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.

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.

2.5 Grafana

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

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.

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.

2.8 Postgres

Postgres is a state of the art high performance RDBMS.

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.

2.10 JSON

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

2.11 JWT

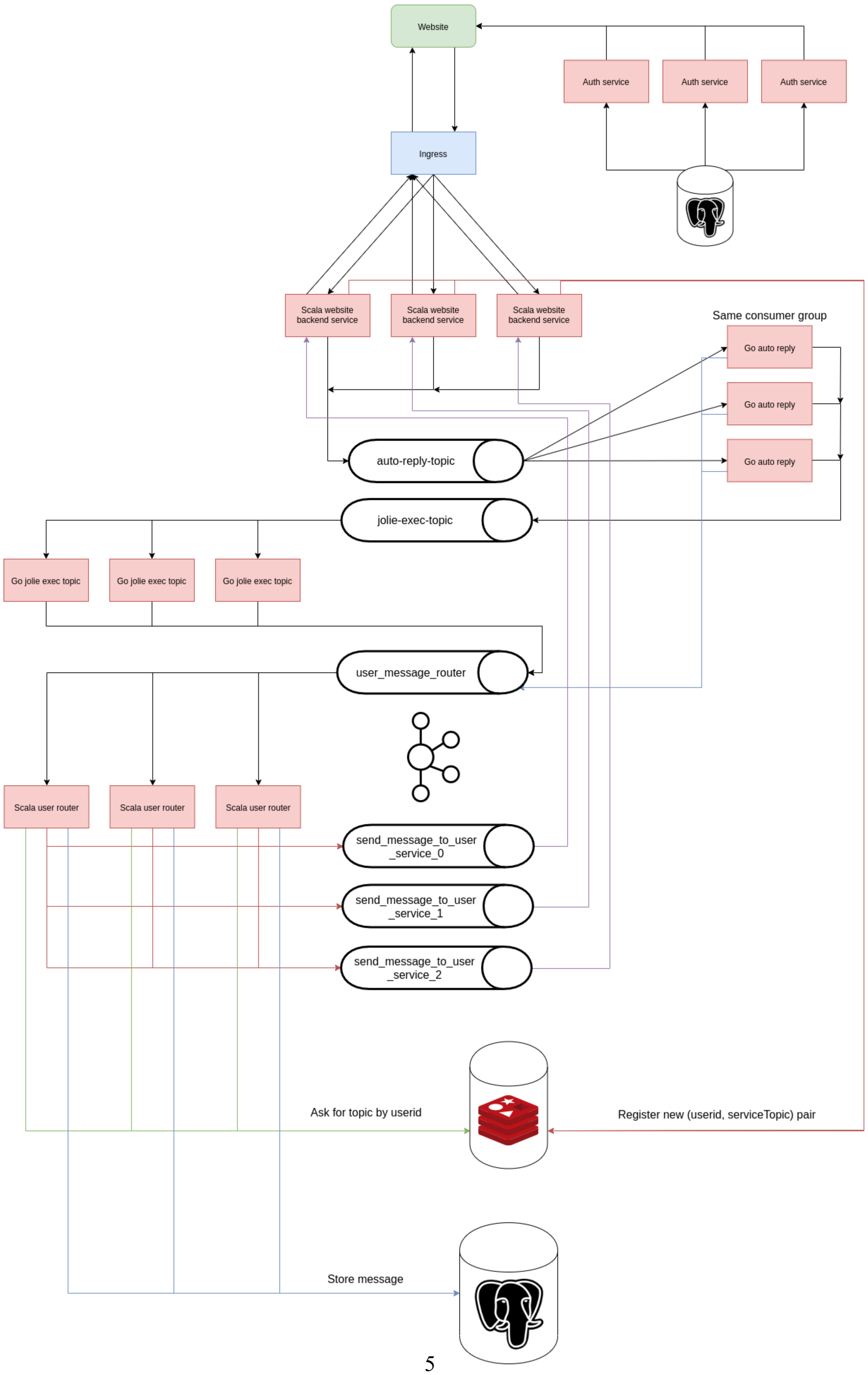
JWT stands for Json-Web-Token, which is a method of dispensing authentication tokens to users.

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. 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.

```json
{
    "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

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

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 Static services

3.3 Deployability

3.4 Reusability

3.5 Performance

3.6 Fault tolerance

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 (TODO: cite https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines), even with small messages as 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.

5 References