Task 16 - Django – Sticky Notes Application Part 1

Practical Task

Sticky Notes Application Part 1

1. Use Case Diagram

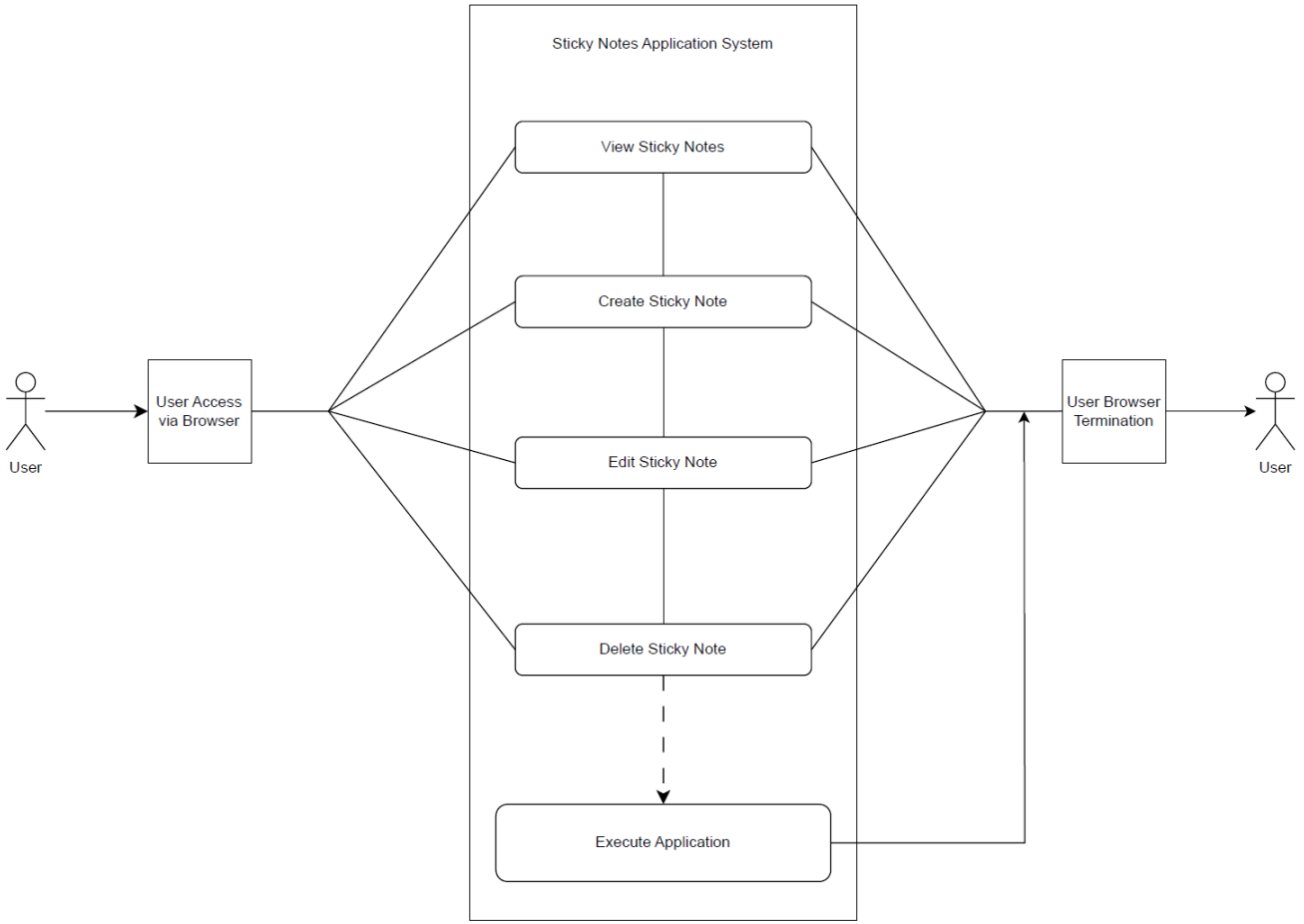

Figure 1: Use Case Diagram

1.1. Overview:

The user initiates all interactions, which encompass viewing Sticky Notes, creating new Sticky Notes, editing existing Sticky Notes, and deleting a particular Sticky Note(s).

1.2. Key Use Cases (CRUD Focus):

- **View Sticky Notes**: Shows all notes and their details.
- **Create Sticky Note**: Open a form to save (submit) a new note.
- **Edit Sticky Note**: Modify an existing note's title and / or content.
- **Delete Sticky Note**: The user is directed to a confirmation page and then deletes it via POST. Confirmation occurs before removing a Sticky Note from the database.
- **System Boundary**: All use cases take place within the Sticky Notes Application.

2. Sequence Diagram (Create Note Flow)

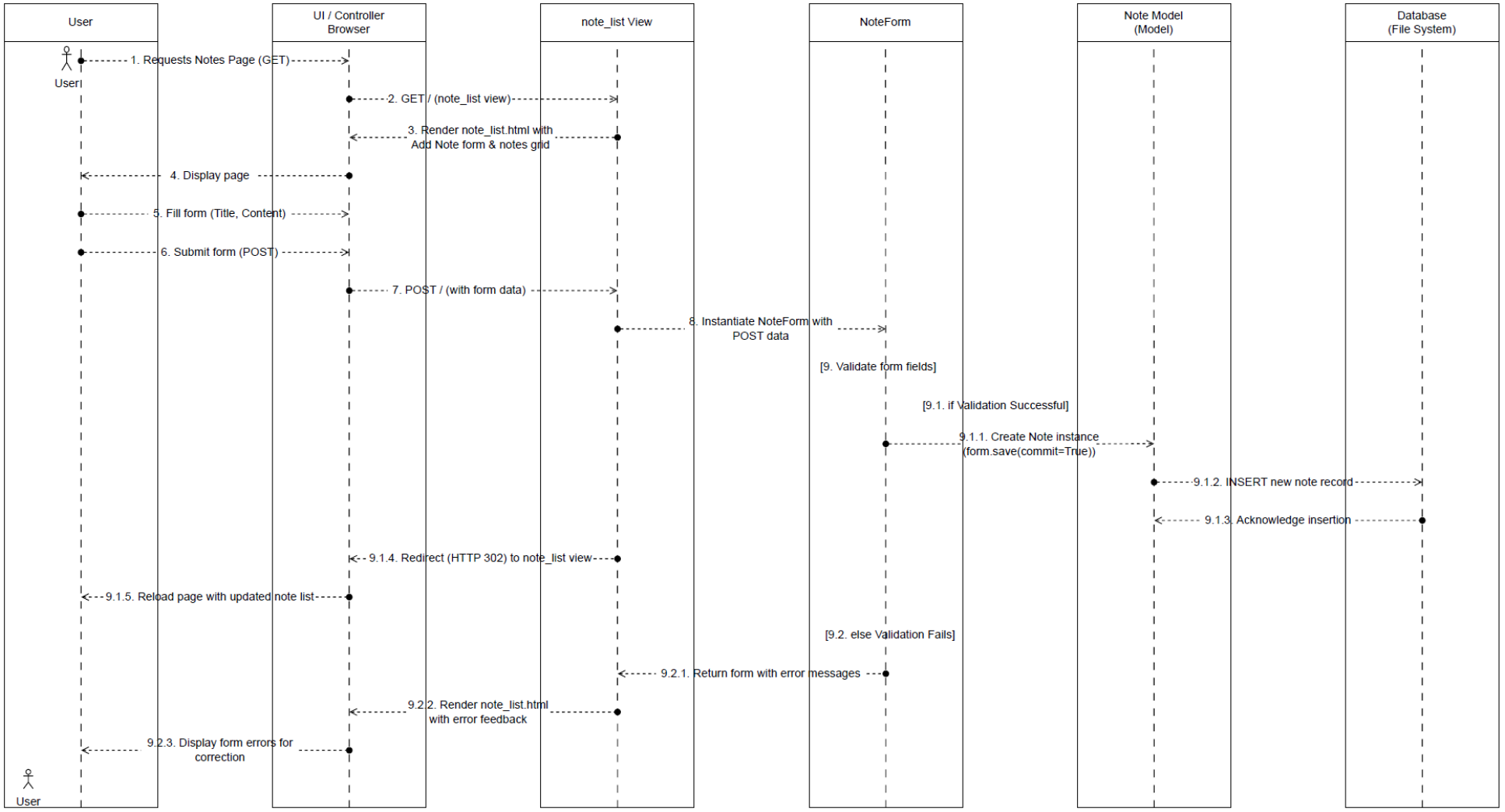

Figure 2: Sequence Diagram

This comprehensive diagram illustrates the flow of each step and emphasises how the application created manages successful and unsuccessful note-creation attempts.

2.1. Scenario:

Creating a New Sticky Note

The above sequence diagram illustrates the chronological process of a user creating a new Sticky Note. It details the interactions among the User, the Browser, the View (controller logic), the Form, the Model, and the Database. This diagram effectively depicts the steps from loading the page, completing the "Add a Note" form, submitting it, and viewing the updated list of notes.

2.2. Lifelines:

- **User (Actor)**: Initiates a request action.
- **User Interface / Controller**: Receives request (e.g., via the GET).
- **note_list View**: Renders the note_list.html template.
- **NoteForm**: Communicates with the view and is called upon when submitting data.
- **Note Model**: Validates and saves new Note object instances and handles interaction with the Database.
- **Database**: Receives Note object instances via INSERT queries and acknowledges final database insertion.

2.3. Sequential Operations:

(Representation – Representation of the above-mentioned scenario):

1. **Page Load Sequence occurs**.
   1.1. When **the User** requests the notes page (GET), the **Browser** sends this request to the note_list view. The view then renders the note_list.html template, which includes the "Add a Note" form and a grid that displays all existing notes, before presenting the page to the User.
2. **Create Note Flow Sequence takes place**.

2.1. **The User** completes the form by entering a title and content before submitting it (POST).
2.2. The **Browser** sends the POST request to the note_list view.
2.3. The view instantiates the NoteForm using the submitted data.
2.4. The **form validates** the input:
2.4.1. If valid:
2.4.1.1. The form calls the save() method to create a new Note instance.
2.4.1.2. The **Note Model** then sends an INSERT query to the Database.
2.4.1.3. **The Database** actively acknowledges this insertion made by the Note Model.
2.4.1.4. The view redirects (HTTP 302) to the notes list, prompting the **Browser** to refresh and display the updated page with the new note.
2.5. If validation fails:
2.5.1. The form displays errors that prompt the view to re-render the page with the appropriate error messages.
2.5.2. The User can view the errors and modify the input form as needed.

3. Class Diagram

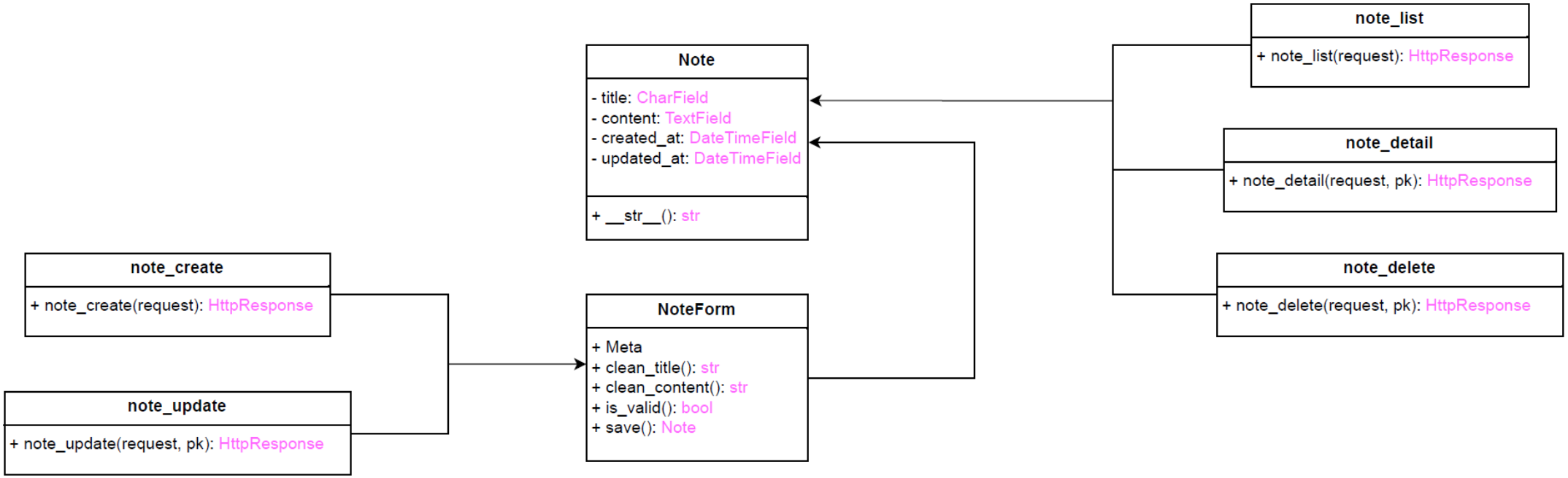

Figure 3: Application Class Diagram

The **Note model** stores essential data, including title, content, and timestamps. NoteForm serves as a Django ModelForm for adding or modifying notes. Conceptual classes represent the **Views** that *manage HTTP requests*: note_list fetches data from the **Note model** to showcase all notes. note_detail accesses a specific note using its primary key. note_create and note_update create instances of **NoteForm** to facilitate adding or modifying notes. The note_delete function retrieves a note, confirms the action, and then deletes it.

4. Conclusions and Findings

4.1. How HTTP Applications Preserve State (User Authentication & Session Management)

HTTP functions as a stateless protocol, meaning that each request from a client to a server is processed independently without retaining the memory of previous requests. However, web applications necessitate state maintenance, particularly for user authentication, personalised content, and session management. Here is the standard approach to achieving this:

- **Cookies & Session IDs**:

When a user logs in, the server generates a unique session identifier (session ID) and sends it to the client via an HTTP header (specifically, a Set-Cookie header). The client, typically a web browser, saves this cookie and automatically attaches it to future requests sent to the server.

- **Server-side Session Storage**:

On the server, frameworks such as Django utilise a session management system that links the session ID (derived from the cookie) to session data stored in a database, cache, or filesystem. This session data contains details like the user's authentication status, preferences, and other contextual information, allowing the server to "remember" the user throughout various request-response cycles.

- **Token-based Authentication (Alternative)**:

APIs and contemporary single-page applications (SPAs) can utilise stateless token mechanisms, such as JSON Web Tokens (JWTs). Upon user login, the server issues a signed token that contains user details. This token is kept on the client side, frequently in localStorage, and is included in every request, typically in an HTTP header. The server authenticates the user by validating the token with each request, eliminating the need for server-side session maintenance.

In summary, HTTP applications maintain state and facilitate user authentication and session management across multiple request-response cycles by combining client-side cookies (or tokens) with server-side storage.

4.2. Performing Django Database Migrations to a Server-based MariaDB Database

To migrate your Django project from the default SQLite or another local database to a robust relational database such as MariaDB, you need to follow several essential steps:

i. **Install MariaDB and the Python Connector**:
Make sure the MariaDB server is properly installed and operating on your server. To connect with Python, install an appropriate connector like `mysqlclient` (frequently used for MariaDB/MySQL) by executing:

```
pip install mysqlclient
```

Figure 4: pip install mysqlclient

ii. **Configure Django to Use MariaDB**:
In your `settings.py` file, modify the `DATABASES` configuration to utilise MariaDB. For instance:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',  # MariaDB is compatible with MySQL
        'NAME': 'your_database_name',
        'USER': 'your_database_user',
        'PASSWORD': 'your_database_password',
        'HOST': 'your_database_host',  # e.g., 'localhost' or a remote IP address
        'PORT': '3306',
        'OPTIONS': {
            'init_command': "SET sql_mode='STRICT_TRANS_TABLES'",
        },
    }
}
```

Figure 5: DATABASES modification configuration

The above configuration instructs Django to utilise the MySQL backend, which is compatible with MariaDB and your connection details.

iii. **Run Django Migrations**:
After updating your settings, execute the commands below to create and apply database migrations:

```
python manage.py makemigrations
python manage.py migrate
```

Figure 6: Migration Commands

iv. **Deployment Considerations**:
In production, ensure your server can access your MariaDB instance and that the user has the necessary privileges. If needed, automate your migrations (for example, through a CI/CD pipeline) to run with each deployment. Monitor your application logs for any connection or permission errors that may occur during the migration process.

This document procedure elucidates the mechanisms of state preservation in HTTP, particularly emphasising user authentication and session management. Additionally, it delineates the procedure for executing Django database migrations to MariaDB.