Heaven's Blind (Front-End)

Introduction

Project Overview

Heaven's Blind is an anonymous blogging platform designed to provide users with a safe, secure, and private space to express their thoughts without fear of judgment or identity exposure. The platform focuses heavily on privacy, ensuring that user data is encrypted and no logs or identifiable information are stored.

Purpose of the Documentation

This document outlines the development of *Heaven's Blind* using the **SAD Waterfall model**, detailing each phase of the project lifecycle. The documentation serves as a guide to understand the decisions made during development and the lessons learned from each phase.

Software Goals

The primary goals of *Heaven's Blind* are:

- Provide a secure and anonymous platform for blogging.
- Ensure user privacy with strong encryption and a no-log policy.
- Deliver a user-friendly, responsive interface.

Phase 1: Requirements Gathering

Detailed List of Software and Hardware Requirements

- **Software**:
    - Frontend: HTML5, CSS3 (Bootstrap 4), JavaScript (jQuery) (blog).
    - Backend: Node.js or Python Flask for API routing and server management.
    - Database: NoSQL or SQL (for encrypted data storage).
    - Encryption: AES or similar strong encryption methods to secure user data.
- **Hardware**:
    - Cloud infrastructure such as AWS or Azure for hosting the platform securely.
    - Adequate memory and processing power for handling concurrent users and secure data encryption.

User Expectations

- **Privacy**: Users expect complete anonymity with no data logging or identifiable information stored.
- **Ease of Use**: Users expect a simple, intuitive interface with easy-to-use blogging features.
- **Security**: Users expect their interactions and data to be encrypted and protected from external threats.

Potential Issues Identified Early

- Implementing encryption without impacting platform performance.
- Ensuring a fully responsive design across all devices.
- Balancing between security and user experience, as some security measures can complicate usability.

Phase 2: System Design

Architectural Design

- **Frontend**:
    - The platform uses Bootstrap for a responsive design with key components like navigation bars, blog sections, and modals for post management(index).
    - JavaScript handles interactive elements such as blog post creation and modals(blog).
- **Backend**:
    - API endpoints are designed to handle post creation, editing, and deletion, ensuring that no user data is logged.
    - The system uses encryption to secure all data, with the database storing only encrypted posts.

Database Schemas

- The database will store user-generated posts, encrypted to ensure privacy. No personally identifiable information will be stored.

- Schemas are designed to handle blog posts without linking them to users, keeping all data anonymous.

Detailed Technical Specifications

- **Encryption**: AES-256 encryption is applied to all user data.
- **API Endpoints**:
    - **POST**: Create new blog posts.
    - **PUT**: Edit existing posts.
    - **DELETE**: Remove posts.

Potential Risks of Design Mistakes Becoming Costly

- Incorrectly implemented encryption could expose user data.
- Poor design of the database schema could make it difficult to maintain anonymity while ensuring platform functionality.
- Inflexible design choices can lead to difficulties in adapting to user needs or fixing errors later.

Phase 3: Implementation

Code Structure Explanation

- **Frontend**:
    - HTML templates and CSS files are organized to provide a clean, responsive layout.
    - JavaScript manages dynamic interactions like opening modals for blog post management(blog)(index).
-

- **Backend**:
    - The backend follows a modular design, with each feature (post creation, editing, deletion) implemented as separate routes.

Modules and Functions

- **Post Management Module**: Handles all interactions related to blog posts, ensuring they are securely stored and retrieved.
- **Encryption Module**: Manages the encryption and decryption of user data to ensure privacy.

Known Challenges Faced During Development

- Ensuring encryption does not degrade performance.
- Balancing between ensuring full anonymity and providing a smooth user experience.
- Difficulties in creating a responsive design across all devices.

Impact of Incorrect Implementations Due to Earlier Design Flaws

- Misconfigurations in encryption could require reworking the entire data handling process.
- Poor API design could result in delays when expanding the platform's features or fixing bugs.

Phase 4: Testing

Unit Testing, Integration Testing

- **Unit Tests**: Ensured that individual functions, such as post creation and encryption, work as expected.
- **Integration Tests**: Verified that the front-end and back-end work together seamlessly, particularly focusing on encryption during data submission.

Bugs Discovered at This Stage

- Data encryption sometimes caused performance issues during post creation.
- Bugs related to the responsiveness of the platform across different screen sizes and browsers.

Time Delays Due to Significant Bug Fixing

- Delays occurred due to the complexity of fixing encryption-related bugs without compromising user anonymity or system performance.
- Additional time was required to fix responsive design issues across different browsers.

Phase 5: Deployment

Deployment Strategy

- **Cloud Deployment**: Deployed the platform on AWS with HTTPS enabled to ensure secure communication.
- **Monitoring**: Used cloud-based monitoring tools to track performance and security post-deployment.

Issues Encountered During the Final Release

- Some performance bottlenecks were discovered post-deployment, related to encryption handling under high user load.
- Minor issues with the responsive design were reported, which required immediate fixes.

Example: Unresolved Errors That Stem from Earlier Stages

- Design flaws in the encryption module led to delays in the deployment as fixing it required reworking certain parts of the back-end code.

Phase 6: Maintenance

Long-term Issues Like Late Discovery of User Dissatisfaction

- Some users found the platform too complex due to additional security measures.

- Responsive design needed continuous improvements based on user feedback, as certain devices were not displaying elements correctly.

Challenges with Updating the Rigid System

- The fixed structure of the database and back-end modules made it difficult to introduce new features without significant rework.
- Adding features or changing security protocols often required revisiting earlier design choices, leading to time delays.

Lessons Learned

- Implementing a more flexible development model (such as Agile) could have allowed for more frequent testing and faster identification of design flaws.
- Better upfront planning of encryption and security features could have prevented many issues discovered during the testing and deployment phases.
- User feedback should be considered more dynamically during the development process to ensure usability aligns with user expectations.

Each phase of the Waterfall model for *Heaven's Blind* shows how mistakes or design flaws compound as the project progresses. While the platform successfully prioritizes privacy and anonymity, some of the challenges encountered highlight the limitations of the rigid Waterfall methodology.

(BACK-END)

1. Introduction

The backend system for *Heaven's Blind* is designed to support a secure, anonymous blogging platform. The key objectives of the backend are to handle user authentication, manage blog posts, and ensure that all user data is protected through encryption and secure session management. This backend is built with Node.js and Express.js, leveraging a PostgreSQL database to store and retrieve user and post data. A key focus of the platform is privacy and security, ensuring users can interact anonymously without compromising data safety.

Key Features of the Backend

1. User Authentication:

- Supports local authentication (email and password) using Passport.js.
- Implements Google OAuth2 for social login, allowing users to sign in using their Google accounts.
- Passwords are securely hashed using bcrypt, with sensitive data encrypted both at rest and in transit**.**

2. Session Management:

- User sessions are managed using Express-Session to maintain user login status across requests.

- Sessions are secured by storing session IDs in cookies, ensuring that user information is not exposed.

3. Data Management:

- All blog posts and user credentials are stored in a PostgreSQL database.
- The system allows CRUD operations (Create, Read, Update, Delete) for blog posts, enabling users to manage their content after logging in.

4. Security:

- The backend employs bcrypt for password hashing, ensuring that plain-text passwords are never stored in the database.
- HTTPS (if deployed properly) ensures secure communication between the client and server, preventing man-in-the-middle attacks.
- A no-log policy is maintained for sensitive user actions, ensuring that user identities remain anonymous.

5. API Integration:

- Axios is used to interact with an external API that handles database operations such as fetching blog posts and handling updates.
- This separation of the API allows for a more modular system where the frontend and backend can interact through clearly defined endpoints.

Technology Stack

- Node.js: As the core runtime environment for running the server, providing a scalable and fast backend solution.
- Express.js: A minimal web framework for Node.js used for routing and handling HTTP requests.
- PostgreSQL: A powerful, open-source relational database system used to store user information and blog posts.
- Passport.js: Authentication middleware that simplifies local and Google OAuth2 login flows.
- bcrypt: Used for hashing user passwords before storing them in the database, ensuring password security.
- Axios: A promise-based HTTP client used for making API requests to a separate API for blog post management.

Purpose of the Backend

The backend's primary role is to handle user authentication and authorization, ensuring that only authenticated users can interact with the platform’s blog features. Additionally, it manages the blog data, ensuring that users can create, edit, and delete posts securely. A key feature of *Heaven's Blind* is its focus on security and privacy, as users should feel confident that their interactions and identities are not exposed or logged.

Challenges Addressed by the Backend:

1. User Privacy: Ensuring user anonymity was a critical challenge. This is handled through encrypted sessions and password hashing.
2. Data Integrity: Guaranteeing that the blog posts are stored securely, with no exposure of personal information.
3. Scalability: The system is designed to handle multiple concurrent users, using PostgreSQL for efficient data management and Node.js for handling multiple requests asynchronously.
4. Security: To ensure secure data transmission, proper use of HTTPS (in deployment) and secure sessions are critical.

The backend's architecture not only ensures a secure and anonymous user experience but also lays the groundwork for future expansion. By keeping the system modular (with separate authentication and blog management components), it can easily scale and adapt to new features, such as additional third-party integrations or more sophisticated content moderation features.

.

---

2. Project Overview

- **Language**: JavaScript (Node.js)
- **Framework**: Express.js for the server.
- **Database**: PostgreSQL for user and blog post management.
- **Authentication**: Passport.js with **local strategy** and **Google OAuth2** for social login.
- **Security**: BCrypt is used for password hashing, and session management with cookies is done via Express sessions.
- **API**: Axios is used to communicate with a separate API that handles data operations.

---

3. Key Components and Configuration

1. Environment Variables Setup

- dotenv is used to load environment variables from the .env file, allowing the application to run with dynamic configurations such as database credentials, API URLs, and third-party secrets (Google OAuth2 client credentials).

import dotenv from "dotenv";

dotenv.config();

- Environment variables used:
    - PORT: Specifies the port where the app runs (default: 3000).
    - API_URL: URL for external API interactions (default: localhost:4000).

- GOOGLE_CLIENT_ID and GOOGLE_CLIENT_SECRET: Used for Google OAuth2 authentication.

2. Database Connection (PostgreSQL)

- The database is connected using pg (PostgreSQL client). Database credentials are read from the environment variables, ensuring sensitive data is not hardcoded into the application.

```
const db = new pg.Client({
  user: process.env.user,
  host: process.env.host,
  database: process.env.database,
  password: process.env.password,
  port_DB: process.env.port_DB,
});

db.connect();
```

- The PostgreSQL database stores user credentials and blog posts. All data is encrypted for privacy, ensuring that personal details (like passwords) are not stored in plaintext.

3. Middleware Setup

- **Body Parser**: Used to parse incoming request bodies in a middleware before handling them.

```
js
app.use(bodyParser.urlencoded({ extended: true }));
```

app.use(bodyParser.json());

- **Session Management**: Express sessions are used to manage user sessions, including keeping users logged in and authenticated. The session's secret is loaded from environment variables to ensure security.

js

```
app.use(session({
  secret: process.env.SESSION_SECRET,
  resave: false,
  saveUninitialized: true,
  cookie: {
    maxAge: 1000 * 60 * 60 * 24 // 24 hours
  }
}));
```

---

4. Authentication (Passport.js)

Passport.js is used for both local authentication and **Google OAuth2** login.

1. Local Strategy

- The **local strategy** uses a combination of email (as username) and password for authentication.

- Upon registration, passwords are hashed using bcrypt, and during login, passwords are compared against the stored hashed password using bcrypt's compare method.

js

```
passport.use("local", new Strategy(async function verify(username, password, done) {
  try {
    const checkResult = await db.query("SELECT * FROM users_login WHERE email_address = $1", [username]);
    if (checkResult.rows.length > 0) {
      const user = checkResult.rows[0];
      const hash = user.password;
      bcrypt.compare(password, hash, (err, result) => {
        if (err) return done(err);
        if (!result) return done(null, false);
        return done(null, user);
      });
    } else {
      return done(null, false);
    }
  } catch (error) {
    return done(error);
  }
}));
```

-

- **Password Hashing**:
    - bcrypt.hash() is used to hash passwords with a salt_round value of 10, ensuring strong password protection.

js

```
bcrypt.hash(password, salt_round, async (err, hash) => {
  // Store hash in the database
});
```

2. Google OAuth2 Strategy

- **Google OAuth2** is configured using passport-google-oauth2 to allow users to sign in using their Google account.
- If a user logs in with Google, the email is used to either find an existing user or create a new one.

js

```
passport.use(
  "google",
  new GoogleStrategy({
    clientID: process.env.GOOGLE_CLIENT_ID,
    clientSecret: process.env.GOOGLE_CLIENT_SECRET,
    callbackURL: "http://localhost:3000/auth/google/blog",
  },
    async (accessToken, refreshToken, profile, cb) => {
      // Logic for finding or creating the user
    }
  )
```

);

3. Session Handling in Passport.js

Session handling in **Passport.js** allows for persistent login sessions, meaning once a user successfully authenticates, they remain logged in as they navigate through the application. This is achieved through two key mechanisms: **serialization** and **deserialization** of user data.

Serialization

Serialization refers to the process of storing user information in the session after a successful login. Passport.js uses the passport.serializeUser() function to define what data should be saved in the session. Typically, rather than storing the entire user object (which could include sensitive information), a unique identifier like the user's ID is stored.

javascript

```
passport.serializeUser((user, done) => {
  done(null, user);  // In this case, the entire user object is stored in the session
});
```

- **What Happens**: After a user logs in, Passport stores the user object in the session. Each subsequent request by the user will include this session information.
- **Best Practice**: While storing the entire user object can work, it's generally recommended to only store the user ID or another unique identifier for efficiency and security. For instance:

javascript

passport.serializeUser((user, done) => {

done(null, user.id); // Only the user ID is stored in the session for security

});

Deserialization

Deserialization is the process of retrieving user information from the session when the user makes additional requests. This is handled by passport.deserializeUser(), which uses the identifier (stored during serialization) to look up the full user details, typically from a database. The retrieved user object is then attached to the request (req.user), making it available in subsequent routes.

javascript

passport.deserializeUser((user, done) => {

done(null, user); // The entire user object is retrieved from the session

});

- **What Happens**: On every request, Passport.js uses the serialized user data to look up and reconstruct the full user object (e.g., from a database) and attaches it to the req.user object for use in routes.
- **Best Practice**: Typically, the deserialization process involves retrieving the user from the database based on the ID or identifier stored in the session:

javascript

passport.deserializeUser((id, done) => {

```
  db.query("SELECT * FROM users WHERE id = $1", [id], (err, result) => {
    if (err) return done(err);
    done(null, result.rows[0]);  // The full user object is retrieved and attached to req.user
  });
});
```

Why Serialization and Deserialization Are Important

- **Session Management**: These functions are essential for keeping users authenticated as they navigate between different routes or pages on your website.
- **Security**: By serializing only an identifier, you minimize the risk of sensitive user data being stored directly in the session.
- **Performance**: Serialization optimizes session storage by reducing the amount of data stored, and deserialization ensures the latest user information is retrieved from the database when needed.

By using **serialization** and **deserialization**, Passport.js ensures that the session contains minimal data while still allowing access to the full user object when required for subsequent requests.

---

5. Blog Post Management

CRUD (Create, Read, Update, Delete) functionality for blog posts is handled using API routes. Posts are managed via Axios requests that interact with an external API for database operations.

1. Create and Update Blog Posts:

- New blog posts are created by submitting forms, which trigger Axios POST requests to the external API.

js

```
app.post("/api/posts", async (req, res) => {
  try {
    await axios.post(`${API_URL}/posts`, req.body);
    res.redirect("/blog");
  } catch (error) {
    handleError(res, "Error creating post");
  }
});
```

2. Edit and Delete Blog Posts:

- Posts can be updated or deleted via Axios PATCH or DELETE requests, respectively.

js

```
app.post("/api/posts/:id", async (req, res) => {
  try {
    await axios.patch(`${API_URL}/posts/${req.params.id}`, req.body);
    res.redirect("/blog");
```

```
  } catch (error) {
    handleError(res, "Error updating post");
  }
});

app.get("/api/posts/delete/:id", async (req, res) => {
  try {
    await axios.delete(`${API_URL}/posts/${req.params.id}`);
    res.redirect("/blog");
  } catch (error) {
    handleError(res, "Error deleting post");
  }
});
```

---

6. Error Handling

A utility function handleError is used for consistent error handling across all routes.

js

```
const handleError = (res, message = "An error occurred", statusCode = 500) => {
  res.status(statusCode).json({ message });
};
```

---

7. Deployment and Running the Server

The server runs on a port specified in the environment variables (or defaults to 3000). When running the server locally or deploying to a production environment, the app will be listening on this port.

js

app.listen(port, () => {

console.log(`Server running on port http://localhost:${port}`);

});

Conclusion

This backend setup provides a secure, scalable foundation for the *Heaven's Blind* platform, using PostgreSQL for data persistence, Passport.js for authentication, and Express.js for routing and session management. The use of bcrypt ensures that user passwords are stored securely, while Google OAuth2 integration provides an additional, secure login option.

API Blog Backend

This documentation outlines the structure and functionality of the blog backend built using **Node.js**, **Express.js**, and **PostgreSQL**. The system enables users to perform CRUD operations on blog posts and manage database tables through a RESTful API.

Introduction

This backend API allows users to interact with a blog system, where posts can be created, retrieved, updated, and deleted. It is built using **Express** for routing and **PostgreSQL** for data storage, utilizing a connection pool for efficient database management. The system supports dynamic table creation and inheritance, enabling flexible management of posts across different categories or sections.

Environment Variables

The following environment variables are required for the system to function properly. They are stored in a .env file and loaded using dotenv.

- PORT: The port on which the API will run. (Default: 4000)
- DB_USER: The username for PostgreSQL.
- DB_HOST: The host of the PostgreSQL database.
- DB_NAME: The name of the PostgreSQL database.
- DB_PASSWORD: The password for the PostgreSQL user.
- DB_PORT: The port of the PostgreSQL database. (Default: 5432)

Database Connection

A connection pool is used to handle multiple concurrent connections to the PostgreSQL database.

javascript

const pool = new pg.Pool({

user: process.env.DB_USER || ,

host: process.env.DB_HOST || ,

database: process.env.DB_NAME || ,

password: process.env.DB_PASSWORD || ,

port: process.env.DB_PORT ||

});

Middleware

- **Body Parser**: Used to parse incoming request bodies, supporting both application/x-www-form-urlencoded and application/json formats.

javascript

Copy code

app.use(bodyParser.urlencoded({ extended: true }));

app.use(bodyParser.json());

- **Error Handling**: A utility function handleError is used to consistently handle errors in the API responses.

javascript

const handleError = (res, error, message = "Internal Server Error", statusCode = 500) => {

* console.error(message, error);*

* res.status(statusCode).json({ error: message });*

};

Endpoints

1. GET /posts/

?

Retrieve all posts from a specified table. If no table is specified, it defaults to retrieving posts from the home table.

- **URL Parameters**:
    - table: *(optional)* Name of the table to fetch posts from. Default: home.
- **Response**:
    - 200 OK: Returns all posts from the specified table.

    - 500 Internal Server Error: If there's an issue retrieving posts.
- **Example**:

bash

GET /posts/home

GET /posts/news

2. GET /tables/all

Retrieve a list of all tables in the public schema of the PostgreSQL database.

- **Response**:
    - 200 OK: Returns a list of tables in the public schema.
    - 500 Internal Server Error: If there's an issue retrieving tables.
- **Example**:

bash

GET /tables/all

3. GET /posts/

Retrieve a specific post by its id from the home table.

- **URL Parameters**:
  - id: The ID of the post to fetch.
- **Response**:
  - 200 OK: Returns the post with the specified ID.
  - 404 Not Found: If the post with the specified ID does not exist.
  - 500 Internal Server Error: If there's an issue retrieving the post.
- **Example**:

bash

GET /posts/1

4. POST /posts

Create a new post in the home table.

- **Request Body**:
  - title: The title of the post.
  - content: The content of the post.
  - author: The author of the post.

- **Response**:
    - 200 OK: Returns the newly created post.
    - 500 Internal Server Error: If there's an issue creating the post.
- **Example**:

bash

```
POST /posts
{
  "title": "New Blog Post",
  "content": "This is the content of the post.",
  "author": "John Doe"
}
```

5. PATCH /posts/

Update a specific post in the home table by its id.

- **URL Parameters**:
    - id: The ID of the post to update.
- **Request Body**:
    - title: *(optional)* The new title of the post.

    - content: *(optional)* The new content of the post.
    - author: *(optional)* The new author of the post.

- **Response**:
    - 200 OK: Returns the updated post.
    - 404 Not Found: If the post with the specified ID does not exist.
    - 500 Internal Server Error: If there's an issue updating the post.

- **Example**:

bash

PATCH /posts/1

{

"title": "Updated Title",

"content": "Updated content"

}

6. DELETE /posts/

Delete a specific post in the home table by its id.

- **URL Parameters**:
  - id: The ID of the post to delete.
- **Response**:
  - 200 OK: Returns a success message.
  - 404 Not Found: If the post with the specified ID does not exist.
  - 500 Internal Server Error: If there's an issue deleting the post.
- **Example**:

bash

Copy code

DELETE /posts/1

7. POST /posts/data

Create a new table in the database with inheritance from the home table. This allows for new sections (like news or events) to be created dynamically while inheriting the structure of the home table.

- **Request Body**:
  - table: The name of the new table to be created.
- **Response**:
  - 200 OK: Returns a success message if the table was created or already exists.
  - 500 Internal Server Error: If there's an issue creating the table.
- **Example**:

bash

POST /posts/data

{

"table": "news"

}

Starting the Server

The API server can be started by running the following command:

bash

npm start

The server will listen on the port specified in the .env file or default to 4000.

-
- **Example**:

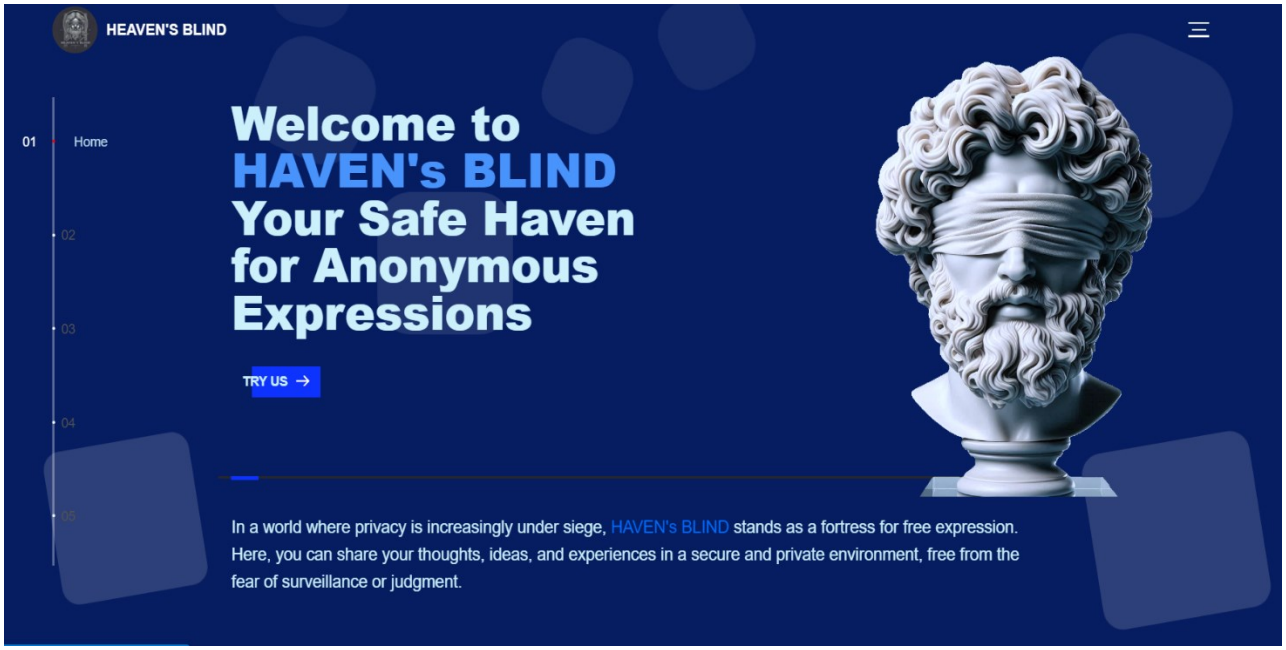
bash

API is running at http://localhost:4000

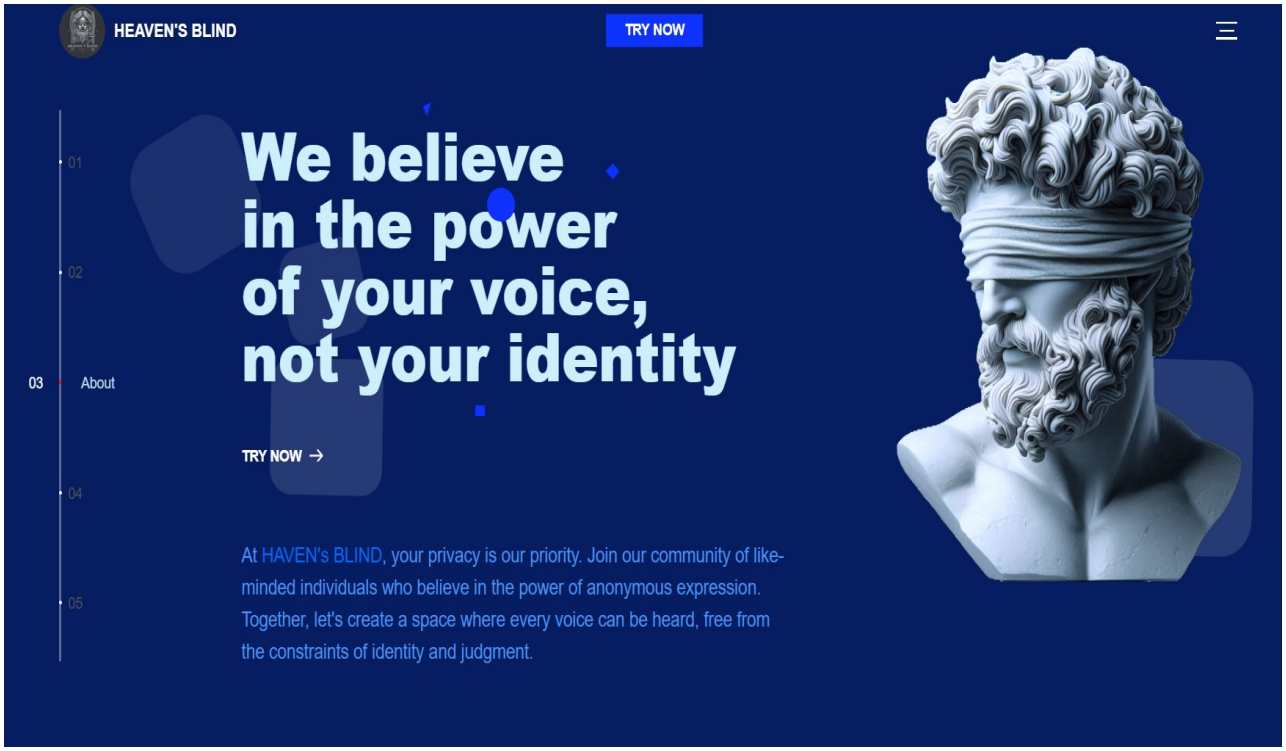
Conclusion

This blog backend API allows for easy management of blog posts with full CRUD capabilities. It also supports dynamic table creation through inheritance, making it adaptable for various sections or categories of posts. With PostgreSQL as the database, the system ensures efficient data management and scalability. The use of error-handling functions ensures that users are provided with meaningful messages in the case of any failures during API calls.

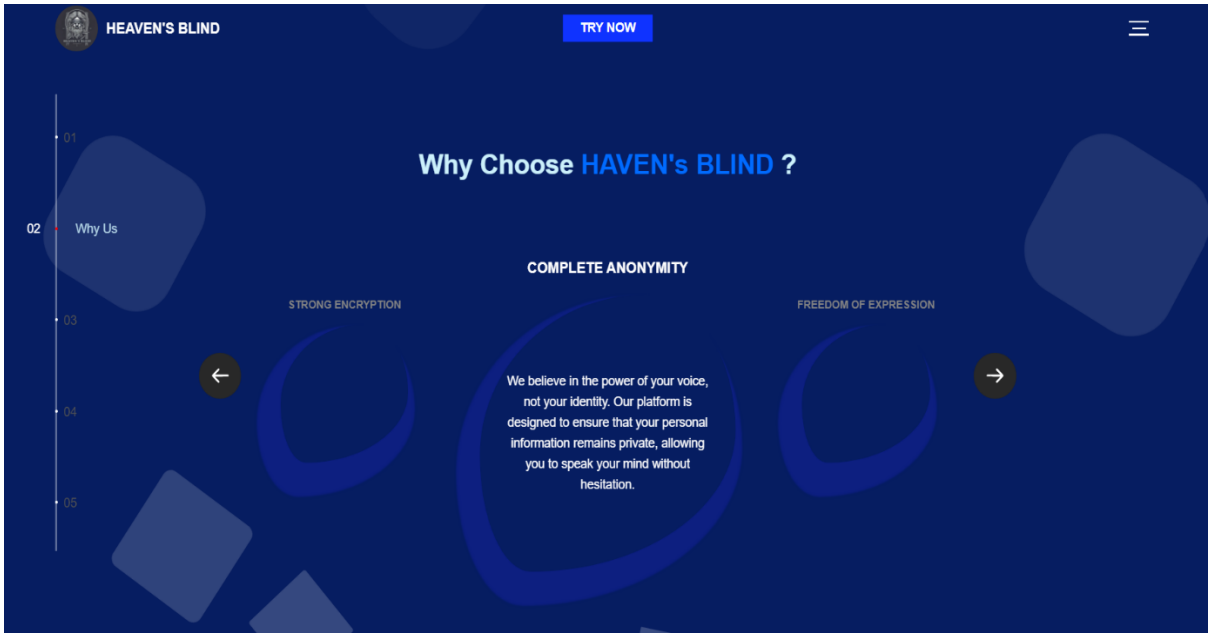
Home page



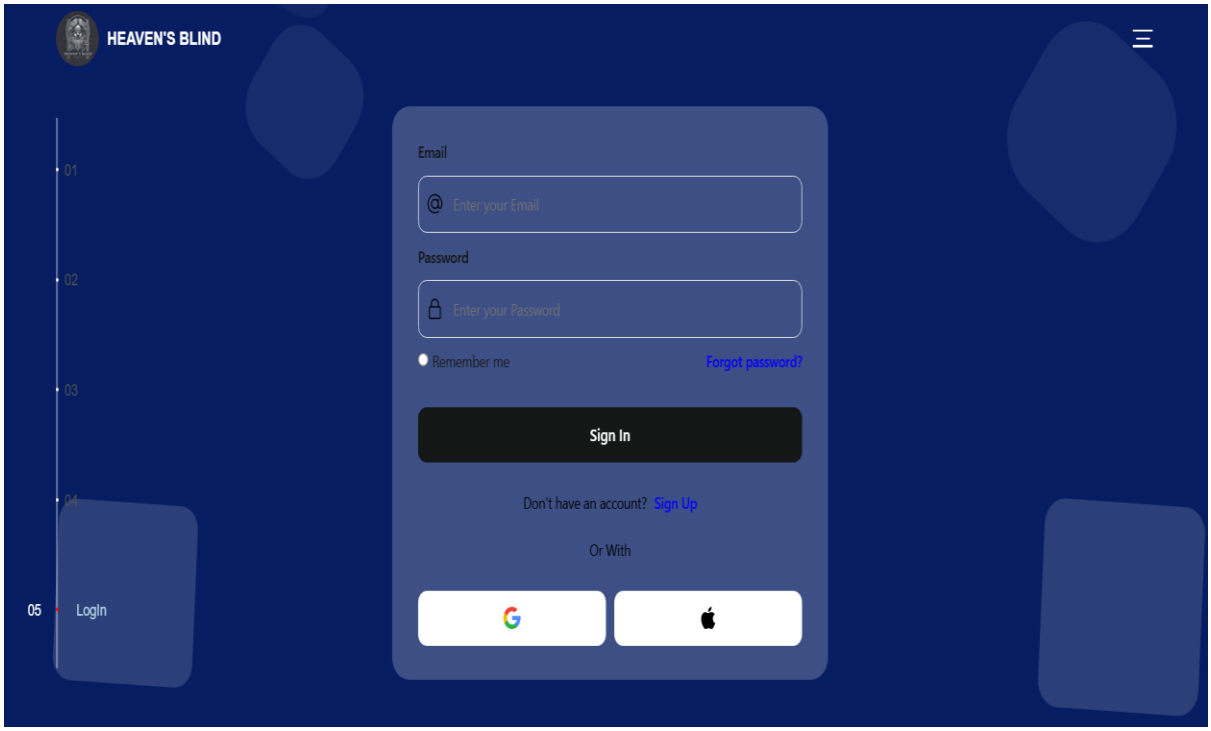
About page



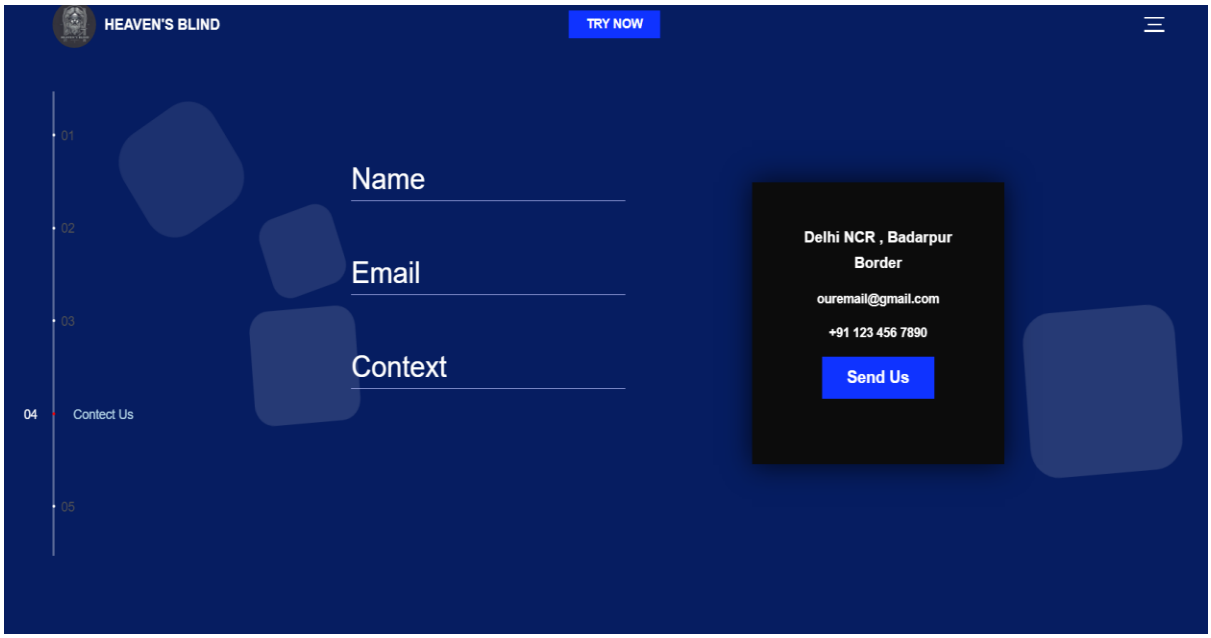
Why us page



Login page

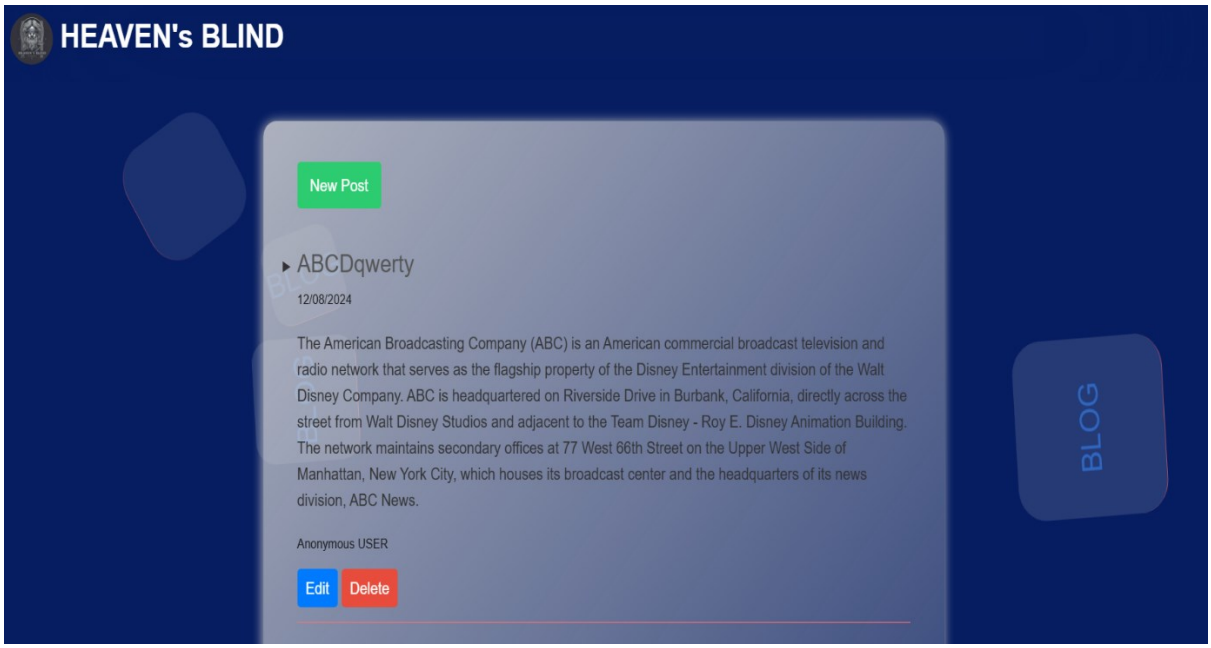


Contect page



Side navbar

Blog page



Post page