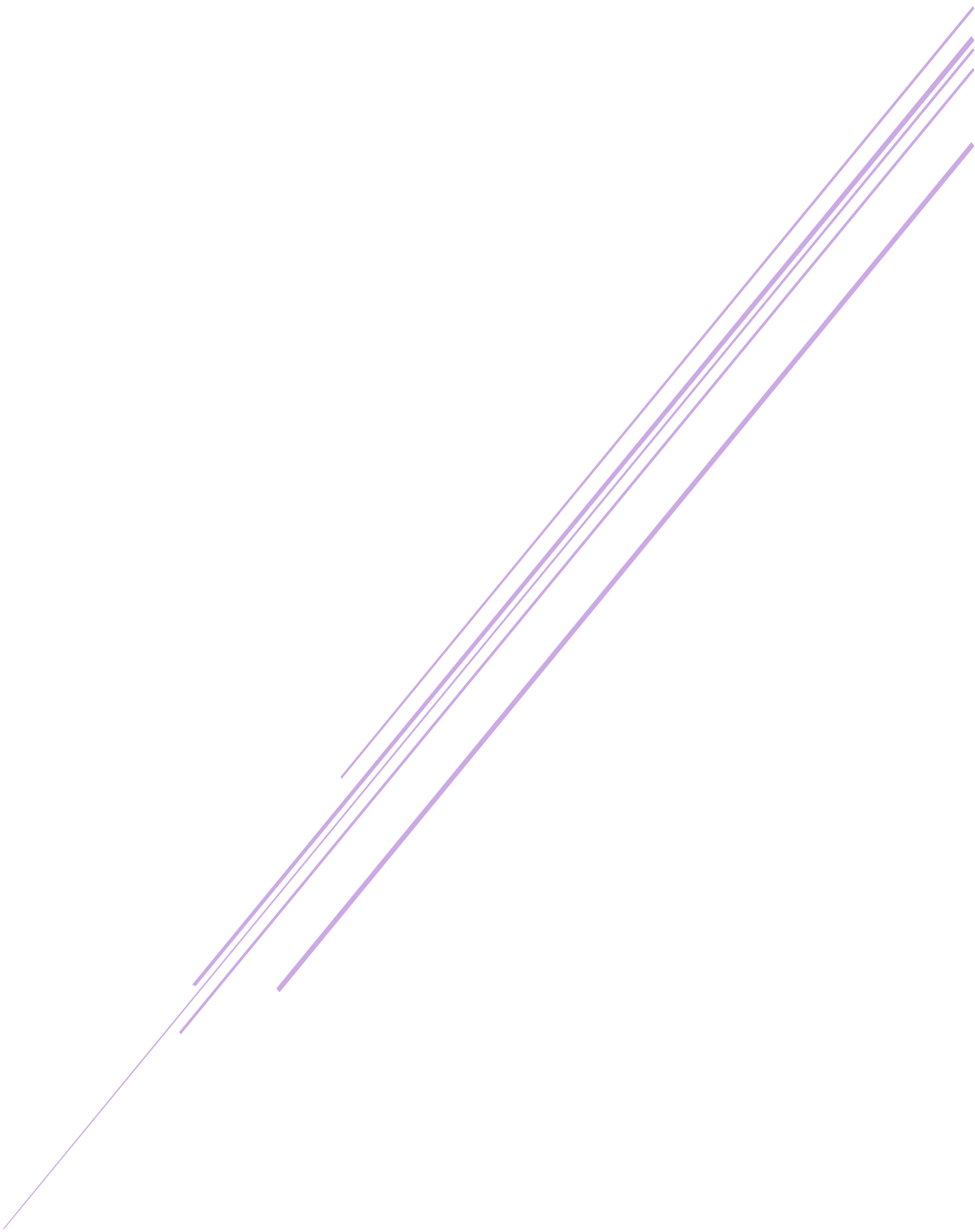
SUPPORT TICKET SYSTEM

write-up by Shay Patel



Source code available @ github.com/0xShay/SupportMe

TABLE OF CONTENTS

ANALYSIS

AIM

My sister works at a dance studio and often has to deal with many enquiries and issues. Usually, people raise their issues via a telephone call or by email.

The development of a web-based support ticket system would provide a central service which users can use to submit issues, and will also allow for assistants to quickly see unclaimed or unanswered tickets and respond to them promptly.

EXISTING ISSUES

Besides using phone calls and email replies, another existing solution for clients to be able to raise concerns and ask questions is a live-chat service, usually embedded into the corner of a website.

After discussions with assistants currently working at the studio, the following issues have arisen:

> "As there are multiple people working in the office at different times, we frequently miss each other's messages and have to fill each other in a lot, taking up lots of precious time."

Phone calls can often lead to miscommunication between both parties, and emails can get easily buried deep into a company's inbox, leaving issues unsolved for large periods of time. A ticket system would allow for both customers and assistants to respond when they're ready to as well as preserving the state of the situation.

> "At the studio we struggle to track inquiries and requests from students and clients and information often gets mixed up between those requesting help."

Allowing for historical ticket messages to be viewable when going to a ticket's page would help alleviate any confusion and miscommunication regarding customers' issues.

EXISTING SOLUTIONS

One existing solution for the studio is hosting and maintaining a live chat service in the form of a website. However, while a live chat service may offer fast response times, the company will need to hire many workers to be able to sit behind a "portal" ready to answer questions

and deal with concerns on a 24/7 basis, and this is simply not financially feasible for the studio.

Another existing solution for the studio is using an already available commercial solution which allows for customers to open support tickets, however many commercial solutions are highly priced and offer services that the studio simply isn't interested in.

For example, Zendesk is a commercially available support ticket system which the studio was initially considering, however it offers a lot of features which the studio is realistically very unlikely to make full use of.

Zendesk offers an "industry-leading" ticket system, with AI-powered automated answers, detailed reporting and analytics, and data and file storage, just to name a few features of the most basic plan.[i] This plan is £39 per agent per month, and considering that the studio won't be using most of the features on the plan, paying a cost this big simply doesn't make financial sense.

The studio isn't interested in AI-automated answers or heavy analytics/data collection – they just want a simple system which allows for customers and assistants to communicate smoothly and solve small-scale problems.

OBJECTIVES

END-USER UTILITY

1. New users must be able to register a new account.
    a. On registration success, they should automatically be logged in.

2. Returning users must be able to log into an existing account.
    a. Once users are logged in, they should be sent an authentication token which they can store in local storage – this will be used as authorization for all future requests.

3. Users should be able to edit their password and profile picture.

4. Users should be able to log out of their account, clearing the authentication token from their local storage and displaying a success message.

5. Customers need to be able to open support tickets.
    a. Once a support ticket has been opened they should be able to access it at a specific link.
    b. They must be able to send messages in the support ticket.

6. Assistants need to be able to read open support tickets.
    a. They must be able to send messages in the support ticket.
    b. They must also be able to mark a support ticket as “closed” or “open”.

STRUCTURE

1. Create SQLite database.
    a. The database should contain any of the tables required and be in the correct format ready to execute SQL statements from the Flask web server running in Python.
2. Create RESTful API with Flask.
    a. GET endpoints will be available so that data and information about tickets, users and messages can be retrieved directly from the database and returned in the request response.
    b. POST endpoints will be available that take JSON inputs which will allow for data regarding tickets, users and messages to be modified, as well as facilitating any other actions that a user may execute, such as (but not limited to):
        i. Logging in
        ii. Signing out
        iii. Creating tickets
        iv. Closing tickets
    c. The RESTful API will interact directly with the local SQLite database in order to read and update stored records.
3. Create front-end endpoints with Flask.
    a. A default set of pages written entirely in HTML, CSS and JS will be accessible via additional endpoints in the Flask server.
    b. These pages will interact directly with the RESTful endpoints, in order to allow users to interact with the support ticket system through a default standardized graphical user interface (GUI).
4. *Scalability*

a. By creating a RESTful API, in the future this will allow me (or other developers) to easily create their own custom interfaces in order to interact with the system. For example, if further down the line a mobile app is to be created that interacts with the system, the app can call the GET and POST endpoints on the RESTful API in order to allow users to interact with the mobile app, allowing for the system to be easily scaled and highly adaptable for the end-user.

DESIGN

WEB SERVER

The server will be split into two parts – a client-side and a server-side. The entire application will work as a client-server model.

The back-end server will be running Flask (Python), and will communicate with a local SQLite database to store data regarding users, tickets and messages.

The Flask server will have POST endpoints and the front-end will be comprised of static HTML/CSS/JS files which communicate via built-in HTML/JavaScript methods (HTML form submission & JS fetch).

Splitting up the application into a client-server model will allow for testing to be done on individual sections in a more organized manner, as well as allowing for the back-end to act as a baseplate for the front-end once completed.

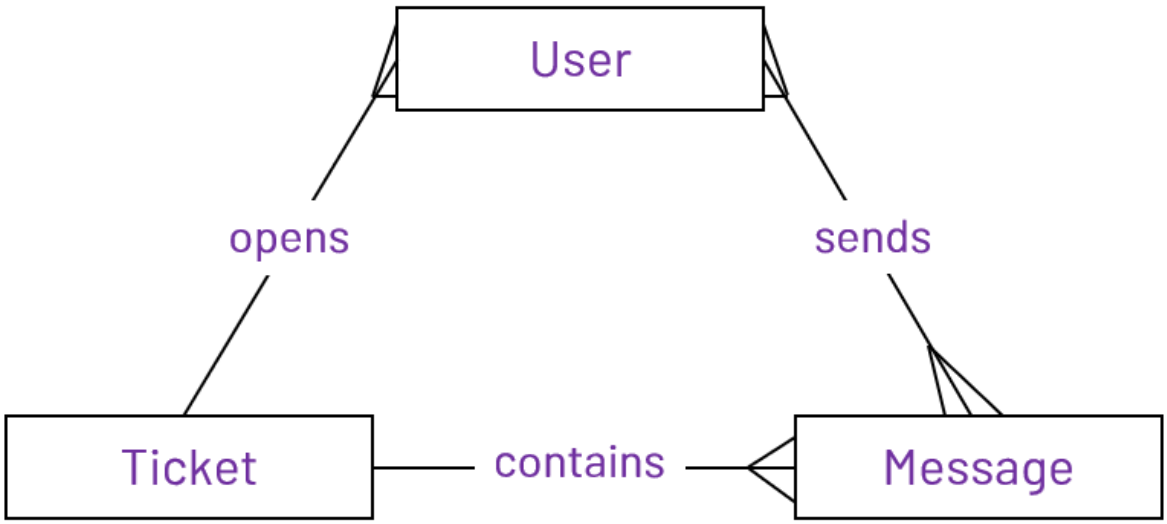
SQL LAYOUT

The Flask back-end will store data in a local SQLite database, which will hold site-related information.

The database will comprise of three main tables:

- The User table will store information about registered users, including both customers and assistants. This will consist of unique user IDs for every user (primary key), as well as key information such as usernames, passwords, and metadata including when the account was created and the type of account.
- The Ticket table will store information about user-opened tickets. They will be identifiable by unique IDs (primary key) and will store information about the customer and assigned assistant (if there is one assigned), as well as metadata including when the ticket was opened and closed.

- The Message table will hold the content and metadata of all messages that have been sent in tickets by users, whether that be by a customer or assistant. Metadata will include the time the message was sent, as well as the ticket status at the time of the message being sent.



"USER" TABLE

User(userID, username, password, createdAt, accountType, profileIcon, email)

Column Name	Data Type	Description	Example Value(s)
userID	INTEGER	A unique numerical identifier for each user, which auto-increments.	1, 2, 3, 4
username	TEXT	A string that can be chosen by the user, which displays as their display name in support tickets.	"John"
password	TEXT	A string that the user will use to authenticate themselves when logging in.	"securepassword123"
createdAt	INTEGER	A UNIX-based timestamp representing when the user account was created.	1678958929
accountType	INTEGER	Either 1 (customer) or 2 (assistant) to represent the user's account level.	1
profileIcon	TEXT	A string representing a relative path location from the web server to the user's profile icon.	"/profile-icons/blue.png"

email	TEXT	A string representing the email address that the user used to create their account.	“john@lavabit.com”

“TICKET” TABLE

Ticket(ticketID, *customerID*, *assistantID*, openedAt, closedAt, title)

Column Name	Data Type	Description	Example Value(s)
ticketID	INTEGER	A unique numerical identifier for each ticket, which auto-increments.	1, 2, 3, 4
customerID	INTEGER	A numerical identifier which refers to the ID of the ticket creator in the “User” table.	1
assistantID	INTEGER	A numerical identifier which refers to the ID of the ticket assistant in the “User” table.	2
openedAt	INTEGER	A UNIX-based timestamp representing when the ticket was opened.	1678958929
closedAt	INTEGER	A UNIX-based timestamp representing when the ticket was closed (or -1 if the ticket is still open).	-1, 1678959041
title	TEXT	A string title which is defined by the customer when the ticket is created.	“Help, the fans aren’t working in Studio 1.”

“MESSAGE” TABLE

Message(messageID, *ticketID*, *authorID*, body, sentAt)

Column Name	Data Type	Description	Example Value(s)
messageID	INTEGER	A unique numerical identifier for each ticket, which auto-increments.	1, 2, 3, 4
ticketID	INTEGER	A numerical identifier which refers to the ID of the ticket in the “Ticket” table.	1
authorID	INTEGER	A numerical identifier which refers to the ID of the message sender in the “User” table.	1

body	TEXT	A string representing the content of the message.	“The fans in the studio don’t seem to be working properly, I think it might be something to do with the switch.”
sentAt	INTEGER	A UNIX-based timestamp representing when the message was sent.	1678959041

ENDPOINTS

These endpoints will act as the backbone, and the entire system’s functionality will be useable with these endpoints alone.

SERVER-SIDE ENDPOINTS

1. POST /register – this endpoint will take in user registration data in JSON and create an account (or return an error)

2. POST /login – this endpoint will take in user credentials (in JSON) and generate and return a web token or return an error

3. GET /get-profile/<int:user_id> – this endpoint will query the database for the user with the given ID and return a JSON response if the user is authenticated to get profile information for that user

4. POST /ticket/new – JSON can be posted here to create a new ticket with a given title and initial message

5. GET /get-ticket/<int:ticket_id> – this endpoint will return a JSON object containing information about the ticket, e.g. when it was opened, when it was closed, the assigned assistant

6. POST /ticket/<int:ticket_id> – this endpoint will be used to append messages to the ticket – the client-side will use this endpoint when users want to send messages or run open/close commands

7. GET /get-open-tickets/<int:user_id> – this endpoint can be used to get a user's open tickets

8. GET /get-closed-tickets/<int:user_id> – this endpoint can be used to get a user's closed tickets

9. GET /get-unclaimed-tickets – this endpoint can be used to get a list of claimable tickets (with no assistant assigned to them)

10. GET /get-messages/<int:ticket_id> – this endpoint will return a JSON response with a list of messages sent in the ticket with the given ID

11. POST /profile – when updating profile information, this endpoint can be used to update any records in the database regarding user information

CLIENT-SIDE ENDPOINTS

Any endpoints below will simply be rendering HTML/CSS/JS and return it to the client – they aren't crucial to backend functionality and simply add a layer over raw API calls from the client's browser to the server.

As a result, server-side and client-side endpoints will be tested separately during the testing phase.

1. GET /home – this will display different information depending on the type of user account that's logged in – if an assistant is logged in a list of unclaimed and claimed tickets will appear, and if a customer is logged in, they will see a list of their open and closed tickets

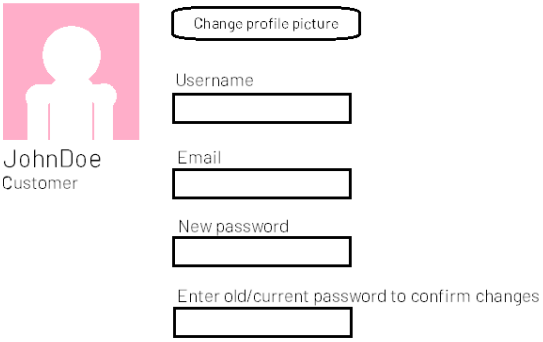
2. GET /register – this will have a form, and on submission it will POST to the /register endpoint, displaying the result of the request on the page

3. GET /login – this will have a form, and on submission it will POST to the /login endpoint, displaying the result of the request on the page

4. GET /ticket/new – this will have a form, and on submission it will POST to the /ticket/new endpoint, displaying the result of the request on the page

5. GET /ticket/<int:ticket_id> – this endpoint will return a rendered template page, displaying the most recent messages in the ticket as well as other key information

6. GET /profile – this page will act as an "edit profile" page where users can update their account information and profile picture



To prevent issues being left unsolved for large periods of time, "inactive tickets" will display on assistants' dashboards, sorted by the time tickets were opened. This will allow old tickets to remain relevant if they are still open, and will hopefully remind assistants that the ticket is still awaiting a response.

DEALING WITH MALFORMED REQUESTS TO FLASK ENDPOINTS

One problem I will face when creating this system is the risk of malicious users sending malformed JSON requests to Flask endpoints. I don't want the server to be negatively impacted by malicious client machines sending requests with invalid JSON in the body of the request.

By wrapping POST requests where JSON input is accepted in the request body in a Python try-except block, I can return an error code if the request is interpreted as malformed by the server.

CONNECTING TO AND INTERACTING WITH THE SQLITE DATABASE

The `sqlite3` PIP library documented extensively at https://docs.python.org/3/library/sqlite3.html will allow me to interact with the local database file to perform queries and retrieve data from the database.

I will need to perform a range of CRUD (create, read, update, delete) functions in response to GET and POST requests sent by clients to the Flask server. A few examples include:

- An INSERT query to create new users on registration
- A SELECT query to read users' login information when verifying credentials
- An UPDATE query to update a ticket's assigned assistant ID when a new assistant claims it

I will also need to ensure that I use prepared statements to prevent any possible SQL injection attacks from occurring, as a result of a malicious user attempting to send in a malicious query which could harm the database and cause data loss or corruption.

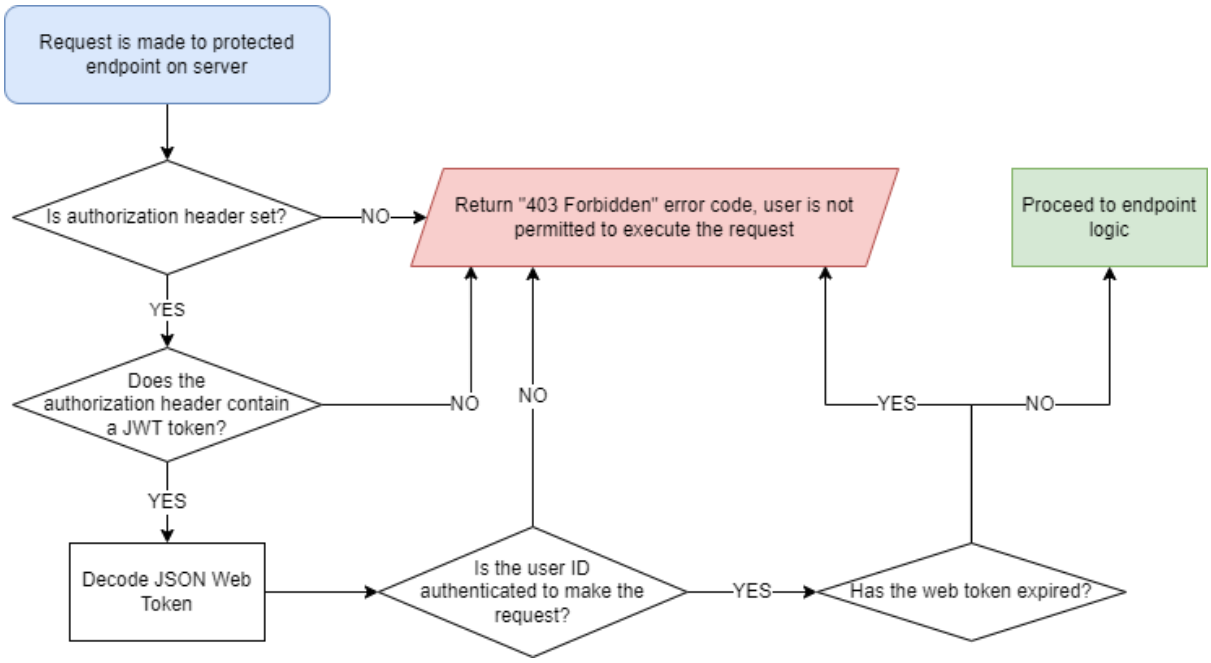
HANDLING USER AUTHENTICATION

To authenticate users, I will be making use of JSON Web Tokens. As outlined in RFC 7519[ii], a JSON Web Token (JWT) is a "compact, URL-safe means of representing claims to be transferred between two parties". I will be able to leverage the use of signed JWTs to encrypt unique tokens for logged in users. As a result, session data will not need to be stored in any substantial capacity on the server-side, and the client-side will be able to use their JWT whenever they make a request to the server. This enables the system to work as a thick-client model, where clients handle their own authentication for the most part, and the server does minimal work to decrypt and verify keys.

On both the client-side and server-side, I will need to be able to quickly encrypt, decrypt and verify web tokens. On the server-side, I will need to predefine a secret phrase, and create a subroutine which takes in inputs regarding the user to generate a token for, returning a web token that the user can store in their local storage on the client-side. When the user then makes requests with their token, the server will also need a subroutine to decrypt the web token to get the user ID of the request sender.

Not only will the server need to be able to use JWTs in this manner, but on the client-side, the only value stored in local storage will be the web token, so if I want to get information about the logged in user's profile, I will need to be able to decode the user's stored token to get their user ID, and then make a request to the server to get the user's profile information.

For the server-side, I plan to use the PyJWT module (outlined at https://pyjwt.readthedocs.io/en/latest/) to encode and decode access tokens when requests are made to protected endpoints.



On the client-side, I will need to decode JWTs in JavaScript – I intend to make use of the open-source JWT decode tool at https://github.com/auth0/jwt-decode to decrypt tokens and get user data where necessary in the front-end. For example, I will need to decrypt the user's stored web token when I want to get the user ID to get profile information about the logged in user.

SERVING DYNAMIC CONTENT ON THE HOMEPAGE

As briefly stated above, the homepage should show logged in users a list of their open and closed tickets. Most recently opened tickets will be displayed at the bottom.

When assistants log in, alongside open and closed tickets, they will also be able to see a list of unclaimed tickets that they can claim. The earliest-most opened tickets will be shown at the top, to create a "priority queue" of tickets. It was a large concern of my client that many customers' concerns were being left abandoned after large periods of time – this can be avoided by ordering unclaimed tickets by the time they were opened.

PROMOTING CUSTOMER ACCOUNTS TO ASSISTANT ACCOUNTS AND VICE-VERSA

I don’t intend to add functionality to the site to change account types – instead I’d like to leave that up to the server administrator. For the studio, a simple interface to change a user’s account type using their user ID will suffice.

I will create a simple Python script, which takes in two inputs (user ID and account type), opens a connection to the local database, performs the single UPDATE operation and prints out a response. To change an account’s type, the server administrator will simply have to run a Python script and enter some values. If they want to update a customer account to an assistant account, they just need the ID of the customer account, and direct access to the server on which the database file is located.

TECHNICAL BUILD

DATABASE INITIALIZATION

As previously stated, the database of choice is a local SQLite database, stored as a file on the same machine as the Flask server which will be delivering content.

I have created an empty *data.db* file in the project's root directory using the DB Browser for SQLite. This file will store all data regarding users, tickets and messages. The table schemas were created with the DB Browser for SQLite (DB4S) software and the table creation commands are shown below:

```
CREATE TABLE "User" (
	"userID" INTEGER,
	"username" TEXT,
	"password" TEXT,
	"createdAt" INTEGER,
	"accountType" INTEGER,
	"profileIcon" TEXT,
	"email" TEXT,
	PRIMARY KEY("userID" AUTOINCREMENT)
);
```

The "User" table will store all the information regarding registered users and assistants. It will store usernames, passwords, as well as when the user account was created, the type of account and the email address that the user registered with.

```
CREATE TABLE "Ticket" (
	"ticketID" INTEGER,
	"customerID" INTEGER,
	"assistantID" INTEGER,
	"openedAt" INTEGER,
	"closedAt" INTEGER,
	"title" TEXT,
	FOREIGN KEY("customerID") REFERENCES "User"("userID"),
	FOREIGN KEY("assistantID") REFERENCES "User"("userID"),
	PRIMARY KEY("ticketID" AUTOINCREMENT)
);
```

The "Ticket" table will store metadata and key information about open and closed tickets. It will store information about who the customer is (customerID) and the assistant assigned to the ticket (assistantID). It will also store when the ticket was opened (and closed if it has been closed).

```
CREATE TABLE "Message" (
	"messageID" INTEGER,
	"ticketID" INTEGER,
	"authorID" INTEGER,
	"body" TEXT,
	"sentAt" INTEGER,
	FOREIGN KEY("ticketID") REFERENCES "Ticket"("ticketID"),
	FOREIGN KEY("authorID") REFERENCES "User"("userID"),
	PRIMARY KEY("messageID" AUTOINCREMENT)
);
```

The “Message” table will store information about messages that customers and assistants send in tickets. They will link to users by authorID foreign key and will be assigned to a ticket via the ticketID foreign key. When the message was sent will also be stored.

```
INSERT INTO "User"
("userID", "username", "password", "createdAt", "accountType",
"profileIcon", "email")
VALUES
(0, "System", "securepassword", 0, 2, "/profile-
icons/admin.png", "Ed_Snowden@lavabit.com");
```

A “System” account will also be pre-created – this account will be responsible for sending system messages in tickets, notifying users when a ticket’s status has changed or when an assistant has claimed a ticket.

CONSTANTS

At the top of the main server.py file, by defining a set of constants, I can maintain referential integrity throughout my program, and instead of having integers spontaneously spread out in code, I can refer to predefined constants.

```
ACCOUNT_TYPE_CUSTOMER = 1
ACCOUNT_TYPE_ASSISTANT = 2

TICKET_STATUS_OPEN = 1
TICKET_STATUS_CLOSED = 2

SYSTEM_USER_ID = 0
```

These constants are also defined in a constants.js static file, accessible from the client-side:

```
const ACCOUNT_TYPE_CUSTOMER = 1
```

```
const ACCOUNT_TYPE_ASSISTANT = 2

const TICKET_STATUS_OPEN = 1
const TICKET_STATUS_CLOSED = 2

const SYSTEM_USER_ID = 0
```

DEALING WITH MALFORMED REQUESTS TO FLASK ENDPOINTS

In order to protect the web server from being susceptible to malformed HTTP requests on the post endpoints, I can wrap any POST request logic in a try-except statement – this means if any JSON is invalid and an error is thrown, the code in the "except" part of the statement will execute, returning an error code in the response and notifying the user that the request was invalid.

```
elif request.method == 'POST':
    try:
        # REQUEST LOGIC GOES HERE
    except:
        return ({ "error": "Server failed to parse the request." }, 400)
```

CONNECTING TO AND INTERACTING WITH THE SQLITE DATABASE

I will make use of the `sqlite3` library as planned, as it is well-documented and updated regularly. It allows me to connect to the database and execute queries easily using cursor objects.

```
import sqlite3
```

To connect to the database, I'm making use of Python's built-in `with` keyboard to maintain an open connection with the local database in different subroutines:

```
with sqlite3.connect("data.db") as connection:
    cursor = connection.cursor()
```

Any user-inputted values need to be handled securely to prevent SQL injection attacks from occurring. This can be achieved by using prepared statements when executing queries:

```
cursor.execute(
    "INSERT INTO User (username, password, createdAt, accountType,
profileIcon, email) VALUES (?, ?, ?, ?, ?, ?)",
    [ username, password, int(time.time()), ACCOUNT_TYPE_CUSTOMER,
"/profile-icons/blue.png", email ]
)
```

I will need to execute queries when creating new user accounts, validating login information, creating new tickets, getting ticket information, creating new messages, and getting a list of messages from the database.

When using the SELECT keyword in queries, I can use `cursor.fetchall()` to get all of the returned rows. For example, I used a SELECT statement to validate a user's login credentials as seen below:

```
def login_user(username, password):

    with sqlite3.connect("data.db") as connection:
        cursor = connection.cursor()

        cursor.execute("SELECT userID, password, accountType FROM User
WHERE username = ?", [ username ])
        user_list = cursor.fetchall()
        if len(user_list) == 0:
            return (False, "A user with the supplied username and password
was not found.")

        _user_id, _password, _account_type = user_list[0]

        if password == _password:
            access_token = generate_access_token(_user_id, _account_type)
            return (True, access_token)
        else:
            return (False, "A user with the supplied username and password
was not found.")
```

I begin by opening a connection to the database, and then establish a local cursor variable. After executing the SELECT statement (searching by the username that is the first parameter of the function), I make use of the cursor's `fetchall` method to get a list of returned rows.

I can check if there is a user registered with that username or not by simply checking the number of records that were returned – if 0 records were returned then the query was unable to find any users with that username.

Further below, if a row was returned from the query it means that a user with the supplied username exists in the database. An if statement is used to check if the passwords match, and if they do I generate and return an access token with the `generate_access_token` method explained in more detail below. However, if the passwords don't match I return an error in the same fashion as when the usernames didn't match.

By returning the same error message in both cases, it eliminates the possibility of a user enumeration vulnerability[iii] – an attacker cannot determine if a username is present or not

in the database by simply brute-forcing different usernames. This reduces the future risk of social engineering attacks occurring amongst others.

HANDLING USER AUTHENTICATION

When encrypting JSON Web Tokens, I will need a server-side secret key. To generate a relatively insecure[iv] key I used list comprehension and Python's `join` function to produce 64 randomly selected hex characters:

```
"".join([random.choice(["0", "1", "2", "3", "4", "5", "6", "7", "8", "9",
"A", "B", "C", "D", "E", "F"]) for i in range(64)])
```

When used in production, a more secure private key can be generated through cryptographically secure methods and the default one can be replaced at the top of the file.

On the server-side, JWTs will need to be generated whenever a user:

a) registers a new account, or
b) logs into an existing account.

When a user registers with a new account, on the server's receipt of a POST request to the /register endpoint, the server will proceed to run `create_user`, which attempts to insert the new user into the database. If successful, the `generate_access_token` method is then called, and the access token is returned to the request sender.

When a user logs in to an account, on the server's receipt of a POST request to the /login endpoint, the server first checks that the user's credentials match the credentials stored in the database, and then proceeds to invoke the `generate_access_token` method, returning the result in the response.

The PyJWT library allows me to encode and decode JSON Web Tokens on the Python server with ease.

```
import jwt

def generate_access_token(user_id, account_type):
    access_token = jwt.encode({
        "user_id": user_id,
        "account_type": account_type,
        "exp": int(time.time()) + 86400
    }, JWT_SECRET_KEY, algorithm="HS256")
    return access_token
```

I start by importing PyJWT (`import jwt`) – this library gives me access to two methods that will come in useful when I generate and verify access tokens: `jwt.encode` and `jwt.decode`.

The `generate_access_token` method is shown above – the subroutine has two parameters: a user id and the user's account type. These two properties, along with an expiry date are encoded using PyJWT's `encode` method and signed using the secret key defined at the top of the file.

The "exp" property of the token allows for the user's requests to become invalidated after 86400 seconds (24 hours), forcing them to revalidate themselves. This allows for increased user security and can minimise the risk of confidential messages being leaked.

On the client-side, on every page, the `getLoggedInUser()` JavaScript method is called, which is defined in the userTools.js script, which is loaded into every page.

```
const getDecodedAccessToken = () => {
    return localStorage.getItem("access_token") != null ?
jwt_decode(localStorage.getItem("access_token")) : null;
}

function getLoggedInUser() {
    let d_at = getDecodedAccessToken();
    if (d_at == null) return null;
    if (d_at["exp"] < (Date.now() / 1000)) {
        localStorage.removeItem("access_token");
        return null
    };
    return d_at;
}
```

In the `getDecodedAccessToken` function, I make use of a `jwt_decode` method – this function is defined in /src/jwt-decode.js, and this open-source tool has been obtained from https://github.com/auth0/jwt-decode.

In the `getLoggedInUser` function shown above, the access token is decoded, and if it has expired, it is removed from local storage, and the function returns "null". On pages where the user must be logged in, the line below is present:

```
if (getLoggedInUser() == null) window.location.href = "/login";
```

If the user is not logged in or their access token is invalidated, the user will be redirected to the /login page.

```
async function register() {

    let username_input = document.getElementById("username_input");
    let email_input = document.getElementById("email_input");
    let password_input = document.getElementById("password_input");
```

```
    let passwordc_input = document.getElementById("passwordc_input");

    if (password_input.value != passwordc_input.value) return
alert("Passwords do not match.")

    const res = await (await fetch('/register', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json'
        },
        redirect: 'manual',
        body: JSON.stringify({
            username: username_input.value,
            email: email_input.value,
            password: password_input.value
        })
    })).json();

    if (res["error"] != undefined) {
        alert(res["error"]);
    } else {
        localStorage.setItem("access_token", res["access_token"]);
        window.location.href = "/home";
    };

}

async function login() {

    let username_input = document.getElementById("username_input");
    let password_input = document.getElementById("password_input");

    const res = await (await fetch('/login', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json'
        },
        redirect: 'manual',
        body: JSON.stringify({
            username: username_input.value,
            password: password_input.value
        })
    })).json();

    if (res["error"] != undefined) {
        alert(res["error"]);
    } else {
        localStorage.setItem("access_token", res["access_token"]);
        window.location.href = "/home";
    };

}
```

The `register()` and `login()` functions are called by buttons in HTML, and they make use of local storage to store access tokens that are returned by the server.

When sending requests to the server from the client-side in JavaScript, I will use `fetch` to pass the stored access token into the Authorization header field:

```
const res = await (await fetch('/ticket/' + ticketID, {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json',
        'Authorization': 'Bearer ' + localStorage.getItem("access_token")
    },
    body: JSON.stringify({
        message: message_input.value
    })
})).json();
```

This access token can then be picked up server-side and verified:

```
def verify_access_token(access_token):
    try:
        d_at = jwt.decode(access_token.split(" ")[1], JWT_SECRET_KEY,
algorithms=["HS256"])
        if d_at["exp"] < time.time():
            return None
        else:
            return d_at
    except:
        return None

@App.route('/get-profile/<int:user_id>', methods=['GET'])
def get_profile_by_user_id_req(user_id):
    if request.method == 'GET':

        auth_user =
verify_access_token(request.headers.get("Authorization"))
        if auth_user == None:
            return ({ "error": "User is not authenticated to make this
request." }, 403)
```

As shown above, the `verify_access_token` method uses PyJWT's `decode` method to get a JSON representation of the encoded value. If the access token is malformed, since it's in a try-except block, the function simply returns `None`. However, if the token is not malformed, the expiry date property ("exp") of the token is checked, and if the token is expired the method returns `None`. If the token is valid, the JSON representation of the token is returned, containing the user's user ID and the expiry date of the token.

If the user is not authenticated for the request, a (Forbidden) 403 error will be returned to the client.

SERVING DYNAMIC CONTENT ON THE HOMEPAGE

If the user is not logged in to an account, they will be redirected to the login page.

This can be easily achieved by calling the getLoggedInUser JavaScript function to check if there is a valid access token, and if there isn't, the page can redirect to the login page:

```
if (getLoggedInUser() == null) window.location.href = "/login";
```

If the user is logged into an account, their account type (decoded from their access token) will be used to decide what content to serve.

Both customers and assistants should be able to see open and closed tickets that they are involved in, however on top of that, assistants should be able to see claimable tickets.

```
@App.route('/get-open-tickets/<int:user_id>', methods=['GET'])
@App.route('/get-closed-tickets/<int:user_id>', methods=['GET'])
@App.route('/get-unclaimed-tickets', methods=['GET'])
```

I made these three routes so that customers and assistants can both request open and closed tickets, where they are either the customerID, or assistantID of that ticket:

```
SELECT ticketID, customerID, assistantID, openedAt, closedAt, title FROM
Ticket WHERE (customerID = ? OR assistantID = ?) AND closedAt = -1 ORDER BY
openedAt ASC;

SELECT ticketID, customerID, assistantID, openedAt, closedAt, title FROM
Ticket WHERE (customerID = ? OR assistantID = ?) AND closedAt != -1 ORDER
BY closedAt DESC;
```

Whilst assistants can also get a list of unclaimed tickets:

```
SELECT ticketID, customerID, assistantID, openedAt, closedAt, title FROM
Ticket WHERE assistantID = -1 ORDER BY openedAt ASC;
```

It was a large concern of my client that many customers' concerns were being left abandoned after large periods of time. By ordering unclaimed tickets by the time they were opened, this allows for assistants to see which tickets were opened first, and prioritize resolving older issues before moving on to solving newer ones.

PROMOTING CUSTOMER ACCOUNTS TO ASSISTANT ACCOUNTS AND VICE-VERSA

A file can be found in the /tools directory, named “changeAccountType.py” – the server admin can run this file, inputting a user ID and the level that they want to set the user account at. This will update the database, or return any relevant errors to the console:

```
import sqlite3

user_id = int(input("Enter the user ID of the account: "))
account_type = int(input("Enter an account type to set (1 = customer, 2 = assistant): "))

with sqlite3.connect("../data.db") as connection:

    try:
        cursor = connection.cursor()
        cursor.execute("UPDATE User SET accountType = ? WHERE userID = ?;", [ account_type, user_id ])
        if cursor.rowcount == 0:
            print("User with given ID was not found in the database. Terminating.")
        else:
            print(f"User with ID {user_id}'s account type has been successfully updated to {account_type}.")

    except sqlite3.Error as error:
        print(error)
```

The server admin can therefore notify any assistants that they should register a customer account, and then their accounts can be manually promoted to assistant accounts so that they can claim tickets. Customers can open tickets; however assistant accounts are unable to open tickets. If an assistant wants support, they will have to create a new customer account and open a ticket through that.

For my client, this is not an issue, since all assistants will be using email accounts on the studio’s business domain for their assistant accounts, and so if the assistants want to request support, they will be encouraged by their employer to create a separate personal account with a personal email address.

TESTING

Testing will be split into two sections – server-side endpoints will be tested with inputs for expected outputs, and client-side endpoints will be tested separately, and tests will be displayed on a video.

TESTING SERVER-SIDE ENDPOINTS (BACK-END)

Each server-side endpoint will be tested with valid, and erroneous data, and the input, expected output and result of each test will be displayed in the table below.

POST /register

Expectation	The details supplied in this request should all be valid, and an access token should be returned.
Request Headers	

Request Body	<pre>{
	    "username": "John",
	    "email": "johndoe@website.net",
	    "password": "securepassword"
	}</pre>

Request Response	<pre>{ "access_token": "eyJh...XLfk" }</pre>
Test Result	✓ - The server returned an access token which can be decoded to get the user ID, account type, and expiry date of the token.

Expectation	The password provided here is less than the minimum number of characters (8), and so an error should be returned.
Request Headers	

Request Body	<pre>{
	    "username": "Jane",
	    "email": "janedoe@website.net",
	    "password": "2short"
	}</pre>

Request Response	<pre>{ "error": "Password must be at least 8 characters." }</pre>
Test Result	✓ - The account was not created and the server responded with an error.

POST /login

Expectation	The details supplied in this request should all be valid, and an access token should be returned.
Request Headers	

Request Body	{
	"username": "John",
	"password": "securepassword"
	}

Request Response	{ "access_token": "eyJh…208M" }
Test Result	✓ - The server returned an access token which can be decoded to get the user ID, account type, and expiry date of the token.

Expectation	A user with the supplied username and password does not exist, and so this request should return an error.
Request Headers	

Request Body	{
	"username": "Tommy",
	"password": "tommyspassword1234"
	}

Request Response	{ "error": "A user with the supplied username and password was not found." }
Test Result	✓ - The username-password pair was not found in the database and the server responded with a suitable error.

GET /get-profile/13

Expectation	The user ID provided (13) is the user ID of the newly created "John" account, and the access token is John's token (who is authorized to make this request), so John's profile information should be returned.

Request Headers	{
	"Authorization": "Bearer eyJh...208M"
	}

Request Body	
Request Response	{ "user": {

	<pre>"account_type": 1,
	"created_at": 1676902471,
	"profile_icon": "/profile-icons/red.png",
	"user_id": 13,
	"username": "John"
	}
	}</pre>

Test Result	✓ - As expected, information about the user's profile was returned.

Expectation	Since no access token has been provided, and this request requires authentication, the server should respond with an authentication error.
Request Headers	
Request Body	
Request Response	<pre>{ "error": "User is not authenticated to make this request." }</pre>
Test Result	✓ - The server returned an authentication error as intended.

POST /ticket/new

Expectation	The details supplied to open the ticket match the validation requirements, and so this request should successfully create a ticket and return the ticket ID.

Request Headers	<pre>{
	"Authorization": "Bearer eyJh...208M"
	}</pre>

Request Body	<pre>{
	"ticket_title": "The toilet roll is finished",
	"message": "The toilet roll… could get it refilled?"
	}</pre>

Request Response	<pre>{ "message_id": 74, "ticket_id": 9 }</pre>
Test Result	✓ - As intended, a new ticket was created and the ID was returned in the response body.

Expectation	In this request, a ticket title was not provided, only a message. The server should respond with an error since the user has not supplied all the required information to open a ticket.

Request Headers	<pre>{
	"Authorization": "Bearer eyJh...208M"
	}</pre>

Request Body	{
	"message": "The toilet roll… could get it refilled?"
	}

Request Response	{ "error": "The request failed." }
Test Result	✓ - The response returned with error code 400 (BAD REQUEST) since the information provided was insufficient to open a ticket.

GET /get-ticket/9

Expectation	The response should contain information about the newly created ticket with ID 9.

Request Headers	{
	"Authorization": "Bearer eyJh...208M"
	}

Request Body	

Request Response	{
	"ticket_data": {
	"assistant_id": -1,
	"closed_at": -1,
	"customer_id": 13,
	"opened_at": 1676904963,
	"ticket_id": 9,
	"title": "The toilet roll is finished"
	}
	}

Test Result	✓ - Information about the ticket was returned as expected.

Expectation	The token provided is for a customer account who is not involved in any way with ticket 9 - they should be unauthorized to get information about this ticket.

Request Headers	{
	"Authorization": "Bearer eyJh...42G4"
	}

Request Body	
Request Response	{ "error": "User is not authenticated to make this request." }
Test Result	✓ - The server returned an authentication error as intended.

POST /ticket/9

Expectation	Since a valid message is being sent to the ticket by the customer who opened it, the request should be successful.
Request Headers	`{ "Authorization": "Bearer eyJh...208M" }`
Request Body	`{ "message": "I believe it ran out last night." }`
Request Response	`{ "message_id": 75, "ticket_id": 9 }`
Test Result	✓ - A new message was created in the database with ID 75, and no errors were returned.

Expectation	Message length should be at least 8 characters - sending this short message should return an error.
Request Headers	`{ "Authorization": "Bearer eyJh...208M" }`
Request Body	`{ "message": "help" }`
Request Response	`{ "error": "Message length must be at least 8 characters - be descriptive." }`
Test Result	✓ - As expected, an error was returned notifying the user of their poor input.

GET /get-open-tickets/13

Expectation	A valid authentication token is being used to get a list of open tickets for the logged in customer - the request should respond with a list of open tickets for the user.
Request Headers	`{ "Authorization": "Bearer eyJh...208M" }`
Request Body	
Request Response	`{ "tickets": [ { "assistant_id": -1, "closed_at": -1,`

	<pre>"customer_id": 13,
	"opened_at": 1676904963,
	"ticket_id": 9,
	"title": "The toilet roll is finished"
	}
	] }</pre>

Test Result	✓ - The server responded with a list of tickets that are marked as OPEN that involve the authorized user.

Expectation	With another user's authentication token, the request should fail, as only the user themselves can see their own tickets.

Request Headers	<pre>{
	"Authorization": "Bearer eyJh...42G4"
	}</pre>

Request Body	
Request Response	<pre>{ "error": "User is not authenticated to make this request." }</pre>
Test Result	✓ - Since the user is not authorized to get user with ID 13's tickets, the server returned an authentication error.

GET /get-closed-tickets/13

Expectation	Using the authentication token of an assistant account, this request should return a list of tickets which have no assistant assigned to them.

Request Headers	<pre>{
	"Authorization": "Bearer eyJh...208M"
	}</pre>

Request Body	
Request Response	<pre>{ "tickets": [] }</pre>
Test Result	✓ - As expected, the server returned an empty list, since none of the user's tickets are closed.

Expectation	With another user's authentication token, the request should fail, as only the user themselves can see their own tickets.

Request Headers	<pre>{
	"Authorization": "Bearer eyJh...42G4"
	}</pre>

Request Body	
Request Response	`{ "error": "User is not authenticated to make this request." }`
Test Result	✓ - Since the user is not authorized to get user with ID 13's tickets, the server returned an authentication error.

GET /get-unclaimed-tickets/13

Expectation	Using the authentication token of an assistant account, this request should return a list of tickets which have no assistant assigned to them.
Request Headers	`{ "Authorization": "Bearer eyJh...WWA4" }`
Request Body	
Request Response	`{ "tickets": [ { "assistant_id": -1, "closed_at": -1, "customer_id": 13, "opened_at": 1676904963, "ticket_id": 9, "title": "The toilet roll is finished" } ] }`
Test Result	✓ - The server responded with a list of tickets that are marked as UNCLAIMED as the user is an assistant who is authorized to make this request.

Expectation	If the authentication token of a customer account is used, an authentication error should be returned, as this request is exclusively for assistant accounts.
Request Headers	`{ "Authorization": "Bearer eyJh...208M" }`
Request Body	
Request Response	`{ "error": "User is not authenticated to make this request." }`
Test Result	✓ - Since the user is a customer account, they are not authorized to get a list of unclaimed tickets, so the server returned an authentication error.

GET /get-messages/9

Expectation	This endpoint should return a list of messages that have been sent in the ticket. It should return the two messages that have been sent by the customer.
Request Headers	`{ "Authorization": "Bearer eyJh...208M" }`
Request Body	
Request Response	`{ "message_list": [ { "author_id": 13, "body": "I believe it ran out last night.", "message_id": 75, "sent_at": 1676911540 }, { "author_id": 13, "body": "The toilet roll… could get it refilled?", "message_id": 74, "sent_at": 1676904963 } ], "ticket_id": 9 }`
Test Result	✓ - A list of messages sent in the ticket were returned to the user, along with message metadata.

Expectation	If an authentication token is not set in the headers, we cannot be certain that the requester is authorized to view the messages so an error should be returned.
Request Headers	`{ "Authorization": "Bearer eyJh...208M" }`
Request Body	
Request Response	`{ "error": "User is not authenticated to make this request." }`
Test Result	✓ - Since no authentication token was set, an authentication error was returned.

POST /profile

Expectation	The user's profile should be updated, and the endpoint should return a success message.
Request Headers	

	<pre>{
	    "Authorization": "Bearer eyJh...208M"
	}</pre>

Request Body	<pre>{
	    "new_password": "securepassword2",
	    "old_password": "securepassword",
	    "profile_icon": "/profile-icons/purple.png"
	}</pre>

Request Response	<pre>{ "msg": "Profile has been updated." }</pre>
Test Result	✓ - The user's profile was successfully updated since a valid old password was provided.

Expectation	Even though the authentication token is valid, since the wrong old password was provided, the request should throw an authentication error and not make any changes to the user's profile.

Request Headers	<pre>{
	    "Authorization": "Bearer eyJh...208M"
	}</pre>

Request Body	<pre>{
	    "new_password": "randompassword",
	    "old_password": "nottoosure",
	    "profile_icon": "/profile-icons/purple.png"
	}</pre>

Request Response	<pre>{ "error": "Incorrect password was provided." }</pre>
Test Result	✓ - The server responded with a 403 code, since an incorrect old password was provided.

Below I test the try-catch statements I implemented, by sending malformed body parameters to POST requests:

POST /register

Expectation	Since an invalid JSON body was provided in the body, the server should respond with an error.
Request Headers	
Request Body	<pre>[]</pre>
Request Response	<pre>{ "error": "Server failed to parse the request." }</pre>
Test Result	✓ - The server responded with a 400 BAD REQUEST error as expected.

POST /login

Expectation	Since an invalid JSON body was provided in the body, the server should respond with an error.
Request Headers	
Request Body	`"Hello world!"`
Request Response	`{ "error": "Server failed to parse the request." }`
Test Result	✓ - The server responded with a 400 BAD REQUEST error as expected.

TESTING CLIENT-SIDE ENDPOINTS (FRONT-END)

FRONT-END TEST PLAN

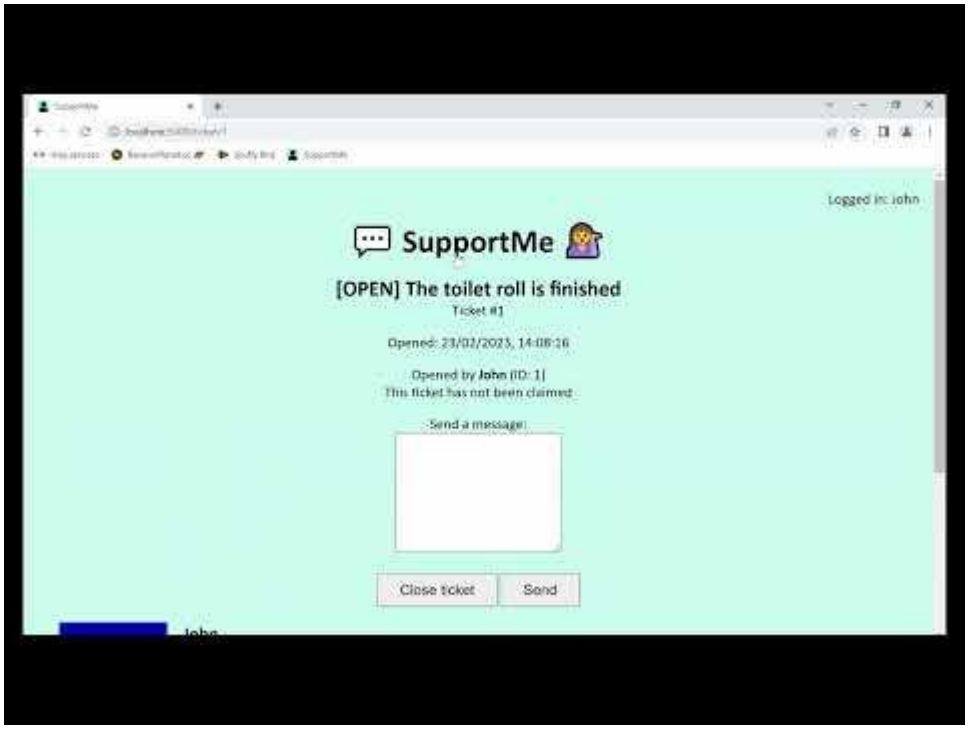
To test the front-end I will create a video which demonstrates the utilization of every page on the website.

1) The customer will first try to log in and they should get an error due to not having an account yet.
2) They should then try to register a new account, but the passwords should not match (so an error should be returned).
3) Then, they should be able to successfully register after matching the passwords, and be automatically logged in to the homepage, where they can see open and closed tickets.
4) The customer will then open a support ticket, and it should take them to the ticket's page.
5) The customer will attempt to send a message in the ticket, and then go back to the homepage.
6) The customer will now attempt to edit their profile.
7) The assistant will then register with a password that is too short, after which they will put in a longer password and create an account.
8) The assistant should be redirected to the homepage after creating their account.
9) The server admin will run the changeAccountType.py script from the terminal to promote the newly created account to an assistant account.
10) The assistant will log out and try to log in with an incorrect password.
11) The assistant will then log in with the correct password, and should be redirected to the homepage, where they can see open, closed, and claimable tickets.
12) The assistant will click on the unclaimed ticket, and proceed to claim it.

13) The assistant will send a message in the ticket, and then close it.
14) The assistant will log out.
15) The customer will go to the homepage, see the ticket in the “closed tickets” section and click on it.
16) The customer should be able to see the messages that the assistant has sent, as well as when the ticket was marked as closed.
17) The customer will then proceed to log out.

FRONT-END TESTS

https://youtu.be/gjKSlZR-KK0 - 7517 NEA SupportMe front-end tests



Customer

- 00:00 – /login
- 00:10 - /register
- 00:23 - /home
- 00:27 - /ticket/new
- 00:42 - /ticket/1
- 00:59 - /profile

Assistant

- 01:18 - /register
- 01:35 - changeAccountType.py tool
- 01:46 - /login

- 01:59 - /home
- 02:02 - /ticket/1

The video above not only tests each front-end endpoint, but also shows where the project has met the end-user utility objectives initially stated in the success criteria.

1. New users must be able to register a new account.
    a. On registration success, they should automatically be logged in.
        ✅ 00:10, 01:18

2. Returning users must be able to log into an existing account.
    a. Once users are logged in, they should be sent an authentication token which they can store in local storage – this will be used as authorization for all future requests.
        ✅ 01:46

3. Users should be able to edit their password and profile picture.
    ✅ 00:59

4. Users should be able to log out of their account, clearing the authentication token from their local storage and displaying a success message.
    ✅ 02:25

5. Customers need to be able to open support tickets.
    a. Once a support ticket has been opened they should be able to access it at a specific link.
        ✅ 00:26
    b. They must be able to send messages in the support ticket.
        ✅ 00:43

6. Assistants need to be able to read open support tickets.
    a. They must be able to send messages in the support ticket.
        ✅ 02:01
    b. They must also be able to mark a support ticket as “closed” or “open”.
        ✅ 02:17

EVALUATION

SUCCESS CRITERIA

END-USER UTILITY

End-User Utility Objective	Met?	Evaluation
1) New users must be able to register a new account. a) On registration success, they should automatically be logged in.	✅	Users can go to the /register page, which will POST to the /register endpoint on the server, with the new user's details. Once a response is received from the server, the /register page will redirect to /home and the user will be automatically logged in. *User feedback* *"Brilliant, but it would be nice to have the set of rules for a password to be valid show before the user enters an invalid password, so that the user can think of a valid password to begin with."*
2) Returning users must be able to log into an existing account. a) Once users are logged in, they should be sent an authentication token which they can store in local storage – this will be used as authorization for all future requests.	✅	If a user already has an account, they can go to the /login page and log in with their details, which will POST to the /login endpoint on the server. The server will respond with an error if the credentials are invalid, or an access token if the credentials are valid. The client-side code on the /login page will take this access token and append it to the user's browser's local storage, which allows for requests in the future to access the user's access token. *User feedback* *"Well designed."*
3) Users should be able to edit their password and profile picture.	✅	On the /profile page, users can edit their password or choose from a range of different profile pictures. This will call the POST /profile endpoint on the server, which will run an UPDATE statement on the SQLite server, updating the database. *User feedback* *"The option to change your profile picture adds a sense of freedom! However, it's not made clear that clicking the title of the site returns to the homepage so I was stuck there for a while."*

4) Users should be able to log out of their account, clearing the authentication token from their local storage and displaying a success message.	✅	Users can log out from the homepage by clicking the “log out” button at any time. This will trigger a function on the client-side, removing the access token from the user’s local storage and displaying a “logged out” message, followed by a redirection to the /login page. *User feedback* *“I like how it immediately redirects me back to the login page, and it confirms when I have logged out.”*
5) Customers need to be able to open support tickets. a) Once a support ticket has been opened they should be able to access it at a specific link. b) They must be able to send messages in the support ticket.	✅	Customers can create a new ticket at the /ticket/new page, which will call the /ticket/new POST endpoint on the server with information about the ticket to be created. The server will respond with an error, or a ticket ID. By going to /ticket/:ticketID the user can view their specific ticket’s details and messages. At that page, they can also send messages in the input box, which will POST to the /ticket endpoint with the message data to be sent. The endpoint will append the message data to the database. *User feedback* *“The unique URL allows for easy accessibility to the ticket, and clients can bookmark the link and come back to check for updates.”*
6) Assistants need to be able to read open support tickets. a) They must be able to send messages in the support ticket. b) They must also be able to mark a support ticket as “closed” or “open”.	✅	Assistants can see a list of open tickets on the /home page, and can send messages in the ticket at any time through the /ticket/:ticketID page, even if they haven’t claimed the ticket for themselves. Furthermore, assistants can use either the buttons, or send !close or !open to mark a ticket as closed or open. *User feedback* *“It’s nice that assistants can easily run commands to edit the ticket status – it makes it a lot easier for assistants to make quick changes to a ticket.”*

STRUCTURE

The Python Flask server uses the `sqlite3` library to interact with the database file stored in the project folder.

The Flask web server also has multiple REST API endpoints which can be used to interact with the system, allowing for tickets to be created, closed, profiles to be updated, and information about tickets, users, and messages to be received (not exhaustive).

As a result, the additional endpoints which provide a default user interface can easily interact with the RESTful endpoints in order to allow for the end-user to easily interact with the system.

Furthermore, anyone can create a user interface and hook up user inputs and outputs to the RESTful endpoints in order to customise or tailor the user experience depending on who their audience is.

This will be beneficial to the studio as they can put their logo on the website or redevelop it to match the theme of their existing website to maintain a level of brand consistency. These pages will interact directly with the RESTful endpoints, in order to allow users to interact with the Support Ticket system through a default standardized interface.

POTENTIAL IMPROVEMENTS

If I had more time and resources, here are some features that I would implement to further improve the system:

RATE LIMITS

Currently there is no rate limiter implemented – adding a rate limit would help protect the server from Distributed-Denial-of-Service (DDoS) attacks. One way I could implement this is keeping account of requests from individual IP addresses, and if an IP address makes too many requests in a set time period, any following requests made within that time period would be ignored and the server would return a “429 Too Many Requests” HTTP error code.

Not only would I implement rate limits for the number of requests that can be made, but a further rate limit could be implemented to prevent users from opening too many tickets. A “maximum ticket limit” could be implemented, where users can only open up to a certain number of tickets, and if they try to open any more before their existing ones are closed, they will be returned an error.

NOTIFICATIONS & AUTO-REFRESHING TICKETS

One criticism that was put forward by the client was that when a new message is sent in a ticket, the ticket page must be refreshed to see any updates. Furthermore, from the homepage, if a user has a lot of tickets, they may want to quickly see which tickets have had any updates since they were last online.

Whilst the system was not intended to be a live-chat service, having a loop running in the background on the client-side, getting all tickets, and refreshing ticket pages when new messages are sent would provide end-users with the illusion of a "live chat" system, where they get updates as soon as they're sent.

In terms of a notification system, whenever a user fetches messages in a ticket, the timestamp could be saved. A "last action" timestamp could also be stored in each ticket's metadata. When the user fetches a list of their open tickets the next time they log in, if a message was sent in any of their tickets (last action timestamp > user's last message fetch timestamp), a small red dot could appear suggesting that there are unread messages in the ticket.

ADMIN PANEL

Currently, to promote a customer account to an assistant account, the server admin must run a command-line tool and input the ID of the assistant account. If the server admin is inexperienced, they may struggle to promote user accounts – having a page dedicated for admins would allow for accounts to be easily managed from a central point, with a more user-friendly interface.

To implement this, I would need to create a new account type - ACCOUNT_TYPE_ADMIN (3) – which would have exclusive access to a newly-created page (/admin-dashboard). On this page they would be able to see a list of registered users and perform actions on behalf of registered user accounts. This would allow for spam users' accounts to be terminated and inappropriate usernames or profile pictures to be monitored and controlled.

ACCOUNT SECURITY

A final improvement I would make is the addition of an email verification system, to prevent multiple accounts being made by the same person. This would entail sending a verification code to the inputted email when a user registers a new account, and having a timer for the user to input the code into the website in order to successfully create their account.

Email verification would also allow for users to recover lost accounts if they lost their password – a /forgot-password page could be created where users input their email or username, and they are sent an email with a password reset link.

Another way to improve account security is implementing two-factor authentication. When users register, a 2FA key could be generated and stored in the User database, and when the user registers for the first time, they can see this 2FA key. Using a 2FA authenticator app on another device, the user can generate unique codes every 30 seconds, so even if their password is compromised, they also need their 2FA code to log in and verify themselves.

SOURCE CODE

MAIN SERVER LOGIC

Appendix A - server.py

```
import json, sqlite3, time, jwt, random, re
from flask import Flask, redirect, url_for, request, send_from_directory,
render_template

JWT_SECRET_KEY = "".join([random.choice(["0", "1", "2", "3", "4", "5", "6",
"7", "8", "9", "A", "B", "C", "D", "E", "F"]) for i in range(64)])


### start of global constants ###

ACCOUNT_TYPE_CUSTOMER = 1
ACCOUNT_TYPE_ASSISTANT = 2

TICKET_STATUS_OPEN = 1
TICKET_STATUS_CLOSED = 2

SYSTEM_USER_ID = 0

### end of global constants ###


App = Flask(
    __name__,
    static_url_path='',
    static_folder='static',
    template_folder='templates'
)


### start of server-side methods ###

def create_user(username, email, password):

    with sqlite3.connect("data.db") as connection:
        cursor = connection.cursor()

        cursor.execute("SELECT * FROM User WHERE username = ?", [ username
])
        if len(cursor.fetchall()) != 0:
            return (False, "That username is already taken.")

        cursor.execute("SELECT * FROM User WHERE email = ?", [ email ])
        if len(cursor.fetchall()) != 0:
            return (False, "A user is already registered with that email
address.")
```

```
        cursor.execute(
            "INSERT INTO User (username, password, createdAt, accountType,
profileIcon, email) VALUES (?, ?, ?, ?, ?, ?)",
            [ username, password, int(time.time()), ACCOUNT_TYPE_CUSTOMER,
f"/profile-icons/{random.choice(['blue', 'green', 'purple', 'red'])}.png",
email ]
        )

        user_id = cursor.lastrowid
        return (True, user_id)

def login_user(username, password):

    with sqlite3.connect("data.db") as connection:
        cursor = connection.cursor()

        cursor.execute("SELECT userID, password, accountType FROM User
WHERE username = ?", [ username ])
        user_list = cursor.fetchall()
        if len(user_list) == 0:
            return (False, "A user with the supplied username and password
was not found.")

        _user_id, _password, _account_type = user_list[0]

        if password == _password:
            access_token = generate_access_token(_user_id, _account_type)
            return (True, access_token)
        else:
            return (False, "A user with the supplied username and password
was not found.")

def get_profile(user_id):

    with sqlite3.connect("data.db") as connection:

        cursor = connection.cursor()

        cursor.execute("SELECT username, createdAt, accountType,
profileIcon FROM User WHERE userID = ?", [ user_id ])
        user_list = cursor.fetchall()
        if len(user_list) == 0:
            return None

        _username, _created_at, _account_type, _profile_icon = user_list[0]

        return {
            "user_id": user_id,
            "username": _username,
            "created_at": _created_at,
            "account_type": _account_type,
            "profile_icon": _profile_icon
        }

def generate_access_token(user_id, account_type):
    access_token = jwt.encode({
```

```
        "user_id": user_id,
        "account_type": account_type,
        "exp": int(time.time()) + 86400
    }, JWT_SECRET_KEY, algorithm="HS256")
    return access_token

def verify_access_token(access_token):
    try:
        d_at = jwt.decode(access_token.split(" ")[1], JWT_SECRET_KEY,
algorithms=["HS256"])
        if d_at["exp"] < time.time():
            return None
        else:
            return d_at
    except:
        return None

def is_valid_username(username):
    if not username.isalnum():
        return False, "Username can only contain alphanumeric characters."
    elif len(username) > 16:
        return False, "Username cannot be more than 16 characters."
    return True, True

def is_valid_password(password):
    if len(password) < 8:
        return False, "Password must be at least 8 characters."
    elif len(password) > 32:
        return False, "Password cannot be more than 32 characters."
    return True, True

def is_valid_email(email):
    if re.search('^[a-z0-9]+[\._]?[a-z0-9]+[@]\w+[.]\w{2,63}$', email):
        return True, True
    else:
        return False, "The provided email was invalid."

def is_valid_profile_icon(profile_icon):
    if profile_icon not in [ f"/profile-icons/{c}.png" for c in ["blue",
"green", "purple", "red"] ]:
        return False, "Profile icon is invalid."
    return True, True

def is_valid_ticket_title(ticket_title):
    if not all(x.isalnum() or x.isspace() for x in ticket_title):
        return False, "Ticket title can only contain spaces and
alphanumeric characters."
    elif len(ticket_title) < 8:
        return False, "Ticket title must be at least 8 characters - be
descriptive."
    elif len(ticket_title) > 64:
        return False, "Ticket title cannot be more than 64 characters -
keep it concise."
    return True, True

def is_valid_message(message):
```

```
    if len(message) < 8 and not message.startswith("!"):
        return False, "Message length must be at least 8 characters - be
descriptive."
    elif len(message) > 512:
        return False, "Message length cannot be more than 512 characters."
    return True, True

### end of server-side methods ###


### start of flask endpoints ###

@App.route('/', methods=['GET'])
@App.route('/home', methods=['GET'])
def home_req():
    if request.method == 'GET':
        return (render_template('home.html'), 200)

@App.route('/register', methods=['GET', 'POST'])
def register_req():

    if request.method == 'GET':

        return (render_template('register.html'), 200)

    elif request.method == 'POST':

        try:

            registration_data = json.loads(request.get_data())

            if list(registration_data.keys()) != ["username", "email",
"password"]:
                return ({ "error": "The request failed." }, 400)

            if "" in list(registration_data.values()):
                return ({ "error": "Please don't leave any blank fields."
}, 400)

            _valid_username =
is_valid_username(registration_data["username"])
            if not _valid_username[0]:
                return ({ "error": _valid_username[1] }, 400)

            _valid_password =
is_valid_password(registration_data["password"])
            if not _valid_password[0]:
                return ({ "error": _valid_password[1] }, 400)

            _valid_email = is_valid_email(registration_data["email"])
            if not _valid_email[0]:
                return ({ "error": _valid_email[1] }, 400)

            cu_success, cu_res = create_user(registration_data["username"],
registration_data["email"], registration_data["password"])
            if cu_success:
```

```
                return ({ "access_token": generate_access_token(cu_res,
ACCOUNT_TYPE_CUSTOMER) }, 200)
            else:
                return ({ "error": cu_res }, 400)

        except:
            return ({ "error": "Server failed to parse the request." },
400)

@App.route('/login', methods=['GET', 'POST'])
def login_req():

    if request.method == 'GET':
        return (render_template('login.html'), 200)

    elif request.method == 'POST':

        try:

            login_data = json.loads(request.get_data())

            if list(login_data.keys()) != ["username", "password"]:
                return ({ "error": "The request failed." }, 400)

            if "" in list(login_data.values()):
                return ({ "error": "Please don't leave any blank fields."
}, 400)

            lu_success, lu_res = login_user(login_data["username"],
login_data["password"])
            if lu_success:
                return ({ "access_token": lu_res }, 200)
            else:
                return ({ "error": lu_res }, 400)

        except:
            return ({ "error": "Server failed to parse the request." },
400)

@App.route('/get-profile/<int:user_id>', methods=['GET'])
def get_profile_by_user_id_req(user_id):
    if request.method == 'GET':

        auth_user =
verify_access_token(request.headers.get("Authorization"))
        if auth_user == None:
            return ({ "error": "User is not authenticated to make this
request." }, 403)

        with sqlite3.connect("data.db") as connection:

            cursor = connection.cursor()

            cursor.execute("SELECT username, createdAt, accountType,
profileIcon FROM User WHERE userID = ?", [ user_id ])
            user_list = cursor.fetchall()
```

```
            if len(user_list) == 0:
                return ({ "error": "User not found." }, 400)

            _username, _created_at, _account_type, _profile_icon =
user_list[0]

            return ({ "user": {
                "user_id": user_id,
                "username": _username,
                "created_at": _created_at,
                "account_type": _account_type,
                "profile_icon": _profile_icon
            } }, 200)

@App.route('/ticket/new', methods=['GET', 'POST'])
def create_ticket_req():
    if request.method == 'GET':
        return (render_template('ticket/new.html'), 200)

    elif request.method == 'POST':

        try:

            auth_user =
verify_access_token(request.headers.get("Authorization"))
            if auth_user == None or auth_user["account_type"] !=
ACCOUNT_TYPE_CUSTOMER:
                return ({ "error": "User is not authenticated to make this
request." }, 403)

            ticket_data = json.loads(request.get_data())

            if list(ticket_data.keys()) != ["ticket_title", "message"]:
                return ({ "error": "The request failed." }, 400)

            if "" in list(ticket_data.values()):
                return ({ "error": "Please don't leave any blank fields."
}, 400)

            _valid_ticket_title =
is_valid_ticket_title(ticket_data["ticket_title"])
            if not _valid_ticket_title[0]:
                return ({ "error": _valid_ticket_title[1] }, 400)

            _valid_message = is_valid_message(ticket_data["message"])
            if not _valid_message[0]:
                return ({ "error": _valid_message[1] }, 400)

            with sqlite3.connect("data.db") as connection:

                cursor = connection.cursor()

                cursor.execute("INSERT INTO Ticket (customerID,
assistantID, openedAt, closedAt, title) VALUES (?, ?, ?, ?, ?);", [
auth_user["user_id"], -1, int(time.time()), -1, ticket_data["ticket_title"]
])
```

```
                ticket_id = cursor.lastrowid

                cursor.execute("INSERT INTO Message (ticketID, authorID,
body, sentAt) VALUES (?, ?, ?, ?);", [ ticket_id, auth_user["user_id"],
ticket_data["message"], int(time.time()) ])
                message_id = cursor.lastrowid

                return ({
                    "ticket_id": ticket_id,
                    "message_id": message_id
                }, 200)

        except:
            return ({ "error": "Server failed to parse the request." },
400)

@App.route('/get-ticket/<int:ticket_id>', methods=['GET'])
def ticket_json_by_id_req(ticket_id):
    if request.method == 'GET':

        auth_user =
verify_access_token(request.headers.get("Authorization"))
        if auth_user == None:
            return ({ "error": "User is not authenticated to make this
request." }, 403)

        with sqlite3.connect("data.db") as connection:

            cursor = connection.cursor()

            cursor.execute("SELECT customerID, assistantID, openedAt,
closedAt, title FROM Ticket WHERE ticketID = ?;", [ ticket_id ])

            ticket_list = cursor.fetchall()
            if len(ticket_list) == 0:
                return ({ "error": "Ticket with given ID was not found in
the database." }, 404)

            _customer_id, _assistant_id, _opened_at, _closed_at, _title =
ticket_list[0]

            if auth_user["user_id"] not in [_customer_id] and
auth_user["account_type"] != ACCOUNT_TYPE_ASSISTANT:
                return ({ "error": "User is not authenticated to make this
request." }, 403)

            return ({
                "ticket_data": {
                    "ticket_id": ticket_id,
                    "customer_id": _customer_id,
                    "assistant_id": _assistant_id,
                    "opened_at": _opened_at,
                    "closed_at": _closed_at,
                    "title": _title
                }
            }, 200)
```

```
@App.route('/ticket/<int:ticket_id>', methods=['GET', 'POST'])
def ticket_by_id_req(ticket_id):
    if request.method == 'GET':
        return (render_template('ticket/ticket.html', ticket_id=ticket_id),
200)

    elif request.method == 'POST':

        try:

            auth_user =
verify_access_token(request.headers.get("Authorization"))
            if auth_user == None:
                return ({ "error": "User is not authenticated to make this
request." }, 403)

            message_data = json.loads(request.get_data())

            if list(message_data.keys()) != ["message"]:
                return ({ "error": "The request failed." }, 400)

            if "" in list(message_data.values()):
                return ({ "error": "Please don't leave any blank fields."
}, 400)

            _valid_message = is_valid_message(message_data["message"])
            if not _valid_message[0]:
                return ({ "error": _valid_message[1] }, 400)

            with sqlite3.connect("data.db") as connection:

                cursor = connection.cursor()

                cursor.execute("SELECT customerID, assistantID, closedAt
FROM Ticket WHERE ticketID = ?;", [ ticket_id ])

                ticket_list = cursor.fetchall()
                if len(ticket_list) == 0:
                    return ({ "error": "Ticket with given ID was not found
in the database." }, 404)

                _customer_id, _assistant_id, _closed_at = ticket_list[0]

                if auth_user["user_id"] not in [_customer_id] and
auth_user["account_type"] != ACCOUNT_TYPE_ASSISTANT:
                    return ({ "error": "User is not authenticated to make
this request." }, 403)

                user_profile = get_profile(auth_user["user_id"])
                if message_data["message"] == "!close":
                    cursor.execute("UPDATE Ticket SET closedAt = ? WHERE
(ticketID = ?);", [ int(time.time()), ticket_id ])
                    cursor.execute("INSERT INTO Message (ticketID,
authorID, body, sentAt) VALUES (?, ?, ?, ?);", [ ticket_id, SYSTEM_USER_ID,
```

```
f"This ticket has been closed by {user_profile['username']} (ID:
{auth_user['user_id']}).", int(time.time()) ])
                elif message_data["message"] == "!open":
                    cursor.execute("UPDATE Ticket SET closedAt = -1 WHERE
(ticketID = ?);", [ ticket_id ])
                    cursor.execute("INSERT INTO Message (ticketID,
authorID, body, sentAt) VALUES (?, ?, ?, ?);", [ ticket_id, SYSTEM_USER_ID,
f"This ticket has been reopened by {user_profile['username']} (ID:
{auth_user['user_id']}).", int(time.time()) ])
                elif message_data["message"] == "!claim" and
auth_user["account_type"] == ACCOUNT_TYPE_ASSISTANT:
                    cursor.execute("UPDATE Ticket SET assistantID = ?,
closedAt = ? WHERE (ticketID = ?);", [ auth_user["user_id"], -1, ticket_id
])
                    if _closed_at != -1:
                        cursor.execute("INSERT INTO Message (ticketID,
authorID, body, sentAt) VALUES (?, ?, ?, ?);", [ ticket_id, SYSTEM_USER_ID,
f"This ticket has been claimed and reopened by {user_profile['username']}
(ID: {auth_user['user_id']}).", int(time.time()) ])
                    else:
                        cursor.execute("INSERT INTO Message (ticketID,
authorID, body, sentAt) VALUES (?, ?, ?, ?);", [ ticket_id, SYSTEM_USER_ID,
f"This ticket has been claimed by {user_profile['username']} (ID:
{auth_user['user_id']}).", int(time.time()) ])
                else:
                    if _closed_at != -1:
                        cursor.execute("INSERT INTO Message (ticketID,
authorID, body, sentAt) VALUES (?, ?, ?, ?);", [ ticket_id, SYSTEM_USER_ID,
f"This ticket has been reopened by {user_profile['username']} (ID:
{auth_user['user_id']}).", int(time.time())-1 ])
                        cursor.execute("UPDATE Ticket SET closedAt = ?
WHERE (ticketID = ?);", [ -1, ticket_id ])
                    cursor.execute("INSERT INTO Message (ticketID,
authorID, body, sentAt) VALUES (?, ?, ?, ?);", [ ticket_id,
auth_user["user_id"], message_data["message"], int(time.time())+1 ])

                message_id = cursor.lastrowid

                return ({
                    "ticket_id": ticket_id,
                    "message_id": message_id
                }, 200)

        except:
            return ({ "error": "Server failed to parse the request." },
400)

@App.route('/get-open-tickets/<int:user_id>', methods=['GET'])
def get_open_tickets_by_user_id_req(user_id):
    if request.method == 'GET':

        auth_user =
verify_access_token(request.headers.get("Authorization"))
        if auth_user == None or auth_user["user_id"] != user_id:
            return ({ "error": "User is not authenticated to make this
request." }, 403)
```

```
        with sqlite3.connect("data.db") as connection:

            cursor = connection.cursor()

            cursor.execute("SELECT * FROM User WHERE userID = ?;", [
user_id ])

            if len(cursor.fetchall()) == 0:
                return ({ "error": "User with given ID was not found in the
database." }, 404)

            cursor.execute("SELECT ticketID, customerID, assistantID,
openedAt, closedAt, title FROM Ticket WHERE (customerID = ? OR assistantID
= ?) AND closedAt = -1 ORDER BY openedAt ASC;", [ user_id, user_id ])
            ticket_list = cursor.fetchall()

            response = []

            for t in ticket_list:
                _ticket_id, _customer_id, _assistant_id, _opened_at,
_closed_at, _title = t
                response.append({
                    "ticket_id": _ticket_id,
                    "customer_id": _customer_id,
                    "assistant_id": _assistant_id,
                    "opened_at": _opened_at,
                    "closed_at": _closed_at,
                    "title": _title
                })

            return ({
                "tickets": response
            }, 200)

@App.route('/get-closed-tickets/<int:user_id>', methods=['GET'])
def get_closed_tickets_by_user_id_req(user_id):
    if request.method == 'GET':

        auth_user =
verify_access_token(request.headers.get("Authorization"))
        if auth_user == None or auth_user["user_id"] != user_id:
            return ({ "error": "User is not authenticated to make this
request." }, 403)

        with sqlite3.connect("data.db") as connection:

            cursor = connection.cursor()

            cursor.execute("SELECT * FROM User WHERE userID = ?;", [
user_id ])

            if len(cursor.fetchall()) == 0:
                return ({ "error": "User with given ID was not found in the
database." }, 404)
```

```
            cursor.execute("SELECT ticketID, customerID, assistantID,
openedAt, closedAt, title FROM Ticket WHERE (customerID = ? OR assistantID
= ?) AND closedAt != -1 ORDER BY closedAt DESC;", [ user_id, user_id ])
            ticket_list = cursor.fetchall()

            response = []

            for t in ticket_list:
                _ticket_id, _customer_id, _assistant_id, _opened_at,
_closed_at, _title = t
                response.append({
                    "ticket_id": _ticket_id,
                    "customer_id": _customer_id,
                    "assistant_id": _assistant_id,
                    "opened_at": _opened_at,
                    "closed_at": _closed_at,
                    "title": _title
                })

            return ({
                "tickets": response
            }, 200)

@App.route('/get-unclaimed-tickets', methods=['GET'])
def get_unclaimed_tickets_req():
    if request.method == 'GET':

        auth_user =
verify_access_token(request.headers.get("Authorization"))
        if auth_user == None or auth_user["account_type"] !=
ACCOUNT_TYPE_ASSISTANT:
            return ({ "error": "User is not authenticated to make this
request." }, 403)

        with sqlite3.connect("data.db") as connection:

            cursor = connection.cursor()

            cursor.execute("SELECT ticketID, customerID, assistantID,
openedAt, closedAt, title FROM Ticket WHERE assistantID = -1 ORDER BY
openedAt ASC;")
            ticket_list = cursor.fetchall()

            response = []

            for t in ticket_list:
                _ticket_id, _customer_id, _assistant_id, _opened_at,
_closed_at, _title = t
                response.append({
                    "ticket_id": _ticket_id,
                    "customer_id": _customer_id,
                    "assistant_id": _assistant_id,
                    "opened_at": _opened_at,
                    "closed_at": _closed_at,
                    "title": _title
                })
```

```
            return ({
                "tickets": response
            }, 200)

@App.route('/get-messages/<int:ticket_id>', methods=['GET'])
def get_messages_by_ticket_id_req(ticket_id):
    if request.method == 'GET':

        auth_user =
verify_access_token(request.headers.get("Authorization"))
        if auth_user == None:
            return ({ "error": "User is not authenticated to make this
request." }, 403)

        with sqlite3.connect("data.db") as connection:

            cursor = connection.cursor()

            cursor.execute("SELECT customerID, assistantID FROM Ticket
WHERE ticketID = ?;", [ ticket_id ])

            ticket_list = cursor.fetchall()
            if len(ticket_list) == 0:
                return ({ "error": "Ticket with given ID was not found in
the database." }, 404)

            _customer_id, _assistant_id = ticket_list[0]

            if auth_user["user_id"] not in [_customer_id] and
auth_user["account_type"] != ACCOUNT_TYPE_ASSISTANT:
                return ({ "error": "User is not authenticated to make this
request." }, 403)

            cursor.execute("SELECT messageID, authorID, body, sentAt FROM
Message WHERE ticketID = ? ORDER BY sentAt DESC;", [ ticket_id ])
            message_list = cursor.fetchall()

            response = []

            for m in message_list:
                _message_id, _author_id, _body, _sent_at = m
                response.append({
                    "message_id": _message_id,
                    "author_id": _author_id,
                    "body": _body,
                    "sent_at": _sent_at
                })

            return ({
                "ticket_id": ticket_id,
                "message_list": response
            }, 200)

@App.route('/profile', methods=['GET', 'POST'])
def profile_req():
```

```
    if request.method == 'GET':
        return (render_template('profile.html'), 200)

    elif request.method == 'POST':

        try:

            auth_user =
verify_access_token(request.headers.get("Authorization"))
            if auth_user == None:
                return ({ "error": "User is not authenticated to make this
request." }, 403)

            profile_data = json.loads(request.get_data())

            if list(profile_data.keys()) != ["new_password",
"old_password", "profile_icon"]:
                return ({ "error": "The request failed." }, 400)

            if profile_data["new_password"] == "":
                profile_data["new_password"] = profile_data["old_password"]

            if "" in list(profile_data.values()):
                return ({ "error": "Please don't leave any blank fields."
}, 400)

            with sqlite3.connect("data.db") as connection:

                cursor = connection.cursor()

                cursor.execute("SELECT password FROM User WHERE userID =
?;", [ auth_user["user_id"] ])

                user_list = cursor.fetchall()
                if len(user_list) == 0:
                    return ({ "error": "User with given ID was not found in
the database." }, 404)

                if user_list[0][0] != profile_data["old_password"]:
                    return ({ "error": "Incorrect password was provided."
}, 403)

                _valid_password =
is_valid_password(profile_data["new_password"])
                if not _valid_password[0]:
                    return ({ "error": _valid_password[1] }, 400)

                _valid_profile_icon =
is_valid_profile_icon(profile_data["profile_icon"])
                if not _valid_profile_icon[0]:
                    return ({ "error": _valid_profile_icon[1] }, 400)

                cursor.execute("UPDATE User SET password = ?, profileIcon =
? WHERE (userID = ?)", [ profile_data["new_password"],
profile_data["profile_icon"], auth_user["user_id"] ])
```

```
                return ({ "msg": "Profile has been updated." }, 200)

        except:
            return ({ "error": "Server failed to parse the request." },
400)

### end of flask endpoints ###

if __name__ == "__main__":
    App.run(host="0.0.0.0")
```

Appendix B - tools/changeAccountType.py

```
import sqlite3

user_id = int(input("Enter the user ID of the account: "))
account_type = int(input("Enter an account type to set (1 = customer, 2 =
assistant): "))

with sqlite3.connect("../data.db") as connection:

    try:
        cursor = connection.cursor()
        cursor.execute("UPDATE User SET accountType = ? WHERE userID = ?;",
[ account_type, user_id ])
        if cursor.rowcount == 0:
            print("User with given ID was not found in the database.
Terminating.")
        else:
            print(f"User with ID {user_id}'s account type has been
successfully updated to {account_type}.")

    except sqlite3.Error as error:
        print(error)
```

CLIENT SCRIPTS

Appendix C - static/src/constants.js

```
const ACCOUNT_TYPE_CUSTOMER = 1
const ACCOUNT_TYPE_ASSISTANT = 2

const TICKET_STATUS_OPEN = 1
const TICKET_STATUS_CLOSED = 2

const SYSTEM_USER_ID = 0
```

Appendix D - static/src/ticketTools.js

```
async function openTicket() {

    let ticket_title_input = document.getElementById("ticket_title_input");
    let message_input = document.getElementById("message_input");

    const res = await (await fetch('/ticket/new', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
          'Authorization': 'Bearer ' + localStorage.getItem("access_token")
        },
        redirect: 'manual',
        body: JSON.stringify({
            ticket_title: ticket_title_input.value,
            message: message_input.value
        })
    })).json();

    if (res["error"] != undefined) {
        alert(res["error"]);
    } else {
        window.location.href = "/ticket/" + res["ticket_id"];
    };

}

async function getTicket(ticket_id) {

    const res = await (await fetch('/get-ticket/' + ticket_id, {
        method: 'GET',
        headers: {
          'Content-Type': 'application/json',
          'Authorization': 'Bearer ' + localStorage.getItem("access_token")
        }
    })).json();

    if (res["error"] != undefined) {
        alert(res["error"]);
        return false;
    } else {
        return res;
    };

}

async function getOpenTicketsByUserID(user_id) {

    const res = await (await fetch('/get-open-tickets/' + user_id, {
        method: 'GET',
        headers: {
          'Content-Type': 'application/json',
          'Authorization': 'Bearer ' + localStorage.getItem("access_token")
        }
    })).json();
```

```
    if (res["error"] != undefined) {
        alert(res["error"]);
        return [];
    } else {
        return res["tickets"];
    };

}

async function getClosedTicketsByUserID(user_id) {

    const res = await (await fetch('/get-closed-tickets/' + user_id, {
        method: 'GET',
        headers: {
          'Content-Type': 'application/json',
          'Authorization': 'Bearer ' + localStorage.getItem("access_token")
        }
    })).json();

    if (res["error"] != undefined) {
        alert(res["error"]);
        return [];
    } else {
        return res["tickets"];
    };

}

async function getUnclaimedTickets() {

    const res = await (await fetch('/get-unclaimed-tickets', {
        method: 'GET',
        headers: {
          'Content-Type': 'application/json',
          'Authorization': 'Bearer ' + localStorage.getItem("access_token")
        }
    })).json();

    if (res["error"] != undefined) {
        alert(res["error"]);
        return [];
    } else {
        return res["tickets"];
    };

}

async function
sendMessage(message_input=document.getElementById("message_input").value) {

    const res = await (await fetch('/ticket/' + ticketID, {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
            'Authorization': 'Bearer ' +
localStorage.getItem("access_token")
```

```
        },
        body: JSON.stringify({
            message: message_input
        })
    })).json();

    if (res["error"] != undefined) {
        alert(res["error"]);
        return [];
    } else {
        window.location.reload();
        return res;
    }

}

async function getMessages(ticket_id) {

    const res = await (await fetch('/get-messages/' + ticket_id, {
        method: 'GET',
        headers: {
          'Content-Type': 'application/json',
          'Authorization': 'Bearer ' + localStorage.getItem("access_token")
        }
    })).json();

    if (res["error"] != undefined) {
        alert(res["error"]);
        return [];
    } else {
        return res["message_list"];
    };

}
```

Appendix E - static/src/usertools.js

```
const getDecodedAccessToken = () => {
    return localStorage.getItem("access_token") != null ?
jwt_decode(localStorage.getItem("access_token")) : null;
}

function getLoggedInUser() {
    let d_at = getDecodedAccessToken();
    if (d_at == null) return null;
    if (d_at["exp"] < (Date.now() / 1000)) {
        localStorage.removeItem("access_token");
        return null
    };
    return d_at;
}

async function register() {
```

```
    let username_input = document.getElementById("username_input");
    let email_input = document.getElementById("email_input");
    let password_input = document.getElementById("password_input");
    let passwordc_input = document.getElementById("passwordc_input");

    if (password_input.value != passwordc_input.value) return
alert("Passwords do not match.")

    const res = await (await fetch('/register', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json'
        },
        redirect: 'manual',
        body: JSON.stringify({
            username: username_input.value,
            email: email_input.value,
            password: password_input.value
        })
    })).json();

    if (res["error"] != undefined) {
        alert(res["error"]);
    } else {
        localStorage.setItem("access_token", res["access_token"]);
        window.location.href = "/home";
    };

}

async function login() {

    let username_input = document.getElementById("username_input");
    let password_input = document.getElementById("password_input");

    const res = await (await fetch('/login', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json'
        },
        redirect: 'manual',
        body: JSON.stringify({
            username: username_input.value,
            password: password_input.value
        })
    })).json();

    if (res["error"] != undefined) {
        alert(res["error"]);
    } else {
        localStorage.setItem("access_token", res["access_token"]);
        window.location.href = "/home";
    };

}
```

```
async function logout() {

    localStorage.removeItem("access_token");
    alert("Successfully logged out.");
    window.location.href = "/login";

}

async function getProfile(user_id) {

    const res = await (await fetch('/get-profile/' + user_id, {
        method: 'GET',
        headers: {
          'Content-Type': 'application/json',
          'Authorization': 'Bearer ' + localStorage.getItem("access_token")
        }
    })).json();

    if (res["error"] != undefined) {
        alert(res["error"]);
        return false;
    } else {
        return res["user"];
    };

}

async function updateProfile() {

    let new_password_input = document.getElementById("new_password_input");
    let new_passwordc_input =
document.getElementById("new_passwordc_input");
    let old_password_input = document.getElementById("old_password_input");
    let profile_icon_select =
document.getElementById("profile_icon_select");

    if (new_password_input.value != new_passwordc_input.value) return
alert("Passwords do not match.");
    if (old_password_input.value == "") return alert("Old password is
needed to confirm changes.");

    const res = await (await fetch('/profile', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
          'Authorization': 'Bearer ' + localStorage.getItem("access_token")
        },
        redirect: 'manual',
        body: JSON.stringify({
            new_password: new_password_input.value,
            old_password: old_password_input.value,
            profile_icon: profile_icon_select.value
        })
    })).json();

    if (res["error"] != undefined) {
```

```
        alert(res["error"]);
    } else {
        alert("Profile updated.");
        window.location.reload();
    };

}
```

Appendix F - static/src/jwt-decode.js (sourced from GitHub)

```
// CODE IS FROM https://github.com/auth0/jwt-decode

(function (factory) {
    typeof define === 'function' && define.amd ? define(factory) :
    factory();
}((function () { 'use strict';

    /**
     * The code was extracted from:
     * https://github.com/davidchambers/Base64.js
     */

    var chars =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/=";

    function InvalidCharacterError(message) {
        this.message = message;
    }

    InvalidCharacterError.prototype = new Error();
    InvalidCharacterError.prototype.name = "InvalidCharacterError";

    function polyfill(input) {
        var str = String(input).replace(/=+$/, "");
        if (str.length % 4 == 1) {
            throw new InvalidCharacterError(
                "'atob' failed: The string to be decoded is not correctly
encoded."
            );
        }
        for (
            // initialize result and counters
            var bc = 0, bs, buffer, idx = 0, output = "";
            // get next character
            (buffer = str.charAt(idx++));
            // character found in table? initialize bit storage and add its
ascii value;
            ~buffer &&
            ((bs = bc % 4 ? bs * 64 + buffer : buffer),
                // and if not first of each 4 characters,
                // convert the first 8 bits to one ascii character
                bc++ % 4) ?
            (output += String.fromCharCode(255 & (bs >> ((-2 * bc) & 6))))
:
```

```
        0
    ) {
        // try to find character in table (0-63, not found => -1)
        buffer = chars.indexOf(buffer);
    }
    return output;
}

var atob = (typeof window !== "undefined" &&
    window.atob &&
    window.atob.bind(window)) ||
polyfill;

function b64DecodeUnicode(str) {
    return decodeURIComponent(
        atob(str).replace(/(.)/g, function(m, p) {
            var code = p.charCodeAt(0).toString(16).toUpperCase();
            if (code.length < 2) {
                code = "0" + code;
            }
            return "%" + code;
        })
    );
}

function base64_url_decode(str) {
    var output = str.replace(/-/g, "+").replace(/_/g, "/");
    switch (output.length % 4) {
        case 0:
            break;
        case 2:
            output += "==";
            break;
        case 3:
            output += "=";
            break;
        default:
            throw "Illegal base64url string!";
    }

    try {
        return b64DecodeUnicode(output);
    } catch (err) {
        return atob(output);
    }
}

function InvalidTokenError(message) {
    this.message = message;
}

InvalidTokenError.prototype = new Error();
InvalidTokenError.prototype.name = "InvalidTokenError";

function jwtDecode(token, options) {
    if (typeof token !== "string") {
```

```
            throw new InvalidTokenError("Invalid token specified");
        }

        options = options || {};
        var pos = options.header === true ? 0 : 1;
        try {
            return JSON.parse(base64_url_decode(token.split(".")[pos]));
        } catch (e) {
            throw new InvalidTokenError("Invalid token specified: " +
e.message);
        }
    }

    /*
     * Expose the function on the window object
     */

    //use amd or just through the window object.
    if (window) {
        if (typeof window.define == "function" && window.define.amd) {
            window.define("jwt_decode", function() {
                return jwtDecode;
            });
        } else if (window) {
            window.jwt_decode = jwtDecode;
        }
    }

}));
//# sourceMappingURL=jwt-decode.js.map
```

STATIC CLIENT FILES

Appendix G - templates/base.html

```
<!DOCTYPE html>
<html lang="en">

    <head>

        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-
scale=1.0">
        <title>SupportMe</title>

        <link rel="stylesheet" href="/style.css">
        <link rel="icon" href="/icon.png">

    </head>

    <body>
```

```
        <p id="loggedInMessage">...</p>

        <br>

        <a href="/home" style="text-decoration: none; color: black;">
            <h1>💬 SupportMe 👩‍💻</h1>
        </a>

        <script src="/src/constants.js"></script>
        <script src="/src/jwt-decode.js"></script>
        <script src="/src/userTools.js"></script>
        <script src="/src/ticketTools.js"></script>

        <script>
            const currentUser = getLoggedInUser();
            if (currentUser != null) {
                getProfile(currentUser["user_id"]).then(profile => {
                    document.getElementById("loggedInMessage").innerText =
"Logged in: " + profile["username"];
                });
            } else {
                document.getElementById("loggedInMessage").innerText = "You
are not logged in.";
            };
        </script>

        {% block content %} {% endblock %}

    </body>

</html>
```

Appendix H - templates/home.html

```
{% extends 'base.html' %}

{% block content %}
    <h2>Home</h2>

    <br>

    <a href="/ticket/new"><button id="open_ticket_button">Open a support
ticket</button></a>
    <br>
    <a href="/profile"><button>Edit profile</button></a>
    <button onclick=logout()>Log out</button>

    <section>

        <h3>Open tickets</h3>

        <ul id="open_tickets_list"></ul>

    </section>
```

```
    <section id="claimable_tickets">

        <h3>Claimable tickets</h3>

        <ul id="unclaimed_tickets_list"></ul>

    </section>

    <section>

        <h3>Closed tickets</h3>

        <ul id="closed_tickets_list"></ul>

    </section>

    <script>
        if (currentUser == null) window.location.href = "/login";

        const openTicketsList =
document.getElementById("open_tickets_list")
        const closedTicketsList =
document.getElementById("closed_tickets_list")
        const unclaimedTicketsList =
document.getElementById("unclaimed_tickets_list")

        getOpenTicketsByUserID(currentUser["user_id"]).then(openTickets =>
{
            if (openTickets.length == 0) openTicketsList.innerHTML = "<i>No
tickets to show.</i>"
            for (t of openTickets) {
                let _a = document.createElement("a");
                _a.innerText = "#" + t["ticket_id"] + " - " + t["title"];
                _a.href = "/ticket/" + t["ticket_id"];
                let _li = document.createElement("li");
                _li.appendChild(_a);
                openTicketsList.appendChild(_li);
            };
        });

        getClosedTicketsByUserID(currentUser["user_id"]).then(closedTickets
=> {
            if (closedTickets.length == 0) closedTicketsList.innerHTML =
"<i>No tickets to show.</i>"
            for (t of closedTickets) {
                let _a = document.createElement("a");
                _a.innerText = "#" + t["ticket_id"] + " - " + t["title"];
                _a.href = "/ticket/" + t["ticket_id"];
                let _li = document.createElement("li");
                _li.appendChild(_a);
                closedTicketsList.appendChild(_li);
            };
        });

        if (currentUser["account_type"] == ACCOUNT_TYPE_CUSTOMER) {
```

```
            document.getElementById("claimable_tickets").style.display =
"none";

        } else if (currentUser["account_type"] == ACCOUNT_TYPE_ASSISTANT) {

            document.getElementById("claimable_tickets").style.display =
"block";
            document.getElementById("open_ticket_button").style.display =
"none";

            getUnclaimedTickets().then(unclaimedTickets => {
                if (unclaimedTickets.length == 0)
unclaimedTicketsList.innerHTML = "<i>No tickets to show.</i>"
                for (t of unclaimedTickets) {
                    _a = document.createElement("a");
                    _a.innerText = "#" + t["ticket_id"] + " - " +
t["title"];
                    _a.href = "/ticket/" + t["ticket_id"];
                    _li = document.createElement("li");
                    _li.appendChild(_a);
                    unclaimedTicketsList.appendChild(_li);
                };
            });

        }
    </script>
{% endblock %}
```

Appendix I - templates/login.html

```
{% extends 'base.html' %}

{% block content %}
    <h2>Login</h2>

    <br>

    <label for="username">Username</label><br>
    <input type="text" id="username_input" name="username" />

    <br><br>

    <label for="password">Password</label><br>
    <input type="password" id="password_input" name="password" />

    <br><br>

    <button type="submit" onclick="login()">Login</button>

    <br><br>

    <a href="register">Don't have an account? Click here to sign up!</a>
```

```
    <script>
        if (currentUser != null) window.location.href = "/home";
    </script>
{% endblock %}
```

Appendix J - templates/profile.html

```
{% extends 'base.html' %}

{% block content %}
    <h2>Profile</h2>

    <br>

    <img id="profile_icon">

    <br><br>

    <label for="user_id">Your user ID</label><br>
    <input type="text" id="user_id_input" name="user_id" disabled />

    <br><br>

    <label for="username">Your username (cannot be changed)</label><br>
    <input type="text" id="username_input" name="username" disabled />

    <br><br>

    <label for="new_password">New password</label><br>
    <input type="password" id="new_password_input" name="new_password" />

    <br><br>

    <label for="new_passwordc">Confirm new password</label><br>
    <input type="password" id="new_passwordc_input" name="new_passwordc" />

    <br><br>

    <label for="old_password">Old password</label><br>
    <input type="password" id="old_password_input" name="old_password" />

    <br><br>

    <label for="profile_icon">Profile picture</label><br>
    <select name="profile_icon" id="profile_icon_select"
oninput="updateProfileIconPreview()">
        <option value="/profile-icons/blue.png">Blue</option>
        <option value="/profile-icons/green.png">Green</option>
        <option value="/profile-icons/purple.png">Purple</option>
        <option value="/profile-icons/red.png">Red</option>
    </select>

    <br><br>
```

```
    <button type="submit" onclick="updateProfile()">Save changes</button>

    <br><br>

    <a href="register">Don't have an account? Click here to sign up!</a>

    <script>
        if (currentUser == null) window.location.href = "/login";

        getProfile(currentUser["user_id"]).then(profile => {
            document.getElementById("profile_icon").src =
profile["profile_icon"];
            document.getElementById("profile_icon").width = 150;
            document.getElementById("user_id_input").value =
profile["user_id"];
            document.getElementById("username_input").value =
profile["username"];
            document.getElementById("profile_icon_select").value =
profile["profile_icon"];
        });

        function updateProfileIconPreview() {
            document.getElementById("profile_icon").src =
document.getElementById("profile_icon_select").value;
        };
    </script>
{% endblock %}
```

Appendix K - templates/register.html

```
{% extends 'base.html' %}

{% block content %}
    <h2>Register</h2>

    <br>

    <label for="username">Username</label><br>
    <input type="text" id="username_input" name="username" />

    <br><br>

    <label for="email">Email</label><br>
    <input type="email" id="email_input" name="email" />

    <br><br>

    <label for="password">Password</label><br>
    <input type="password" id="password_input" name="password" />

    <br><br>

    <label for="passwordc">Confirm Password</label><br>
    <input type="password" id="passwordc_input" name="passwordc" />
```

```
    <br><br>

    <button type="submit" onclick="register()">Register</button>

    <br><br>

    <a href="login">Already have an account? Click here to log in!</a>

    <script>
        if (currentUser != null) window.location.href = "/home";
    </script>
{% endblock %}
```

Appendix L - templates/ticket/new.html

```
{% extends 'base.html' %}

{% block content %}
    <h2>Open a new support ticket</h2>

    <br>

    <label for="ticket_title">Ticket Title</label><br>
    <input type="text" id="ticket_title_input" name="ticket_title" />

    <br><br>

    <label for="message">Ticket Description</label><br>
    <textarea type="text" id="message_input" name="message"
rows="10"></textarea>

    <br><br>

    <button type="submit" onclick="openTicket()">Open ticket</button>

    <br><br>

    <script>
        if (currentUser == null) window.location.href = "/login";
    </script>
{% endblock %}
```

Appendix M - templates/ticket/ticket.html

```
{% extends 'base.html' %}

{% block content %}
    <h2 id="ticket_title">...</h2>
    <p id="ticket_id">Ticket #{{ ticket_id }}</p>
```

```
    <p>
        <span id="opened_at"></span><br>
        <span id="closed_at"></span>
    </p>
    <p>
        <span id="opened_by">...</span><br>
        <span id="claimed_by">This ticket has not been claimed</span>
    </p>

    <br>

    <label for="message">Send a message:</label><br>
    <textarea type="text" id="message_input" name="message"
rows="10"></textarea>

    <br><br>

    <button id="close_ticket_btn" type="submit"
onclick="sendMessage('!close')">Close ticket</button>
    <button id="open_ticket_btn" type="submit"
onclick="sendMessage('!open')">Reopen ticket</button>
    <button id="send_message_btn" type="submit"
onclick="sendMessage()">Send</button>
    <button id="claim_ticket_btn" type="submit"
onclick="sendMessage('!claim')">Claim ticket</button>

    <br><br>

    <table id="ticket_messages"></table>

    <script>
        if (currentUser == null) window.location.href = "/login";

        const profiles = {};

        const ticketID = parseInt("{{ ticket_id }}");

        getTicket(ticketID).then(ticket => {
            if (ticket != false) {
                ticket = ticket["ticket_data"];
                document.getElementById("ticket_title").innerText = "[" +
(ticket["closed_at"] == -1 ? "OPEN" : "CLOSED") + "] " + ticket["title"];
                document.getElementById("opened_at").innerText = "Opened: "
+ new Date(ticket["opened_at"] * 1000).toLocaleString();
                if (ticket["closed_at"] != -1) {
                    document.getElementById("closed_at").innerText =
"Closed: " + new Date(ticket["closed_at"] * 1000).toLocaleString();

document.getElementById("open_ticket_btn").style.display = "inline";
                } else {

document.getElementById("close_ticket_btn").style.display = "inline";
                };
                getProfile(ticket["customer_id"]).then(customer => {
```

```
                    document.getElementById("opened_by").innerHTML =
"Opened by <b>" + customer["username"] + "</b> (ID: " +
ticket["customer_id"] + ")";
                });
                if (currentUser["account_type"] == ACCOUNT_TYPE_ASSISTANT
&& ticket["assistant_id"] != currentUser["user_id"]) {

document.getElementById("claim_ticket_btn").style.display = "inline";
                };
                if (ticket["assistant_id"] != -1) {
                    getProfile(ticket["assistant_id"]).then(assistant => {
                        document.getElementById("claimed_by").innerHTML =
"Claimed by <b>" + assistant["username"] + "</b> (ID: " +
ticket["assistant_id"] + ")";
                    });
                };
            };
        });

        const ticketMessagesTable =
document.getElementById("ticket_messages");

        getMessages(ticketID).then(async messages => {
            for (a_id of messages.map(m => m["author_id"])) {
profiles[a_id] = await getProfile(a_id); }

            for (msg of messages) {

                let _tr = document.createElement("tr");
                let _td1 = document.createElement("td");
                let _img = document.createElement("img");
                _img.src = profiles[msg["author_id"]]["profile_icon"];
                _img.width = 150;
                let _td2 = document.createElement("td");
                let _h3_sender = document.createElement("h3");
                _h3_sender.classList.add("message_sender");
                _h3_sender.innerText =
profiles[msg["author_id"]]["username"];
                let _p_body = document.createElement("p");
                _p_body.classList.add("message_body");
                _p_body.innerText = msg["body"];
                let _br1 = document.createElement("br");
                let _p_sent_at = document.createElement("p");
                _p_sent_at.classList.add("message_sent_at");
                _p_sent_at.innerText = new Date(msg["sent_at"] *
1000).toLocaleString();
                let _br2 = document.createElement("br");
                _td1.appendChild(_img);
                _td2.appendChild(_h3_sender);
                _td2.appendChild(_p_body);
                _td2.appendChild(_br1);
                _td2.appendChild(_p_sent_at);
                _td2.appendChild(_br2);
                _tr.appendChild(_td1);
                _tr.appendChild(_td2);
                ticketMessagesTable.appendChild(_tr);
```

```
            };
        });

    </script>
{% endblock %}
```

Appendix N - static/style.css

```
body {
    background-color: rgb(203, 255, 238);
    font-family: 'Barlow', 'Gill Sans Nova', 'Calibri';
    padding: 20px;
    text-align: center;
}

p {
    margin: 20px 0 0 0;
}

h1 {
    text-align: center;
    font-size: min(8vw, 45px);
    margin: 20px 0 0 0;
}

h2 {
    font-size: 30px;
    margin: 20px 0 0 0;
}

h3 {
    font-size: 25px;
    margin: 20px 0 0 0;
}

p, label, input, button, a, li {
    font-size: 20px;
}

p#loggedInMessage {
    position: absolute;
    right: 20px;
    top: 10px;
}

p#ticket_id {
    margin: 0 0 0 0;
}

ul {
    text-align: left;
}
```

```
input {
    width: min(225px, 80%);
}

textarea {
    width: min(225px, 80%);
}

button {
    padding: 10px 30px;
    margin: 5px 0 0 0;
}

button#close_ticket_btn, button#open_ticket_btn, button#claim_ticket_btn {
    display: none;
}

section#claimable_tickets {
    display: none;
}

table#ticket_messages tr img {
    padding-bottom: 20px;
}

table#ticket_messages tr td {
    text-align: left;
    padding: 0 0 0 20px;
    vertical-align: top;
}

table#ticket_messages tr td * {
    margin: 0;
}

table#ticket_messages p.message_sent_at {
    font-size: 15px
}

@media only screen and (max-width: 600px) {
    input {
        width: min(200px, 80%);
    }
}
```

REFERENCES

[i] Zendesk. *Zendesk Pricing.* https://www.zendesk.co.uk/pricing/#everyone (accessed 17/03/2023)
[ii] Internet Engineering Task Force (IETF). *JSON Web Token (JWT).* https://www.rfc-editor.org/rfc/rfc7519 (published May 2015, accessed 05/02/2023)
[iii] Virtue Security. *Application Penetration Testing – Username Enumeration*. https://www.virtuesecurity.com/kb/username-enumeration/ (accessed 05/02/2023)
[iv] Vaarun Sinha. *Why you should never use random module for generating passwords.* https://dev.to/vaarun_sinha/why-you-should-never-use-random-module-for-generating-passwords-38nl (last updated 07/11/2021, accessed 05/02/2023)