DM874 – Report
Group 2

January 2020

Hello

Contents

1 Introduction

The purpose of this project has been to develop a chat application that allows its users to automate in- and outbound messages, through user uploaded scripts written in the service oriented programming language, Jolie, while maintaining security.

The underlying infrastructure of the application itself has been built as a micro-service architecture with the goal of affording developers greater atonomy and agility in the limited scope that they are responsible for.

The choice of a micro-service architecture also helps another key goal, namely that of good horizontal scalability; we have utilized the venerable container orchestration framework, Kubernetes to facilitate this. Every service is containerized using Docker, and managed as pods in Kubernetes. Hello

2 Preliminaries

In this section we will provide an overview of the concepts, tools and platforms that we have used and what they are.

2.1 Containerization

Containerization is the concept of packaging the environment your application needs with the application itself into a small self-contained package. In our project we have used Docker for this.

2.2 REST

REST is an API spec, that is built ontop of HTTP. We use REST for interacting with our service from the outside, since it is almost an ISO standard for how to expose your web-services.

2.3 Kubernetes

Kubernetes is a self-healing, fast, fault-tolerant, and pluggable spec, for container orchestration and networking abstraction at scale. [7]

2.3.1 Self-healing

In the context of kubernetes, self-healing means that the kubernets master actively makes sure that containers are healthy, if not they are restarted (usually).

2.3.2 Fast

Kubernetes services are usually configured at one of the networking layers, making it extremely fast, and runtime-configurations is implemented at systemcall-level. The whole system is also written in Go, which is known to have very good performance.

2.3.3 Pluggable

Kubernetes itself is more of an api that usually has pre built-in modules. An example is the kube-dns, which can easily be plugged with another DNS. The kubernets ingress controllers can also be plugged for something like nginx, if chosen to.

2.3.4 Container orchestration

Usually people know this by automatic scheduling and resource management for containers. Imagine having 3 virtual machines and 50 containers, kubernetes automatically handles all the deploying and networking for you.

2.4 Prometheus

Prometheus is a applications monitoring spec, like JVM memory details, Go garbage collector details and much much more. It is also a server software, that can pull from REST endpoints (usually /prometheus), and then exposes another api with all the collected data that can be consumed by eg Grafana.[1]

2.5 Grafana

Grafana is a visualization tool, it can be connected to various sources (such as prometheus and loki), for data visualization.[8]

2.6 Loki

Loki is a brand new software from the grafana team, that harvests logs stores them in cassandra and tokenizes them for extremely fast lookup. Loki exposes an endpoint that adheres to the prometheus spec, so it is also extremely pluggable.[9]

2.7 Redis

Redis is an extremely high-performance in-memory simple key-value store but extensible through an plugin environment. Redis is used often for query caching and session management.[10]

2.8 Postgres

Postgres is a state of the art high performance RDBMS.[3]

2.9 Kafka

Kafka is a fully distributed, pub-sub like messaging system that employs log techniques instead of the ordinary ack-pop of pub-sub systems. Kafka is the most widely used modern messaging system of it's category of pub-sub systems.[4]

2.10 JSON

JSON is a data modelling language, it is simple and easy to read.

2.11 JWT

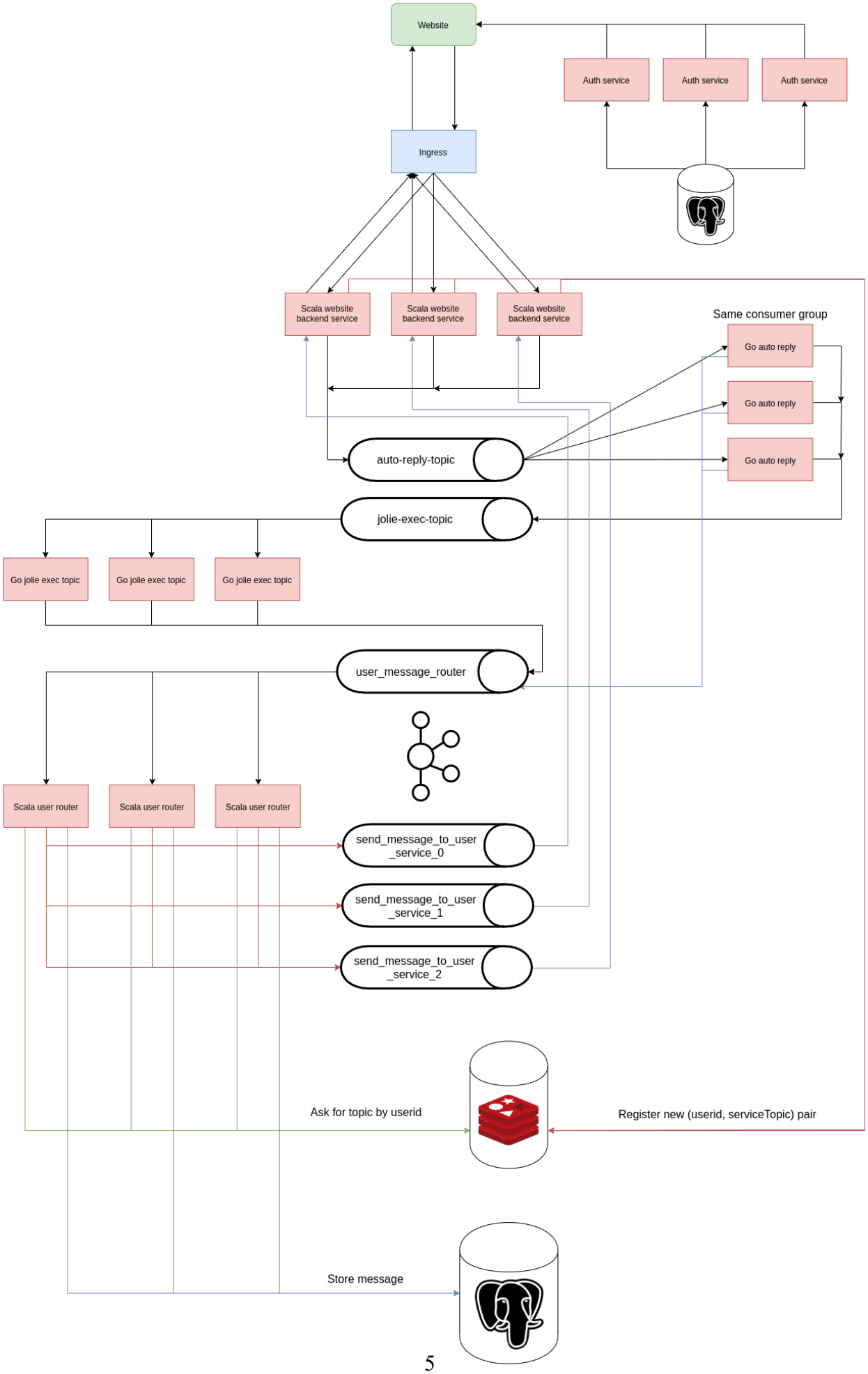
JWT stands for Json-Web-Token, which is a method of dispensing authentication tokens to users.

2.12 GCS

GCS stands for google cloud storage, its a thirdparty object storage service from google cloud.

3 Technical Description

In this section the technicalities of the project is explained. First the spec will explained and then every aspect into more detail. A flowchart of the stack can be seen below:


Website
Auth service
Auth service
Auth service
Ingress
Scala website backend service
Scala website backend service
Scala website backend service
Same consumer group
Go auto reply
Go auto reply
Go auto reply
auto-reply-topic
jolie-exec-topic
Go jolie exec topic
Go jolie exec topic
Go jolie exec topic
user_message_router
Scala user router
Scala user router
Scala user router
send_message_to_user_service_0
send_message_to_user_service_1
send_message_to_user_service_2
Ask for topic by userid
Register new (userid, serviceTopic) pair
Store message

In the spec we use an event sourcing like technique.[2] This includes a stateful data-structure that every service knows the structure of. This structure contains any data about a message that a service would need to know, as well as the sender user.

```
{
    "messageUid": "c0a630d2-8db3-4a03-9e19-7141582f37aa",
    "sessionUid": "cf2bd7ca-ba13-40d9-8fb7-bab2064028d4",
    "messageBody": "Hello, world!",
    "senderId": 42,
    "recipientIds": [12, 8],
    "fromAutoReply": false,
    "eventDestinations": ["TOPIC1", "TOPIC2"]
}
```

1. `messageUid` a UUID that represents the message id.
2. `sessionUid` a UUID to represent the user session (websocket session).
3. `messageBody` this is the full message.
4. `senderId` this is the user id of the sender.
5. `recipientIds` these id's represent the recipient user id's.
6. `fromAutoReply` this represents if the message was generated through pragmatic methods.
7. `eventDestinations` this represents the topics (services) that the message must go through, usually a message ends in the router.

3.1 Microservices

This section covers the responsibilities and design choices of the different microservices that comprise the chat application.

3.1.1 Webserver (website backend)

This service has 5 simple but important objectives.

1. To serve the static single page React application.
2. To handle user websockets through the kubernetes ingress & insert itself as the manager of a user websocket.
3. To check authentication when a user makes a request.

4. To ”bootstrap” a message, by inserting into one of these event sourcing models (see the JSON structure in above).

5. To route messages back to the recipient user websockets.

The service is written in scala and is independently scalable, has no state, storage or any stateful requirements. It is fault tolerant and secure through JWT using HMAC256 symmetric private key encryption.

3.1.2 Auto reply

The auto reply service is an internal (to the cluster) service responsible for providing users with a simpler alternative to writing Jolie scripts to automate replying to messages. The feature works like how auto reply does in most e-mail clients; letting users write a static message that will be sent to everyone trying to message them. The message can be toggled on or off in the settings of the chat application.

The service is written in the Go programming language, which lends itself well to microservice development for a few reasons:

- It is modern enough to have many built-in facilities useful for micro service development, like easy JSON encoding and decoding.
- It is a compiled language which means the service will require fewer hardware resources for the same performance.
- It is a strongly typed language which helps catch many bugs during compilation.
- Since it is developed by Google who themselves promote using a microservice architecture[6], there exists many tools and resources for containerizing go applications and deploying them with Kubernetes which also happens to be by Google.

State related to the auto reply service is kept in a dedicated PostgreSQL database, seperate from the service itself; this makes the service easier to scale horizontally.[11]

3.1.3 Jolie exec

3.1.4 Router

The router service is responsible for routing messages to the correct services, eg the one managing the websocket for the recipient user. The router service simply makes a lookup in redis for the key of the self-registered managed websocket, and sends the message to the found web-servers.

3.2 Deployability

We have chosen to use kubernetes for this project to manage almost all of our deployability. One of the key points of kubernetes is that if your deployment configurations are written well, it handles all deployment oriented aspects. Kubernetes is an extension to containerization, and since containerization envisions the idea of easy deployments kubernetes really pushes it to the next level.

All runtime aspects and configuration has deployability and scalability in mind, using many built-in parts of kubernetes. We have the flexibility to say here is a service and it's configuration (which is dynamically injected at runtime if changed), now run it.

3.3 Reusability

Reusability is employed on two different levels. One is the level that we re-use Kafka for every aspect of cross-service communication, meaning that kafka is the only pre-requisite to understand the communication model.

On the other side, we have services with clear API definitions. These services can be chained in any order chosen, thus they are completely reusable if the message flow needs it.

3.4 Performance

Performance was a big part of our considerations, both in designing the system and the physical performance. One of the most important parts of performance is the ability to do scaling seamlessly. An instance of this is how we introduced a router service and a redis instance to remove the need to check all websocket managers if they managed a connection to whatever user was receiving a message.

All services are completely stateless and scalable except the websocket manager (webserver) instance, since it needs to do some rebalancing if a downscale must happen. All services are also written in high performance asynchronous languages, which include Go and Scala.

The idea of introducing kafka has also been for performance aspects, since kafka has an obscene throughput (at least 100k req/sec without message bulking).

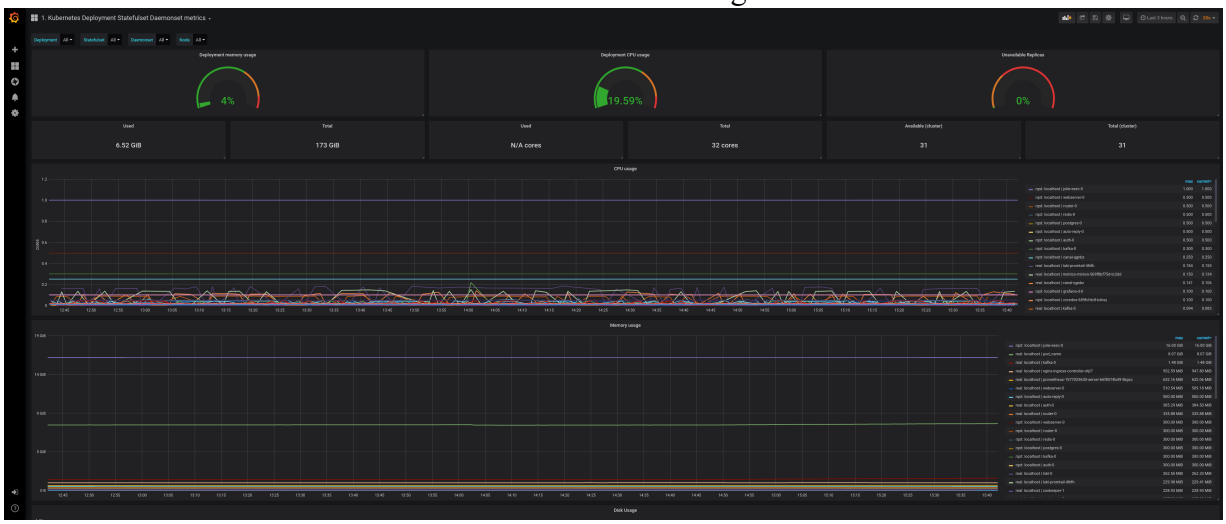
3.5 Fault tolerance

Since we have chosen to use Kafka for this project, we basically get fault tolerance for free. Kafka is fully distributed and might be one of the most fault tolerant pieces of software out there. It uses >2 instances of apache zookeeper, a distributed store (basically a filesystem structure) to manage the Kafka metadata. Kafka has the ability to have multiple brokers (brokers just means instances) crash and still go on as if nothing has happened. It uses leader election and built-in heartbeat methods which also the producers and consumers know of when connecting, and has automatic load balancing through broker rebalancing.

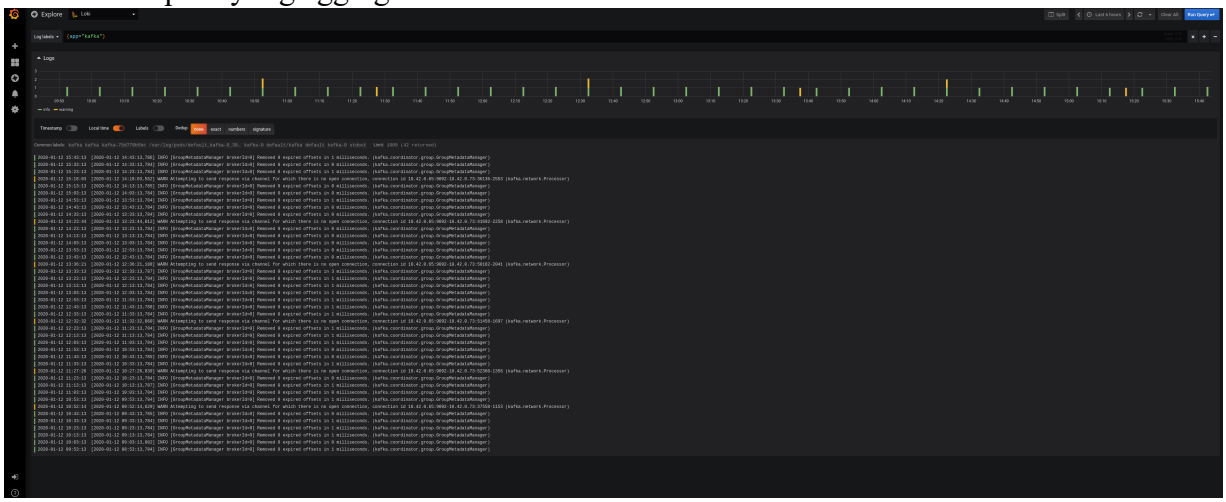
As stated earlier all services are completely stateless (again, escept the websocket manager), so it compliments Kafka to keep track of the messages, since Kafka is more of a log store than a traditional pub-sub.

3.6 Monitoring

For monitoring the system we have chosen to go the opensource, modern and standardized way with Prometheus, Grafana and Loki. These three are an equivalent of the ELK stack, but seem to have much less resource requirements and be less of a pain to maintain. We have used public Grafana dashboards for Kafka and Kubernetes monitoring:



And Loki is purely log aggregation:



4 Related Work and Discussion

The first intersting point of discussion is the choice of utilizing Apache Kafka for inter-service communication, rather than the commonly used REST architechture. There are multiple reasons as to why we ended up using Kafka.

Firstly, Kafka is very efficient, with the ability to saturate a 1GiB connection [5], even with small messages which are expected in a messaging/chat application.

Another positive aspect of using Kafka is the rather limited amount of boiler plate code required in each service to use the communication medium. Since REST runs on top of HTTP, using REST would require each service to run a webserver of its own, which, depending on the language, could be cumbersome. On the other hand, while most modern languages have Kafka clients readily available, it is not all. As such, the programmer might be limited in choice of language.

The primary argument one could make for REST over Kafka, is the simplicity of providing API access to the individual services, however this can be mitigated by using an API Gateway which translates REST requests to Kafka messages and vice-versa, which is considered good practice for microservice systems either way.

The use of Kafka eliminated Jolielang as an option for implementing microservices in the system easily. However, as per verbal agreement, a user should be able to write Jolie code snippets to run on their messages. Instead, the `Jolie-exec` service is implemented in golang, which in many ways is quite similar to an extended version of `C`.

As we decided to run the service on an ubuntu docker image, ubuntu group management is used along iptables to provide some level of security when running the user code snippets, and we are able to limit the runtime and memory usage of each snippet, something which seems unavailable in pure Jolie.

Since Jolie runs on the Java Virtual Machine, a rather large memory overhead is present for each snippet running, which brings us to the primary drawback of this approach. Jolie would not run if the memory was limited to anything below 6 GiB, which is a ridiculous amount of memory for simple message processing. It is possible that the could be avoided with some configuration, however, we did not find a nice solution.

If the application was to be used in production, the first obvious change would be discontinuation of Jolie snippets, migrating to a lower-overhead interpreted language such as Python or Lua. If we assume a python runtime memory overhead of around 32MiB (testing this locally shows ~16MiB in python 2.7.17), we could fairly safely (assuming no file transfers through regular messages) limit the snippet's available memory to 32Mib, for a total of 64MiB per instance. With each user snippet instance being limited to 64MiB memory, ~96 times more user snippets could run simultaneously, greatly reducing infrastructure costs.

5 References

[1] Prometheus Authors. Overview. 2020. URL: https://prometheus.io/docs/introduction/overview/ (visited on 12/01/2020).

[2] Martin Fowler. Event Sourcing. 2005. URL: https://martinfowler.com/eaaDev/EventSourcing.html (visited on 12/01/2020).

[3] The PostgreSQL Global Development Group. PostgreSQL: About. 2020. URL: https://www.postgresql.org/about/ (visited on 12/01/2020).

[4] Confluent Inc. What is Apache Kafka? 2020. URL: https://www.confluent.io/what-is-apache-kafka/ (visited on 12/01/2020).

[5] Jay Kreps. Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines). 2014. URL: https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines (visited on 12/01/2020).

[6] Google LLC. Migrating a monolithic application to microservices on Google Kubernetes Engine. 2019. URL: https://cloud.google.com/solutions/migrating-a-monolithic-app-to-microservices-gke (visited on 12/01/2020).

[7] Google LLC. What is Kubernetes? 2020. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (visited on 12/01/2020).

[8] Grafana Labs Ltd. Features. 2020. URL: https://grafana.com/grafana/ (visited on 12/01/2020).

[9] Grafana Labs Ltd. Grafana Loki. 2020. URL: https://grafana.com/oss/loki/ (visited on 12/01/2020).

[10] Redis Labs Ltd. Introduction to Redis. 2020. URL: https://redis.io/topics/introduction (visited on 12/01/2020).

[11] XenonStack. Stateful and Stateless Applications Best Practices and Advantages. 2018. URL: https://www.xenonstack.com/insights/stateful-and-stateless-applications/ (visited on 12/01/2020).