Internet Technology Implementation Report Death Notes

Arshia Kaul
Harish Ravichandran
Siddhartha Pratim Dutta

School of Computing Science

Sir Alwyn Williams Building

University of Glasgow

G12 8RZ

20th March 2025

Contents

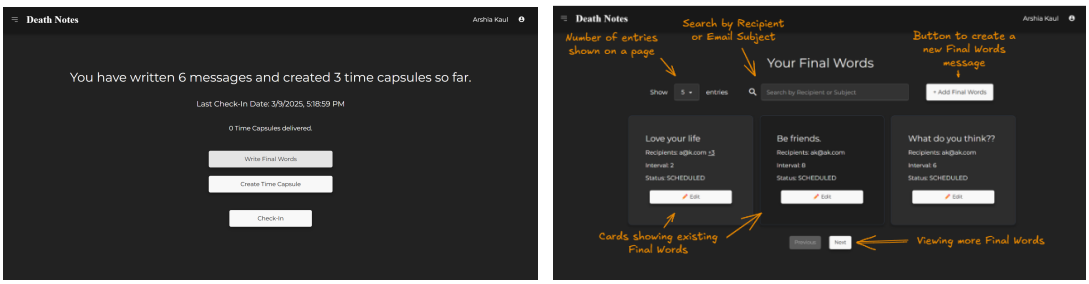
Chapter 1 Introduction

Deployed Web Application: https://www.deathnotes.tech/

Death Notes is a web application that enables users to schedule messages for recipients. These messages can either be sent on a specified future date or automatically triggered after a period of user inactivity. This service ensures that important messages reach intended recipients, whether at a meaningful future moment or after the user's passing, through two core features: **Final Words** and **Time Capsule**. While death is an uncomfortable and difficult topic, Death Notes is designed to provide peace of mind, ensuring that no message remains unsaid.

The design specification of Death Notes provided a solid foundation during development, guiding with system architecture, database entities, and website organisation through the sitemap and wireframes. However, a couple of adjustments were made during detailed implementation planning to streamline development and improve user experience (UX):

1. **Separation of Concerns**: The initial system architecture assumed Django would serve both the backend and frontend using Bootstrap and JavaScript with server-side rendering. However, to support asynchronous development, the application was separated into a Vue.js frontend and a Django backend, with Django exposing REST APIs through the Django Rest Framework**.**
2. **UX Adjustments**: A Check-In button was added to the dashboard (see Figure 1(a)). While this could have been achieved automatically during user login, achieving this through an element on the UI is more intuitive for the user. Furthermore, it was noted that using a table on the pages for viewing Final Words and Time Capsules did not provide a responsive interface on smaller devices. Therefore, cards-elements with details about recipients, interval and status were used to display the records on both the pages (see Figure 1(b)).



(a) **(b)**

Figure 1: Design Adjustments (a) Check-In Button on Dashboard (b) Card elements for viewing messages

These adjustments ensured that Death Notes not only met the original design specification goals but also provided a better development process and a more intuitive and responsive user experience.

Chapter 2 Design Specification

2.1 Overview

- A digital safety net for scheduled messaging: Death Notes allows users to schedule messages to ensure that important words are delivered at the right time, even when they are no longer around.
    - **Final Words:** Users can schedule heartfelt messages, important instructions, or final thoughts to be sent after a defined period of inactivity, with a customisable delay. This ensures that final messages, whether farewells, confessions, or advice, are delivered when the user is no longer able to speak for themselves.
    - **Time Capsule:** Users can schedule messages for delivery at a specific future date and time. This feature allows users to send notes of encouragement, reminders, or meaningful messages for future milestones, helping them stay connected across time.
- Automate life's important messages, from last words to financial details or personal reflections, and have them delivered exactly when intended.
- A blend of practicality and emotional depth, Death Notes ensures that important words are never left unsaid in an unpredictable world with an inevitable end.

2.2 Requirements

- Users must be able to log in using Microsoft Single Sign-On to create or access their account.
- Users must be able to log out of their account.
- Users must be able to define a custom inactivity period (interval) in the user settings.
- Users must be able to schedule **Final Words** messages to recipients with a customisable delay triggered after the defined inactivity interval.
- Users must be able to schedule **Time Capsule** messages to recipients for a specific future date and time.
- Users must be able to check in to their account to indicate user activity and prevent final words from being sent.
- Users must be able to view, edit and delete scheduled messages.
- Users must be able to view their activity log and message statuses on a dashboard.

2.3 System Architecture Diagram

This web application uses Vue 3, a lightweight JavaScript framework for the frontend, while Django Rest Framework (DRF) is used for the backend (see Figure 2). The interaction between the frontend and backend occurs through REST API requests, with responses exchanged in JSON format. The Django backend handles core logic, including check-ins, message creation, and scheduling. The Microsoft Identity Platform is used for the authentication of users via the OAuth 2.0 authorisation flow, and sqlite3 serves as the database for storing user and scheduling data. A scheduler service runs as a background process to send scheduled messages via an SMTP server for email delivery.

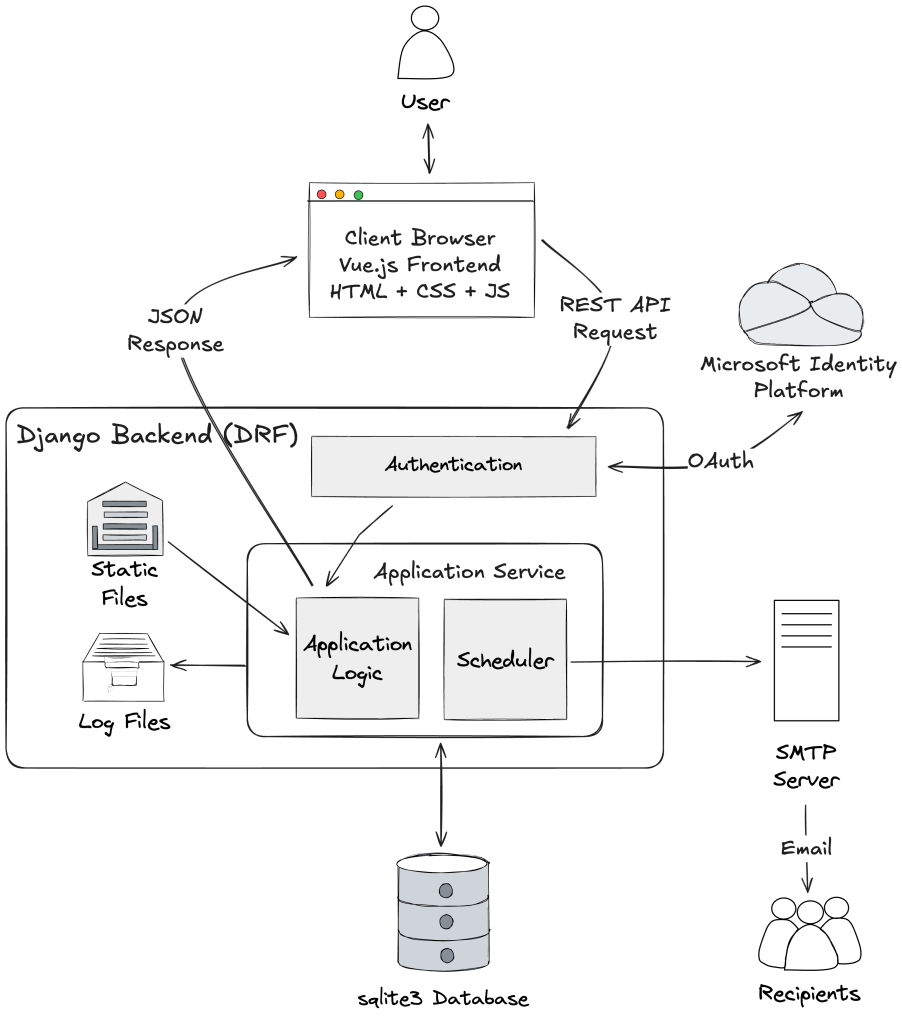


Figure 2: System Architecture Diagram for Death Notes

2.4 Entity Relationship Diagram

The ER diagram for this application consists of four entities. Each user can track their various events, like check-in and message creation, in the Activity Log. Further, a single user can create multiple Messages. In addition, each message is scheduled as a single Job which gets executed when the appropriate conditions are met.

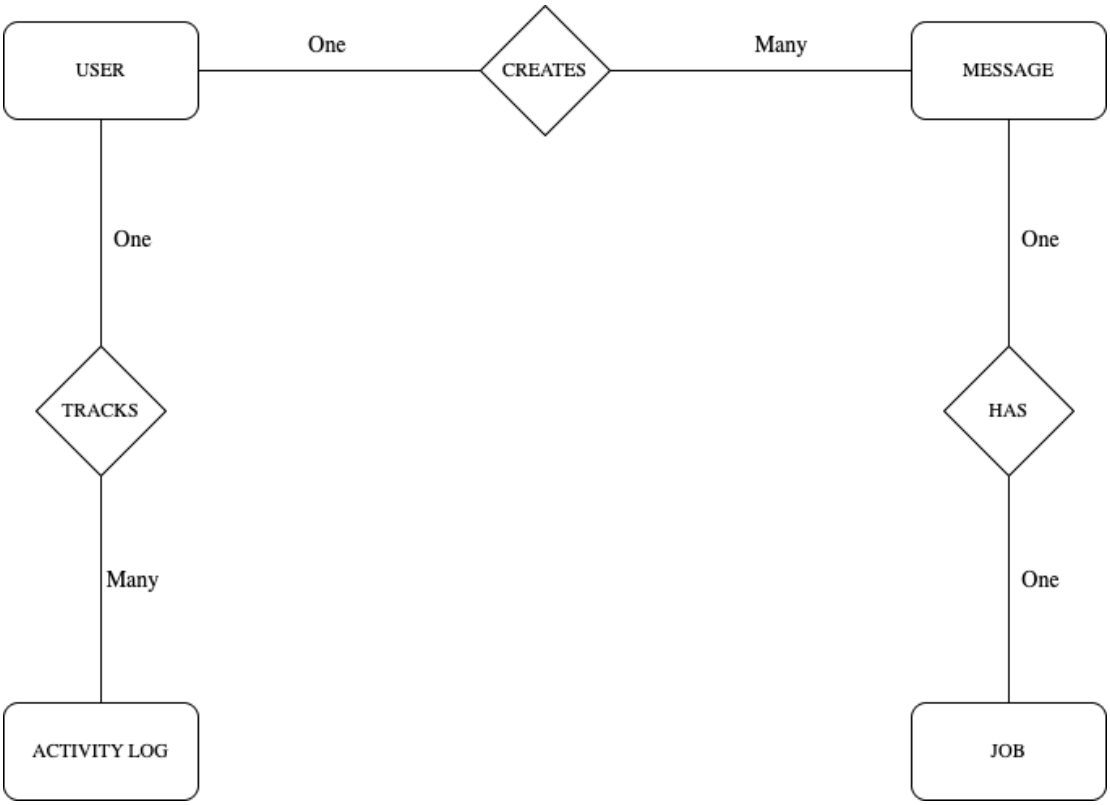


Figure 3: Entity Relationship Diagram for Death Notes

2.4.1 Entities

User: Personal details of the user, such as their name and email address, are stored in this entity. It also includes an interval field which stores the number of days after which a user is considered inactive (or having possibly passed away).

USER		
id (PK)	integer	Unique identifier for User
first_name	varchar(50)	User's first name
last_name	varchar(50)	User's last name
email	varchar(50)	User's email for authentication
interval	integer	Number of days after which the user is considered inactive

Table 1: The User Entity

Message: This entity stores the actual message content to be delivered, along with a comma-separated list of recipients. The message can either be a Final Words message to be delivered after the user is deemed inactive or a Time Capsule that gets delivered at a scheduled date regardless of the user's inactivity. For a Final Words message, a *delay* is stored, which signifies the number of days after the user's inactivity when the message gets delivered. For Time Capsules, the future date of delivery is stored in the *scheduled_at* field.

MESSAGE		
id (PK)	integer	Unique identifier for Message
user_id (FK)	integer	Reference to User [id]
type	enum	Type of message [FINAL_WORDS | TIME_CAPSULE]
recipients	text	Comma-separated list of message recipients
status	enum	Current status of the message [SCHEDULED | DELIVERED | FAILED]
subject	varchar(100)	Message title i.e., the email subject
content	text	Content of the message i.e., the email body
delay	integer	Delay before sending message for FINAL_WORDS (in days)
scheduled_at	timestamp	Date of deliver of the message for TIME_CAPSULE

Table 2: The Message Entity

Job: Message delivery is managed through this entity. Every time a message is created, a corresponding record is inserted into the Job entity. For a Time Capsule, the *Job.scheduled_at* is set to be the same value as *Message.scheduled_at*. For a Final Words message, it is initially set to the sum of the user's interval and the delay of the message. However, the date of delivery for a Final Words message is dependent on user inactivity. Therefore, *Job.scheduled_at* for Final Words messages is set depending upon the last check-in by the user.

JOB		
id (PK)	integer	Unique identifier for Message
message_id (FK)	integer	Reference to Message [id]
scheduled_at	timestamp	Scheduled time of job execution
is_completed	boolean	Completion status of the job

Table 3: The Job Entity

Activity Log: This entity consolidates user activities such as check-ins, message creation, deletion, scheduling, and delivery, making it easier for users to view their actions.

ACTIVITY LOG		
id (PK)	integer	Unique identifier for ActivityLog
user_id (FK)	integer	Reference to User [id]
timestamp	timestamp	Time of the activity
type	enum	Type of the activity [CHECKED_IN | MESSAGE_CREATED | MESSAGE_DELIVERED]
description	text	Description of the activity

Table 4: The Activity Log Entity

2.5 Site Map

At the topmost level, the user logs in from the Home page and is redirected to the dashboard after authentication. The dashboard allows navigation to the user account and activity log. It also provides access to main features like Final Words and Time Capsules. Next, each main feature allows the user to perform more specific actions at the second level. This involves Manage Profile and Logout options for the user account. Meanwhile, both Final Words and Time Capsules provide Create and List (view) functionality, whereas Activity allows the user to sort logs. Furthermore, the Operations Level provides more granular functions like viewing listed items from the previous level and then updating or deleting them.

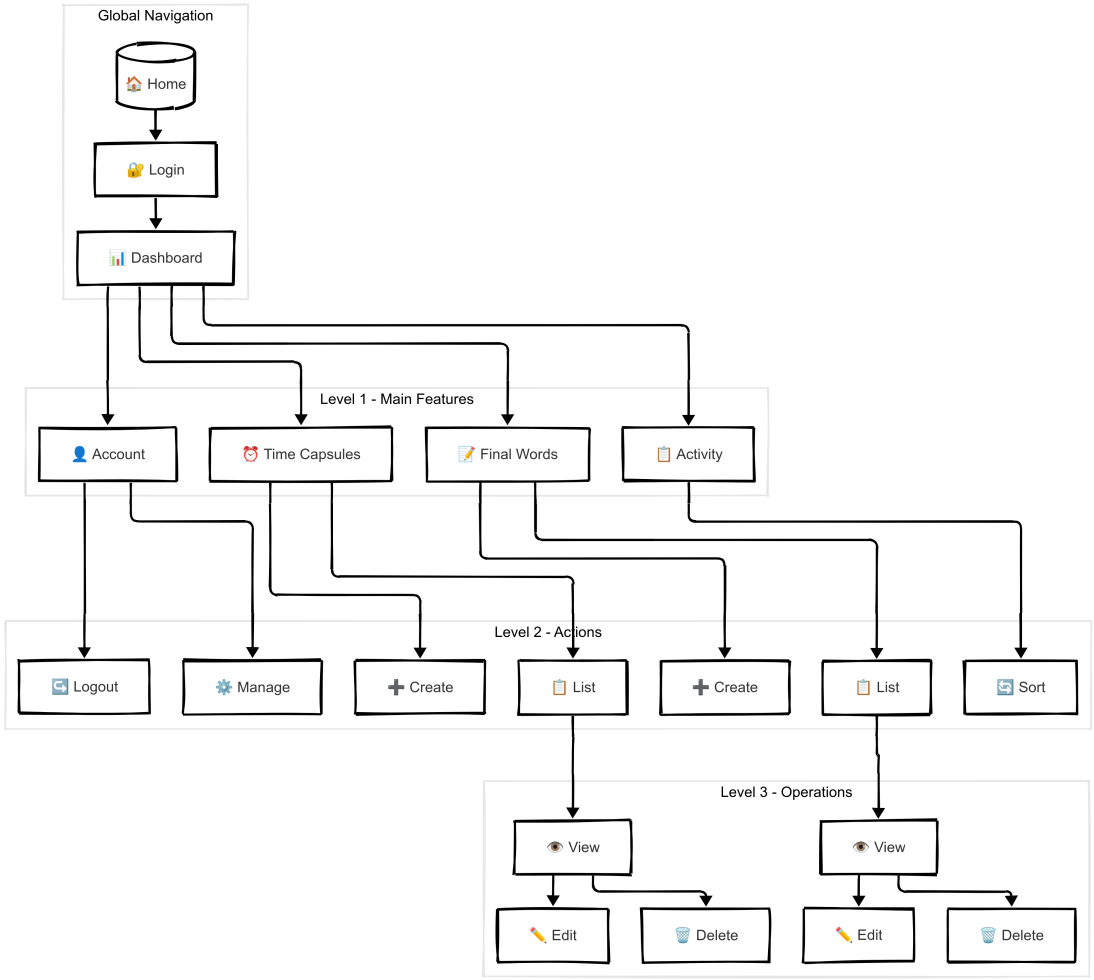
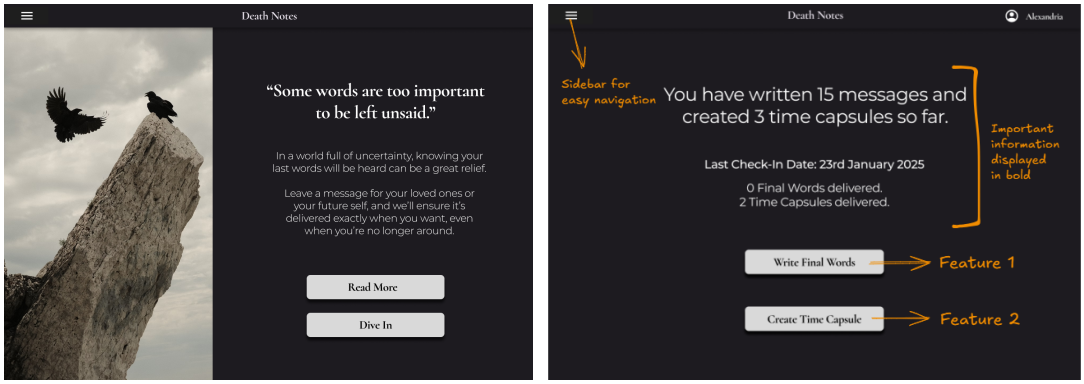


Figure 4: Site Map for Death Notes

2.6 Wireframes

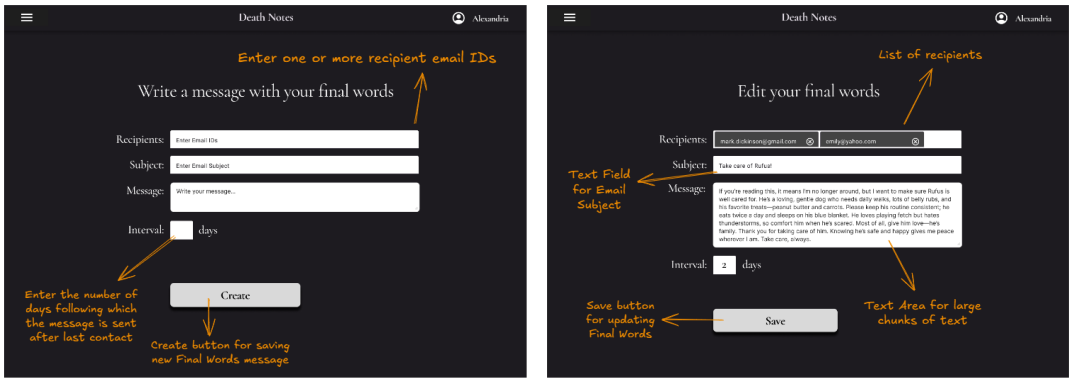
The wireframe creation process used Figma and emphasised consistent style and component reuse. These wireframes were created to closely resemble the final product, thus leaving very little room for ambiguity during the development process. Since this application deals with an emotional subject, it was ensured that the design remained minimal and uncluttered, thus letting the user focus on writing their message without being distracted by the elements on the screen.



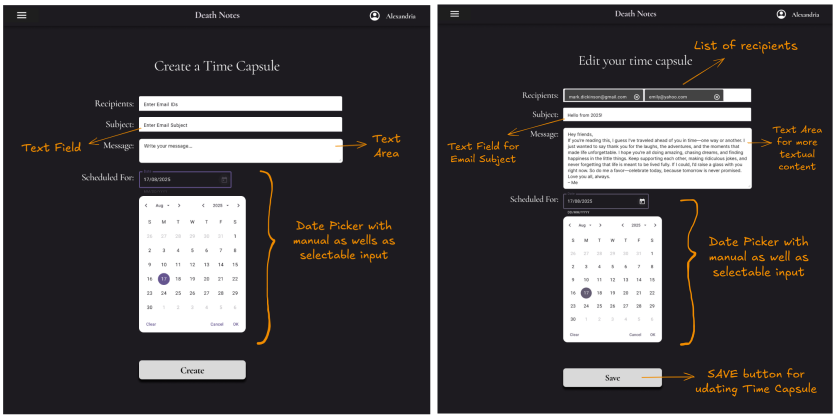
(a) **(b)**

Figure 5: (a) Home Page with Login (b) User Dashboard

Further, frequently used functionalities like writing Final Words and creating a Time Capsule were provided on the dashboard, which is the first page displayed after logging in (see Figure 5(a)). Also, relevant information like the number of final words written, time capsules created, and last check-in date are displayed on the dashboard. Next, a similar interface was used to create and edit Final Words, with them differing only in the content displayed inside the input fields and the text displayed on the button for form submission (see Figure 6). The same approach was employed for wireframes for creating and editing Time Capsules.



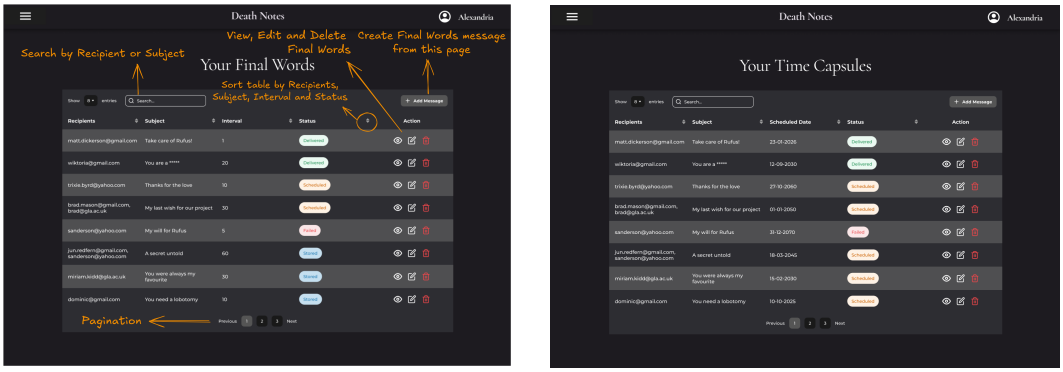
(a) **(b)**



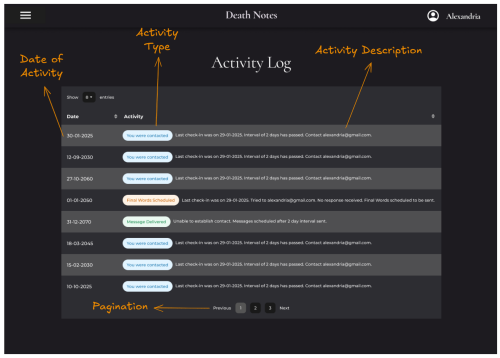
(c) **(d)**

Figure 6: (a) Create Final Words (b) Edit Final Words (c) Create Time Capsule (d) Edit Time Capsule

Finally, the pages to view Final Words, Time Capsules and Activity Log were designed in a tabular form to allow convenient sorting and reuse components like pagination elements and search bars (see Figure 7).



(a) **(b)**



(c)

Figure 7: (a) View Final Words (b) View Time Capsules (c) View Activity Log

Chapter 3 Implementation Description

3.1 Frontend Development

Frontend Repository: https://github.com/siddydutta/death-notes-app/

The frontend utilises Vue 3, which is a lightweight JavaScript framework that allows for fast development, making it a suitable choice for this project. Further, it provides smaller bundle sizes and lesser boilerplate compared to other JavaScript frameworks like React. Also, unlike React, which requires manual state updates, Vue automatically tracks changes and updates the UI without having to manually trigger updates, thus providing for a more intuitive reactivity.

Firstly, the source code for frontend is logically segregated into different folders based on the functionality being implemented. Inside `api/http.ts`, Axios is used to make HTTP requests to the backend APIs. It is configured to provide interceptors and handle authentication tokens. HTTP requests involving fetching and updating messages, activity log and user profile are placed in separate files inside the `api` folder. Further, routing is handled using Vue Router, and all the relevant configurations are present inside `router/index.ts`. Meanwhile, the `assets` folder holds global CSS files and images displayed in the web application.

Secondly, the auth store is defined in `stores/auth.ts` using Pinia, Vue's official state management library, and is used in various components to manage user authentication state and information.

Initially, it was planned to use Tailwind CSS, a framework that provides low-level utility classes to build custom elements. However, DaisyUI - a wrapper over Tailwind CSS- worked better for our case as it provided a set of pre-designed components and themes to speed up the development process. It enabled us to create easily customisable and highly responsive components like App-Bars, Toast Notifications and Message Cards that could be reused in multiple views. This was especially useful in allowing users to navigate the website through the App-Bars on every page and to provide a confirmation/failure of user actions via Toast Notifications.

Furthermore, all the repetitive elements were abstracted into reusable components which reside under the `components` folder. Also, any custom CSS class that had to be reused for multiple components was defined inside `assets/main.css`. These components were then used inside views, thus enhancing code readability and reuse while ensuring consistent styling across all the pages.

Finally, although DaisyUI enabled the development of a reliable responsive interface, CSS media queries were used to customise certain parts of the interface to ensure a uniform experience on devices with smaller screens.

3.2 Backend Development

Backend Repository: https://github.com/siddydutta/death-notes/

The backend is built using the Django 5 and is structured into three Django apps: `accounts`, `cron` and `web`. The `accounts` app manages the custom user model, which uses email instead of username for authenticating users and user-related functionality. The `cron` app, as the name suggests, is responsible for scheduling-related functionality involving the `Job` entity. The `web` app implements the core service logic and exposes REST APIs to enable users to create, read, update or delete their profile settings, messages and activity log.

User authentication and authorisation are handled via the Microsoft Identity Platform. An app registration for Death Notes is configured, specifying that the application only requires the `User.Read` scope to access user information like email, first name and last name. On successfully authenticating the user credentials through Microsoft APIs, the backend returns access and refresh JSON Web Tokens (JWT) to the frontend application, which are then used to authorise subsequent user actions.

The backend extensively uses the Django Rest Framework (DRF) to implement REST API functionality and JWT authentication. DRF is leveraged to simplify the development of APIs for GET (list and retrieve records), POST (create records), PATCH (update records), and DELETE (delete records) and supporting query parameter handling for filtering or ordering and paginated JSON responses when appropriate. A custom pagination class is implemented that extends the `rest_framework.pagination.LimitOffsetPagination` class to integrate it with the frontend application. HTTP methods and status codes were used according to standard REST practices with appropriate status codes returned, for example, 200 (ok), 204 (no content) or 400 (bad request).

For job scheduling and execution, the `django_q` library is used, which allows scheduled jobs to run without the need for external brokers or messaging services. For the scale of this application, a single worker is configured to run a job every 30 minutes to check if any pending `Job` instances need to be processed. The execution of a `Job` instance involves sending messages via emails for which Django's wrapper of the `smtplib` module was used to send emails through Google's SMTP servers. A custom HTML email template is maintained to provide formatting, which uses the Django templating language to add dynamic values, improving the user experience of receiving messages.

Most importantly, the core application logic lies in the usage of Django signals, which are triggered on specific events. For example, the creation or updating of a `Message` instance requires the updates to cascade to the `Job` instance. Similarly, any event updates from `Job` instances need to be caught and handled for populating the `ActivityLog` table. The signal dispatcher helps to decouple components and maintain consistency during database updates, thus implementing the core logical flows through an event-driven architecture.

A logging configuration stores access, application, and scheduler logs in external log files on the server, which is managed using a rotating log mechanism that overwrites old logs, retaining newer logs while optimising storage.

3.3 Deployment and Infrastructure

To ensure a seamless deployment workflow, a combination of Vercel, Azure, Docker, Nginx, and GitHub Actions was utilised, enabling continuous delivery with minimal manual intervention.

The frontend deployment is handled by Vercel, which makes the deployment process seamless. Linking the public GitHub repository to Vercel allows automatic deployments whenever changes are pushed to the `master` branch. This ensures that any updates to the frontend application are instant and hassle-free.

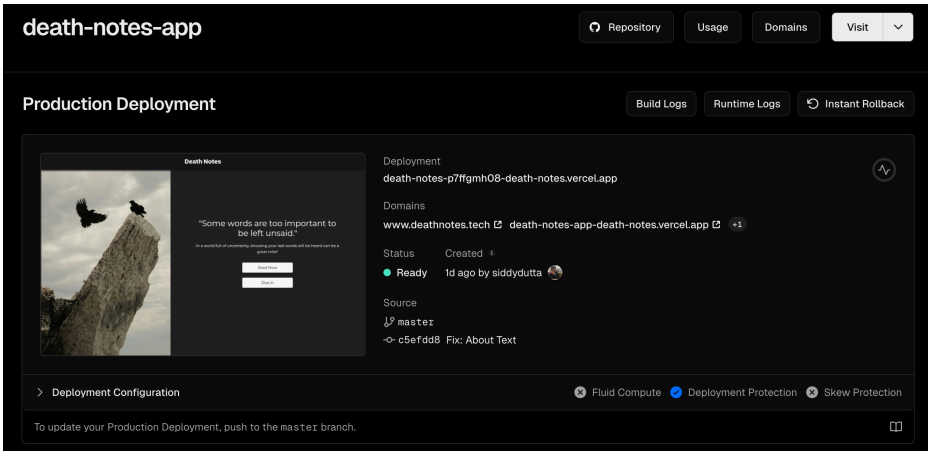


Figure 8: Vercel for Frontend Deployments

The backend project required a more manual approach. It is hosted on an Azure Linux Virtual Machine with a `Standard B2ats v2` disk. The application itself is containerised using Docker, which ensures consistency across deployments. The container setup script includes installing dependencies, running any database migrations, and starting the Django server. To handle incoming requests, Nginx is also configured as a reverse proxy with HTTPS enabled using Let's Encrypt for SSL certificates.

While the initial server setup was manual, deployment automation has been integrated using GitHub Actions. A custom YAML script ensures that when changes are pushed to the master branch, the latest code is pulled and the Docker container is rebuilt and restarted. Since SQLite3 is used, some extra steps are taken to persist the database across restarts, preventing data loss.

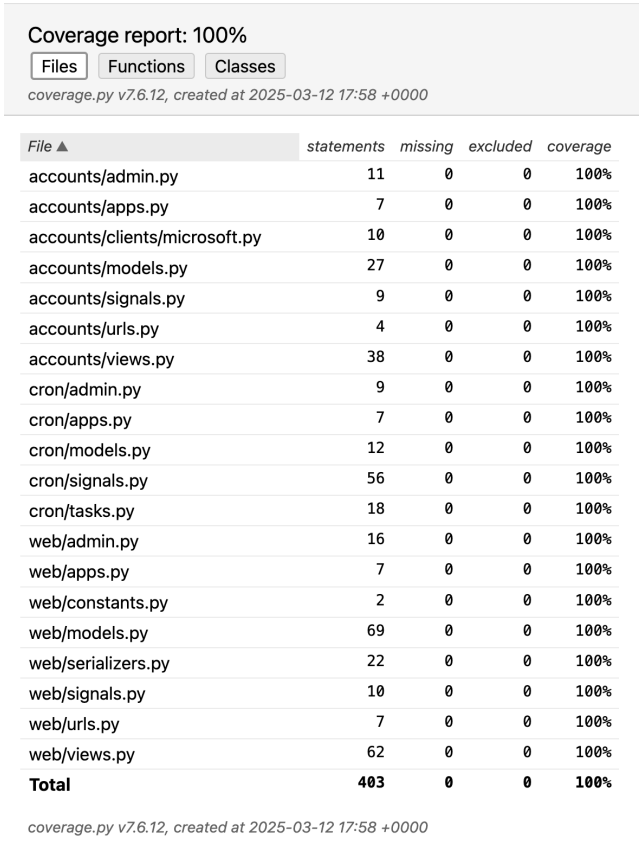
To lend the project more authenticity, a custom domain `deathnotes.tech` was purchased and configured appropriately. The frontend is linked via Vercel at https://deathnotes.tech/, while the backend is set up on https://api.deathnotes.tech/ via Nginx. This required updating the DNS A records to map the domain to the correct server IP addresses. While the frontend's A record was automatically managed by Vercel, the backend's A record was manually configured to point to the Azure VM's public IP address. Using a structured domain setup helps with the project organisation, ensuring a smooth integration between the frontend and backend servers.

Thus, leveraging automated deployment options like Vercel and GitHub actions enables the Death Notes project to follow DevOps principles, ensuring that new features and bug fixes can be deployed continuously with minimal downtime.

Chapter 4 System Testing

As the Death Notes backend contains the bulk of the application logic, including APIs and scheduling logic, unit tests were developed to ensure robustness across models, signals, authentication flows, and API endpoints. Unit tests were writing using both Django's builtin TestCase library as well as Django REST Framework's APITestCase library. These tests include:

- Model and Signal Testing: Validates the behaviour of database models, especially custom validation logic, and ensures that the signals triggered on database actions perform the business logic as expected.
- API Testing: Simulates authentication flows with JWT tokens as well as CRUD operations asserting that the views return the correct response based on API requests, along with access control.
- Mocking & Request Simulation: The `unittest.mock.path` is leveraged to mock dependencies such as email sending and Microsoft OAuth responses to prevent unnecessary external API calls.



Coverage report: 100%

Files | Functions | Classes

coverage.py v7.6.12, created at 2025-03-12 17:58 +0000

File ▲	statements	missing	excluded	coverage
accounts/admin.py	11	0	0	100%
accounts/apps.py	7	0	0	100%
accounts/clients/microsoft.py	10	0	0	100%
accounts/models.py	27	0	0	100%
accounts/signals.py	9	0	0	100%
accounts/urls.py	4	0	0	100%
accounts/views.py	38	0	0	100%
cron/admin.py	9	0	0	100%
cron/apps.py	7	0	0	100%
cron/models.py	12	0	0	100%
cron/signals.py	56	0	0	100%
cron/tasks.py	18	0	0	100%
web/admin.py	16	0	0	100%
web/apps.py	7	0	0	100%
web/constants.py	2	0	0	100%
web/models.py	69	0	0	100%
web/serializers.py	22	0	0	100%
web/signals.py	10	0	0	100%
web/urls.py	7	0	0	100%
web/views.py	62	0	0	100%
Total	**403**	**0**	**0**	**100%**

coverage.py v7.6.12, created at 2025-03-12 17:58 +0000

Figure 9: Unit Test Coverage Report

Unit test coverage was measured using the `coverage.py` tool, achieving 100% coverage with 44 tests across 403 statements, confirming that all critical backend components are tested. To enforce this standard, a GitHub action CI pipeline is configured to automatically run test coverage reports on every pull request, enforcing a minimum coverage threshold of 95% to prevent untested code from being merged, thus maintaining code standards and system stability.

Chapter 5 Appendix

1. Arshia Kaul (2976917K) – 33.33%
    - Sitemap
    - Wireframes
    - Frontend Development
    - Report Writing
2. Harish Ravichandran (2973284R) – 33.33%
    - Requirements
    - Entity Relationship Diagram
    - Testing & Deployment
    - Report Writing
3. Siddhartha Pratim Dutta (2897074D) – 33.33%
    - Introduction & Overview
    - System Architecture Diagram
    - Backend Development
    - Report Writing