Software architecture document

TwitterV2

Daniil Blagoev

Contents

[Introduction 2](#_Toc169780263)

[TwitterV2’s architecture 2](#_Toc169780264)

[System context diagram 2](#_Toc169780265)

[Container diagram 2](#_Toc169780266)

[Component diagram 4](#_Toc169780267)

[User service 4](#_Toc169780268)

[Tweet service 5](#_Toc169780269)

[Reasoning behind TwitterV2’s architectural choices 6](#_Toc169780270)

[Why microservices? 6](#_Toc169780271)

[Scaling of individual components 6](#_Toc169780272)

[Reliability, Uptime & Continuous deployment 6](#_Toc169780273)

[Why a gateway? 6](#_Toc169780274)

[Why a separate database for each microservice? 6](#_Toc169780275)

[Reliability & Uptime 6](#_Toc169780276)

[Data distribution & Security 6](#_Toc169780277)

[Why messaging instead of direct communication between microservices? 7](#_Toc169780278)

[Reliability & Uptime 7](#_Toc169780279)

[Performance 7](#_Toc169780280)

# Introduction

The purpose of this document is to provide an insight into TwitterV2’s architecture and the reasoning of the choices behind it.

# TwitterV2’s architecture

## System context diagram

TwitterV2 does not use any external systems, therefore the system context is quite simple.

A diagram of a user

Description automatically generated

Figure 1 TwitterV2 context diagram

## Container diagram

A diagram of a company

Description automatically generated

Figure 2 TwitterV2 container diagram

## Component diagram

### User service

A diagram of a computer

Description automatically generated

Figure 3 TwitterV2 User API server component diagram

### Tweet service

A diagram of a diagram

Description automatically generated

Figure 4 TwitterV2 Tweet API server component diagram

# Reasoning behind TwitterV2’s architectural choices

## Why microservices?

### Scaling of individual components

I created an [estimation](ResourceMapping.docx)/resource mapping of the use of TwitterV2’s different functionalities, and it became clear that some would be bearing several times heavier loads than others. For example, tweet functionalities will be used a lot more than user functionalities. Based on this information, it made sense to go for microservices, since each service can be scaled individually based on its needs. After all, why would you scale the whole application when only a small part of it needs to be scaled?

### Reliability, Uptime & Continuous deployment

Due to the decoupled nature of microservices, each service is self-reliant – it can still function when the rest of the services are down. The problematic services can be fixed and redeployed without having to take the whole web application down for maintenance and losing users in the process. In addition, deployment of new features or services does not require the whole application to be taken down, giving more flexibility and efficiency to the development process.

## Why a gateway?

I chose to route the communication from the frontend to the services through a gateway mostly for the sake of security. The gateway makes it incredibly easy to do multiple things which are of great importance to a web application:

* Security configuration – Configurable request limits and authentication for each microservice in a single place.
* Monitoring – Aids in understanding the tendencies of how users use the APIs.

## Why a separate database for each microservice?

### Reliability & Uptime

If there are any problems with a service’s database, it would not affect the rest of the services and the web application would keep working. Meanwhile, if all services were to work with a single database, if that database goes down, the web application is going down with it.

### Data distribution & Security

It allows each service to only have access to data which is relevant to it. In case a service’s database is compromised, the surface area of the attack is greatly reduced.

## Why messaging instead of direct communication between microservices?

### Reliability & Uptime

If a service is supposed to communicate with another service which is down, the web application can keep working without the first service being affected. It just asynchronously sends a message to the other one and does not wait for an answer. Meanwhile, with HTTP communication, the first service would be stuck waiting for an answer from the second one, which would result in two services being rendered unusable instead of just one.

### Performance

When a service is supposed to communicate with another service, it can asynchronously send a message to the other service and keep working on the rest of the incoming requests. Meanwhile, with HTTP communication, the first service would be stuck waiting for an answer from the second one.