberikan, semakin banyak *request* yang dilakukan, maka *container* yang disediakan akan semakin banyak. Nilai *container* tersebut didapatkan dari perhitungan *proactive model*. Selain melihat jumlah *request*, penentuan *container* yang dibentuk juga dari jumlah sumber daya yang digunakan *container* berdasarkan perhitungan

menggunakan *reactive model*. Pada Gambar 5.1 dapat dilihat grafik dari jumlah *container* yang terbentuk berdasarkan jumlah *request* yang dilakukan.

Tabel 5.8: Jumlah *Container*

Concurrent Users	**Maksimal *Container***	**Rata-rata *Container***
800	6	2
1.600	8	3
2.400	14	6
3.200	18	7
4.000	30	11



Gambar 5.1: Grafik Jumlah *Container*

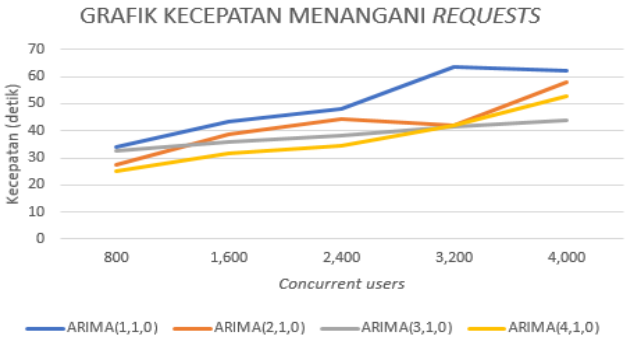
5.3.2.1 Kecepatan Menangani *Request*

Dari hasil uji coba kecepatan menangani *request*, dapat dilihat pada Table 5.9 dalam satuan detik bahwa semakin banyak *concurrent users*, semakin lama pula waktu yang diperlukan untuk menyeselaikannya. Request paling cepat ditangani dengan menggunakan prediksi ARIMA(4,1,0) dan paling lambat

menggunakan ARIMA(1,1,0). Hal tersebut terjadi karena kurang bagusnya hasil prediksi yang dihasilkan oleh ARIMA(1,1,0) yang mana kadang hasil prediksinya terlalu rendah atau terlalu tinggi. Dari hasil percobaan tersebut, dapat dilihat bahwa hampir semua *request* dapat ditangani di bawah satu menit. Lalu grafik hasil uji coba perhitungan kecepatan menangani *request* ditunjukkan pada Gambar 5.2.

Tabel 5.9: Kecepatan Menangani *Request*

	800	**1600**	**2400**	**3200**	**4000**
ARIMA(1,1,0)	34.167	43.286	48.143	63.857	62.286
ARIMA(2,1,0)	27.429	38.571	44.143	42.143	57.857
ARIMA(3,1,0)	32.429	36.000	38.429	41.571	43.857
ARIMA(4,1,0)	24.857	31.571	34.429	42.143	52.714



Gambar 5.2: Grafik Kecepatan Menangani *Request*

5.3.2.2 Penggunaan CPU

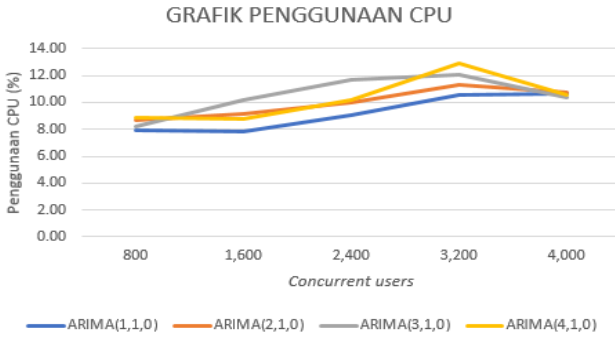
Dari hasil uji coba penggunaan CPU pada *server master host*, penggunaan CPU berada di bawah 15%. Penggunaan CPU yang diukur adalah penggunaan CPU yang dilakukan oleh *container*

dari aplikasi, tidak termasuk sistem. Jumlah *core* yang dimiliki oleh *processor* di *server master host* adalah 8 buah, yang artinya kurang lebih hanya satu core yang digunakan untuk menangani semua *request*. Hasil pengukuran penggunaan CPU dapat dilihat pada Tabel 5.10

Tabel 5.10: Penggunaan CPU

	800	**1600**	**2400**	**3200**	**4000**
ARIMA(1,1,0)	7.1%	7.8%	9.1%	10.5%	10.7%
ARIMA(2,1,0)	8.5%	9.2%	10.1%	11.3%	10.7%
ARIMA(3,1,0)	8.8%	10.2%	11.6%	12.1%	10.3%
ARIMA(4,1,0)	8.0%	8.3%	10.1%	12.9%	10.5%

Dari hasil uji coba, penggunaan prediksi yang berbeda tidak terlalu berpengaruh terhadap penggunaan CPU. Lalu, penggunaan CPU tergolong rendah, yaitu hanya sebesar ±10% untuk menangani semua *request* yang diberikan. Hasil uji coba performa penggunaan CPU ditunjukkan oleh dalam grafik pada Gambar 5.3.



Gambar 5.3: Grafik Penggunaan CPU

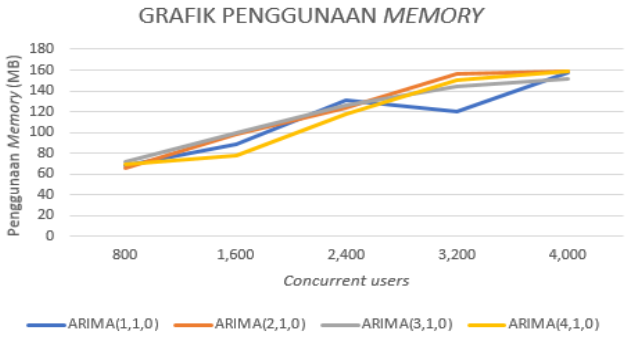
5.3.2.3 Penggunaan *Memory*

Dari hasil uji coba penggunaan *memory*, semakin banyak *request* yang diterima, semakin banyak *memory* yang diperlukan. Perhitungan penggunaan *memory* adalah rata-rata penggunaan dari masing-masing *container* sebuah aplikasi. Untuk masing-masing *container*, dibatasi penggunaan maksimal *memory* adalah 512 MB. Dari hasil uji coba ini, dapat dilihat pada Tabel 5.11 bahwa penggunaan terbesar hanya sebesar 158.71 MB. Artinya jumlah tersebut hanya menggunakan sepertiga dari keseluruhan *memory* yang bisa digunakan.

Tabel 5.11: Penggunaan *Memory*

	800	**1600**	**2400**	**3200**	**4000**
ARIMA(1,1,0)	67.91	88.97	130.79	120.14	157.73
ARIMA(2,1,0)	65.89	97.98	123.47	156.64	158.33
ARIMA(3,1,0)	72.20	99.72	125.56	144.42	152.14
ARIMA(4,1,0)	69.60	77.34	117.39	149.76	158.71

Hasil uji coba performa penggunaan *memory* dalam grafik ditunjukkan pada Gambar 5.4.



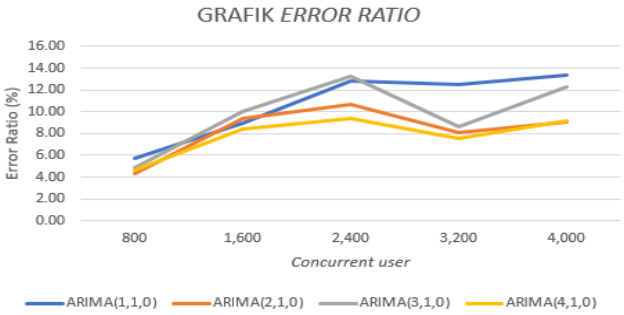
Gambar 5.4: Grafik Penggunaan Memory

5.3.2.4 Keberhasilan *Request*

Pada uji coba ini, dilakukan perhitungan seberapa besar jumlah *request* yang gagal dilakukan. Untuk jumlah *concurrent user* pada tingkat 800 dan 1600, dapat dilihat pada Table 5.12 *error* yang terjadi hampir sama. Prediksi menggunakan ARIMA(4,1,0) berhasil unggul karena menggunakan parameter yang lebih banyak. Namun hal tersebut tidak berlaku untuk ARIMA(3,1,0) karena walaupun parameternya lebih banyak dari ARIMA(2,1,0), tapi hasil prediksinya bisa meleset saat terjadi kondisi dimana koefisien negatif atau koefisien ke dua dikalikan dengan sebuah parameter bukan nol, dan koefisien lain dikalikan dengan parameter nol, maka hasil prediksinya akan negatif, yang mana seharusnya tidak mungkin ada *request* negatif.

Tabel 5.12: *Error Ratio Request*

	800	**1600**	**2400**	**3200**	**4000**
ARIMA(1,1,0)	5.72%	8.96%	12.85%	12.54%	13.38%
ARIMA(2,1,0)	4.31%	9.35%	10.68%	8.11%	9.04%
ARIMA(3,1,0)	4.84%	10.02%	13.22%	8.63%	12.24%
ARIMA(4,1,0)	4.62%	8.41%	9.39%	7.52%	9.21%



Gambar 5.5: Grafik Error Ratio

Dari uji coba itu, 90% lebih *request* berhasil ditangani. Hasil uji coba jumlah *request* yang gagal ditunjukkan dengan grafik pada Gambar 5.5.

(Halaman ini sengaja dikosongkan)

BAB VI

PENUTUP

Bab ini membahas kesimpulan yang dapat diambil dari tujuan pembuatan sistem dan hubungannya dengan hasil uji coba dan evaluasi yang telah dilakukan. Selain itu, terdapat beberapa saran yang bisa dijadikan acuan untuk melakukan pengembangan dan penelitian lebih lanjut.

6.1 Kesimpulan

Dari proses perencangan, implementasi dan pengujian terhadap sistem, dapat diambil beberapa kesimpulan berikut:

1. Sistem dapat menjalankan dan menyajikan satu atau lebih aplikasi web berbasis *docker* kepada *end-user* melalui domain yang disediakan.
2. Sistem dapat menyesuaikan sumber daya secara otomatis berdasarkan jumlah *request* dengan menggunakan *proactive model* dan penggunaan sumber daya, yaitu CPU dan *memory*, pada *container* dengan menggunakan *reactive model*.
3. Penggunaan *load balancer* cocok digunakan dengan aplikasi yang berjalan di atas *docker container*. Hal tersebut karena semua *request* ke aplikasi akan melalui *load balancer*. Jika terjadi penambahan dan pengurangan sumber daya, penyesuaian dengan cepat dilakukan dan hanya perlu merubah sedikit konfigurasi pada *load balancer* dan pengguna tidak perlu tahu apa yang terjadi di dalam sistem.
4. Prediksi jumlah *request* menggunakan ARIMA sudah bisa menangani skenario uji coba. Perbedaan *order* ARIMA yang digunakan mempengaruhi akurasi dalam menentukan *request* yang akan terjadi ke depannya. Dalam pengujian ini, ARIMA(4,1,0) memiliki hasil pengujian paling bagus dengan jumlah rata-rata *error request* yang paling rendah,

yaitu sebesar 7.83%. Lalu untuk kecepatan menerima *request*, ARIMA(2,1,0) dan ARIMA(4,1,0) memiliki konsistensi yang berbanding lurus dengan jumlah *request*.

5. Penggunaan sumber daya CPU dan *memory* tidak dipengaruhi oleh penggunaan ARIMA yang berbeda. Penggunaan sumber daya tersebut bergantung kepada jumlah *request*, semakin banyak *request* yang diberikan, penggunaan CPU dan *memory* akan semakin tinggi. Penggunaan CPU paling tinggi yaitu sebesar 12.9% dan penggunaan *memory* paling tinggi sebesar 158.71 MB. Dengan penggunaan tersebut, masih tersisa lebih dari setengah sumber daya yang bisa digunakan.
6. Sebuah *container* dari sebuah aplikasi dapat dibentuk dalam waktu $\pm$ 1 detik sehingga penambahan sumber daya bisa dilakukan dengan cepat dan proses untuk memperbarui konfigurasi dari HAProxy memerlukan waktu $\pm$ 5 detik. Selama proses tersebut, akses pengguna akan tertunda, namun tidak menunjukkan terjadinya *down*.

6.2 Saran

Berikut beberapa saran yang diberikan untuk pengembangan lebih lanjut:

1. Mengamankan komunikasi antar *server* karena saat ini *endpoint server* bisa diakses oleh siapapun. Hal tersebut bisa dilakukan dengan mengimplentasikan *private* IP dan menggunakan token untuk komunikasinya.
2. Pemodelan menggunakan ARIMA cukup baik, namun perlu dicoba untuk melakukan pembuatan model dengan *dataset* yang lebih baru. Selain itu, bisa mencoba alternatif pemodelan *time series* yang lain, seperti ARCH (Autoregressive Conditional Heteroskedasticity).

DAFTAR PUSTAKA

[1] C. Kan, "DoCloud: An elastic cloud platform for Web applications based on Docker," in *2016 18th International Conference on Advanced Communication Technology (ICACT)*, Jan. 2016, hal. 478–483.

[2] "What is Docker?" 2016, 12 Desember 2016. [Daring]. Tersedia pada: https://www.docker.com/what-docker. [Diakses: 12 Desember 2016].

[3] C. Boettiger, "An introduction to Docker for reproducible research, with examples from the R environment," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, hal. 71–79, Jan. 2015, arXiv: 1410.0846.

[4] "1998 World Cup Web Site Access Logs," 10 April 2017. [Daring]. Tersedia pada: http://ita.ee.lbl.gov/html/contrib/WorldCup.html. [Diakses: 10 April 2017].

(Halaman ini sengaja dikosongkan)

LAMPIRAN A

INSTALASI PERANGKAT LUNAK

Instalasi Lingkungan Docker

Proses pemasangan Docker dpat dilakukan sesuai tahap berikut:

- Menambahkan repository Docker
  Langkah ini dilakukan untuk menambahkan *repository* Docker ke dalam paket `apt` agar dapat di unduh oleh Ubuntu. Untuk melakukannya, jalankan perintah berikut:

```
sudo apt-get -y install \
   apt-transport-https \
   ca-certificates \
   curl

curl -fsSL https://download.docker.com/linux/
    ubuntu/gpg | sudo apt-key add -

sudo add-apt-repository \
    "deb [arch=amd64] https://download.docker.com/
     linux/ubuntu \
    $ (lsb_release -cs) \
    stable"

sudo apt-get update
```

- Mengunduh Docker
  Docker dikembangkan dalam dua versi, yaitu CE (*Community Edition*) dan EE (*Enterprise Edition*). Dalam pengembangan sistem ini, digunakan Docker CE karena merupakan versi Docker yang gratis. Untuk mengunduh Docker CE, jalankan perintah `sudo apt-get -y install docker-ce`.

- Mencoba menjalankan Docker
  Untuk melakukan tes apakah Docker sudah terpasang dengan benar, gunakan perintah `sudo docker run hello-world`.

Instalasi Docker Registry

Docker Registry dikembangkan menggunakan Docker Compose. Dengan menggunakan Docker Compose, proses pemasangan Docker Registry menjadi lebih mudah dan fleksibel untuk dikembangkan ditempat lain. Docker Registry akan dijalankan pada satu *container* dan Nginx juga akan dijalankan di satu *container* lain yang berfungsi sebagai perantara komunikasi antara Docker Registry dengna dunia luar. Berikut adalah proses pengembangan Docker Registry yang penulis lakukan:

- Pemasangan Docker Compose
  $ `sudo apt-get -y install python-pip`
  $ `sudo pip install docker-compose`
- Pemasangan paket `apache2-utils`
  Pada paket `apache2-utils` terdapat fungsi `htpasswd` yang digunakan untuk membuat *hash password* untuk Nginx. Proses pemasangan paket dapat dilakukan dengan menjalankan perintah `sudo apt-get -y install apache2-utils`.
- Pemasangan dan pengaturan Docker Registry
  Buat folder `docker-registry` dan `data` dengan menjalankan perintah berikut:
  $ `mkdir /docker-registry && cd $_`
  $ `mkdir data`
  Folder `data` digunakan untuk menyimpan data yang dihasilkan dan digunakan oleh *container* Docker Registry. Kemudian di dalam folder `docker-registry` buat sebuah berkas dengan nama `docker-compose.yml` yang akan

digunakan oleh Docker Compose untuk membangun aplikasi. Tambahkan isi berkasnya sesuai dengan Kode Sumber A.1.

```
nginx:
image: "nginx:1.9"
ports:
  - 443:443
  - 80:80
links:
  - registry:registry
volumes:
  - ./nginx/:/etc/nginx/conf.d
registry:
  image: registry:2
  ports:
    - 127.0.0.1:5000:5000
  environment:
    REGISTRY_STORAGE_FILESYSTEM
        _ROOTDIRECTORY: /data
  volumes:
    - ./data:/data
    - ./registry/config.yml:/etc/docker
        /registry/config.yml
```

Kode Sumber A.1: Isi Berkas docker-compose.yml

- Pemasangan *container* Nginx Buat folder `nginx` di dalam folder `docker-registry`. Di dalam folder `nginx` buat berkas dengan nama `registry.conf` yang berfungsi sebagai berkas konfigurasi yang akan digunakan oleh Nginx. Isi berkas sesuai denga Kode Sumber A.2.

```
upstream docker-registry{
  server registry:5000;
}
```

```
server{
  listen 80;
  server_name registry.nota-no.life;
  return 301 https://
     $server_name$request_uri;
}
server{
  listen 443;
  server_name registry.nota-no.life;
  ssl on;
  ssl_certificate /etc/nginx/conf.d/
     cert.pem;
  ssl_certificate_key /etc/nginx/conf.d
     /privkey.pem;
  client_max_body_size 0;
  chunked_transfer_encoding on;
  location /v2/{
    if ($http_user_agent ~ "^(docker
       \/1\.(3|4|5(?!\.[0-9]-dev))|Go )
       .*$" ){
      return 404;
    }
    auth_basic "registry.localhost";
    auth_basic_user_file /etc/nginx/
       conf.d/registry.password;
    add_header 'Docker-Distribution-Api
       -Version' 'registry/2.0' always;
    proxy_pass http://docker-registry;
    proxy_set_header Host $http_host;
    proxy_set_header X-Real-IP
       $remote_addr;
    proxy_set_header X-Forwarded-For
       $proxy_add_x_forwarded_for;
```

```
        proxy_set_header X-Forwarded-Proto
            $scheme;
        proxy_read_timeout 900;
    }
}
```

Kode Sumber A.2: Isi Berkas registry.conf

Instalasi Pustaka Python

Dalam pengembangan sistem ini, digunakan berbagai pustaka pendukung. Pustaka pendukung yang digunakan merupakan pustaka untuk bahasa pemrograman Python. Berikut adalah daftar pustaka yang digunakan dan cara pemasangannya:

- Python Dev
  $ `sudo apt-get install python-dev`
- Flask
  $ `sudo pip install Flask`
- docker-py
  $ `sudo pip install docker`
- MySQLd
  $ `sudo apt-get install python-mysqldb`
- Redis
  $ `sudo pip install redis`
- RQ
  $ `sudo pip install rq`

Instalasi HAProxy

HAProxy dapat dipasang dengna mudah menggunakan `apt-get` karena perangkat lunak tersebut sudah tersedia pada *repository* Ubuntu. Untuk melakukan pemasangan HAProxy, gunakan perintah `apt-get install haproxy`.

Setelah HAProxy diunduh, perangkat lunak tersebut belum berjalan karena belum diaktifkan. Untuk mengaktifkan *service haproxy*, buka berkas di `/etc/default/harpoxy` kemudian ganti nilai `ENABLED` yang awalnya bernilai `0` menjadi `ENABLED=1`. Setelah itu service haproxy dapat dijalankan dengan menggunakan perintah `service harpoxy start`. Untuk konfigurasi dari HAProxy nantinya akan diurus oleh *confd*. *confd* akan menyesuaikan konfigurasi dari HAProxy sesuai dengan kebutuhan aplikasi yang tersedia.

Instalasi etcd dan confd

etcd dapat di unggah dengan menjalankan perintah berikut, `curl https://github.com/coreos/etcd/releases/download/v3.2.0-rc.0/etcd-v3.2.0-rc.0-linux-amd64.tar.gz`. Setelah proses unduh berhasil dilakukan, selanjutnya yang dilakukan adalah melakukan ekstrak berkasnya menggunakan perintah `tar -xvzf etcd-v3.2.0-rc.0-linux-amd64.tar.gz`. Berkas binary dari etcd bisa ditemukan pada folder `./bin/etcd`. Berkas inilah yang digunakan untuk menjalankan perangkat lunak etcd. Untuk menjalankannya, dapat dilakukan dengan menggunakan perintah `etcd --listen-client-urls http://0.0.0.0:5050 --advertise-client-urls http://128.199.250.137:5050`. Perintah tersebut memungkinkan etcd diakses oleh *host* lain dengan IP 128.199.250.137, yang merupakan host dari *load balancer* dan confd. Setelah proses tersebut, etcd sudah siap untuk digunakan.

Setelah etcd siap digunakan, selanjutnya adalah memasang confd. Untuk menginstall confd gunakan rangkaian perintah berikut:

```
$ mkdir -p $GOPATH/src/github.com/kelseyhightower
$    git clone https://github.com/kelseyhightower/
```

```
confd.git $GOPATH/src/github.com/kelseyhightower/
confd
$ cd $GOPATH/src/github.com/kelseyhightower/confd
$ ./build
```

Setelah berhasil memasang confd, selanjutnya buka berkas `/etc/confd/confd.toml` dan isi berkas sesuai dengan Kode Sumber A.3. Pengaturan tersebut bertujuan agar confd melakukan *listen* terhadap server etcd dan melakukan tindakan jika terjadi perubahan pada etcd.

```
confdir = "/etc/confd"
interval = 20
backend = "etcd"
nodes = [
        "http://128.199.250.137:5050"
]
prefix = "/"
scheme = "http"
verbose = true
```

Kode Sumber A.3: Isi Berkas confd.toml

Setelah melakukan konfigurasi confd, selanjutnya adalah membuat *template* konfigurasi untuk HAProxy. Buka berkas di `/etc/confd/templates/haproxy.cfg.tmpl`. Jika berkas tidak ada maka buat berkasnya dan isi berkas sesuai dengan Kode Sumber A.4.

```
global
        log /dev/log    local0
        log /dev/log    local1 notice
        chroot /var/lib/haproxy
        stats socket /run/haproxy/admin.
           sock mode 660 level admin
        stats timeout 30s
```

```
        daemon
defaults
        log     global
        mode    http
        option  httplog
        option  dontlognull
        timeout connect 5000
        timeout client  50000
        timeout server  50000
        errorfile 400 /etc/haproxy/errors
           /400.http
        errorfile 403 /etc/haproxy/errors
           /403.http
        errorfile 408 /etc/haproxy/errors
           /408.http
        errorfile 500 /etc/haproxy/errors
           /500.http
        errorfile 502 /etc/haproxy/errors
           /502.http
        errorfile 503 /etc/haproxy/errors
           /503.http
        errorfile 504 /etc/haproxy/errors
           /504.http
frontend http-in
        bind *:80

        # Define hosts
        {{range gets "/images/*"}}
        {{$data := json .Value}}
                acl host_{{$data.image_name
                   }} hdr(host) -i {{$data.
                   domain}}.nota-no.life
        {{end}}
```

```
        ## Figure out which one to use
        {{range gets "/images/*"}}
        {{$data := json .Value}}
                use_backend {{$data.
                   image_name}}_cluster if
                   host_{{$data.image_name
                   }}
        {{end}}
{{range gets "/images/*"}}
{{$data := json .Value}}
backend {{$data.image_name}}_cluster
        mode http
        balance roundrobin
        option forwardfor
        cookie JSESSIONID prefix
        {{range $data.containers}}
        server {{.name}} {{.ip}}:{{.port}}
           check
        {{end}}
{{end}}
```

Kode Sumber A.4: Isi Berkas haproxy.cfg.tmpl

Langkah terakhir adalah membuat berkas konfigurasi untuk HAProxy di `/etc/confd/conf.d/haproxy.toml`. Jika berkas tidak ada, maka buat berkasnya dan isi berkas sesuai dengan Kode Sumber A.5.

```
[template]
src = "haproxy.cfg.tmpl"
dest = "/etc/haproxy/haproxy.cfg"
keys = [
        "/images"
]
```

```
reload_cmd = "iptables –I INPUT –p tcp ––
    dport 80 ––syn –j DROP && sleep 1 &&
    service haproxy restart && iptables –D
    INPUT –p tcp ––dport 80 ––syn –j DROP"
```

Kode Sumber A.5: Isi Berkas haproxy.toml

Setelah melakukan konfigurasi, selanjutnya adalah menjalankan confd dengan menggunakan perintah `confd &`.

Pemasangan Redis

Redis dapat dipasang dengan mempersiapkan kebutuhan pustaka pendukungnya. Pustaka yang digunakan adalah `build-essential` dan `tcl8.5`. Untuk melakukan pemasangannya, jalankan perintah berikut:

```
$ sudo apt-get install build-essential
$ sudo apt-get install tcl8.5
```

Setelah itu unduh aplikasi Redis dengan menjalankan perintah `wget http://download.redis.io/releases/redis-stable.tar.gz`. Setelah selesai diunduh, buka file dengan perintah berikut:

```
$ tar xzf redis-stable.tar.gz && cd redis-stable
```

Di dalam folder `redis-stable`, bangun Redis dari kode sumber dengan menjalankan perintah `make`. Setelah itu lakukan tes kode sumber dengan menjalankan `make test`. Setelah selesai, pasang Redis dengan menggunakan perinah `sudo make install`. Setelah selesai melakukan pemasangan, Redis dapat diaktifkan dengan menjalankan berkas bash dengan nama `install_server.sh`.

Untuk menambah pengaman pada Redis, diatur agar Redis hanya bisa dari *localhost*. Untuk melakukannya, buka file `/etc/redis/6379.conf`, kemudian cari baris `bind`

`127.0.0.1`. Hapus komen jika sebelumnya baris tersebut dalam keadaan tidak aktif. Jika tidak ditemukan baris dengan isi tersebut, tambahkan pada akhir berkas baris tersebut.

Pemasangan kerangka kerja React

Pada pengembangan sistem ini, penggunaan pustaka React dibangun di atas konfigurasi Create React App. Untuk memasang Create React App, gunakan perintah `npm install -g create-react-app`. Setelah terpasang, untuk membangun aplikasinya jalankan perintah `create-react-app fe-controller`. Setelah proses tersebut, dasar dari aplikasi sudah terbangun dan siap untuk dikembangkan lebih lanjut.

(Halaman ini sengaja dikosongkan)

LAMPIRAN B

KODE SUMBER

Let's Encrypt Cross Signed

```
-----BEGIN CERTIFICATE-----
MIIEkjCCA3qgAwIBAgIQCgFBQgAAAVOF
    c2oLheynCDANBgkqhkiG9w0BAQsFADA/
MSQwIgYDVQQKExtEaWdpdGFsIFNpZ25h
    dHVyZSBUcnVzdCBDby4xFzAVBgNVBAMT
DkRTVCBSb290IENBIFgzMB4XDTE2MDMx
    NzE2NDA0NloXDTIxMDMxNzE2NDA0Nlow
SjELMAkGA1UEBhMCVVMxFjAUBgNVBAoT
    DUxldCdzIEVuY3J5cHQxIzAhBgNVBAMT
GkxldCdzIEVuY3J5cHQgQXV0aG9yaXR5
    IFgzMIIBIjANBgkqhkiG9w0BAQEFAAOC
AQ8AMIIBCgKCAQEAnNMM8FrlLke3cl03
    g7NoYzDq1zUmGSXhvb418XCSL7e4S0EF
q6meNQhY7LEqxGiHC6PjdeTm86dicbp5 gWAf15Gan/
    PQeGdxyGkOlZHP/uaZ6WA8
SMx+yk13EiSdRxta67nsHjcAHJyse6cF 6
    s5K671B5TaYucv9bTyWaN8jKkKQDIZ0
Z8h/pZq4UmEUEz9l6YKHy9v6Dlb2honz hT+Xhq+
    w3Brvaw2VFn3EK6BlspkENnWA
a6xK8xuQSXgvopZPKiAlKQTGdMDQMc2P
    MTiVFrqoM7hD8bEfwzB/onkxEz0tNvjj
/PIzark5McWvxI0NHWQWM6r6hCm21AvA 2
    H3DkwIDAQABo4IBfTCCAXkwEgYDVR0T
AQH/BAgwBgEB/wIBADAOBgNVHQ8BAf8E
    BAMCAYYwfwYIKwYBBQUHAQEEczBxMDIG
CCsGAQUFBzABhiZodHRwOi8vaXNyZy50
    cnVzdGlkLm9jc3AuaWRlbnRydXN0LmNv
bTA7BggrBgEFBQcwAoYvaHR0cDovL2Fw
    cHMuaWRlbnRydXN0LmNvbS9yb290cy9k
```

```
c3Ryb290Y2F4My5wN2MwHwYDVR0jBBgw
    FoAUxKexpHsscfrb4UuQdf/EFWCFiRAw
VAYDVR0gBE0wSzAIBgZngQwBAgEwPwYL
    KwYBBAGC3xMBAQEwMDAuBggrBgEFBQcC
ARYiaHR0cDovL2Nwcy5yb290LXgxLmxl
    dHNlbmNyeXB0Lm9yZzA8BgNVHR8ENTAz
MDGgL6AthitodHRwOi8vY3JsLmlkZW50
    cnVzdC5jb20vRFNUUk9PVENBWDNDUkwu
Y3JsMB0GA1UdDgQWBBSoSmpjBH3duubR
    ObemRWXv86jsoTANBgkqhkiG9w0BAQsF
AAOCAQEA3TPXEfNjWDjdGBX7CVW+dla5
    cEilaUcne8IkCJLxWh9KEik3JHRRHGJo
uM2VcGfl96S8TihRzZvoroed6ti6WqEB
    mtzw3Wodatg+VyOeph4EYpr/1wXKtx8/
wApIvJSwtmVi4MFU5aMqrSDE6ea73Mj2
    tcMyo5jMd6jmeWUHK8so/joWUoHOUgwu
X4Po1QYz+3dszkDqMp4fklxBwXRsW10K  XzPMTZ+
    sOPAveyxindmjkW8lGy+QsRlG
PfZ+G6Z6h7mjem0Y+iWlkYcV4PIWL1iw
    Bi8saCbGS5jN2p8M+X+Q7UNKEkROb3N6
KOqkqm57TH2H3eDJAkSnh6/DNFu0Qg==
−−−−−END  CERTIFICATE−−−−−
```

Kode Sumber B.1: Let's Encrypt X3 Cross Signed.pem

BIODATA PENULIS

Muhammad Fahrul Razi, akbrab dipanggil Razi lahir pada tanggal 23 November 1994 di Ilung, Kalimantan Selatan. Penulis merupakan seorang mahasiswa yang sedang menempuh studi di Jurusan Teknik Informatika Institut Teknologi Sepuluh Nopember. Memiliki hobi antara lain membaca novel dan futsal. Selama menempuh pendidikan di kampus, penulis juga aktif dalam organisasi kemahasiswaan, antara lain Staff Departemen Pengembangan Sumber Daya Mahasiswa Himpunan Mahasiswa Teknik Computer-Informatika pada tahun ke-2. Pernah menjadi staff National Programming Contest Schematics tahun 2014 dan 2015. Selain itu penulis pernah menjadi asisten dosen di mata kuliah Pemrograman Jaringan, serta asisten praktikum pada mata kuliah Dasar Pemrograman dan Struktur Data.