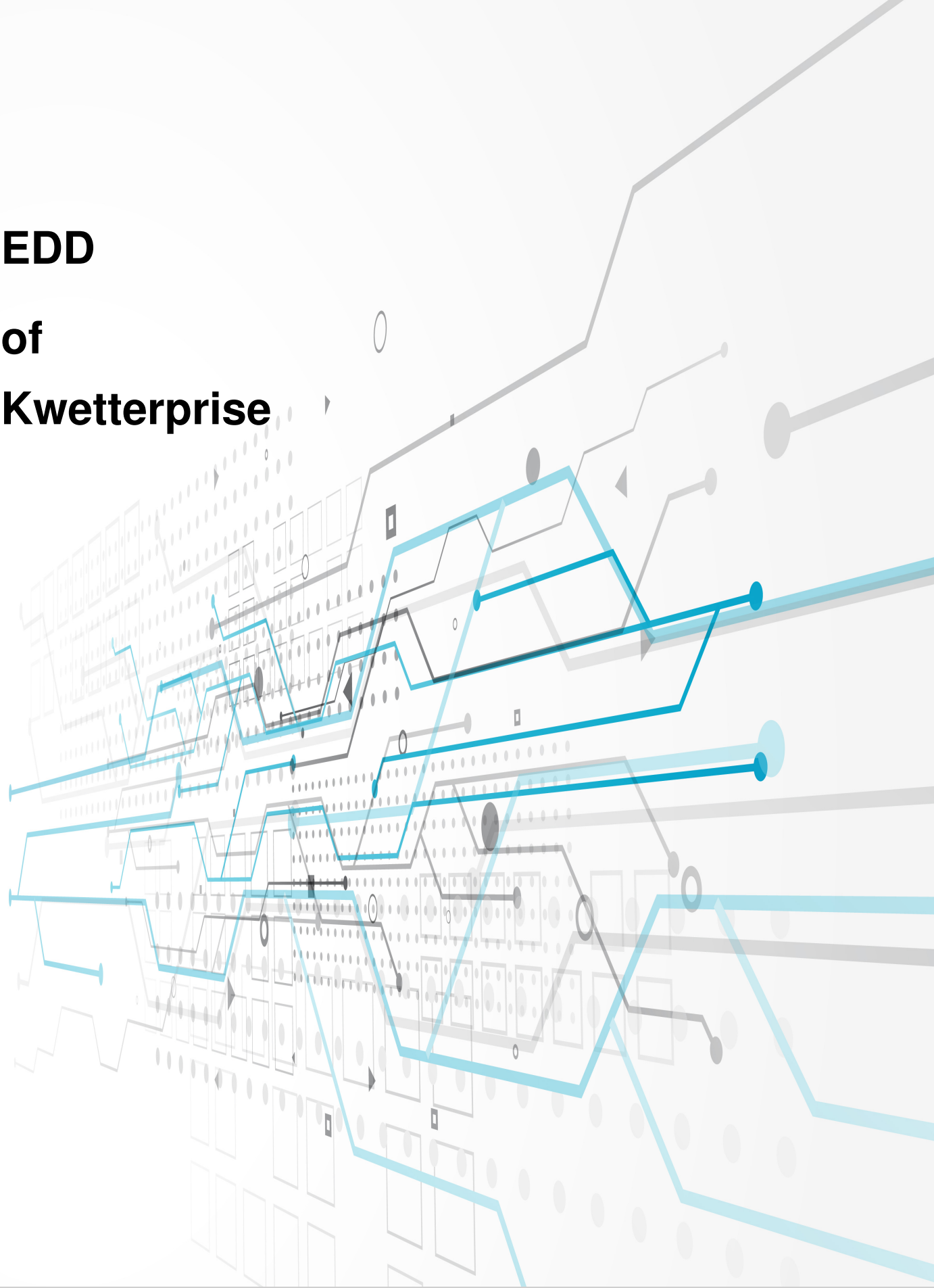
EDD

of

Kwetterprise



Date:	2020-04-15	Course:	S6
Author(s):	Tim van den Essen	Student number:	3118525
Status:	Concept		

Document history

Rel.	Date	Status	Changes
R01	2020-04-15	Concept	First version

Distribution list

Name	R01	R02	R03
Kwetterprise			
Tim van den Essen	A		
Fontys			
Erik van der Schriek	X		
Frank Coenen	X		

Document

Text marking

Marked text	Text needs to be changed or completed.
Marked text	Text has changed compared to the previous release.
Marked section	Section headers that are intended for review.

Contents

List of Figures

List of Tables

List of Listings

1. Introduction

This document describes the design of Kwetter.

1.1. Definitions and abbreviations

RSD	Requirements Document
SPD	Specification Document
EDD	Engineering Design Document
CQRS	Command–Query Responsibility Segregation
SPoT	Single Point of Truth

2. High Level Design

Figure 2.1 contains a diagram of Kwetterprise's architecture.

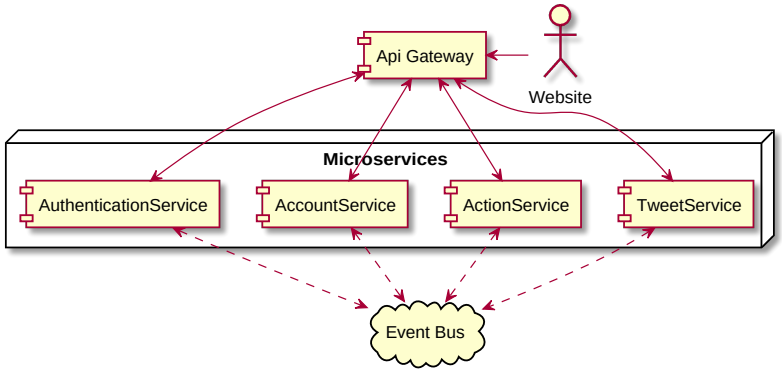


Figure 2.1: High level design.

This architecture consists of four main parts:

- **Microservices**
  The business logic of Kwetterprise is handled by multiple microservices. Microservices are chosen to seperate responsibilies and functionality.
- **API gateway**
  The API gateway is used to dyamically route requests from the front-end (or other API users) to the correct micro-service.
- **Event bus**
  Communication between microservices is event-driven. A so called "event bus" is responsible for accepting and delivering events.

2.1. Service design

Services are designed with the "command–query responsibility segregation" principle (CQRS) in mind. Figure 2.2 shows this archtecture.

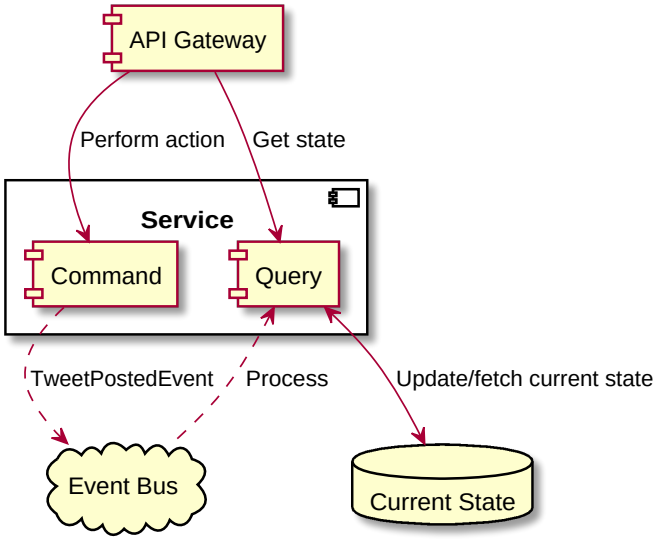


Figure 2.2: High level design of a service.

The service consists of two parts. The *Command* part handles commands which are invoked by an

actor such as tweeting or following a user. These commands result in events being published to the event bus.

The *Query* part of the service is responsible for presenting this data to other components. It processes the events created by the *Command* component and keeps a database with the current state. This database functions as a “cache” for query requests like retrieving a user’s tweets or followers.

The reason this cache exists is because otherwise a “retrieve all followers from this user” request must fetch all events relating to that user (following, unfollowing, account deletion, etc) and process them to determine the final result. This would put high load on the service the moment a query is executed instead of balancing the load whenever updates are pushed.

Thus, the SPoT (single-point of truth) is the events in the events bus. Note that the “current state cache” database can always be recreated by processing all historic events.