ReTwitter report - Group 3

Felix Edholm, Patrik Olsson, Omar Sulaiman

Introduction

This report will describe a web application modeled after the social media service Twitter. The web application, named ReTwitter, contains a selection of the features available on regular Twitter, but not all. Some of these features are: posting and deleting tweets, following accounts and seeing a feed of tweets. But it also includes a feature not available on Twitter, displaying a feed of news for the user. Below, this report will describe all use cases present in the application, how to use it, the API specification as well as the underlying architecture and design.

The Github repository for the application is:
https://github.com/Omar-su/ReTwitter-WebApplications-Project/

Use cases

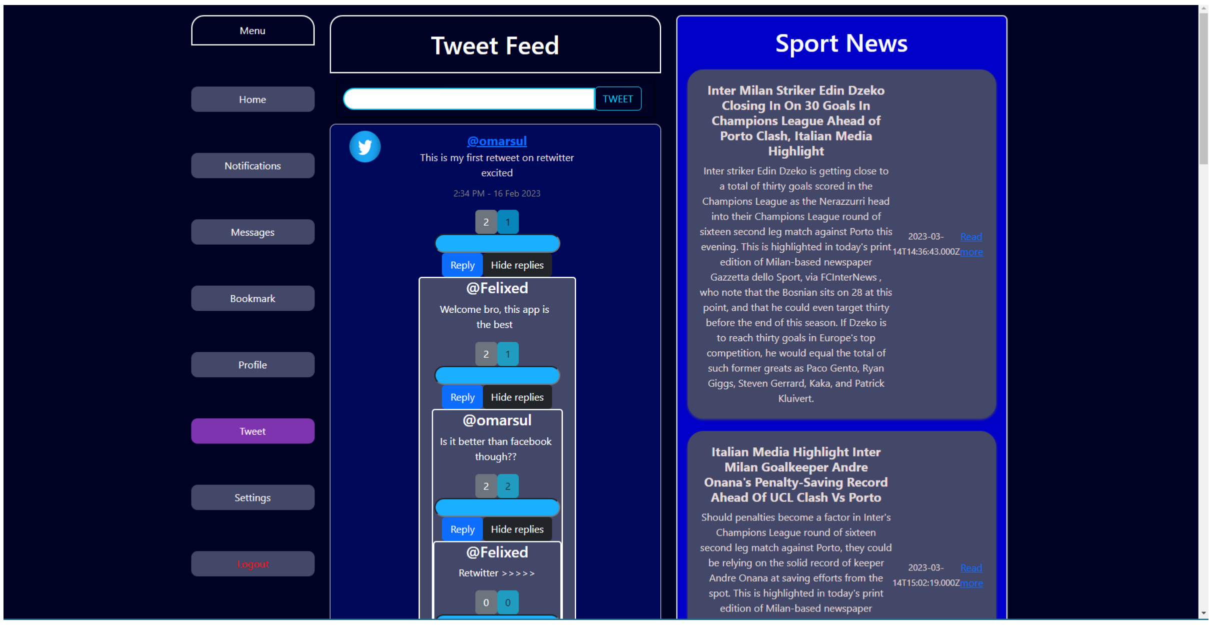
1. The user should be able to post a tweet.
2. The user should be able to see a feed of tweets containing their own tweets and tweets from all accounts followed.
3. The user should be able to like and unlike a tweet
4. The user should be able to like/unlike a reply on a tweet
5. The user should be able to reply to a tweet
6. The user should be able to reply to a reply on a tweet
7. The user should be able to see replies to a tweet
8. The user should be able to delete a tweet
9. The user should be able to create an account with a name and email and bio
10. The user should be able to log in/log out
11. The user should be able to go to a profile and see that users list of tweets
12. The user should be able to follow/unfollow another account
13. The user should be able to navigate through different pages using a sidebar
14. The user should be able to see a feed of news (Using an external api Newscatcher)

User manual

The application can be used by downloading the code from the Github repository, running the command *npm install* in the directory ReTwitter-WebApplications-Project/server/, and *npm install* in the directory ReTwitter-WebApplications-Project/client_mockup/client/. This will install all the necessary dependencies needed to run the application.

After this, the user should run the command *npm run dev* in the ReTwitter-WebApplications-Project/server/ directory, starting up the backend, as well as the

command *npm start* in ReTwitter-WebApplications-Project/client_mockup/client/ which will bring up the running applications login page.
The user can then choose to log in if they have an account or click a button to register an account containing a name, a username, a biography, an email address and a password. The user will then be sent to the login page to use the information to log in. After they have logged in, they are presented with the /home/ page, which is the main page of the application.



@omarsul
This is my first retweet on retwitter excited
2:34 PM - 16 Feb 2023
2 1

@Felixed
Welcome bro, this app is the best
2 1

@omarsul
Is it better than facebook though??
2 2

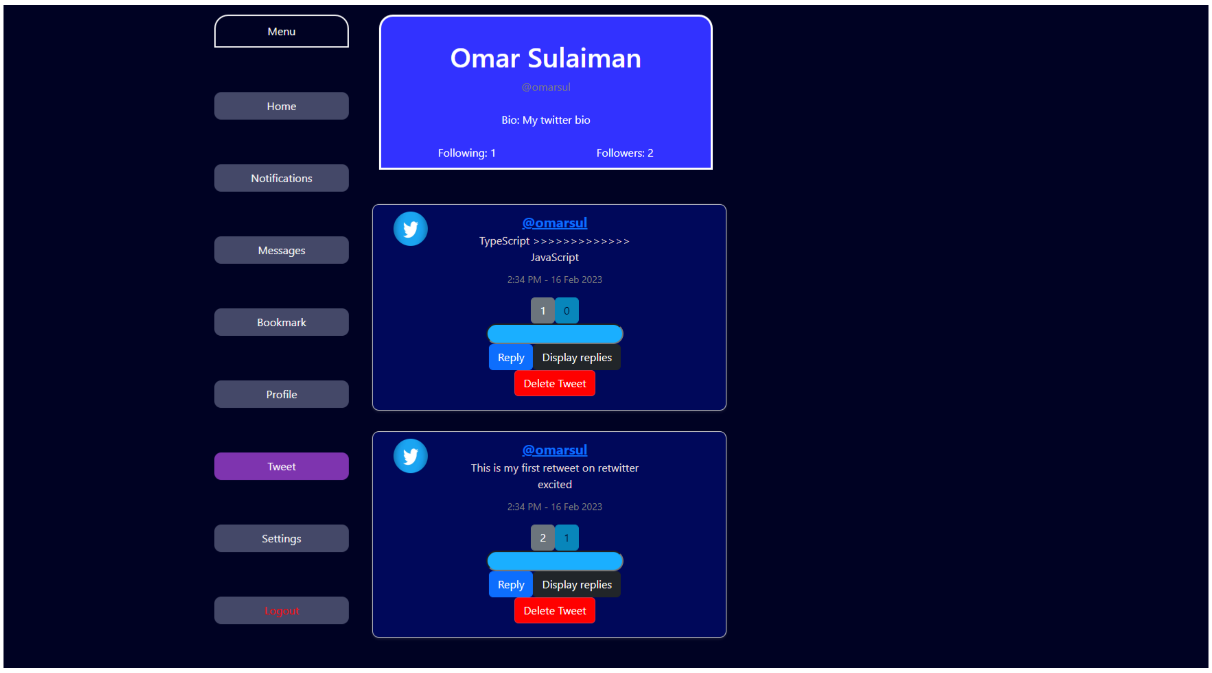
@Felixed
Retwitter >>>>>
0 0

Inter Milan Striker Edin Dzeko Closing In On 30 Goals In Champions League Ahead of Porto Clash, Italian Media Highlight
Inter striker Edin Dzeko is getting close to a total of thirty goals scored in the Champions League as the Nerazzurri head into their Champions League round of sixteen second leg match against Porto this evening. This is highlighted in today's print edition of Milan-based newspaper Gazzetta dello Sport, via FCInterNews , who note that the Bosnian sits on 28 at this point, and that he could even target thirty before the end of this season. If Dzeko is to reach thirty goals in Europe's top competition, he would equal the total of such former greats as Paco Gento, Ryan Giggs, Steven Gerrard, Kaka, and Patrick Kluivert.
2023-03-14T14:36:43.000Z

Italian Media Highlight Inter Milan Goalkeeper Andre Onana's Penalty-Saving Record Ahead Of UCL Clash Vs Porto
Should penalties become a factor in Inter's Champions League round of sixteen second leg match against Porto, they could be relying on the solid record of keeper Andre Onana at saving efforts from the spot. This is highlighted in today's print edition of Milan-based newspaper
2023-03-14T15:02:19.000Z

The main page includes a feed of tweets made by accounts followed by the logged in user as well as the users own tweets. The user can choose to like or unlike a tweet, reply to it, or if it is their own tweet, delete it. The page also includes a feed of news stories as well as a navigation bar that currently can take them to their own profile, as well as log them out. The navigation bar is available on all pages in the application. It also has navigation to notification, messages, bookmarks and settings pages but they are not implemented at this time.

The user can navigate to another user's page by pressing their username on a tweet. It is also possible to go to any available user's profile by using the URL /profile/username. A profile page displays the account information for the given username as well as a list of tweets made by the user. On a profile that is not their own, the user will be able to follow or unfollow the account. This option is of course not available on their own profile page. Below is a screenshot of a profile page.



Omar Sulaiman
@omarsul
Bio: My twitter bio
Following: 1 Followers: 2

@omarsul
TypeScript >>>>>>>>>>>>
JavaScript
2:34 PM - 16 Feb 2023
1 0

@omarsul
This is my first retweet on retwitter
excited
2:34 PM - 16 Feb 2023
2 1

Design

This section will describe the overall architecture of the application, what frameworks and technologies are used as well as give a specification of the API.

Layered architecture

The web applications architecture is mainly made up of a series of layers. The backend layers include a database layer for handling persistence, a service layer for handling the business logic and a router layer responsible for handling API requests and sending a response from the server. The frontend consists of one layer, the presentation layer, responsible for displaying the graphical interface that the user is interacting with.

Having these responsibilities between the layers leads to a clear separation of concerns in the application. This is important as it enables greater extensibility and makes it easier to make modifications. When you make a change in the service layer for instance, it should not greatly affect what happens in the router layer as long as the contract set by the interface for the service layer is not changed. This was apparent when we made the switch from in memory storage to the use of a Mongo database. The service layer had to be changed in order to handle the new way of storing data but having a clear contract for the services made the switch quite easy. Below we will go into a little more detail about each layer.

Database layer

The database layer in the application consists of a MongoDB database and the use of Mongoose to create the connection between the database and the Node.js runtime as well as help with modeling the data using schemas. The database itself consists of three collections. One for storing the information for all created users, one for storing all the tweets tweeted from the application and one for storing all the replies. The data in the database is inserted and retrieved by the service layer using the Mongoose model.

Service layer

The service layer's responsibility is to handle the business logic of the application as well as getting and inserting the data in the database. It currently consists of three service classes, one for handling users, one for tweets and one for replies. Each of these classes implement interfaces for their specific service with a contract of what functionality they should have in order to facilitate the extensibility mentioned above.

The layer handles deleting and creating tweets, creating users, the following and unfollowing between accounts etc. It also makes sure that the database is up to date following all these operations.

Router layer

This layer is where the applications API endpoints are handled. When a call is made to the API of the application, the router layer handles this request and sends back the appropriate response. For example, when a request is sent for creating a user, the router layer checks that the type of the request body is appropriate. If it is not, the router layer sends back an error response with a given error message. But if the information is correct, the router then calls the service layer’s method for creating a user and if that operation is successful, it sends a response confirming that the creation was successful. The requests to the API are sent by the presentation layer.

Presentation layer

This is the layer that the user actually sees and directly interacts with. It is made up of several ReactJS components, each with their own responsibilities of what they display. For instance, the CreateAccount component is what the user interacts with when creating their account. This component displays input boxes for the user to input their information and when the user is done and presses the button to create an account, the component sends a request to the router layer with the given information and awaits a response to see if it was successful or not.

Frameworks, libraries and other technologies

React

React is an open source Javascript frontend framework and library that is used to make interactive web applications. Using React, you break down the user interface into a series of small reusable components where each component is responsible for rendering a specific part of the page. This components-based system can cut down on development time because of the ability to reuse components across different places of the application. In our application for instance, we use the same TweetItem component to display tweets on the home page as well as on a profile page.

React Router

React router is a framework that enables client side routing. This means that the app can be updated by changing URL without sending a new request for a specific document. This gives the app a faster response time.

Node.js

Node.js is the Javascript runtime environment that the application uses. Javascript was originally created to be used for scripting in a web browser, but Node.js enables the use of Javascript to write server-side code without having to use another language such as PHP or Python. It provides a set of built-in modules for handling operations such as networking or HTTP requests.

Express.js

In order to get the most out of Node.js you can use Express.js which is an open-source framework for Node.js which contains features for easier handling of for instance HTTP requests. For instance, our application uses Express for creating the routers used in the router layer as well as handling session information including which user is currently logged in.

MongoDB and mongoose

MongoDB is an open-source NoSQL document-based database. The data in a Mongo database is stored as documents, similar to JSON objects, in a set of collections. Each document can have its own unique structure, and fields within a document can vary in type, size, and content. This flexible schema format allows developers to easily store and manage data without having to define a rigid schema up front.

But if you want an easier way of defining your schema and interacting with your database, you can use the Object Data Modeling library Mongoose, which is what is used in our application. This allows us a more intuitive way to work with the database, defining our schemas, and provides a series of methods for querying and manipulating data in MongoDB.

MUI

MUI is a library with tools that helps to create new features faster. It comes with well designed components like the text field and buttons on the login / register page. It also comes with themes that can be used instantly like dark / light themes. In the end, it makes development faster by offering wellmade components. MUI is used by many large websites like Spotify, Amazon and Netflix.

News Catcher API

Newscatcher API is a powerful news data extraction and aggregation service that empowers developers to access news articles from thousands of sources worldwide. By leveraging advanced algorithms and natural language processing techniques, Newscatcher API efficiently identifies and collects articles on a wide range of topics, from business and technology to sports and entertainment.

Incorporating this API into our project has a specific goal: to provide users with engaging and thought-provoking news topics to discuss and explore. We understand that news can evoke strong emotions, ranging from controversy and complexity to sadness and tragedy. In today's tech-savvy world, we believe that writing and discussing news online is the most effective way to express these emotions and facilitate meaningful discussions.

Security

It should be noted that the application has a lot of points of improvements regarding security. There is currently no form of encryption happening in regards to account information, including passwords. It is also possible to get a full user object with all information by making the right API request. In a production setting this would of course not be acceptable but we chose not to focus on this when creating this application.

API specification

Endpoints per use case

1. POST request to /tweet should tweet a new tweet.
   Request body: {“description”: string}
   Request parameters: None
   Request header: session: {user: UserInterface}.
   Success response:
   Code 201 with a body containing the created tweet.
   Error responses:
   Code 400 with body “Bad POST call to ${req.originalUrl} --- description has type ${typeof (description)}” if request body does not contain description field or is not a string.
   Code 401 with body “Not logged in” if session user is null.
   Code 500 with body of error message for any other unhandled exception.

2. GET request to /tweet/feed should return a body with the list of tweets by the logged in user as well as the tweets of the account followed by the current user.
   Request body: None
   Request parameters: None
   Request header: Session: {user: UserInterface}.
   Success response:
   Code 200 with body containing the list of tweets for the feed.
   Error responses:
   Code 401 with body “Not logged in” if session user is null.
   Code 404 with body “Failed to get feed tweets” if there was an error while retrieving the tweets.
   Code 500 with body of error message for any other unhandled exception.

3. & 4. POST to /tweet/:ID should toggle like status for tweet with id ID for logged on user and increase its number of likes.
    Request body: None
    Request parameters: {“id”: string}
    Request header: Session: {user: UserInterface}.
    Success response:
        Code 200 with body saying “Like status toggled”
    Error responses:
        Code 400 with body “Bad POST call to ${req.originalUrl} --- missing id param” if parameter id is null.
        Code 400 with body “Bad POST call to ${req.originalUrl} --- id number must be a positive integer” if parameter id is negative.
        Code 401 with body “Not logged in” if session user is null.
        Code 404 with body “No tweet with id number ${id}” if the given ID does not exist.
        Code 500 with body of error message for any other unhandled exception.

5. & 6. POST to /reply/:ID with body should tweet a reply to the tweet with id ID.
    Request body: {“author”: string, “description”: string}
    Request parameters: {“id”: string}
    Request header: Session: {user: UserInterface}.
    Success response:
        Code 201 with body saying “Succeeded”
    Error responses:
        Code 400 with body “Bad POST call to ${req.originalUrl} --- missing body data” if type of description is not string.
        Code 400 with body “Id not found” if parameter id is negative.
        Code 401 with body “Not logged in” if session user is null.
        Code 404 with body “Tweet not found” if the given ID does not exist.
        Code 500 with body of error message for any other unhandled exception.

7. GET to reply/feed/replies/:ID should get all replies on a tweet with id ID.
    Request body: none
    Request parameters: {“id”: string}
    Request header: Session: {user: UserInterface}.
    Success response:
        Code 200 with body with all replies on a tweet with the id ID.
    Error responses:
        Code 400 with body “Id not found” if parameter id is negative or if id is null.
        Code 401 with body “Not logged in” if session user is null.
        Code 404 with body "Replies for the tweet not found" if the tweet does not exist.
        Code 500 with body of error message for any other unhandled exception.

8. DELETE to /tweet/:ID should delete the tweet with id ID.
    Request body: None
    Request parameters: {“id”: string}
    Request header: Session: {user: UserInterface}.
    Success response:
        Code 200 with body saying “Tweet was deleted”
    Error responses:
        Code 400 with body “Id not found” if parameter id is null.
        Code 400 with body “Id not found” if parameter id is negative.
        Code 401 with body “Not logged in” if session user is null.
        Code 400 with body “Tweet not found or not owner of tweet” if the given ID does not exist or if the logged in user is not the owner of the tweet.
        Code 500 with body of error message for any other unhandled exception.
9. POST to /user should create a new user with the given information.
    Request body: {“userid”: string, “ownerName”: string, “bio”: string, “email”: string, “password”: string}
    Request parameters: None
    Request header: Session: {user: UserInterface}.
    Success response:
        Code 201 with body saying “User created successfully”
    Error responses:
        Code 400 with body “Bad POST call to ${req.originalUrl} --- userid has type ${typeof (userid)” if type of userid is not string.
        Code 400 with body “Bad POST call to ${req.originalUrl} --- bio has type ${typeof (bio)” if type of bio is not string.
        Code 400 with body “Bad POST call to ${req.originalUrl} --- ownerName has type ${typeof (ownerName)” if type of ownerName is not string.
        Code 400 with body “Bad POST call to ${req.originalUrl} --- email has type ${typeof (email)” if type of email is not string.
        Code 400 with body “Bad POST call to ${req.originalUrl} --- password has type ${typeof (password)” if type of password is not string.
        Code 409 with body “User with userid ${userid} or email ${email} already exists” if there already is a user with the given userid or email.
        Code 500 with body of error message for any other unhandled exception.

10. POST to /user/login should log in a user with the given account information
    Request body: {“userid”: string, “ownerName”: string, “bio”: string, “email”: string, “password”: string}
    Request parameters: None
    Request header: Session: {user: UserInterface}
    Success response:
        Code 200 with body containing the logged in user object

Error responses:

Code 400 with body “Bad POST call to ${req.originalUrl} --- email has type ${typeof (email)” if type of email is not string.

Code 400 with body “Bad POST call to ${req.originalUrl} --- password has type ${typeof (password)” if type of password is not string.

Code 409 with body “Invalid username or password” if no user exists with the given email and password combination.

Code 409 with body “User already logged in” if the user is already logged in.

Code 500 with body of error message for any other unhandled exception.

POST to /user/logout should log out the currently logged in user.

Request body: None

Request parameters: None

Request header: Session: {user: UserInterface}.

Success response:

Code 200 with body saying “User logged out”

Error responses:

Code 409 with body “User already logged out” if session user is null.

Code 500 with body of error message for any other unhandled exception.

11. GET to /profile/username should return a list of tweets made by the user with username and the information for that profile.

Request body: None

Request parameters: {“username”: string}

Request header: None

Success response:

Code 200 with body saying containing the user object with the given ID

Error responses:

Code 400 with body “Bad GET call to ${req.originalUrl} --- missing username param” if parameter username is null.

Code 404 with body “No user exists with username ${userName}” if no user exists with the given username.

Code 500 with body of error message for any other unhandled exception.

POST to /profile/username/follow should add account with username to followers list of currently logged in user as well as add currently logged in user to following list of account with username.

Request body: None

Request parameters: {“username”: string}

Request header: Session: {user: UserInterface}

Success response:

Code 200 with body saying “Added users to follower and following list"

Error responses:

Code 400 with body “Bad POST call to ${req.originalUrl} --- missing account to follow” if parameter username is null.
Code 400 with body “Bad POST call to ${req.originalUrl} --- account to follow has type ${typeof (followee)}” if type of parameter username is not string.
Code 401 with body “Not logged in” if session user is null.
Code 404 with body “No user with username ${followeeUserName} or no user with username ${userFollowingUserName}” if one of or both ids does not correspond to an existing user.
Code 500 with body of error message for any other unhandled exception.

12. POST to /profile/username/unfollow should remove account with username from following list of currently logged in user as well as remove currently logged in user from followers list of account with username.

Request body: None
Request parameters: {“username”: string}
Request header: Session: {user: UserInterface}
Success response:
Code 200 with body saying “Added users to follower and following list"
Error responses:
Code 400 with body “Bad POST call to ${req.originalUrl} --- missing account to unfollow” if parameter username is null.
Code 400 with body “Bad POST call to ${req.originalUrl} --- account to unfollow has type ${typeof (toBeUnfollowedUserName)}” if type of parameter username is not string.
Code 401 with body “Not logged in” if session user is null.
Code 404 with body “No user with username ${toBeUnfollowedUserName} or no user with username ${userUnfollowingUsername}” if one of or both usernames do not correspond to an existing user.
Code 500 with body of error message for any other unhandled exception.

13. Does not require an API call

14. GET to /news/ should return a list of news stories.

Request body: None
Request parameters: None
Request header: Session: {user: UserInterface}
Success response:
Code 200 with body containing list of news
Error responses:
Code 401 with body “not logged in” is session user is null
Code 404 with body "Failed external api call" if newscatcher api does not respond with data back.
Code 500 with body of error message for unhandled exception.

Other endpoints

GET to /user/current_user should return the object for the currently logged in user.

Request body: None
Request parameters: None
Request header: Session: {user: UserInterface}
Success response:
Code 200 with body containing the currently logged in user
Error responses:
Code 401 with body “No user is logged in” if session user is null.
Code 404 with body “Could not find user data” if there was an issue retrieving the data for currently logged in user from the database.
Code 500 with body of error message for unhandled exception.

Responsibilities

Felix: Mainly responsible for the profile page, including follow functionality. Created the first implementations of the DBService classes and connected them to the routers. Wrote the first iterations of the introduction, the user manual and API specification of the report. Wrote most of the Layered architecture section of the report as well as the technologies section. Also wrote unit tests.

Omar: Mainly responsible for handling tweeting and everything that is related to the communication between the users. The user cases which I have done are these :

- The user should be able to post a tweet.
- The user should be able to see a feed of tweets containing their own tweets and tweets from all accounts followed.
- The user should be able to like and unlike a tweet
- The user should be able to reply to a tweet
- The user should be able to delete a tweet
- The user should be able to like/unlike a reply on a tweet
- The user should be able to reply to a reply on a tweet

I worked on these use cases both in the backend and frontend. I did also contact the newscatcher api so that we get more calls to be able to user the api. I did also setup the project and added some css to it.

I also did super session tests for endpoints in reply.ts user.ts and tweet.ts routers.

Patrik: Worked mainly with the login and create account pages. Also worked with page navigation (react router) which also includes automatic redirects if already logged in etc. Also worked with the documentation.