SOEN 487 – Final Project Report

Giovanni Prattico – 27316860
Andres Augusto Nunez Zegarra – 27194331
Joel Dusablon Senécal – 40035704
Vartan Benohanian – 27492049

Team Giovanni Prattico

April 27th, 2019

OVERVIEW

For the final project of SOEN 487, the team decided to develop a messaging service. We used Python's web framework Flask to develop the routes in the backend. While it wasn't a requirement for the project, we decided to improve the user experience by connecting a frontend user interface to the backend. The frontend is completely built with ReactJS. We used SQLite as our preferred method of storing data.

This project was broken down into 3 microservices:

1. Authentication service, used for registering users and authenticating them.
2. Messaging service, used to send and receive messages from other users.
3. Notification service, to notify users when they receive a new message.

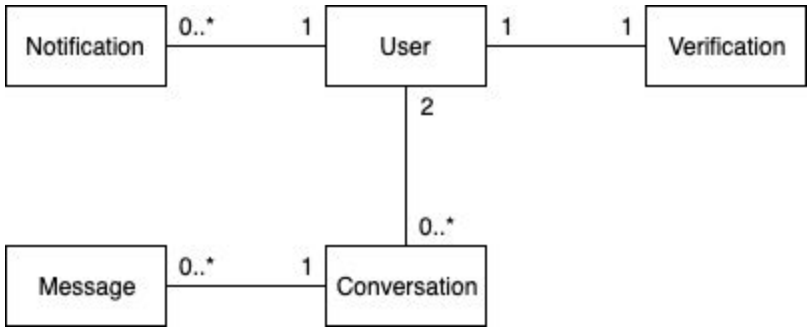
The code can be viewed here. The project is currently hosted here, but it is highly encouraged to read to full document before trying out the application. Demo steps are outlined in the *README.md* of the repository.

GOALS

- Learn how to develop an application using a microservices approach.
- Learn how to develop a REST API with the Python Flask framework.
- Use a version control system to develop our project. We used Git with GitHub.

CLASS DIAGRAM

Here is the class diagram used for this project. Where the entities are (i.e. in which microservice) will be broken down in the next section of the document.



The *diagrams* directory in the root of the repository contains more fine-grained representations of each service's class diagrams.

OUR MICROSERVICES

To develop any messaging service, we need users who are identifiable. Without users, our application wouldn't make much sense, so we need a way for users to register, and then log in to be able to send messages to other users.

Much like the name "messaging service" suggests, we would also need messages. The messaging service handles all the functionality having to do with creating conversations and sending messages.

To update a user on when they receive a new message, we've also developed a notification service. Each time a user receives an incoming message, they get notified.

Instead of sharing one large database, each microservice establishes a connection its own unique, smaller database. We wanted to make sure to follow the separation of concerns principle.

Authentication Service

The authentication service is arguably the simplest to develop and understand of all the services. It's also the riskiest, since every other service depends on this one. For users to be able to use the application for its intended purpose, we need a fully developed and working authentication service.

This service contains the basic method for registering a user, verifying an account, and logging in. Since it's directly connected to the SQLite instance containing the users table, it also has some convenient methods to retrieve all users, get one by their id, or filter by their username.

Once a user registers an account in the application, they get an email with a verification link, which contains their corresponding key to verify their account. A user will not be able to login unless their account is verified. Once they visit that link, the verification row associated to them will be deleted. A user is considered verified, i.e. can log in, if they don't have a verification model attached to them.

IMPORTANT: The email credentials are in a .env file, not checked into the repository. They need to be kept secret. There is a *.env-example* template file in the *server/authentication_service/app* directory of the project. It must be copied to a *.env* file with the following contents:

```
EMAIL=soen487projectms@gmail.com
PASSWORD=wnjcsbnmuezvgesi
```

The following are the REST API endpoints for the authentication service and their specifications. All but the register, verify, and login methods require the user to be authenticated, i.e. have a valid JSON web token as a Bearer token in the Authorization header of the HTTP request.

```
register()
POST /api/auth/register
Registers a user. Once registered, they will receive an email with a
verification link.

Request body:
email: string
username: string
password: string
```

```
verify()
GET or POST /api/auth/verify?key=[key]
Verifies user's account. The verification key is the "key" request
parameter.

Request body:
N/A
```

```
login()
POST /api/auth/login
Logs in a user. Cannot login unless verified beforehand. Logging in
gives the user a JSON web token with claims inside, such as user id,
username, etc.

Request body:
username: string
password: string
```

```
get_all_users()
GET /api/user
Returns list of all users.

Request body:
N/A
```

```
get_user_by_id()
GET /api/user/<int:user_id>
Returns user with specified id in URL.

Request body:
N/A
```

```
filter_users_by_username()
GET /api/user/filter?username=[username]
Return list of users whose username contains the “username” request
parameter as a substring (case insensitive).

Request body:
N/A
```

Messaging Service

The messaging service is another key component in our application. It lets authenticated users send messages to other users of the system.

There are two models in this microservice: conversations and messages. Each conversation is associated to its creator, a participant (i.e. the user chosen by the creator). Conversations also have a *created_at* attribute. There is a one-to-many relationship between conversations and messages, i.e. each conversation can have multiple messages.

Messages are associated to a conversation by the conversation id. Each message is associated to a user by the *sender_id* attribute, which has to be either of the creator or participant id's in the conversation model. A message obviously has text, which is what the sender decided to send to the receiver, and similar to the conversation entity, has a *created_at* attribute. Once a message is created, the receiver gets a notification, meaning that this microservice communicates with the notification service. We will see that in detail in the next section.

Since all of the functionality in this microservice requires users to be logged in, the endpoints are protected in such a way that the request needs a valid Authorization header. We do this by prepending the *@jwt_required* decorator before each endpoint.

```
get_conversations()
GET /api/conversation
Returns all conversations that the authenticated user is involved in.

Request body:
N/A
```

```
get_messages(conversation_id)
GET /api/conversation/<int:conversation_id>
Returns a given conversation by id, with all of its messages

Request body:
N/A
```

```
get_message_with_limit(conversation_id, message_limit)
GET /api/conversation/<int:conversation_id>/<int:message_limit>
Returns a given conversation by id, with a limit of messages.

Request body:
N/A
```

```
create_conversation()
POST /api/conversation
Creates a conversation, by grabbing the creator_id from the subject
claim in the JSON web token.

Request body:
participant_id: integer
```

```
create_message()
POST /api/message
Creates a message, by grabbing the sender_id from the subject claim in
the JSON web token.

Request body:
conversation_id: integer
receiver_id: integer
text: string
```

Notification Service

The notification service communicates with the messaging service. It's comprised of only a handful of endpoints, and its primary purpose is to listen for incoming requests, for storing, deleting and updating notifications. When a message is sent through the messaging service, a request to the notification microservice made to create a notification for the message receiver. All endpoints in this microservice are protected in such a way that a valid JWT is required in the Authorization header of the request.

Each notification entity has a unique id, a sender id, a receiver id, and a message attached to it. The message can be anything, for example, "You've received a message from username!".

```
get_all_notifications()
GET /api/notifications
Get all of your notifications.

Request body:
N/A
```

```
create_notification()
POST /api/notifications
Create a notification for another user with a message.

Request body:
receiverID: integer
message: string
```

```
delete_notification()
DELETE /api/notifications/<int:notification_id>
Delete notification with specified id.

Request body:
N/A
```

Image Service (deprecated)

We originally had an image service for image-based messages, but due to time constraints, especially during finals, we had to drop it. We figured it was best to do so, since we already have 3 microservices that communicate with each other integrated within the application.

The code was done by Joel. The pull request he made can be viewed here.

CONCLUSION

We learned a lot doing this project. Some of the team members had never worked with Python before so it was a nice challenge to learn a language, while working with one of its most popular frameworks to go along with it.

The biggest takeaway from the project is how the microservices approach is beneficial in decoupling an application's components from one another. We had very little "merge hell" since each feature was in its own service, isolated from the rest. It helped speed up the development process, while minimizing headaches.

Overall, it was a fun experience, and it definitely taught us the importance of starting a project with the future high-scale potential in mind.

This document was made on Google Docs by Giovanni Prattico and Vartan Benohanian.