Integration Application Development Project

Module Name	DSE204/03 INTEGRATION APPLICATION DEVELOPMENT
Course Name	Bachelor of Software Engineering (Honours)
Start Date	12 June 2025
Submission Date	4 July 2025
Lecturer	Ms Grace Tok Bee Choo
Student ID	041240370
Student Name	Eng Wen Ping

Table of Content

Table of Content

Table of Content

Table of Content

Table of Figures

Table of Figures

Table of Figures

Table of Tables

Introduction

The Scope of the Project is to research of different existing APIs and **compare** for each of your examples and **evaluate** their suitability while also **identifying** any potential security issues and **build a login** with selected **API** in "Know-Your-Neighborhood" website application.

The Know-Your-Neighborhood website consists of the following key pages:

1. Home Page
2. Registration Page
3. Login Page with API link
4. Contact us Page
5. About us Page
6. Terms and Conditions Page

Customers can login using the existing API and fetch basic information such as name, email from API.

Tools Used in Development

The tools used are for the assignment are:

- Visual Studio Code
- Node v22.16.0, Npm v11.1.0, React v 19.1.0
- HTML, CSS, Java, JavaScript
- Eclipse STS 4.30.0 (Java 17)
- Postman
- XAMPP PhpMyAdmin

Task 1 APIs Concepts and Types

1.1 Role & Concepts of API

API (**Application Programming Interface**) is a set of **rules**, **protocols**, and **tools** that allows different software applications to **communicate** with **each other**, enabling them to **exchange data** or **perform functions** without exposing their internal workings. APIs define **how requests** should be **made**, how **data** should be **formatted**, and what **responses to expect**.

APIs are essential in modern software development to

1. Enable **diverse systems** to **work together**
2. Work **Efficiency**
3. Provide **Innovation** Opportunity to Other Developers
4. Company **Monetization** to generate revenue by providing access to their APIs

1.1.1 Existing API

- **Google Maps API**, Provides mapping, geocoding, and routing services for location-based applications.
- **Twitter API**, Enables access to tweets, user data, and posting capabilities for social media integration.
- **Stripe API**, Facilitates online payment processing for e-commerce platforms.
- **OpenWeatherMap API**, Delivers weather data for applications requiring real-time weather updates.
- **RESTful APIs**, Widely used in web services (e.g., GitHub API, Spotify API) due to their simplicity and scalability.

1.2 API and SDK

An SDK (**Software Development Kit**) is a broader set of tools, libraries, and documentation designed to **help developers build applications** for a specific platform or framework. SDKs often include APIs but provide additional resources like code samples, debugging tools, and emulators.

Tables of API vs SDK

Differences	API	SDK
Defination	Interface for communitcation between applications	A collection of tools, libraries and APIs for building applications
Scope	Focus on specific function or data exchange	Includes documentations and various development tools or feature
Usage	Used to access specific feature	Normally used to develop entire applications (Mobile Apps)

1.3 Types and Use of APIs

1.3.1 Access Model APIs

1. **Public APIs**, available for **third-party developers** to build their own **applications** such as OpenWeatherMap API for weather Data
2. **Private APIs**, used within **organization** to connect internal systems such as Database, HR and payroll systems
3. **Partner APIs**, shared with specific **partners** under controlled access usually used to facilitate **business-to-business integrations** such as Paypal and Stripe

1.3.2 Architecture APIs

1. **Rest (Representational State Transfer)**, Uses **HTTP methods** (GET, POST, PUT, DELETE) in web services, mobile apps and cloud applications such as Github API repository management.
2. **SOAP (Simple Object Access Protocol)**, XML Protocol used for **structured and secure communication** usually used in financial systems.
3. **GraphQL**, allow clients to **retrieve data flexibly** for complex applications such as social media platforms.
4. **gRPC**, A **high performance** RPC framework using HTTP/2 and P**rotocol Buffers** for **microservices** and **real-time applications** which are used for live straming services and E-commerce Platform

1.3.3 Functionality APIs

1. **Data APIs**, provide access to database or datasets.
2. **Functional APIs**, Enable specific actions like sending emails.
3. **Hardware APIs**, Interact with device hardware like cameras, sensors and microcontroller (Arduino UNO)

1.4 Potential Security Issues

1. **Authentication Weakness**, APIs without Authentication may lead to attacks to **data leaks** or perform **unauthorized actions.**
2. **Data Exposure**, Leaks sensitive information when returning full user profile data.
3. **Injection Attacks**, backend system ir data steal by the attackers throught unvalidated inputs
4. **Insecure Endpoints**, poor configured endpoints (Http instead of Https) might leads to data expose to intercept or attacks

1.5 Evaluation of Suitable API

1. **Security**, API used must have **OAuth 2.0** for authentication **use** and **http expect JWT Token** encrypt with **user information** to prevent data exposure, authentication weakness and insecure endpoints
2. **Scalabilty**, API must expect to **handle high transaction or request** during peak shopping periods for e-commerce or live-streaming platforms.
3. **Ease of integration**, the API should **integrate easily** with e-commerce and third-party services that **supports multiple** programming **languages** such as (Javscript, Python and Java)
4. **Data Privacy**, The APIs that **handling payment and user data** are required to **support tokenization** such as **JWT Tokens** to prevent data exposure
5. **Cost and Maintenance**, should be **cost-effective and easy to maintain** when developing small projects

Task 2 API Selection and Application Design

2.1 Scenario Analysis

The application is a web-based platform designed to provide more information “Know Your Neighbourhood” of ABC Condo. The application requires a user-friendly interface with seamless authentication and possible implementation of API Services.

The pages required to be developed are HomePage, Login with API link, Register, Contact Us, Terms and Conditions, About Us Page. Extra Pages to be developed are Profile and Blog Page.

2.1.1 Requirements

1. **Authentication**, secure user login with third-party OAuth provider to fetch basic information.
2. **API Integration**, Utilize APIs to fetch and display relevant data
3. **Security**, Ensure secure handling of user data and API communications.
4. **Frontend Development**, Build an interactive UI using **React JS**.
5. **Backend Development**, Build a backend to interact seamlessly to the database using **Spring Boot** and **JWT token implementation.**

2.1.2 Selected APIs

1. **Google OAuth 2.0**, Google Authentication Option for **Login and registration (first-time Login).**
2. **Facebook SDK Login**, **Alternative authentication** option using Facebook
3. **Google Geocoding API, Convert**s Condo **Static Address** to **geographic coordinates** used in weather API
4. **Google Geolocation API**, Determine **user location** based on IP or device data

5. **Google Weather API**, Provide **real-time weather data** for the user's neighbourhood

6. **Google Gemini API**, used to provide **information of "About Us"**, helps using in creating **post contents**, post **images** & **profile picture**.

2.2 Wireframe Developed

2.2.1 HomePage Wireframe

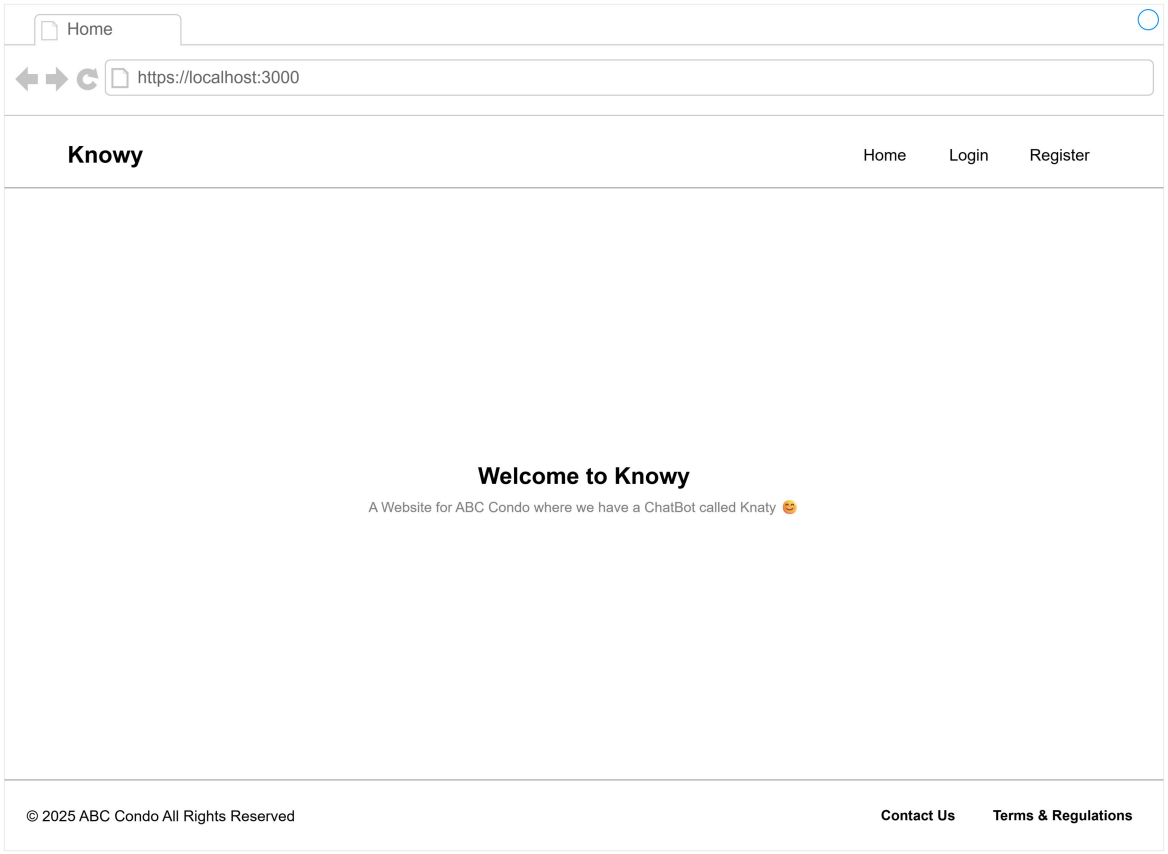

Figure 1 HomePage Wireframe

2.2.2 About Us Page

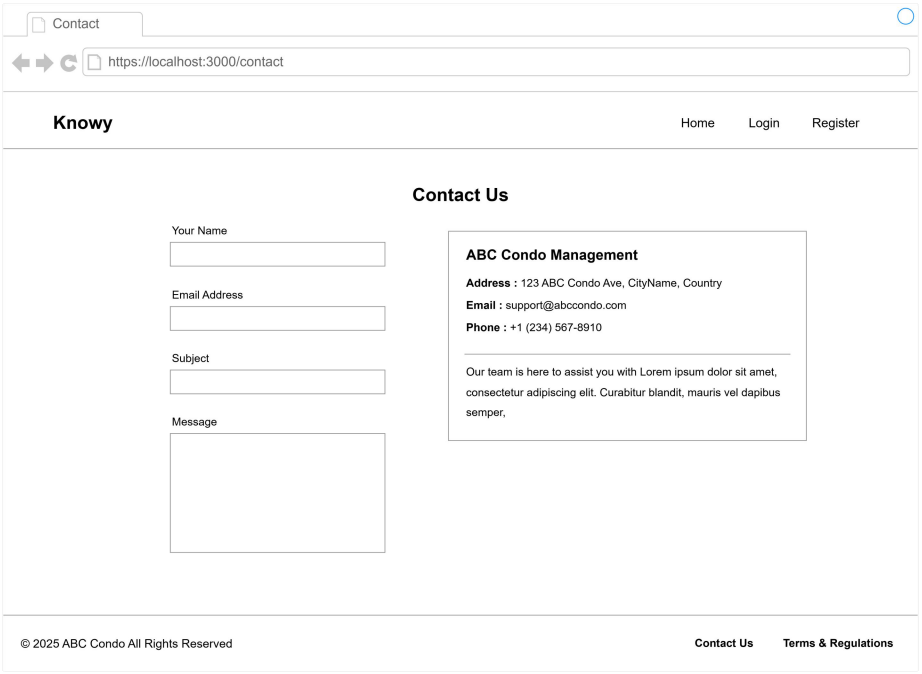

Figure 2 About Us Wireframe

2.2.3 Terms & Regulations Page

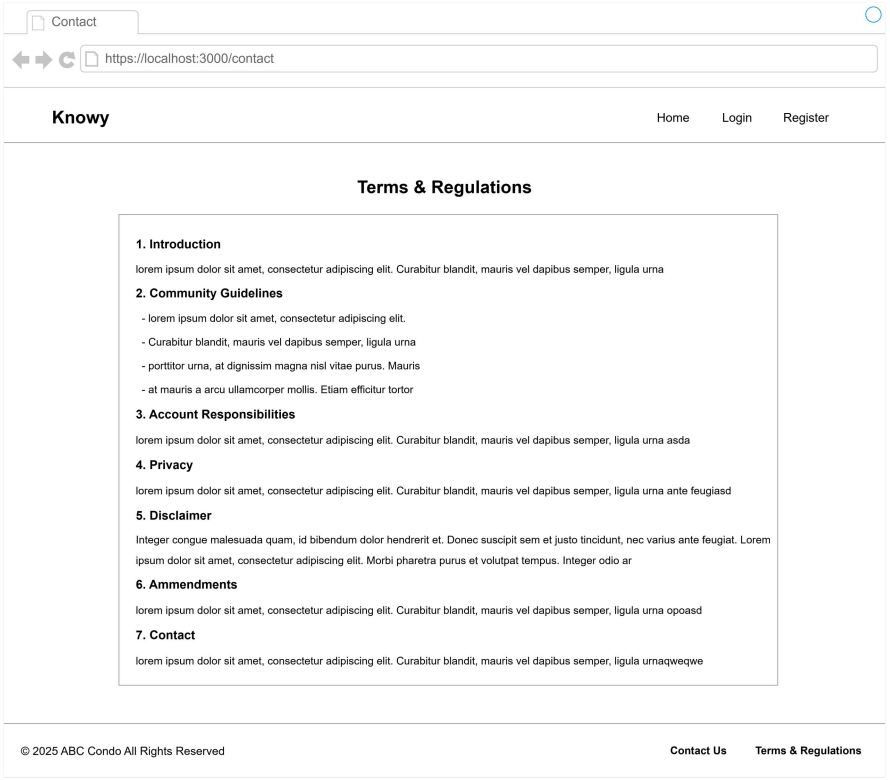

Figure 3 Terms & Regulations Wireframe

2.2.4 Login Page

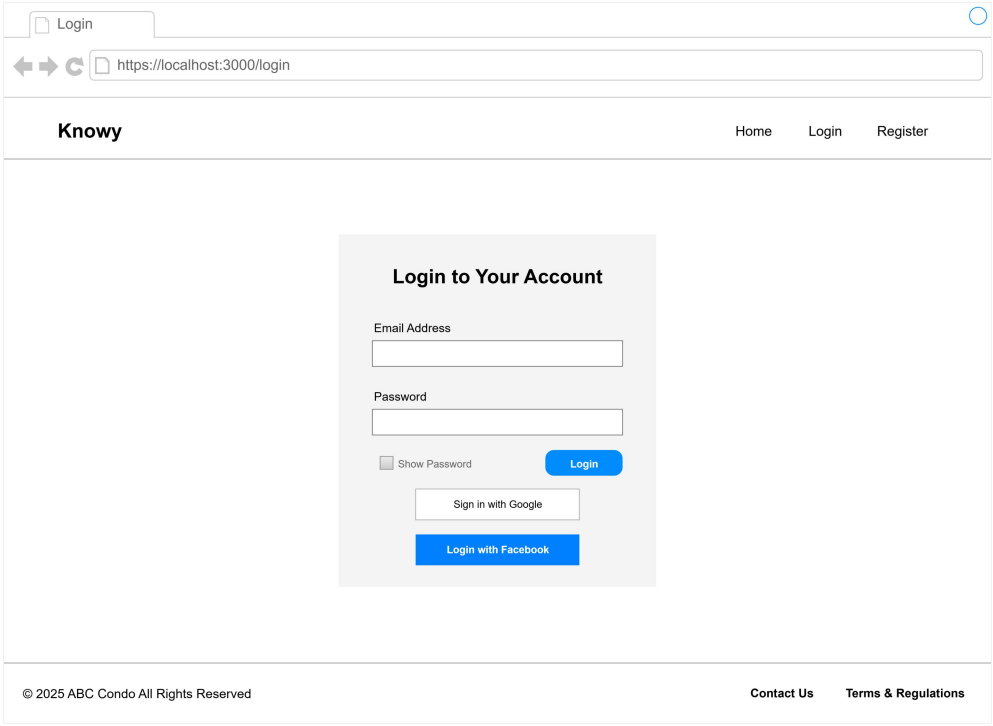

Figure 4 Login Wireframe

2.2.5 Register Page

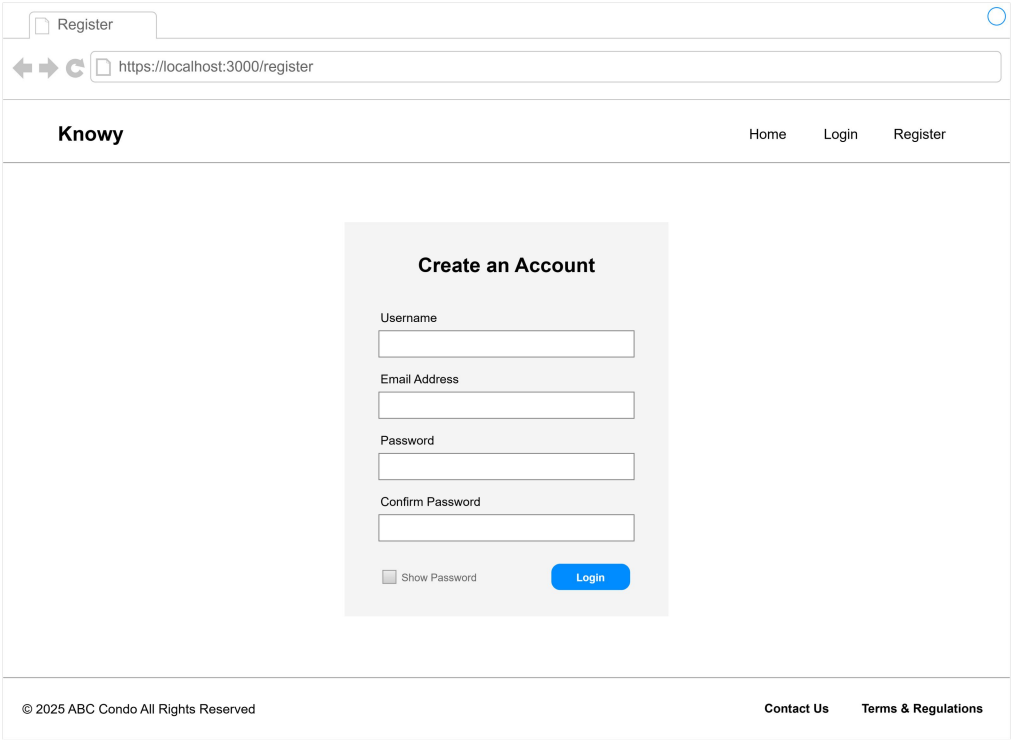

Figure 5 Register Wireframe

2.2.6 Blog Page

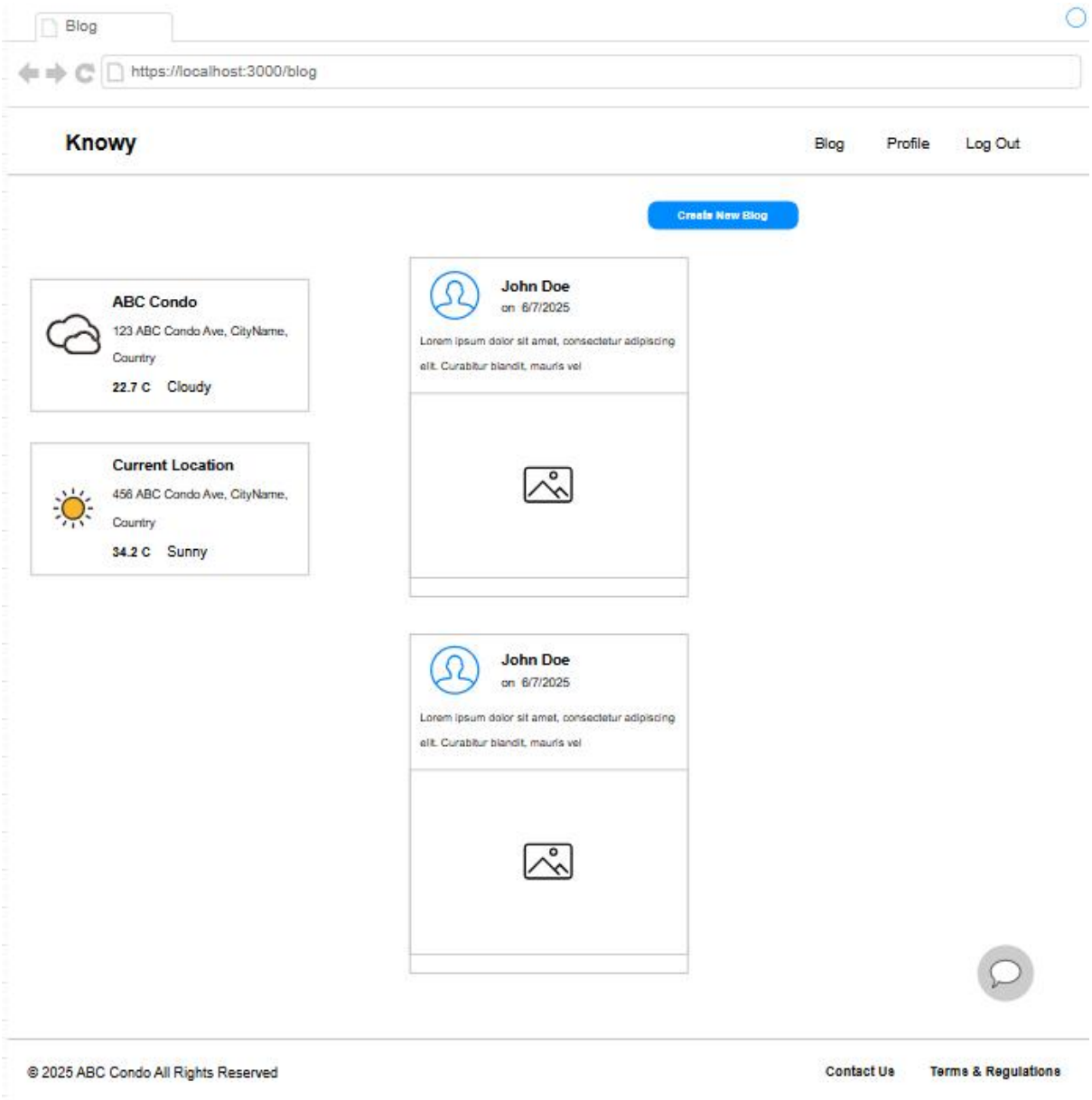

Figure 6 Blog Wireframe

2.2.7 Profile Page

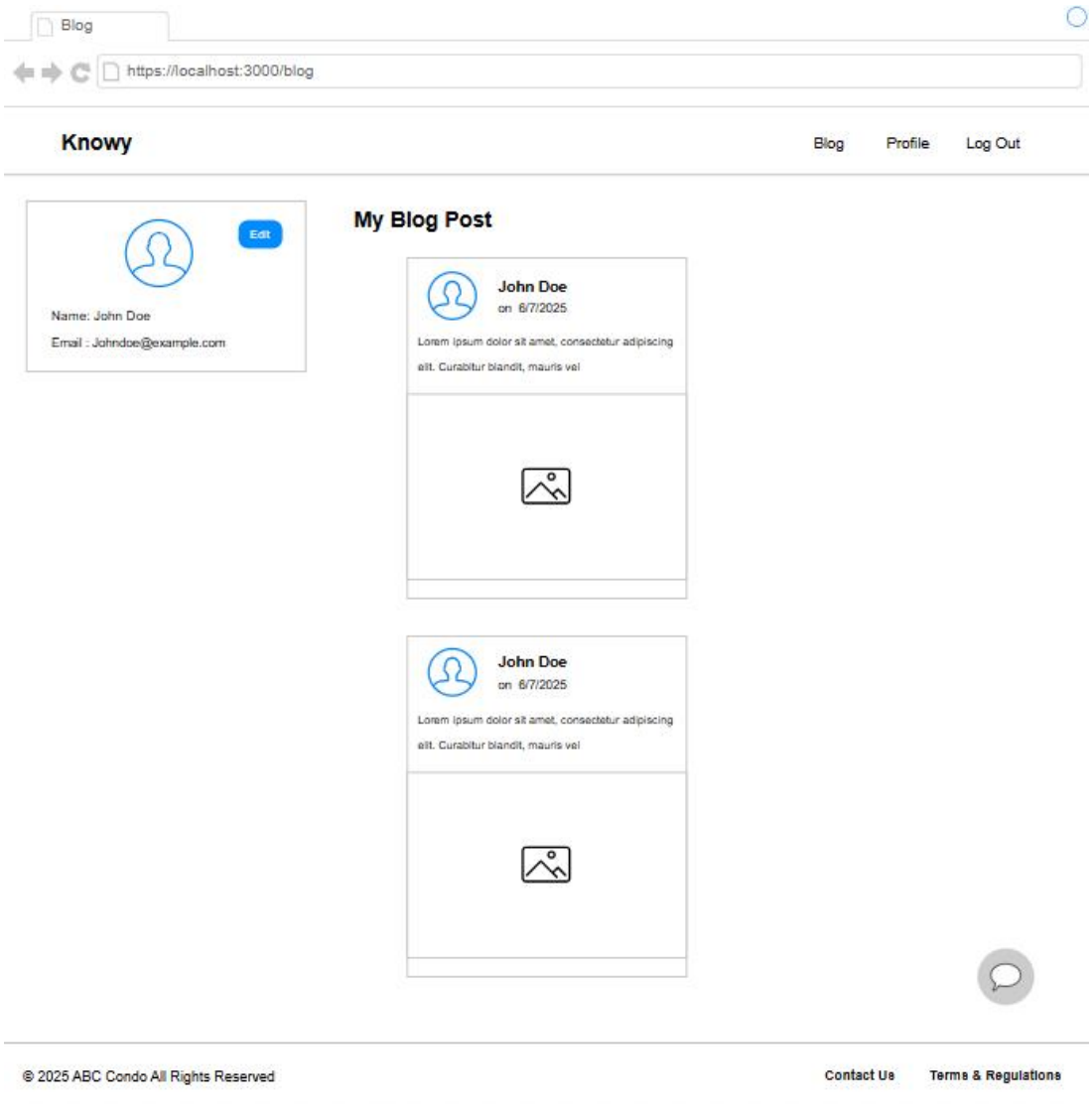

Figure 7 Profile Wireframe

2.2.8 About Us Page (Chatbot)

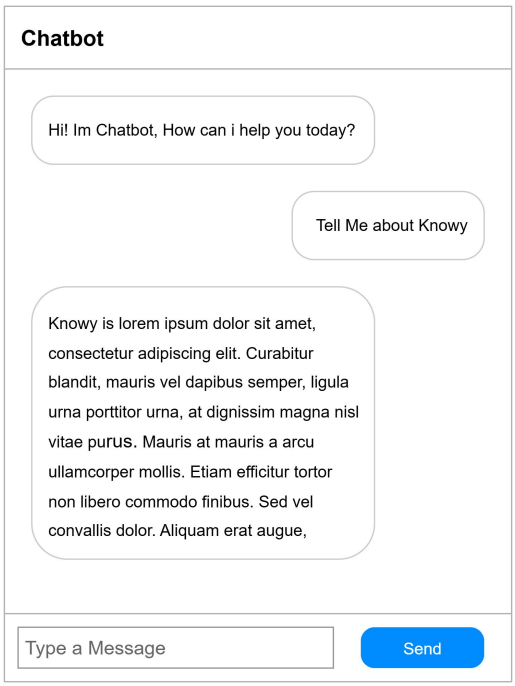

Figure 8 About Us Wireframe

2.3 Scope and Target Platforms

2.3.1 Scope

1. User authentication via Google OAuth 2.0 and Facebook SDK
2. Fetch and display user data (name, email, profile picture)
3. Display Condo location and user current location real-time weather condition
4. Offer AI-driven chatbot (provide “About Us” information, post contents, images if certain keyword were trigger from user question to chatbot)

2.3.2 Target Platforms

- Primary Platform: Web Browser
- Frontend Framework: React JS
- Backend Framework: Spring Boot
- Database : XAMPP PhpMyAdmin MySQL

2.4 Evaluation of Chosen APIs

2.4.1 Authentication API

Google OAuth 2.0 and Facebook SDK both **uses OAuth 2.0**, which employs secure **token-based authentication**, minimizing risk of credential theft over HTTPS ensuring encryption to permit **limit data access** to only what is necessary (name,email, profile picture) or based on what user has **set on Facebook Developer** and **Google Console**.

“ Google responds to this request by returning a JSON object that contains a short-lived access token and a refresh token.”(Using OAUth 2.0 for Web Server Applications, n.d.)

Figure 9 Google OAuth Token Security Proof

2.4.2 Google Geocoding, Geolocation, Weather and Gemini APIs

Google Geocoding, Geolocation, Weather API **prevent unauthorized access over HTTPS**, ensuring encryption for secure endpoints of the website. Data Privacy of Gemini API is **does'nt allow parameters** in the url link other than the key for security reinforcement and is **maintained by limiting AI interactions** to non-sensitive, non-political, contextual queries for Gemini APIs.

```
// Get user weather
try {
    const weatherResp = await axios.get(
        `https://weather.googleapis.com/v1/currentConditions:lookup?location.latitude=${lat}&location.l
    );
    setUserWeather(weatherResp.data);
} catch {
    setUserWeather(null);
}

// Get user address
try {
    const addrResp = await axios.get(
        `https://maps.googleapis.com/maps/api/geocode/json?latlng=${lat},${lng}&key=${GOOGLE_API_KEY}`
    );
    setUserAddress(addrResp.data.results[0]?.formatted_address || "");
} catch {
    setUserAddress("");
}
```

Figure 10 Google API https Proof

Task 3 Application Implementation

3.1 Three-Tier Architecture

3.1.1 Frontend Architecture Table

Framework	Description	Pros	Cons
React	A **JavaScript library** for building user interfaces, maintained by Meta. It uses a component-based architecture and a virtual DOM for efficient rendering.	Highly **flexible**, **large ecosystem**, **strong** community **support**, reusable components.	**Steep learning** curve for **beginners**, requires additional libraries for state management.
Vue	A progressive **JavaScript framework** for building UI, known for its simplicity and integration capabilities. It supports reactive data binding and component-based development.	**Easy to learn**, lightweight, good documentation, flexible for small to large projects.	**Smaller ecosystem** compared to React or Angular, fewer job opportunities.
Angular	A **Typescript-based** framework developed by Google, designed for building robust, large-scale applications with two-way data binding and dependency injection.	Comprehensive framework, strong typing with **TypeScript, built-in tools** for large apps.	**Complex syntax**, heavier footprint, **slower initial load** times.

3.1.2 Backend Architecture Table

Framework	Description	Pros	Cons
Spring Boot	A **Java-based framework** that simplifies the development of production-ready applications with minimal configuration. It includes embedded servers and extensive libraries..	Robust, scalable, great for microservices, **strong enterprise support**.	**Steeper learning** curve for **non-Java** developers, **heavier** resource **usage**.
Django	A high-level **Python web framework** that encourages rapid development and clean, pragmatic design. It includes an ORM and admin interface.	**Fast development**, secure by default, excellent documentation, built-in features like authentication..	**Less flexible** for non-standard projects, monolithic **structure** can be **limiting**.
Laravel	A **PHP framework** designed for elegant syntax and developer productivity, with features like Eloquent ORM and Blade templating.	Comprehensive framework, strong typing with TypeScript, built-in tools for large apps.	**Slower performance** compared to Django or Spring Boot, PHP-specific limitations.

3.1.3 API Implementation Process

3.1.3.1 Design and Planning

1. **Define API purpose and functionality**, Clearly outline the problem the API solves, the data it will handle, and the operations it will support.
2. **Choose an API style**, Select the appropriate architectural style (REST, SOAP, GraphQL, etc.) based on your needs and constraints.
3. **Define endpoints and resources**, Identify the specific URLs (endpoints) and the data structures (resources) that will be accessed through the API.
4. **Establish request/response formats**, Determine how data will be formatted in requests and responses (e.g., JSON, XML).
5. **Design for security**, Implement authentication, authorization, and encryption mechanisms to protect the API and its data.
6. **Consider performance and scalability**, Design the API to handle expected traffic and data volume efficiently.

3.1.3.2 Implementation

1. **Choose a technology stack**, Select the programming languages, frameworks, and libraries for building the API.
2. **Implement API endpoints and logic**, Write the code to handle requests, process data, and generate responses based on the API design.
3. **Develop API documentation**, Create clear and comprehensive documentation for developers, including endpoint descriptions, request/response formats, and usage examples.
4. **Implement API security**, Integrate authentication, authorization, and other security measures.

3.1.3.3 Testing

1. **Functional testing**, Verify that the API performs its intended functions correctly.
2. **Performance testing**, Evaluate the API's speed, responsiveness, and ability to handle load.
3. **Security testing**, Identify and address vulnerabilities in the API's security mechanisms.
4. **Integration testing**, Ensure the API integrates properly with other systems and applications.
5. **User acceptance testing (UAT)**, Involve users to test the API in a real-world scenario.

3.2 Development Environment

3.2.1 Frontend Development Environment (.env)

The .env file is used to configure environment-specific variables for frontend applications such as API endpoints, keys, and environment-specific settings, ensuring flexibility and security.

In my project my .env stores my backend url, facebook App Id, Google API Key (Geocoding, Geolocation, Weather), Gemini API and Condo Location for weather condition of the static condo.

Figure 11 React Application .env

3.2.2 Backend Development Environment (application.properties)

The application.properties file configures backend settings,database, server, and application-specific settings, such as database connections and server ports.

In my project my .env stores my application name, database url, username, password, jwt secret, JPA configurations, Google and Facebook Secret along with its app id and client id.

```
spring.application.name=backend

# Database connection

spring.datasource.url=jdbc:mysql://localhost:3306/abccondo
spring.datasource.username=root
spring.datasource.password=

# JWT Secret
jwt.secret=[redacted]

# JPA/Hibernate
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
server.port=8080

# Id & Secret
google.client-id=[redacted]
google.client-secret=[redacted]
facebook.app-id=[redacted]
facebook.app-secret=[redacted]
```

Figure 12 Spring Boot application.properties

3.3 API Integration

3.3.1 Frontend to Backend to API Integration

3.3.1.1 Google OAuth Authentication

1. When Click on the Google Login button , user are to fill in their google accounts and login, and receive the Google JWT Token. Then frontend will navigate to **`/api/auth/google`** in backend with the id Token.
2. In the backend ,the **`GoogleIdTokenVerifier`** will **verify the Id** with **googleClientID** to **ensure** the **token was intended** for the application. (if fail to get idToken, return with **error 401**)
3. Backend will then Extract the user information to the model and save it to the database using repository
4. Then it will return to frontend with the generated JWT token and save it to the localstorage of the frontend.

```
<div className="d-flex justify-content-center mt-2">
    <GoogleLogin
        onSuccess={handleGoogleLoginSuccess}
        onError={handleGoogleLoginError}
        useOneTap
    />
</div>
 // Google OAuth Success handler
 const handleGoogleLoginSuccess = async (credentialResponse) => {
    try {
        // Send Google ID token to your backend for verification
        const { credential } = credentialResponse;
        // Optionally decode for UI (not for security!)
        const decoded = jwtDecode(credential);
        console.log(decoded);
        const response = await axios.post(`${API_URL}/api/auth/google`, {
            idToken: credential,
        });
        localStorage.setItem('token', response.data.token); // Your backend's JWT
        setIsAuth(true);
        navigate('/', { replace: true });
    } catch (error) {
        setError('Google login failed');
        console.error('Google login error:', error);
    }
 };
```

Figure 13 OAuth Frontend Setup

```java
@PostMapping("/google")
public ResponseEntity<?> googleLogin(@RequestBody Map<String, String> body) {
    //recieve Google JWT Token from frontend
    String idTokenString = body.get("idToken");
    if (idTokenString == null) {
        return ResponseEntity.badRequest().body(Map.of("error", "Missing idToken"));
    }

    try {
        //verify the Google JWT Token
        GoogleIdTokenVerifier verifier = new GoogleIdTokenVerifier.Builder(new NetHttpTransport(),
                GsonFactory.getDefaultInstance()).setAudience(Collections.singletonList(googleClientId)).build();

        GoogleIdToken idToken = verifier.verify(idTokenString);
        if (idToken == null) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body(Map.of("error", "Invalid ID token"));
        }

        GoogleIdToken.Payload payload = idToken.getPayload();
        String email = payload.getEmail();
        String name = (String) payload.get("name");
        String picture = (String) payload.get("picture");

        // Find or create user in your DB
        Optional<UserModel> optionalUser = userRepo.findByEmail(email);
        UserModel user;
        if (optionalUser.isPresent()) {
            user = optionalUser.get();
        } else {
            user = new UserModel();
            user.setEmail(email);
            user.setName(name);
            user.setProvider("google");
            user.setPicture(picture);
            // You may want to generate a random password or leave blank
            userRepo.save(user);
        }

        // Generate your own JWT/session token
        String token = jwtUtil.generateToken(user.getId(), user.getEmail());

        return ResponseEntity.ok(Map.of("token", token));

    } catch (Exception e) {
        e.printStackTrace();
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(Map.of("error", "Google login failed"));
    }
`
```

Figure 14 Google OAuth 2.0 Integration

3.3.1.2 Facebook Authentication

1. At Frontend, it will first load the Facebook SDK as a **<script>** under **<head>**
2. After the user click on the Login with Facebook Button, it will first check the SDK is loaded and proceed to **/api/auth/facebook** in backend through axios with the **accesstoken** if true and wait the backend response with the data. If false, it will error log of "**Facebook SDK not loaded**".
3. Backend will first check if accesstoken is present. If not, it will return with **error 400**.
4. Then it will call the facebook api link and will create a new user with the name, email and profile image url into database users with `https://graph.facebook.com/me?fields=id,name,email,picture&access_token=" + accessToken`.
5. Then it will return to frontend with the **JWT token** and frontend will save it to the **localstorage** and set **isAuth** to **true** then navigate to the Home Page.

```
<div className="d-flex justify-content-center mt-0 ">
    <FacebookLoginButton className='social-button' onClick={handleFacebookLoginClick} />
</div>
```

Figure 15 Facebook Button

```
useEffect(() => {
    // Load Facebook SDK
    window.fbAsyncInit = function () {
        window.FB.init({
            appId: process.env.REACT_APP_FACEBOOK_APP_ID,
            cookie: true,
            xfbml: true,
            version: 'v19.0'
        });
    };


    // Check if Facebook JSSDK is found
    if (!document.getElementById('facebook-jssdk')) {
        // create <script>
        const script = document.createElement('script');
        // assign <script> value
        script.id = 'facebook-jssdk';
        script.src = 'https://connect.facebook.net/en_US/sdk.js';

        //assign to in the <head>
        document.head.appendChild(script);
    }
}, []);// run only once

const handleFacebookLoginClick = () => {
    if (!window.FB) {
        setError('Facebook SDK not loaded');
        return;
    }
    window.FB.login(function (response) {
        if (response.authResponse) {
            handleFacebookLoginResponse(response.authResponse);
        } else {
            setError('Facebook login was cancelled or failed');
        }
    }, { scope: 'email,public_profile' });
};

const handleFacebookLoginResponse = async (authResponse) => {
    try {
        const { accessToken } = authResponse;
        console.log(authResponse);
        if (accessToken) {
            const response = await axios.post(`${API_URL}/api/auth/facebook`, {
                accessToken
            });
            localStorage.setItem('token', response.data.token);
            setIsAuth(true);
            navigate('/', { replace: true });
        } else {
            setError('Facebook login failed: No access token');
        }
    } catch (error) {
        setError('Facebook login failed');
        console.error('Facebook login error:', error);
    }
};
```

Figure 16 Frontend Facebook SDK Handler Logic

```java
@PostMapping("/facebook")
public ResponseEntity<?> facebookLogin(@RequestBody Map<String, String> body) {
    String accessToken = body.get("accessToken");
    if (accessToken == null) {
        return ResponseEntity.badRequest().body(Map.of("error", "Missing accessToken"));
    }

    try {
        String userInfoUrl = "https://graph.facebook.com/me?fields=id,name,email,picture&access_token=" + accessToken;
        RestTemplate restTemplate = new RestTemplate();
        Map<String, Object> fbUser = restTemplate.getForObject(userInfoUrl, Map.class);

        String id = (String) fbUser.get("id");
        String email = (String) fbUser.get("email");
        String name = (String) fbUser.get("name");

        String picture = null;
        Object pictureObj = fbUser.get("picture");
        if (pictureObj instanceof Map) {
            Map<?, ?> pictureMap = (Map<?, ?>) pictureObj;
            Object dataObj = pictureMap.get("data");
            if (dataObj instanceof Map) {
                Map<?, ?> dataMap = (Map<?, ?>) dataObj;
                Object urlObj = dataMap.get("url");
                if (urlObj instanceof String) {
                    picture = (String) urlObj;
                }
            }
        }

        if (email == null) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED)
                    .body(Map.of("error", "Email permission is required"));
        }

        // Find or create user in your DB
        Optional<UserModel> optionalUser = userRepo.findByEmail(email);
        UserModel user;
        if (optionalUser.isPresent()) {
            user = optionalUser.get();
        } else {
            user = new UserModel();
            user.setEmail(email);
            user.setName(name);
            user.setProvider("facebook");
            user.setPicture(picture);
            userRepo.save(user);
        }

        // Generate your own JWT/session token
        String token = jwtUtil.generateToken(user.getId(), user.getEmail());

        return ResponseEntity.ok(Map.of("token", token));
    } catch (Exception e) {
        e.printStackTrace();
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                .body(Map.of("error", "Facebook login failed"));
    }
```

Figure 17 Backend Facebook API Handler Logic

3.3.2 Frontend API Integration

3.3.2.1 Google Geocoding / Geolocation API / Weather API

- **Google Geolocation** is used to get the **user current geographic location** with **`navigator.geolocation.getCurrentPosition`** only if the **`GOOGLE_API_KEY`** is present
- **Google Geocoding** will then get the **geographic location** of the **static address** (Condo address) for weather api call and get the **user current address location** for **UI display** in the weather card with **`setUserAddress`**.
- **Weather API** is used to fetch weather information with axios under `https://weather.googleapis.com/v1/currentConditions:lookup?location.latitude=${lat}&location.longitude=${lng}&key=${GOOGLE_API_KEY}` and then used in user and condo weather card with **`setUserWeather`** and **`setCondoWeather.`**

```
useEffect(() => {
    if (!CONDO_ADDRESS || !GOOGLE_API_KEY) return;

    const fetchCondoLocation = async () => {
        try {
            const locResp = await axios.get(
                `https://maps.googleapis.com/maps/api/geocode/json?address=${encodeURIComponent(CONDO_ADDRESS)}&key=${GC
            );
            const result = locResp.data.results[0];
            const lat = result.geometry.location.lat;
            const lng = result.geometry.location.lng;
            setCondoLatLng({ lat, lng });
            setCondoAddress(result.formatted_address);

            // 2. Get condo weather
            const weatherResp = await axios.get(
                `https://weather.googleapis.com/v1/currentConditions:lookup?location.latitude=${lat}&location.longitude=
            );
            setCondoWeather(weatherResp.data);

        } catch (err) {
            setErrorMsg("Unable to load condo location or weather.");
        }
    };

    fetchCondoLocation();
}, [CONDO_ADDRESS, GOOGLE_API_KEY]);
```

Figure 18 Condo Google API integration

```
useEffect(() => {
    if (!GOOGLE_API_KEY) return;

    const fetchUserLocation = () => {
        if (!navigator.geolocation) {
            setErrorMsg("Geolocation is not supported by this browser.");
            return;
        }
        navigator.geolocation.getCurrentPosition(
            async (position) => {
                const lat = position.coords.latitude;
                const lng = position.coords.longitude;
                setUserLatLng({ lat, lng });

                // Get user weather
                try {
                    const weatherResp = await axios.get(
                        `https://weather.googleapis.com/v1/currentConditions:lookup?location.latitude=${lat}&location.
                    );
                    setUserWeather(weatherResp.data);
                } catch {
                    setUserWeather(null);
                }

                // Get user address
                try {
                    const addrResp = await axios.get(
                        `https://maps.googleapis.com/maps/api/geocode/json?latlng=${lat},${lng}&key=${GOOGLE_API_KEY}`
                    );
                    setUserAddress(addrResp.data.results[0]?.formatted_address || "");
                } catch {
                    setUserAddress("");
                }
            },
            (error) => {
                setErrorMsg("Unable to get your location.");
            }
        );
    };

    fetchUserLocation();
}, [GOOGLE_API_KEY]);
```

Figure 19 User Google API Integration

3.3.2.2 Google Gemeni AI API

The Gemini API is implement in **Knatty.js** with **`{ GoogleGenAI, Modality }`** from "**`@google/genai`**" (Using Gemini API Keys, n.d.). With this, user can ask anything to the chatbot as it will respond as how a Gemini AI does.

```
const genAI = useRef(null);

const GEMINI_API_KEY = process.env.REACT_APP_GEMINI_API_KEY;

if (!genAI.current && GEMINI_API_KEY) {
  genAI.current = new GoogleGenAI({ apiKey: GEMINI_API_KEY });
}
```

Figure 20 Gemini API Setup

GoogleGenAI and Modality were used for API called different Gemini API models along with their Mapped of object or string as it doesn't allow parameters for better security and logging purposes.

```
const response = await genAI.current.models.generateContent({
  model: "gemini-2.5-flash",
  contents: userMsg,
});
```

Figure 21 Gemini Text Model Setup

```
const response = await genAI.current.models.generateContent({
  model: "gemini-2.0-flash-preview-image-generation",
  contents: contents,
  config: {
    responseModalities: [Modality.TEXT, Modality.IMAGE],
  }
});
```

Figure 22 Gemini Image Model Setup

Gemini API were customized to perform **Message Handling and Conditional Logic** below:

```
const isAboutUsQuestion = (msg) => {
  const aboutUsKeywords = ["about you", "knowy", "abc condo", "about us information", "who are you", "who is knatty"];
  return aboutUsKeywords.some((kw) => msg.toLowerCase().includes(kw));
};

const isPostContentRequest = (msg) => /post|blog|refine|improve|generate.*content/.test(msg.toLowerCase());
const isImageRequest = (msg) => /image|picture|photo|draw|avatar|profile picture|generate.*image/.test(msg.toLowerCase());
```

Figure 23 Keywords Logic for About Us, Blog Post, Image Gen

1. **About Us Questions** (Non-API),

   If the user's message contain keywords of **(about you, knowy, abc condo, about us information, who are you, who is knatty)**. It will return with a predefined response (**ABOUT_US_INFO**) without calling the Gemini API. This is to handles simple queries to introduce on what the website and chatbot does.

```
const ABOUT_US_INFO = `
Knowy, ABC Condo is a vibrant community located in the heart of the city. Our mission is t
We value inclusivity, safety, and a warm welcome for all newcomers. Knatty, was a friendly
`;
```

Figure 24 Predefined About Us Information

```
// About Us
if (isAboutUsQuestion(userMsg)) {
  setMessages((msgs) => [
    ...msgs,
    {
      from: "knatty",
      text: "Here's some information about us:\n\n" + ABOUT_US_INFO,
    },
  ]);
  setLoading(false);
  setTimeout(scrollToBottom, 10);
  return;
}
```

Figure 25 About Us Question Logic

2. **Blog Post Content,**

   similar logic as normal text prompt but if user message contains of **(post, blog, refine, improve, generate content).toLowerCase()** and is not logged in It will then ask the user to first login to use this feature. If logged in, it will generate content related with Blog Formats which will contains Emojis, Hashtag (#) and others social media post style elements.

```
// Blog Post Generation/Refine (require login)
if (isPostContentRequest(userMsg)) {
  if (!isAuth) {
    setMessages((msgs) => [
      ...msgs,
      {
        from: "knatty",
        text: "To help you generate or refine post content, please login first.",
        loginPrompt: true,
      },
    ]);
    setLoading(false);
    setTimeout(scrollToBottom, 10);
    return;
  }
}
```

Figure 26 Gemini Post Generation Logic

```
  // Text Generation (general, blog, etc.)
  try {
    const response = await genAI.current.models.generateContent({
      model: "gemini-2.5-flash",
      contents: userMsg,
    });
    const text = response.text;
    console.log(response.text);
    setMessages((msgs) => [...msgs, { from: "knatty", text }]);
  } catch (err) {
    setMessages((msgs) => [
      ...msgs,
      { from: "knatty", text: "Sorry, I couldn't process your request. Please try again later." },
    ]);
    console.log(err);
  }
  setLoading(false);
  setTimeout(scrollToBottom, 10);
};
```

Figure 27 Gemini Text Respond Logic

3. **Image Generation**,

   if the user message matches image-related keywords **`(image, picture, photo, draw, avatar, profile picture, generate image).toLowerCase()`.** Similar with Post, it ask the user to **login before using the feature**. If **logged in**,It will called the image model API and will expect the model to **generate an image with a text response** because of the **ResponseModalities include "Image" and "text".** It will then set the image as a chat message from Knatty in the chatbot with the text and image responded from the Gemini API.

```
// Image Generation (require login)
if (isImageRequest(userMsg)) {
  if (!isAuth) {
    setMessages((msgs) => [
      ...msgs,
      {
        from: "knatty",
        text: "To generate images, please login first.",
        loginPrompt: true,
      },
    ]);
    setLoading(false);
    setTimeout(scrollToBottom, 10);
    return;
  }
  try {
    const contents = userMsg;
    const response = await genAI.current.models.generateContent({
      model: "gemini-2.0-flash-preview-image-generation",
      contents: contents,
      config: {
        responseModalities: [Modality.TEXT, Modality.IMAGE],
      }
    });


    const candidate = response.candidates?.[0];
    const parts = candidate?.content?.parts || [];
    const imagePart = parts.find(part => part.inlineData && part.inlineData.mimeType.startsWith("image/"));
    const textPart = parts.find(part => part.text);
    if (imagePart) {
      const { data, mimeType } = imagePart.inlineData;
      const imageUrl = `data:${mimeType};base64,${data}`;
      setMessages((msgs) => [
        ...msgs,
        {
          from: "knatty",
          text: "Here is your generated image, " +  (textPart?.text || ""),
          imageUrl,
        },
      ]);
    } else {
      setMessages((msgs) => [
        ...msgs,
        {
          from: "knatty",
          text: "Sorry, I couldn't generate an image for your request.",
        },
      ]);
    }
  } catch (err) {
    setMessages((msgs) => [
      ...msgs,
      { from: "knatty", text: "Sorry, I couldn't generate the image. Please try again later." },
    ]);
  }
  setLoading(false);
  setTimeout(scrollToBottom, 10);
  return;
}
```

Figure 28 Gemini Image Generation Logic

Task 4 Application Testing

4.1 Whitebox Testing

White box testing usually used for backend testing involves testing an application with **detailed inside information** of its **source code, architecture** and **configuration**. It can expose issues like security vulnerabilities, broken paths or data flow issues, which **increase the development logic** for the application. For my case, I use **JUnit testing** for **Spring Boot Whitebox testing**. Total of 12 Whitebox Testing had been tested shown in below.

4.1.1 Login testing

Test Objective:

Verify the each of the **endpoint** correctly **verifies a valid token**, finds or creates the user, and returns a **JWT token** for **API Logins**. On the other hand, **Non-API Logins** requires verification on the endpoint to **authenticates existing user** (email/password), **issues JWT**, and **fails on invalid credentials**.

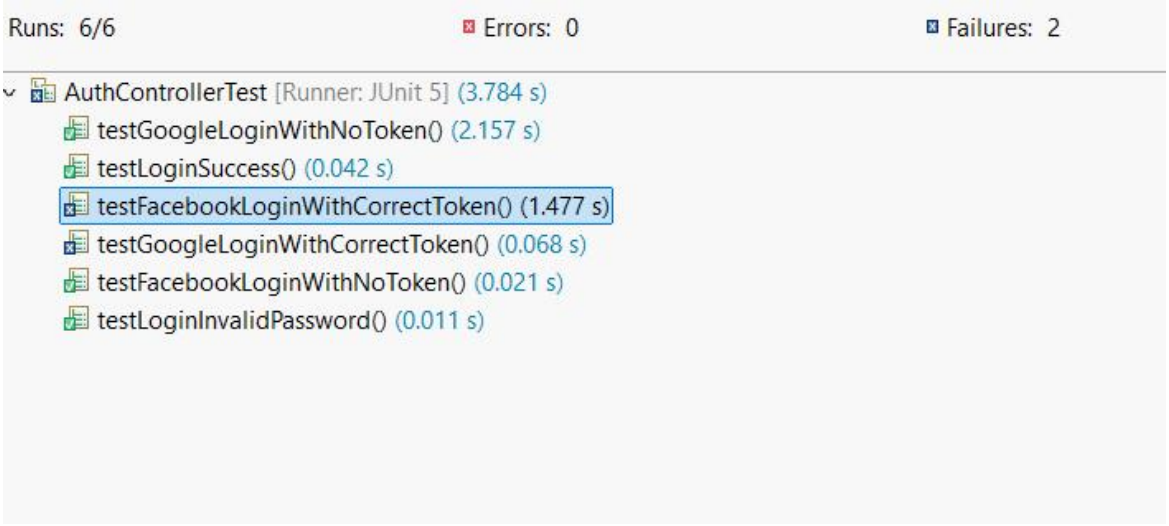

Figure 29 All Login testing Results

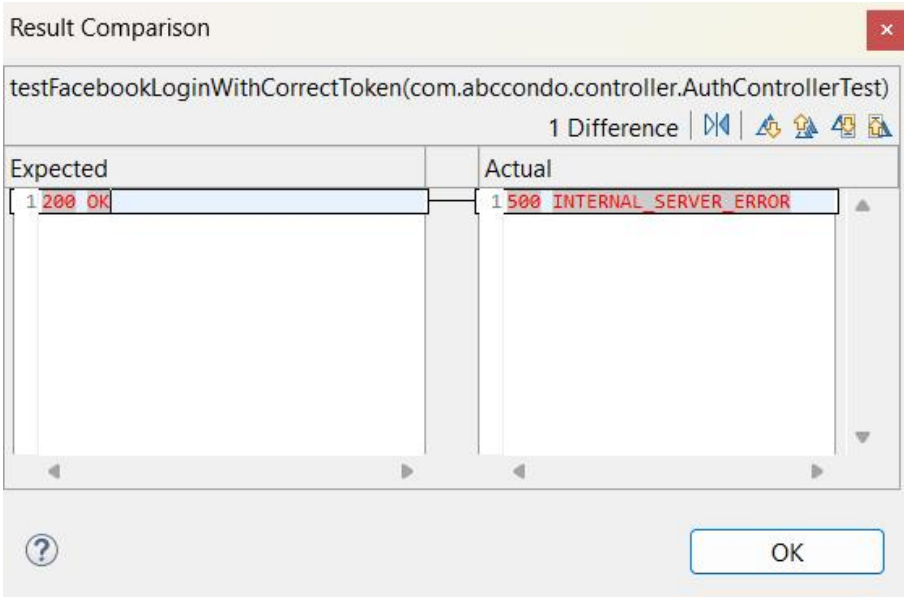

Figure 30 Facebook Login with Correct Token Result

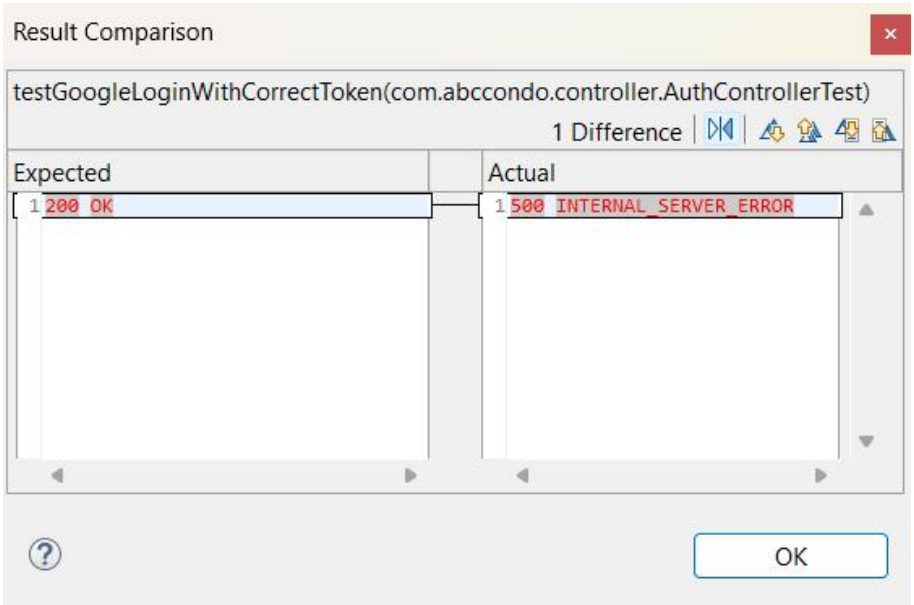

Figure 31 Google Login with Correct Token Result

Login Whitebox Testing Table

Test Case	Scenario	Expected Outcome	Explanation
WB_TC_01	Login Success	200 OK	Return a map with a valid JWT Token
WB_TC_02	Login with Invalid Password	500 Internal Server Error	Invalid Password
WB_TC_03	Google Login with Correct Token	500 Internal Server Error	Accept 500 is because the /google endpoint itself are require to make verifications to the token received from the frontend, whereas cant be implemented in JUnit Testing with a fake idToken
WB_TC_04	Google Login without Token	400 Bad Request, missing idToken	Missing Token
WB_TC_05	Facebook Login with Correct Token	500 Internal Server Error	Accept 500 is because the /facebook endpoint itself are require to make verifications to the token received from the frontend, whereas cant be implemented in JUnit Testing with a fake accessToken.
WB_TC_06	Facebook Login without Token	400 Bad Request, missing accessToken	Missing Token

```java
@Test
void testGoogleLoginWithCorrectToken() {
    // Arrange
    Map<String, String> body = new HashMap<>();
    body.put("idToken", "validGoogleToken");
    body.put("email", "test@example.com");
    body.put("name", "Test User");
    body.put("picture", "http://example.com/picture.jpg");

    when(userRepo.findByEmail("test@example.com")).thenReturn(Optional.of(testUser));
    when(jwtUtil.generateToken(1L, "test@example.com")).thenReturn("jwtToken");

    // Act
    ResponseEntity<?> response = authController.googleLogin(body);

    // Expect 500, this is because GoogleTokenVerifier cannot verify the Test token as it is a FAKE token
    // Will leave expect 200 because if the verification works, it should perform actions as below
    assertEquals(HttpStatus.OK, response.getStatusCode());
    assertEquals(Map.of("token", "jwtToken"), response.getBody());
    verify(userRepo).findByEmail("test@example.com");
    verify(jwtUtil).generateToken(1L, "test@example.com");
    verify(userRepo, never()).save(any(UserModel.class));
}
```

Figure 32 Google Login with Correct Token Test Logic

```java
@Test
void testFacebookLoginWithCorrectToken() {
    // Arrange
    Map<String, String> body = new HashMap<>();
    body.put("accessToken", "validFacebookToken");
    body.put("email", "test@example.com");
    body.put("name", "Test User");
    body.put("picture", "http://example.com/picture.jpg");

    when(userRepo.findByEmail("test@example.com")).thenReturn(Optional.of(testUser));
    when(jwtUtil.generateToken(1L, "test@example.com")).thenReturn("jwtToken");

    // Act
    ResponseEntity<?> response = authController.facebookLogin(body);

    // Expect 500, this is because FacebookTokenVerifier cannot verify the Test token as it is a FAKE token
    // Will leave expect 200 because if the verification works, it should perform actions as below
    assertEquals(HttpStatus.OK, response.getStatusCode());
    assertEquals(Map.of("token", "jwtToken"), response.getBody());
    verify(userRepo).findByEmail("test@example.com");
    verify(jwtUtil).generateToken(1L, "test@example.com");
    verify(userRepo, never()).save(any(UserModel.class));
    verify(restTemplate, never()).getForObject(anyString(), any());
}
```

Figure 33 Facebook Login with Correct Token Test Logic

4.1.2 Register testing

Test Objective:

Verify the user registration can be success, since there is **no validation** for the forms, the test case will only proceed with registering an account.

Register Whitebox Testing Table

Test Case	Scenario	Expected Outcome	Explanation
WB_TC_07	Register Success	**200** OK	Test email and password were verified in the test database and no JWT operations performed

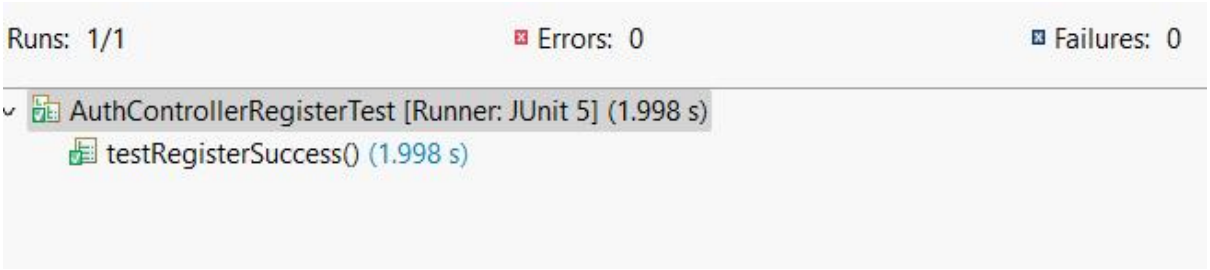

Figure 34 Register Testing results

```
@Test
void testRegisterSuccess() {
    // Arrange
    Map<String, String> body = new HashMap<>();
    body.put("email", "newuser@example.com");
    body.put("name", "New User");
    body.put("password", "password123");

    when(userRepo.findByEmail("newuser@example.com")).thenReturn(Optional.empty());
    when(passwordEncoder.encode("password123")).thenReturn("encodedPassword");
    when(userRepo.save(any(UserModel.class))).thenAnswer(invocation -> invocation.getArgument(0));

    // Act
    Map<String, String> response = authController.register(body);

    // Assert
    assertEquals("registered", response.get("status"));

    // check if the database has the test email and password using userRepo and verify no JWT interactions
    verify(userRepo).findByEmail("newuser@example.com");
    verify(passwordEncoder).encode("password123");
    verify(userRepo).save(any(UserModel.class));
    verifyNoInteractions(jwtUtil);
```

Figure 35 Register Testing Logic

4.1.3 Blog testing

Test Objective:

Verify the blog information are return to frontend and can update the database with the CRUD function requested from frontend. Since there are no validation, the expected outcome are only in success.

Blog Whitebox Testing Table

Test Case	Scenario	Expected Outcome	Explanation
WB_TC_08	Display Blog List	**200** OK	Verify the JWT Token, **blog listed** and ensure that jwtutil, userRepo, BlogRepo has **none unverified actions**
WB_TC_09	Edit Blog Successfully	**200** OK	Verify the JWT Token, **blog data is updated** and ensure that jwtutil, userRepo, BlogRepo has **none unverified actions**
WB_TC_10	Delete Blog Successfully	**200** OK	Verify the JWT Token, **blog is deleted** and ensure that jwtutil, userRepo, BlogRepo has **none unverified actions**

Figure 36 Blog Testing results

4.1.4 Profile testing

Test Objective:

Verify the blog information of only with the user return to frontend and update the user information.

Profile Whitebox Testing Table

Test Case	Scenario	Expected Outcome	Explanation
WB_TC_11	Display User Blog Only	**200** OK	Verify the JWT Token, **blog listed** with the **author** and ensure that jwtutil, userRepo, BlogRepo has **none unverified actions**
WB_TC_12	Update User Information	**200** OK	Verify the JWT Token, **User Data is Updated** and ensure that jwtutil, userRepo, BlogRepo has **none unverified actions**

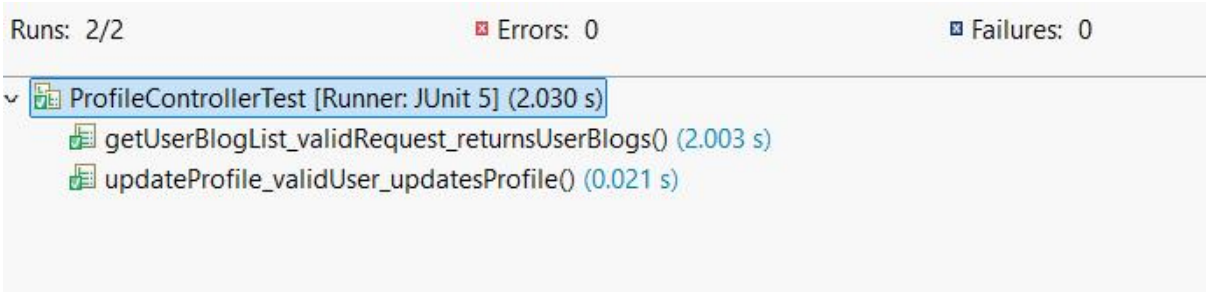

Figure 37 Profile Testing results

4.2 Blackbox API Testing

Black box testing used in frontend testing involves testing a system with **no prior knowledge** of its internal workings. A tester **provides an input**, and **observes the output** generated by the system under test. This makes it possible to **identify** how the system **responds to expected** and **unexpected** user **actions**, its response time, usability issues and reliability issues.

4.2.1 Get Location Testing

Verify the API can :

1. Get user current geographic location with **Geolocation API**
2. Convert geographic location into address and vice-versa address into geographic location with **Geocoding API**

Location Blackbox Testing Table

Test Case	Scenario	Expected Output	Explanation
BB_TC_01	Valid geolocation request with user permission	{lat: number, lng: number}	Receive a JSON Map with lat and lng variable
BB_TC_02	User denies location permission	errorMsg set to "Unable to get your location."	Without User permissions, Google are unable to track the user current location
BB_TC_03	Convert Geographic location into address	{formatted_address: address}	Receive a JSON Map with formatted_address variable
BB_TC_04	Convert Address into Geographic location	{navigation_points: {location: {lat : number, lng: number}}}	Receive a JSON Map with the lat and lng variable

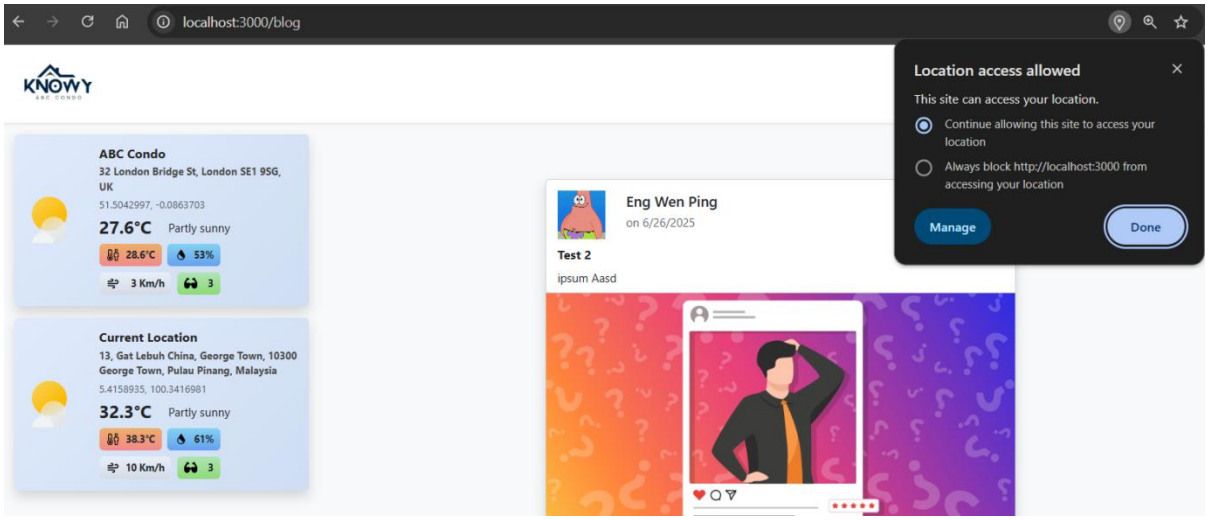

Figure 38 BB_TC_01, Permission Access

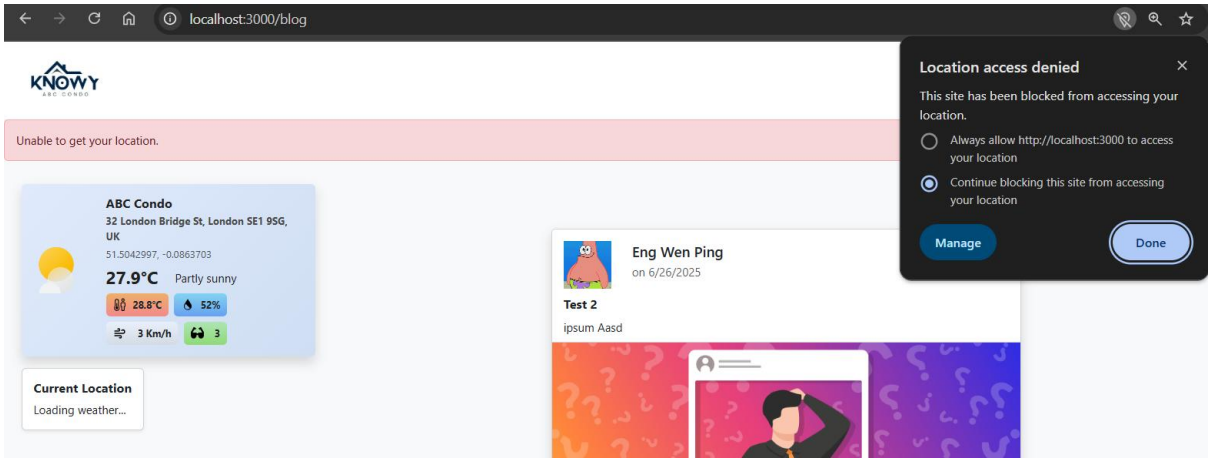

Figure 39 BB_TC_02, Permission Denied

Figure 40 BB_TC_03, JSON Map of formatted_address

Figure 41 BB_TC_04, JSON Map of navigation_points

4.2.2 Get Weather Testing

Verify that Weather API is working functionally

Weather Blackbox Testing Table

Test Case	Scenario	Expected Output	Explanation
BB_TC_05	Get Current Weather with the latitude and longitude provided	JSON Map of Weather Conditions	Receive a JSON Map with bunch of Weather variables

```
{
  "currentTime": "2025-07-01T07:50:25.797906208Z",
  "timeZone": {
    "id": "Asia/Kuala_Lumpur"
  },
  "isDaytime": true,
  "weatherCondition": {
    "iconBaseUri": "https://maps.gstatic.com/weather/v1/party_cloudy",
    "description": {
      "text": "Partly sunny",
      "languageCode": "en"
    },
    "type": "PARTLY_CLOUDY"
  },
  "temperature": {
    "degrees": 32.4,
    "unit": "CELSIUS"
  },
  "feelsLikeTemperature": {
    "degrees": 38.4,
    "unit": "CELSIUS"
  },
  "dewPoint": {
    "degrees": 23.7,
    "unit": "CELSIUS"
  },
  "heatIndex": {
    "degrees": 38.4,
    "unit": "CELSIUS"
  },
  "windChill": {
    "degrees": 32.4,
    "unit": "CELSIUS"
  },
  "relativeHumidity": 61,
  "uvIndex": 4,
  "precipitation": {
    "probability": {
      "percent": 6,
      "type": "RAIN"
    },
    "snowQpf": {
      "quantity": 0,
      "unit": "MILLIMETERS"
    },
    "qpf": {
      "quantity": 0.0457,
      "unit": "MILLIMETERS"
    }
  },
  "thunderstormProbability": 0,
  "airPressure": {
    "meanSeaLevelMillibars": 1007.06
  },
  "wind": {
    "direction": {
      "degrees": 225,
      "cardinal": "SOUTHWEST"
    },
    "speed": {
      "value": 8,
      "unit": "KILOMETERS_PER_HOUR"
```


Figure 42 BB_TC_05, JSON Map of Weather Information

4.2.3 Gemini Chatbot Testing

Verify the Chatbot can :

1. Generate Post Content
2. Generate Non-sensitive, non political, branding Images
3. Provide About Us and Chatbot Information
4. Respond like how an AI does

Chatbot Blackbox Testing Table

Test Case	Scenario	Expected Output	Explanation
BB_TC_06	Greetings with introduction of the user	Knatty Respond user with greetings	Respond User with Google AI, Gemini
BB_TC_07	Ask the chatbot to generate non-sensitive image with user logged in	Knatty will Provide the image requested	Generate with Google Trained Image Model
BB_TC_08	Ask the chatbot to generate image with user logged out	Knatty ask the user to login with a login button	Only logged-in user can use the feature
BB_TC_09	Ask the chatbot to generate content for a post with user logged in	Knatty will Provide the Post Content requested	Generate with Google Trained Text Model
BB_TC_10	Ask the chatbot to generate post content with user logged out	Knatty ask the user to login with a login button	Only logged-in user can use the feature
BB_TC_11	Ask the Chatbot About Knowy	About Us information	Preset About Us information

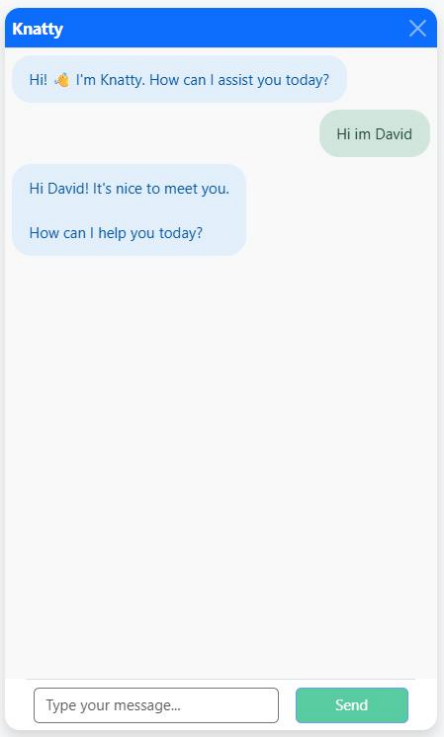

Figure 43 BB_TC_06, User Greeting

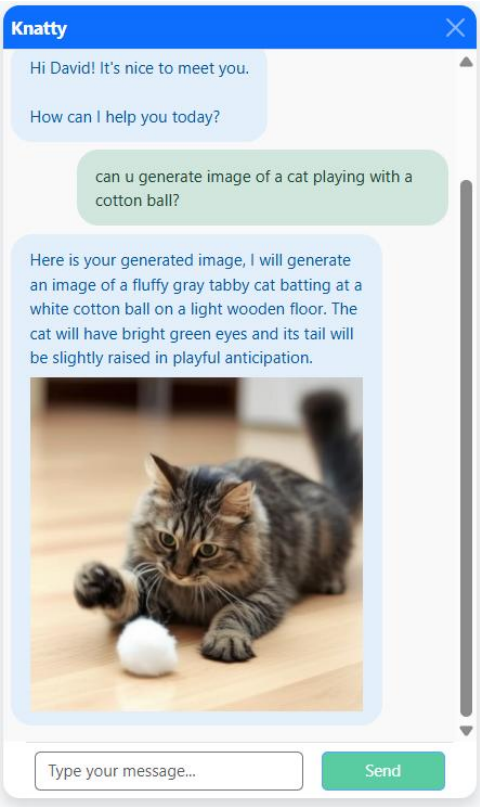

Figure 44 BB_TC_07, Generate valid Image

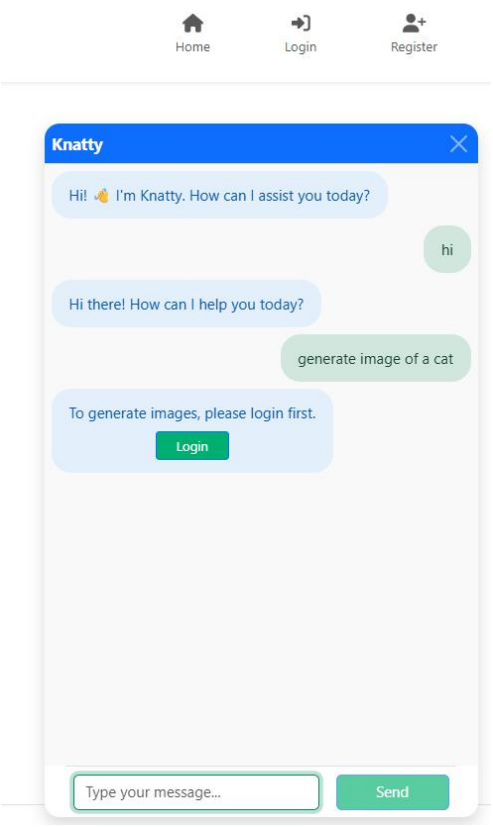

Figure 45 BB_TC_08, Generate image while logged out

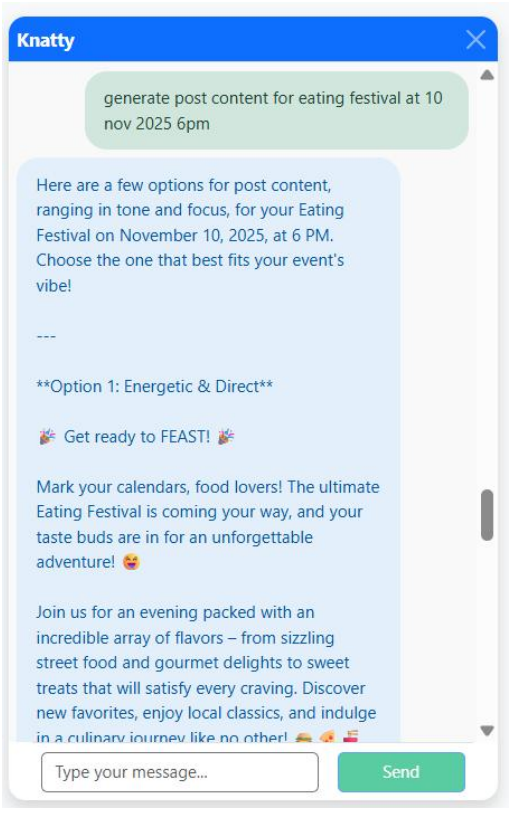

Figure 46 BB_TC_09, Generate post content

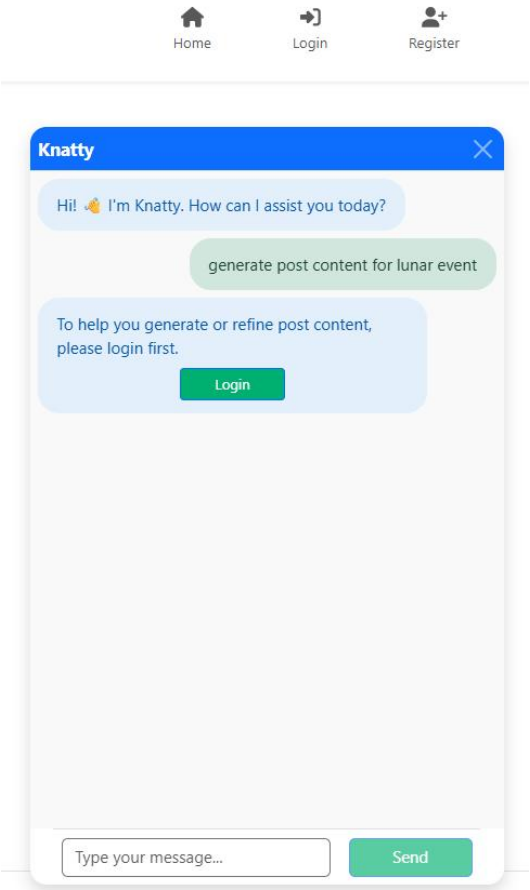

Figure 47 BB_TC_10, Generate post content while logged out

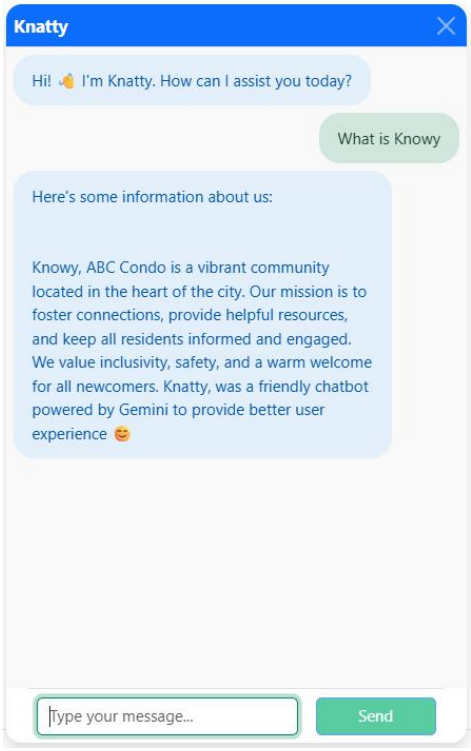

Figure 48 BB_TC_11, Provide Knowy Information

Task 5 API Review

5.1 Strength & Weakness of API Table

API	Strength	Weakness
Google OAuth2.0	- OAuth 2.0 industry standard - Granular scope control and token revocation - Secure flow using HTTPS + JWT - tutorials and docs support - Easy integration with multiple language	- Setup complexity for new devs - Redirect URI and consent screen must be exact - Fully dependent on Google availability
Facebook SDK	- Simple login Integration - SDKs for Web, IOS, Android	- App review can block features - Low trust in Facebook Privacy because of development state of the application
Geocoding API	- High accucracy and global coverage - Integrates well with other Google APIs - Key-based security with domain/IP restrictions	- Strict terms of service (must use with Google display)
Geolocation API	- Works well with WIFI - Secure via API key - will ask user permissions before getting locations	- Requires IP or Wifi info (less accurate than GPS) - Paid beyond free usage

Weather API	- Has huge Weather information - Can be Intergrated with location APIs -Easy REST API design	- Poor documentations - Has different value but same variable logic (Weather : Partly Cloudy, Partly Sunny)
Gemini API	- Does not accept raw input parameter, reducing injection risk - Secure access via Google OAuth or API keys - Multimodal support (text, images) - Built-in guardrails, moderation, safety filters	- Free tier may reject requests - Prompt results vary with small word change - Requires careful usage quota management for apps

Task 6 Application Screenshots

6.1 Home Page

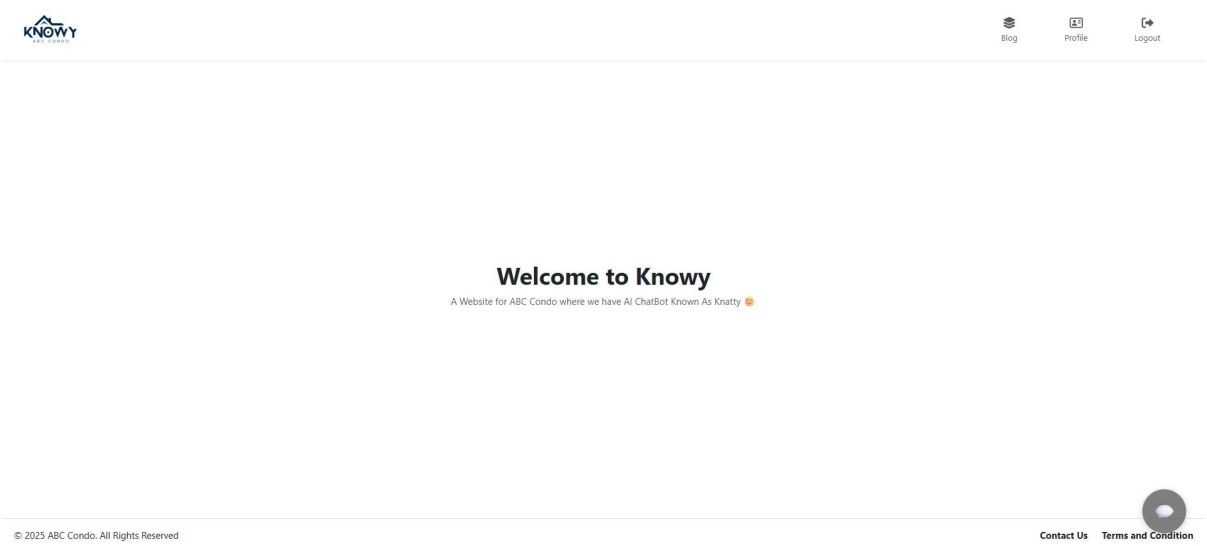

Figure 49 Home Page

6.2 Contact Us Page

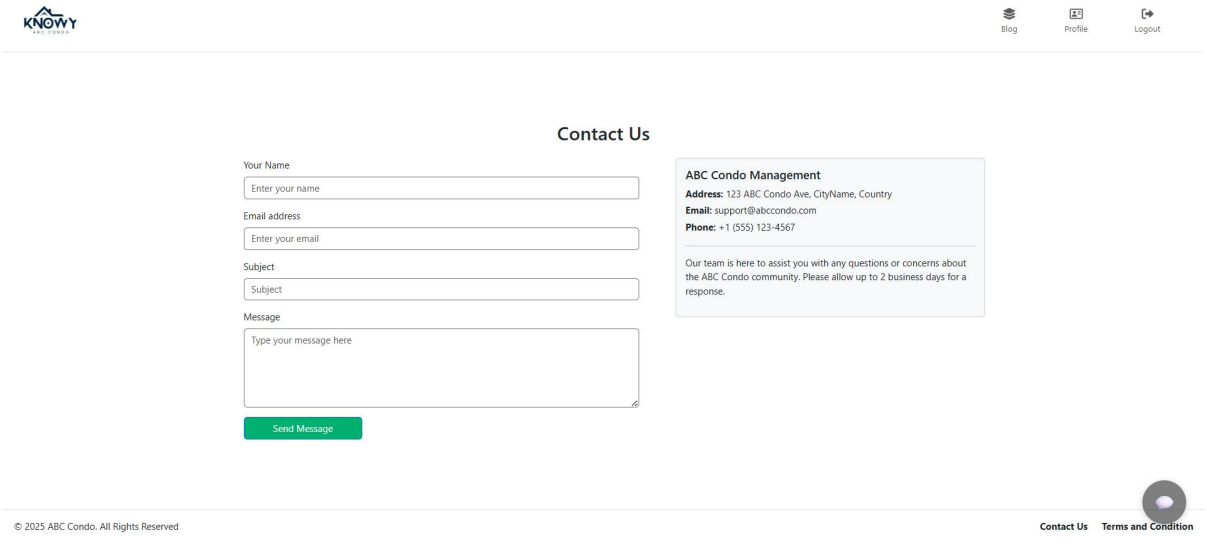

Figure 50 Contact Us Page

6.3 Terms & Regulations Page

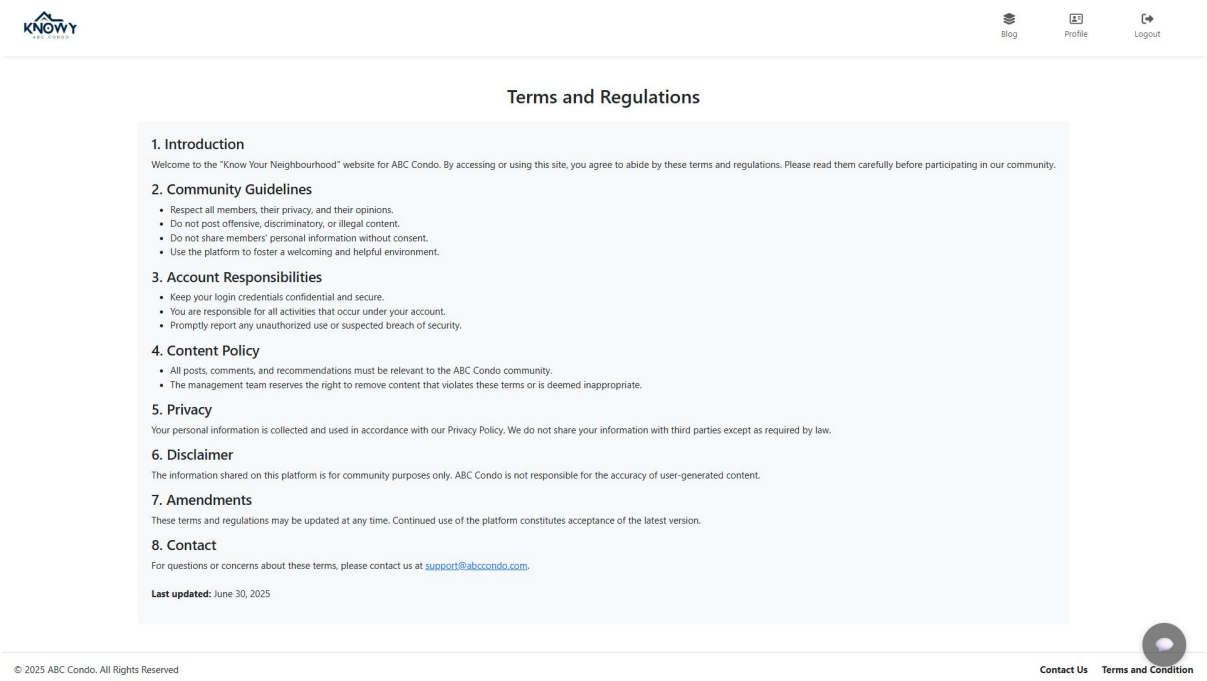

Figure 51 Terms & Regulations Page

6.4 Chatbot Application (About Us)

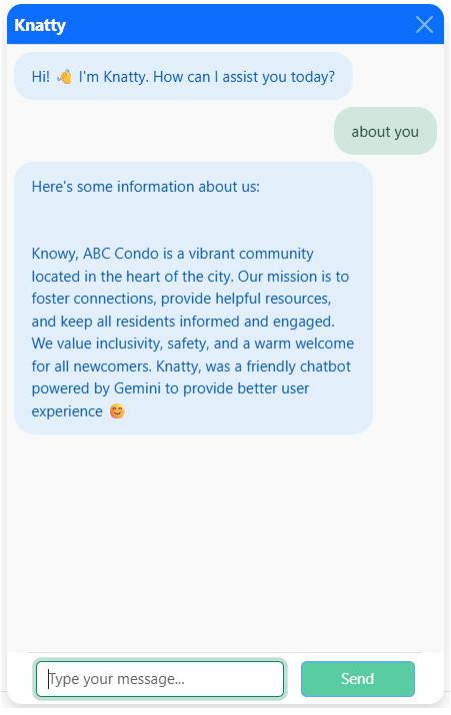

Figure 52 Chatbot Application About Us

6.5 Login Page

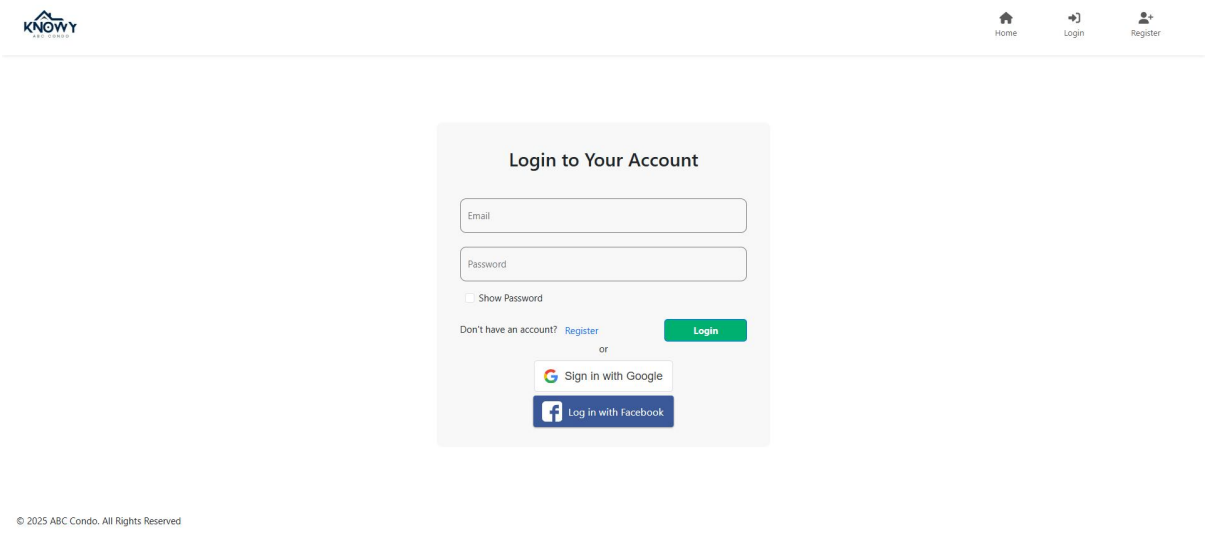

Figure 53 Login Page

6.6 Register Page

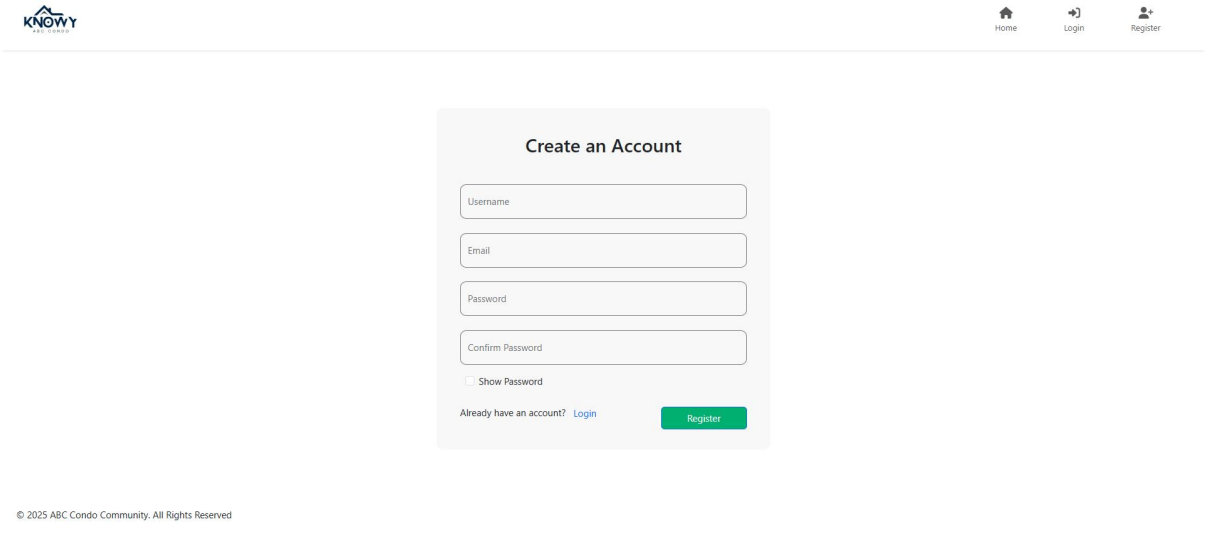

Figure 54 Register Page

6.7 Blog Page

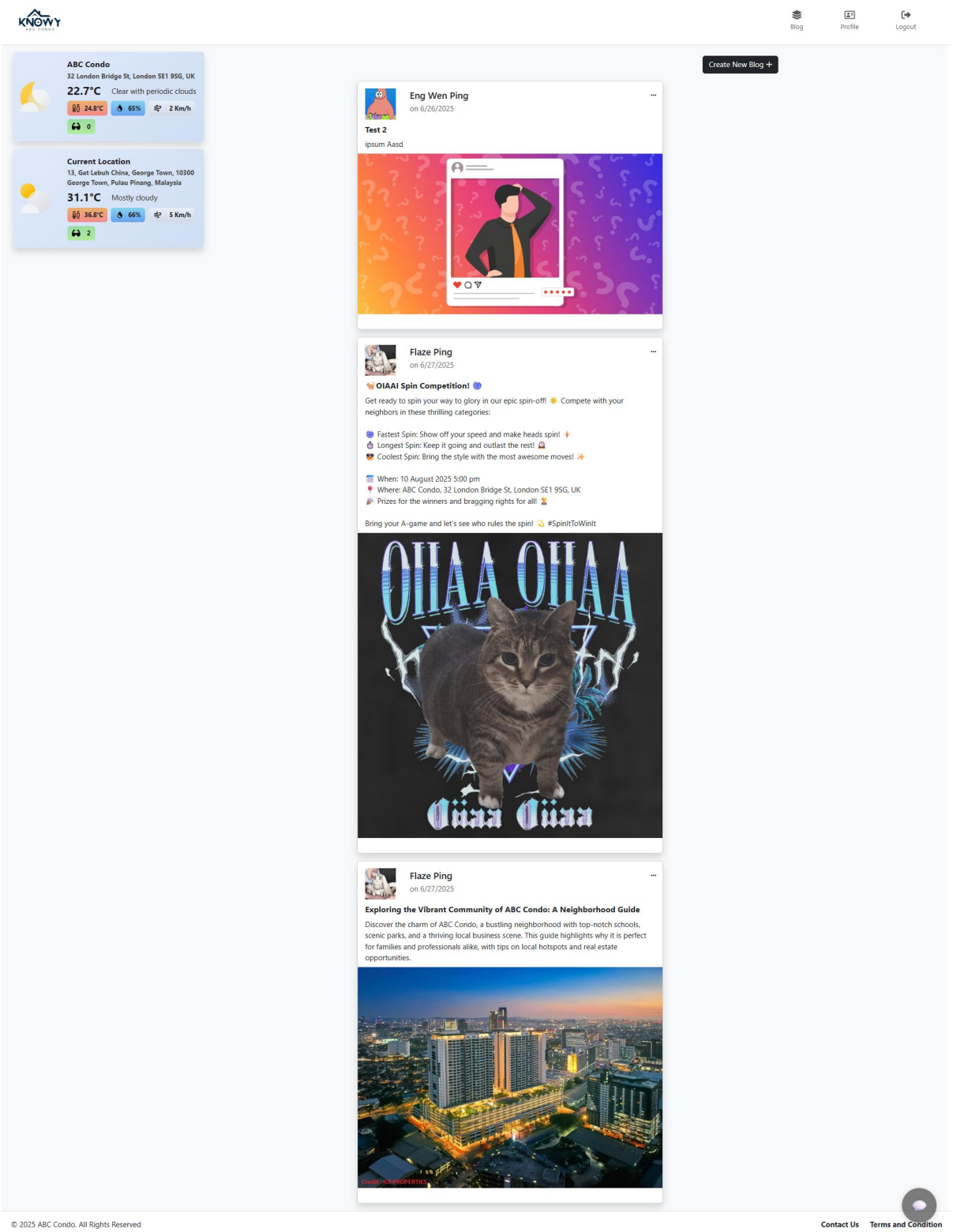

Figure 55 Blog Page

6.7.1 Blog Modal

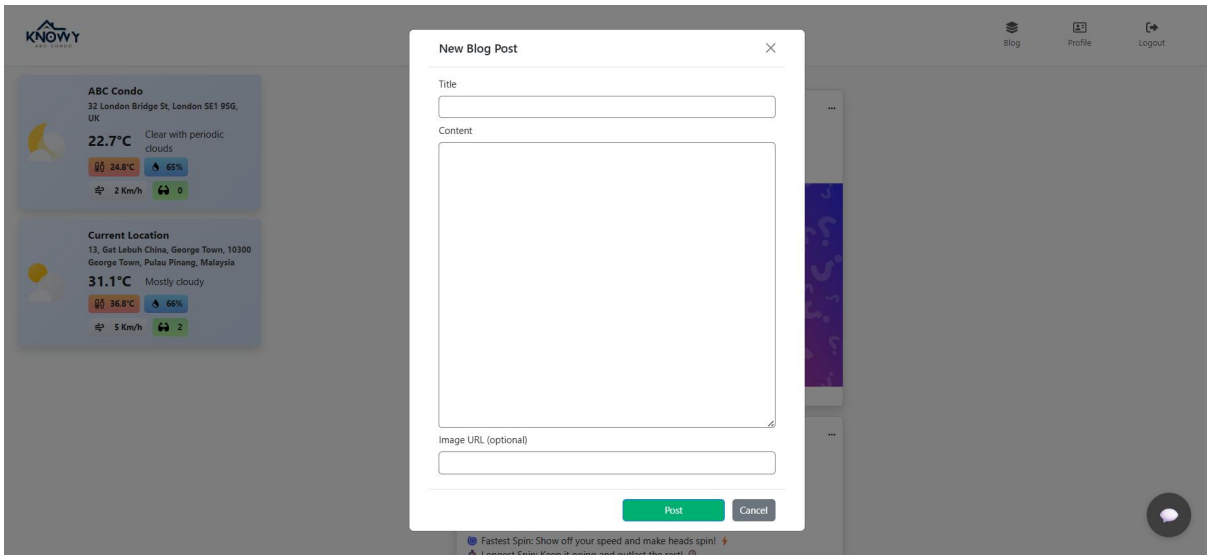

Figure 56 Blog Modal

6.7.2 Blog Owner Actions

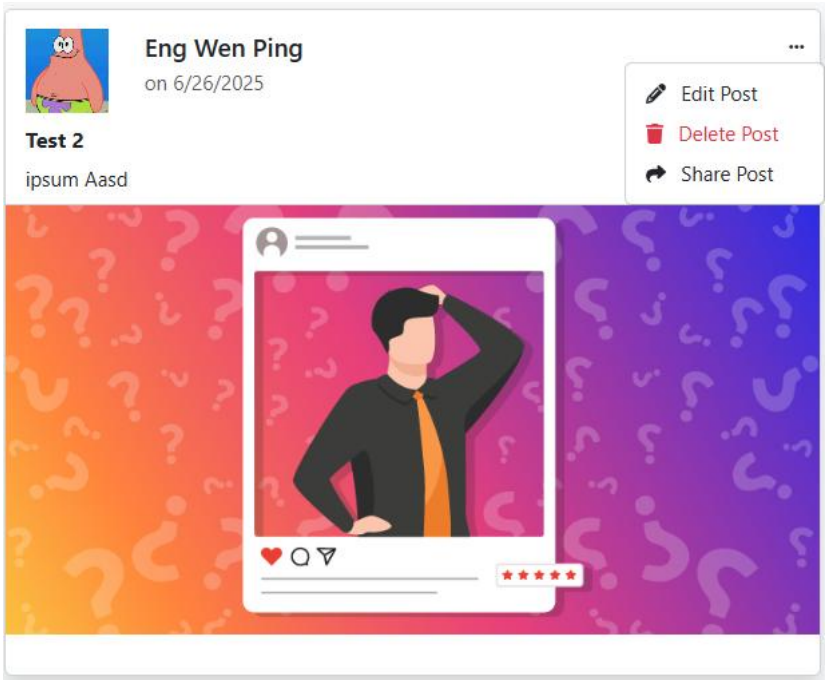

Figure 57 Blog Owner Action Modal

6.7.3 Blog User Actions

Figure 58 Blog User Action Modal

6.8 Profile Page

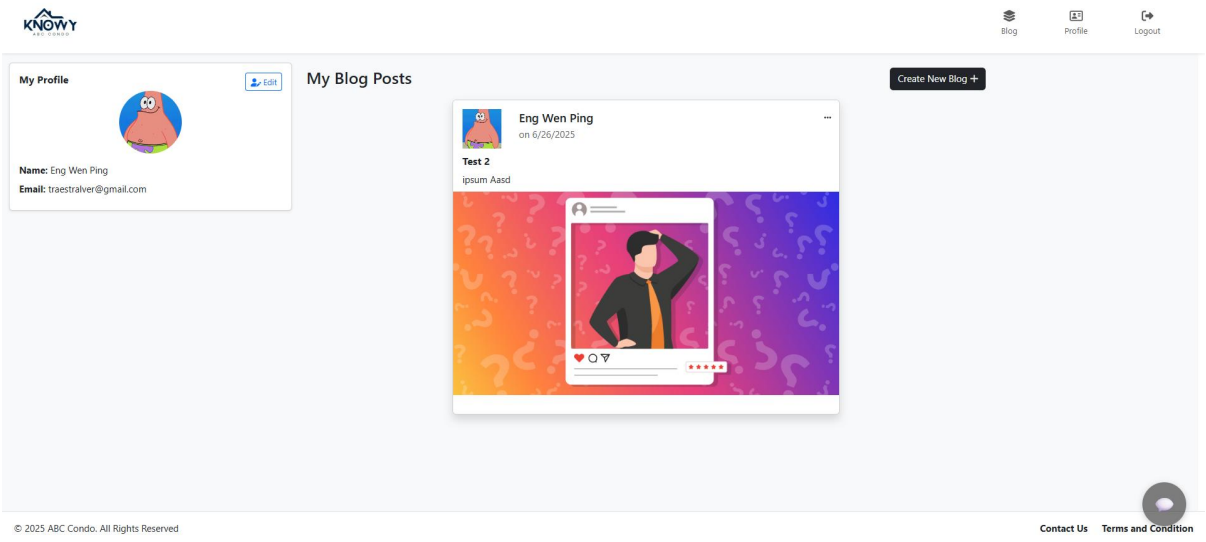

Figure 59 Profile Page

Conclusion

As a conclusion of the report, I will summarize all the task I had done which are: Introduction of the project, Discuss about the API Concepts and Types, How to Select API, Application Design, Develop 8 Wireframe, Provide Scope of the project & its target platform, Application Implentation of 3 arhictecture along with development environment & API Integration, total of 12 Whitebox and 11 Blackbox testing, API strength & weakness review and 10 Application Screenshots. Lastly, I appreciate that I participate in this project as i become more API integration with configurations and setup, AI application and simple machine learning logic which helps me in my future career.

References

1. *Introduction to GraphQL | GraphQL*. (2025, June 17). https://graphql.org/learn/

2. *Introduction to GRPC*. (2024, November 12). gRPC. https://grpc.io/docs/what-is-grpc/introduction/

3. Salesforce. (n.d.). *About SOAP API | SOAP API Developer Guide | Salesforce Developers*. https://developer.salesforce.com/docs/atlas.en-us.api.meta/api/sforce_api_quickstart_intro.htm

4. *GitHub REST API documentation - GitHub Docs*. (2022, November 28). GitHub Docs. https://docs.github.com/en/rest?apiVersion=2022-11-28

5. *Using OAUth 2.0 for web server applications*. (n.d.). Authorization | Google for Developers. https://developers.google.com/identity/protocols/oauth2/web-server#httprest

6. *Using Gemini API keys*. (n.d.). Google AI for Developers. https://ai.google.dev/gemini-api/docs/api-key

7. *Image generation*. (n.d.). Gemini API | Google AI for Developers. https://ai.google.dev/gemini-api/docs/image-generation

Appendix

Github Link

Knowy, https://github.com/Trae-ralv/Knowy

Marking Rubrics