Software architecture document

Version 1.0

March 20th, 2021

Document history

Versions

Version	Date	Author(s)	Changes	Status
0.1	03-03-2021	Kevin Heijboer	Document setup, technologies and tools	Concept
0.2	07-03-2021	Kevin Heijboer	Microservices domain models, CI/CD pipeline overview	Concept
1.0	20-6-2021	Kevin Heijboer	Technologies, Architecture, CI/CD	Concept

Table of Contents

1 Introduction

1.1 Purpose

This document contains the architectural design for Quacker. This document describes the composition of the software in separate components. For each component, the interfaces and dependencies of other components are described, as well as the software requirements they meet. This document also describes a brief overview and the context of the system.

1.2 Scope

Quacker consists of a containerized microservice architecture back end and a web application front end. The system web interface is designed for all modern web browsers. Quacker allows users to communicate with each other through quacks and follows. The application consists of one web application where certain functionalities are locked behind permissions. Data is stored in databases per microservice.

2 System overview

2.1 Microservices

A microservices architecture consists of a collection of small, autonomous services. Each service is self-contained and must implement a single business goal.

Each microservice is a separate code base. Microservices are relatively small and independent, and they can be developed and maintained by a small development team. Microservices can also be deployed independently of each other. This means that unlike a monolithic application, the entire system does not have to be deployed again for each update.

Each microservice is responsible for storing its own data. This could be a database per microservice. Only data that is relevant to the business processes of the service in question is stored per microservice.

Due to the distributed nature of this architecture, it can fall short in terms of performance. Within traditional monolithic applications, communication between the various components takes place directly via code calls. This is significantly faster than communication protocols such as HTTP or AMPQ. The benefits such as agility, scalability, etc. that microservices bring far outweigh this performance shortcoming.

Scalability is one of the biggest advantages of the microservice architecture. Because microservices are so distributed and independent of each other, they can be scaled up or down on a per-service basis. When a service receives more traffic than other services, only the corresponding server needs to be scaled up. This is in contrast to traditional applications, where the entire application needs to be scaled up. Component scaling saves money and server space.

2.2 Publisher/Subscriber messaging

The publish-subscribe pattern is a messaging pattern in which the message publishers do not program the messages to be sent directly to specific subscribers, but instead divide the published messages into classes without knowing which subscribers, if any, are present. Similarly, subscribers can subscribe to one or more classes and receive only messages that are important to the subscriber, without knowing which publishers are present.

Within distributed applications, components often need to send information to other components as events occur. Asynchronous messaging is a way to keep publishers and subscribers independent of each other and to avoid blocking the sender from waiting for a reply. It is also possible that some components are only interested in part of the information.

Furthermore, this messaging pattern increases scalability. The sender can quickly send a single message to the input channel and then return to his normal functions. The messaging infrastructure is responsible for delivering messages to subscribed subscribers.

3 Technologies

3.1 Client-side

Vue

The client-side web application is built on the Vue JavaScript framework. Vue helps with developing user-interfaces as well as most of the client-side logic present in the web application. The major benefit of using Vue is development speed. Since the developer already has experience with Vue, it will not be required to learn any new frameworks such as React or Angular. This allows to spend more time on more complex subjects such as CI/CD, DevOps and microservices, which require more attention in this project.

Cypress

Cypress is the testing framework used for the client. It allows for easy to write tests. Cypress is mainly used for all end-to-end tests in the client but can also be used for any front-end unit or integration tests. Cypress simulates a browser on which it executes all commands written in tests. It has great support for CI pipelines and allows for artifacts such as video recordings and screenshots of the browser simulation.

3.2 Server-side

.NET 5

The C# framework .NET is used for the development of the application server. .NET provides many utilities for quick development of web APIs as well as support for authentication and authorization. .NET Core also makes it easy to add middleware and has great support for dependency injection. A benefit besides previous .NET Core experience is the extensive microservices and containerization documentation that Microsoft has released.

Entity Framework Core

An object-relational mapped (ORM) is used to efficiently handle transactions with the database. The ORM EntityFramework Core is used in the application server. It is developed by Microsoft and works seamlessly with .NET Core. EF Core removes the need for writing any SQL for most basic use cases.

Identity

Microsoft's Identity framework is used for handling all processes related to authentication and authorization. The identity framework offers are pre-built database structure for storing and authenticating users. It also offers many ways of authenticating such as JWT and session based.

Ocelot

Ocelot is an API gateway framework for .NET. It allows for a unified point of entry into the back-end system.

3.3 Database

MySql

A MySQL database is used for storing all the data related to Quacker. MySQL is a well-known, free, and open-source database system that is supported by Entity Framework.

3.4 Cloud

DigitalOcean Kubernetes

DigitalOcean is the cloud provider that is used to host Quacker. It offers Kubernetes clusters with configurable resources such as CPU and memory. It also supports automatic vertical autoscaling.

DigitalOcean Managed Database Cluster

DigitalOcean's Managed Databases are a fully managed, high performance database cluster service. It supports multiple databases and takes away most of the effort and time it costs when installing, configuring, maintaining, and securing databases by hand.

Azure Service Bus

Azure Service Bus is the cloud service used for handling all messaging communication between microservices. Azure Service Bus supports the publisher-subscriber pattern and works well with .NET 5.

Amazon S3

Amazon's S3 service is a storage service that is used for storing profile pictures in Quacker. It also works as a CDN, allowing for faster loading of images on Quacker. Using S3 removes the effort of storing images in a database, or on the system locally.

3.5 Monitoring

Istio Service Mesh

Istio Service Mesh is used as a way to proxy and measure data between microservices. Istio injects a so-called “sidecar” into each microservice. Using Istio allows for use of extra monitoring tools and data.

Prometheus

Prometheus is used in combination with Istio for recording real-time metrics in a time series database. It allows for extra insights with queries.

Kiali

Kiali is a management console for an Istio-based service mesh. It provides dashboards, and observability for the Kubernetes Cluster. It shows the structure the cluster by measuring traffic and displays the health of each microservice.

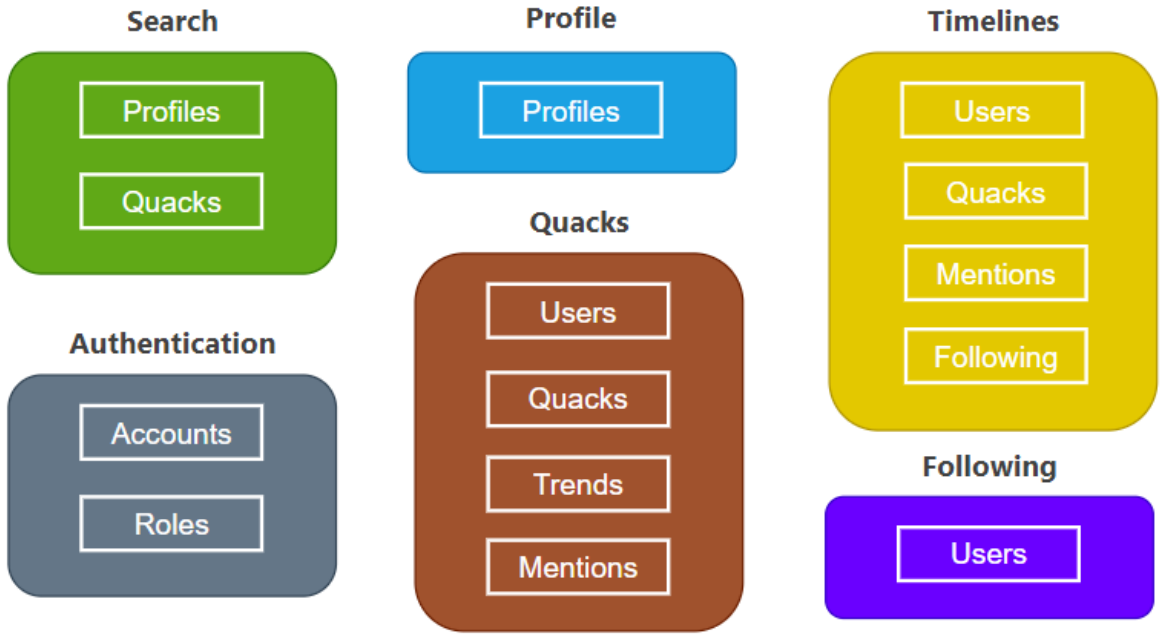
Grafana

Grafana is used as a monitoring dashboard for measuring load. It allows for custom dashboards. Istio provides are couple of pre-built dashboards.

4 System architecture

4.1 Domain model per microservice

The below figure illustrates the different microservices used in Quacker, along with their domain models. The domain models help in understanding the purpose of each microservice, as well as how they differ from each other. Throughout the domains, the same entity appears as “Profiles”, “Accounts”, “Users” depending on the service. Even though these entities have the same identity, they have different shapes. The “profiles” entity has no business in knowing a user’s password where the “accounts” entity does.



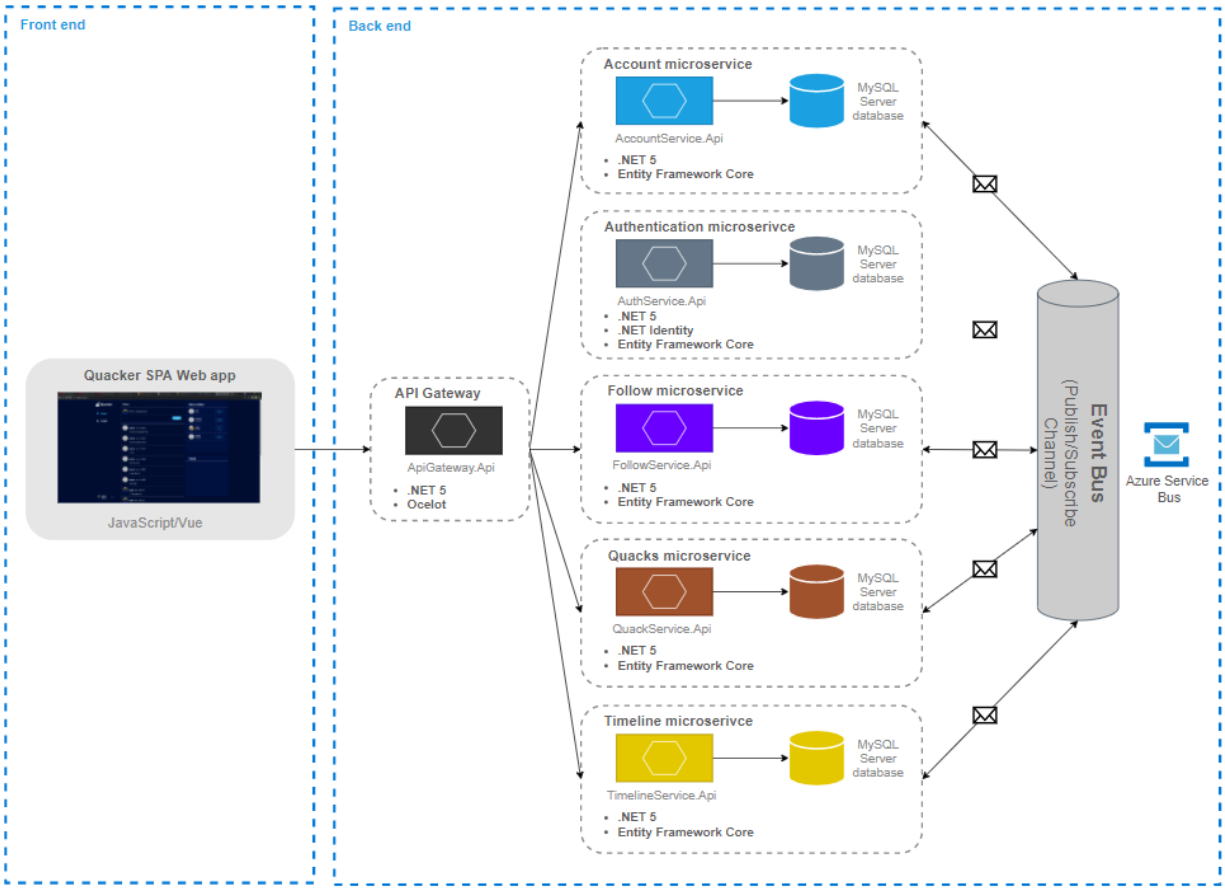
Each microservice has their own database containing most of the data that they need in executing business processes.

4.2 Architecture

The below diagram shows the overall architecture and workflow of Quacker. Users can interact with the system via the web client. The web client communicates directly with the API gateway, which serves as a unified point of entry into the server. The API gateway reroutes every request to its designated microservice as configured.

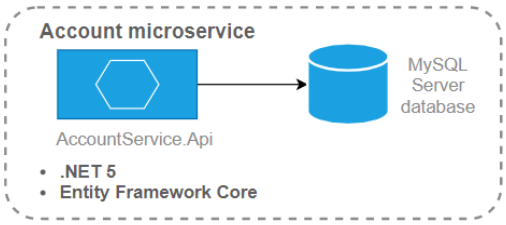
Every microservice is a web API itself. These APIs can not be accessed externally. Every microservice has their own MySQL database in which they store their data. These databases are managed by DigitalOcean in a database cluster.

Microservices communicate with each other via messaging. Messaging is done over the Azure Service Bus using a publisher/subscriber pattern.



The following sections describe each microservice in more detail. Each section consists of a component diagram, class diagram and a database diagram of the corresponding microservice.

Account microservice

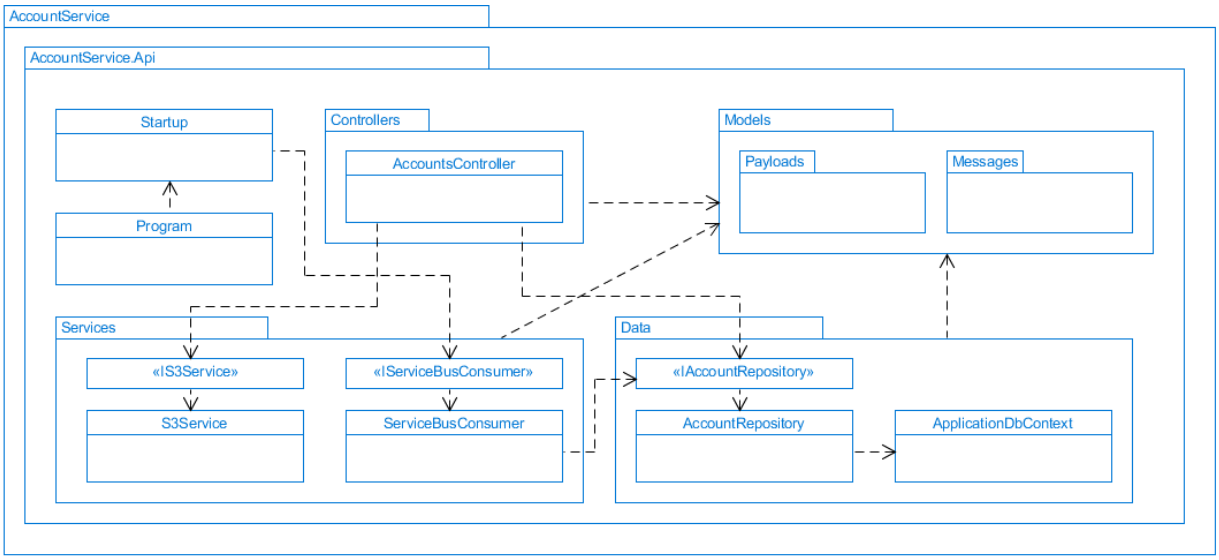


The account microservice handles all business processes regarding user profiles. New registrations are received via messaging and stored in the corresponding database. Of these new registrations only the user id and the username are stored.

The account service allows for editing user profiles. This also includes changing profile pictures by communicating with the Amazon S3 buckets.

Components

The following diagram describes all the components in the account service. Models are not shown as they are described in more detail in the following section.

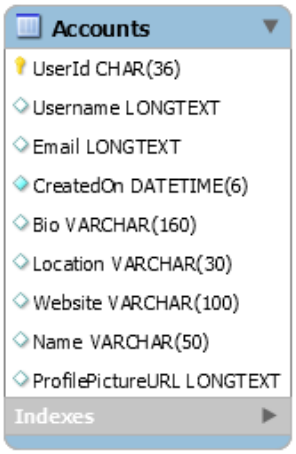


Class diagram

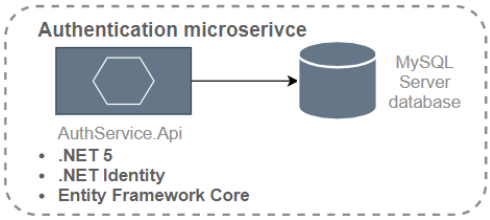
The class diagram for this service is relatively simple. It consists only of an account class which contains all the information needed for storing a user profile.

Database diagram

The following diagram shows the database structure for the account service database.



Authentication microservice

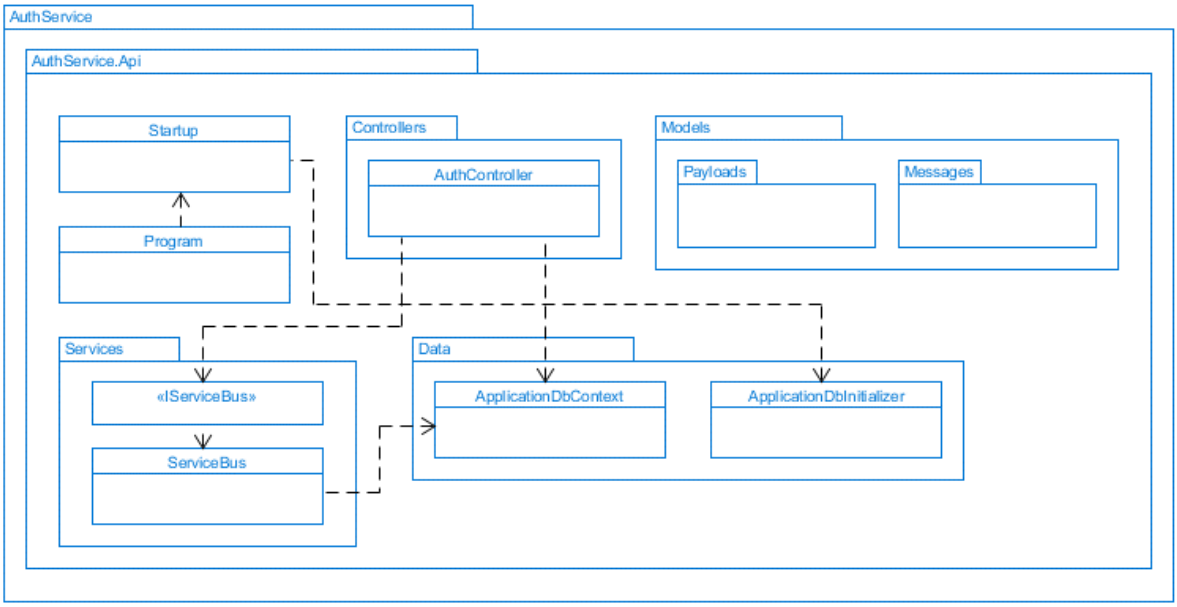


The account microservice handles all business processes regarding authentication. All registration and login requests are routed to the authentication service. After a successful login, the service will return a generated JWT.

After a successful registration, the service will store the newly registered user and publish a message to let other services know of the new user.

Components

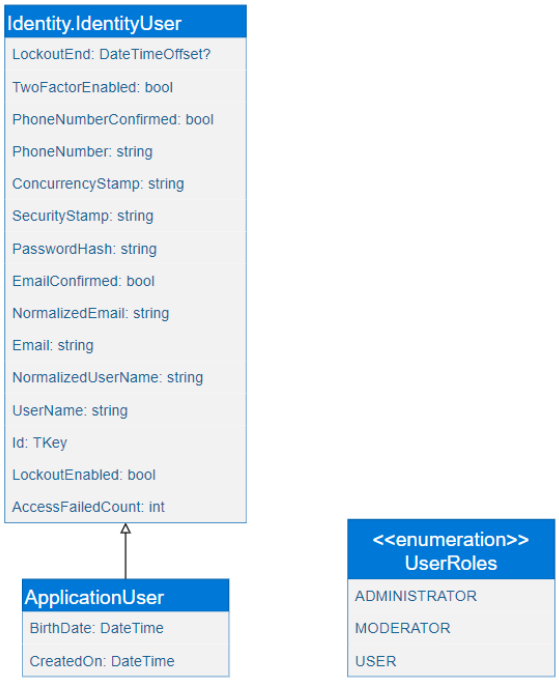
The following diagram describes all the components in the authentication service. Models are not shown as they are described in more detail in the following section.



Class diagram

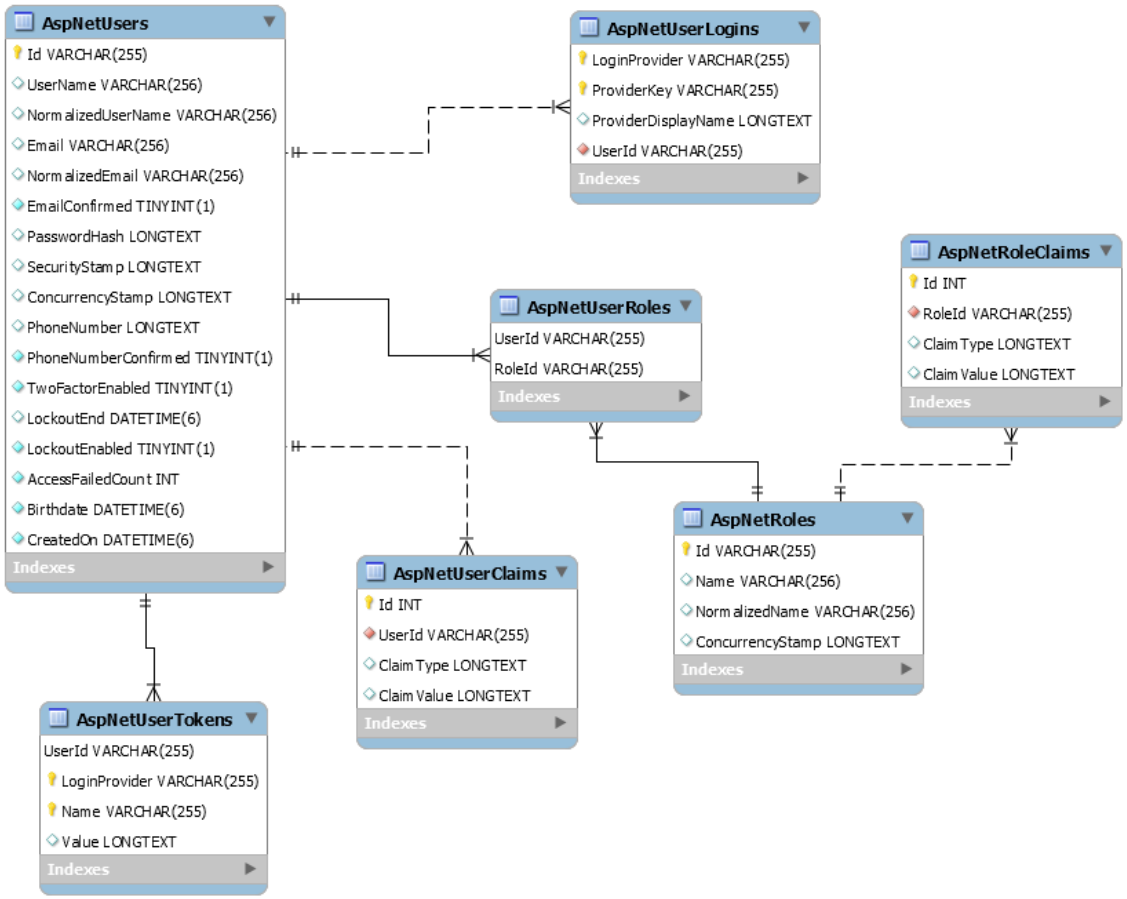
The only model used in this microservice is the ApplicationUser model. This class inherits from the IdentityUser class provided by the Identity framework. As shown in the diagram below, the Identity framework adds many fields for authenticating, authorizing and storing users.

The UserRoles enumeration is used as a way to keep track of different user roles.

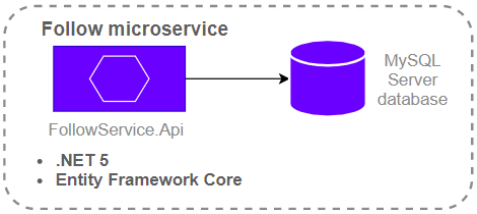


Database diagram

The following diagram shows the database structure for the authentication service database. All columns in this database are generated by the Identity framework.



Follow microservice

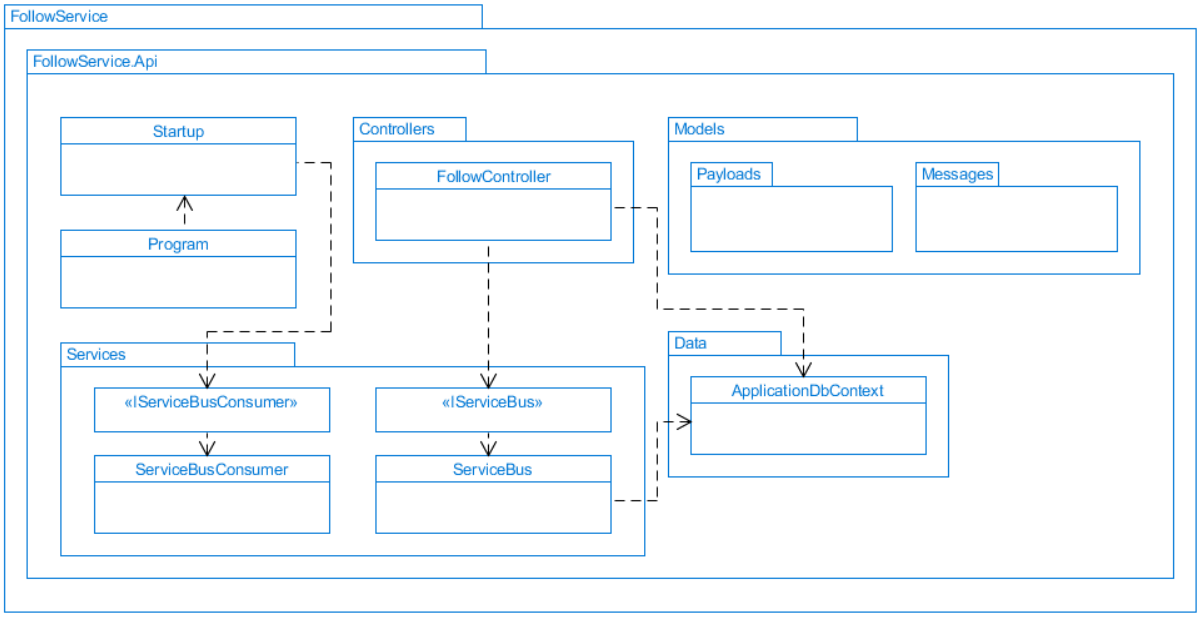


The follow service handles all functionalities regarding following and unfollowing users. Like the account microservice, new registrations are received via messaging and stored in the corresponding database. Of these new registrations only the user id and the username are stored.

The follow service publishes messages to the service bus whenever a user follows or unfollows another user.

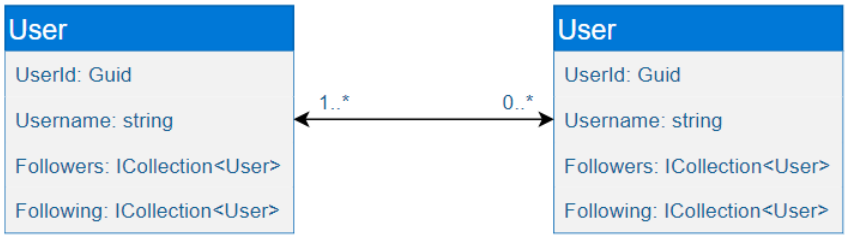
Components

The following diagram describes all the components in the follow service. Models are not shown as they are described in more detail in the following section.



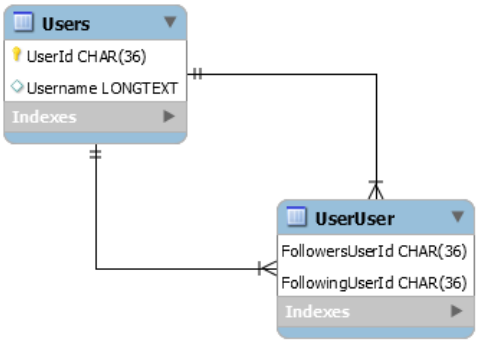
Class diagram

The below class diagram shows the relation between a user and its followers/following.

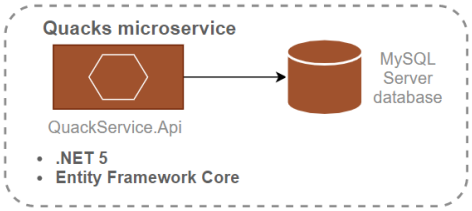


Database diagram

The following diagram shows the database structure for the follow service database. This structure is generated by Entity Framework based on class relations described in the above class diagram.



Quacks microservice



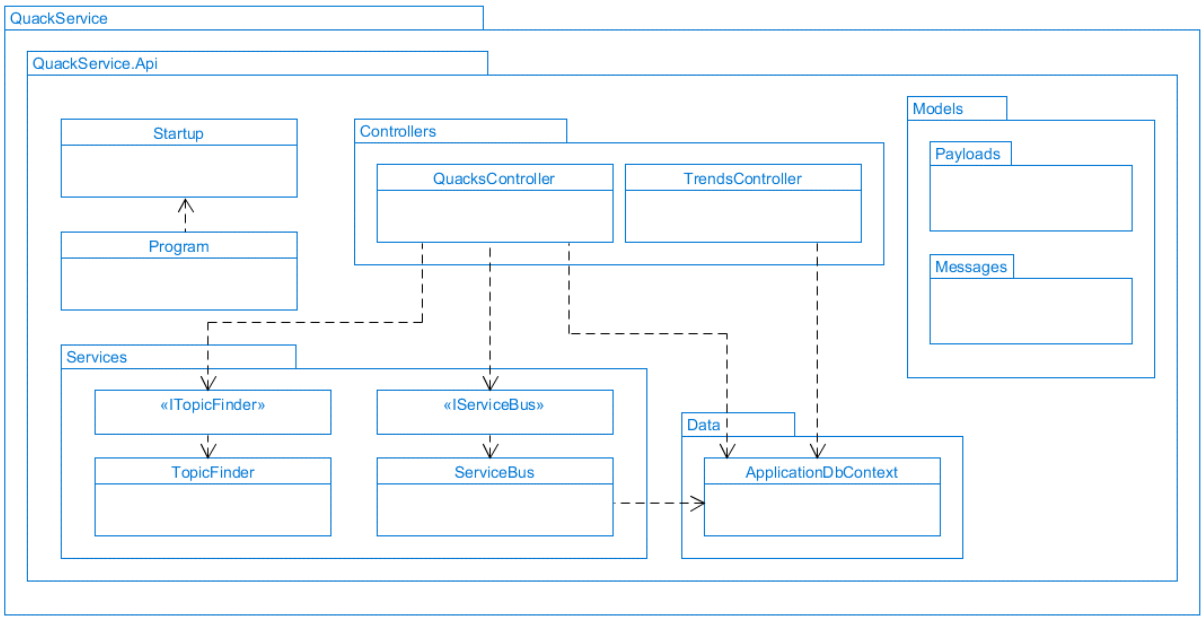
The quack service handles all functionalities regarding posting quacks, retrieving quacks, mentions, topics and determining trends.

On posting a quack, the quack service will determine all topics and mentions inside the quack by looking for the # and @ prefixes. Furthermore, a profanity check is performed on every quack by calling the corresponding Azure Function. A message is published to the service bus for each newly posted quack.

The trends controller determines trends by looking at popular topics in the last week.

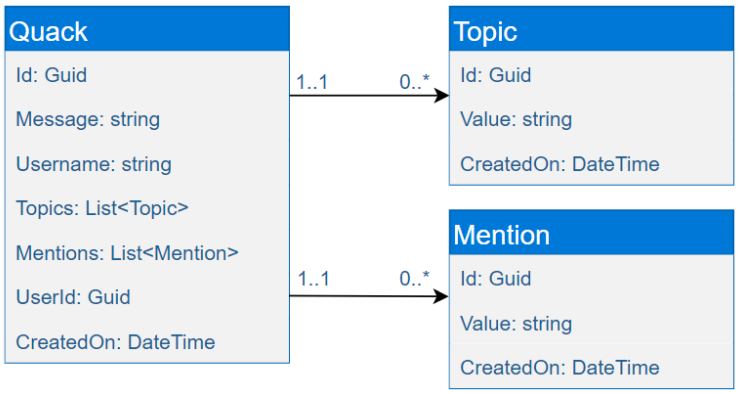
Components

The following diagram describes all the components in the quack service. Models are not shown as they are described in more detail in the following section.



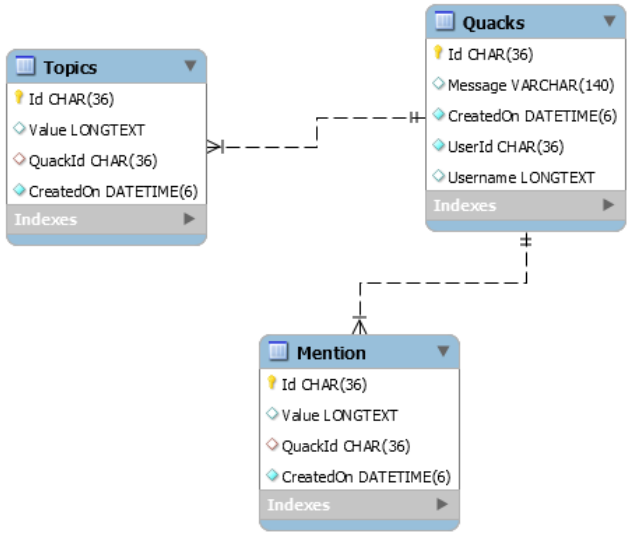
Class Diagram

The following class diagram shows the relation between a quack and its topics and mentions. A quack can contain multiple topics and mentions.

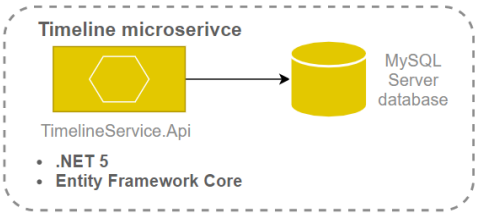


Database diagram

The following diagram shows the database structure for the quack service database. This structure is generated by Entity Framework based on class relations described in the above class diagram.



Timeline microservice

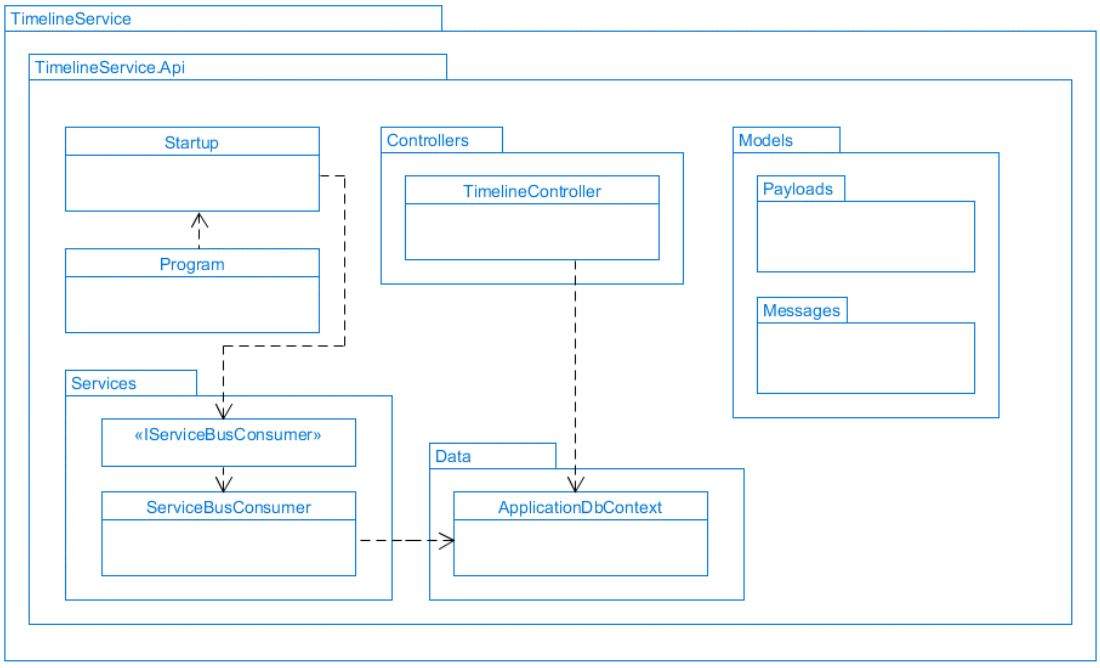


The timeline service determines all functionality regarding generating a timeline for each user. The timeline is generated when a user visits their home/timeline page. The timeline service is subscribed to multiple message topics which are required to generate the timeline.

A timeline is generated based on who a user is following, their own quacks and quacks in which they are mentioned.

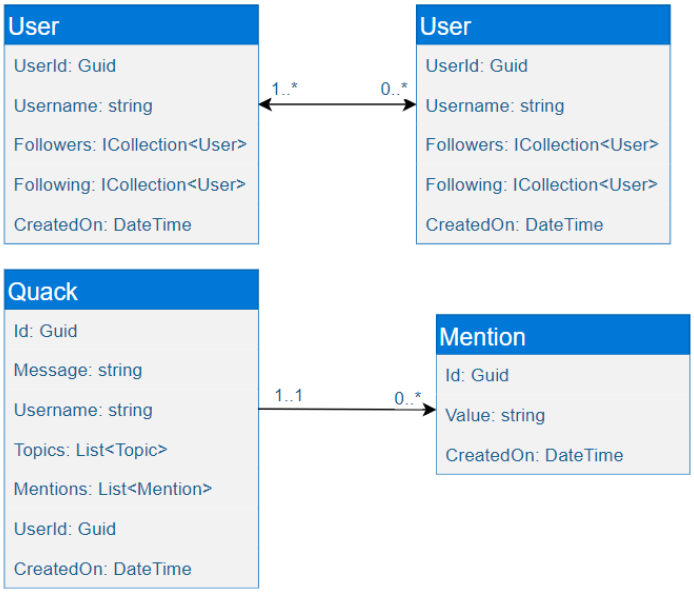
Components

The following diagram describes all the components in the timeline service. Models are not shown as they are described in more detail in the following section.



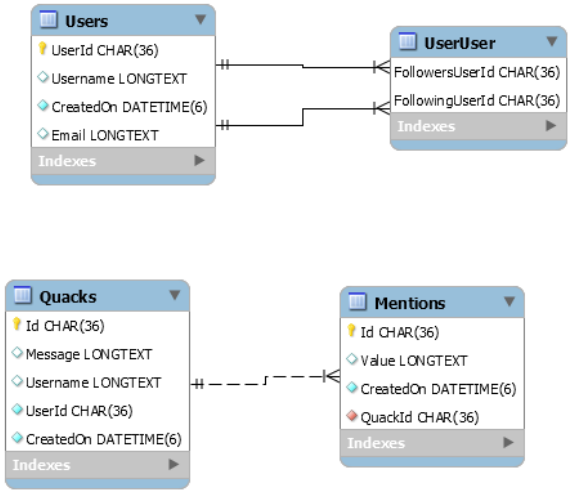
Class Diagram

The following diagram shows the different models used in the timeline service. All models in this service are received via messaging from either the authentication service, follow service, or the quack service.



Database diagram

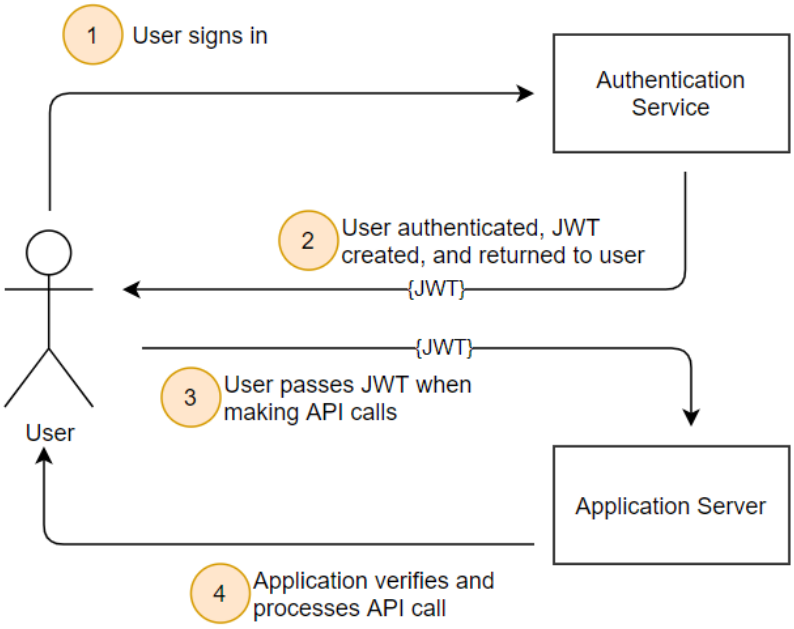
The following diagram shows the database structure for the timeline service database. This structure is generated by Entity Framework based on class relations described in the above class diagram.



5 Authentication and authorization

The authentication and authorization of Quacker are achieved using JSON web tokens (JWTs). This section will describe the process of authenticating by describing every step in detail.

The below diagram shows a global overview of how the authentication is implemented.



5.1 Microsoft Identity

The back-end authentication and authorization is mostly handled by the Microsoft Identity package. This package takes care of all database structures and transactions regarding users and roles.

JWT

The Identity package is configured to use JWT authentication. This requires a secret key which will be used for generating and validating tokens.

5.2 Registration

New users register on a registration form in the front end. Any valid registration requests will be handled by the authentication microservice. This service first checks for duplicate usernames, duplicate email addresses and valid field lengths.

5.3 Authenticating

Users can log in via the front end. This will send a request to the api gateway which in turn routes the request to the authentication microservice.

The authentication service will generate a JWT with an expiration date of two days on a successful login. The function will then return the JWT, along with additional user information such as user roles, back to the client.

5.4 Storing the JWT on the client side

Upon receiving a successful login response, the front end stores the user information and the token in the browser's cookies.

5.5 Authentication header

Every request to the server carries an authentication header. This way, the server will receive the JWT with every request. From the JWT, the server can determine the logged in user and their permissions.

5.6 Authorization

Users are only allowed to perform operations if they have the right user role, such as the administrator and moderator role. The client-side prevents most unauthorized requests in various ways.

Some buttons and other elements are only rendered if the logged-in user has valid right user role. Most pages also have redirect guards that redirect unauthorized users.

Every microservice will always check for authorization on requests in case the client fails to prevent any unauthorized requests.

6 CI/CD

6.1 Pipeline overview

The CI/CD pipelines consists of four parts. Building, testing, staging, and deploying.

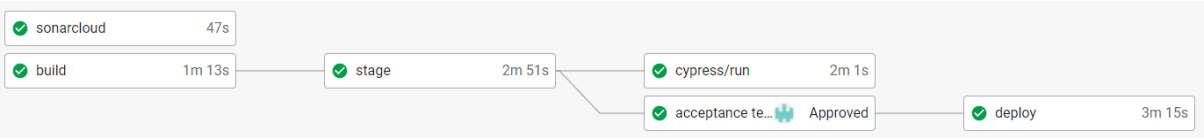
After each Git push to the master branch, the pipeline is triggered. All the code is compiled and built. Afterwards, any unit tests and integration tests are executed.

If the previous steps pass, the application is deployed to a staging server (staging.quacker.nl). This staging server is live alongside the production server, under a different namespace in the Kubernetes Cluster. It allows for end-to-end tests and users tests using dummy data. By executing tests on the staging server, the production server is not affected.

If all end-to-end tests pass, the pipeline will ask for a manual approval. This allows for performing acceptance tests. If the staging environment is approved, the application is automatically deployed to the production server.

Front end pipeline

The following image shows the pipeline that is used for the front end. For each execution, the application is built and at the same time scanned by SonarCloud. Afterwards, the application is put on a staging server. The staging server can then be used for end-to-end tests (cypress/run) and acceptance testing, after which the application can be deployed on the production server.



Microservice pipeline

The following image shows the pipeline used for every microservice. For each execution, the application is built. After a successful built, both unit tests and integration tests are executed. If all tests pass, the application is deployed on the staging server. The staging server can then be used for user acceptance testing, after which the application can be deployed on the production server.