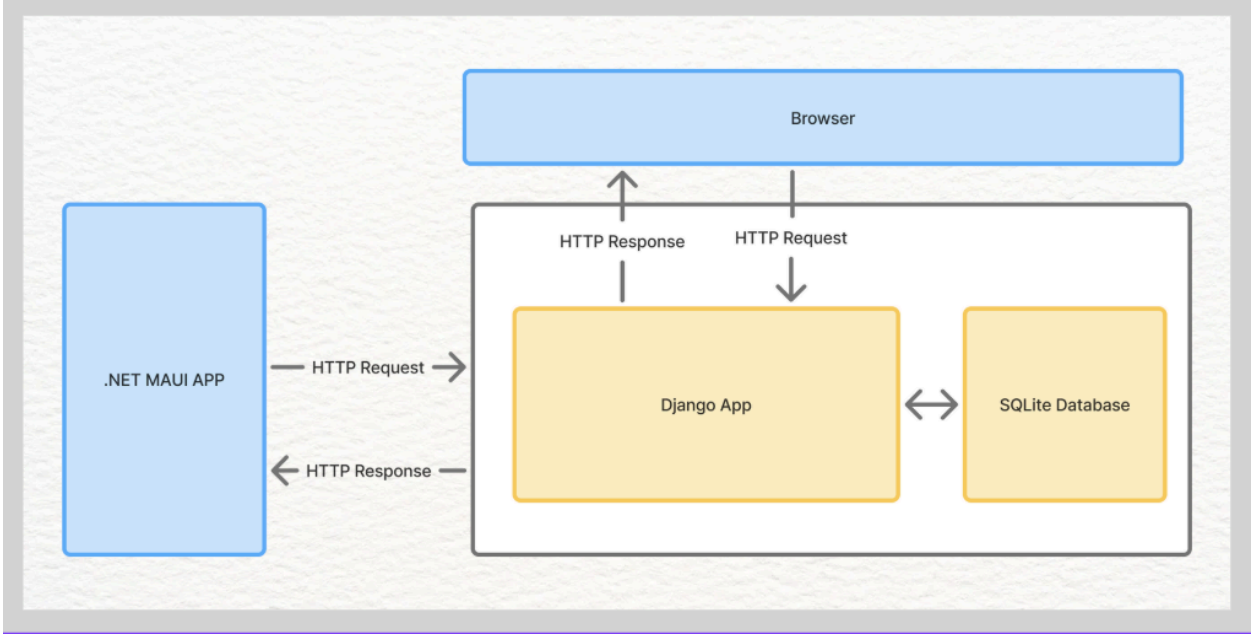
Report on Art Work Management System

1. Introduction

The aim of our system is to design and build a smart IoT system that can track digital artwork for galleries or museums. Django and .NET MAUI would be used to create our system. We will use Django to handle the server-side logic and .NET MAUI to deal with mobile applications, which are apps' user interface etc. And REST API is to communicate between Django and .NET MAUI.

2. System Architecture

- **High-level diagram** (client-server model)

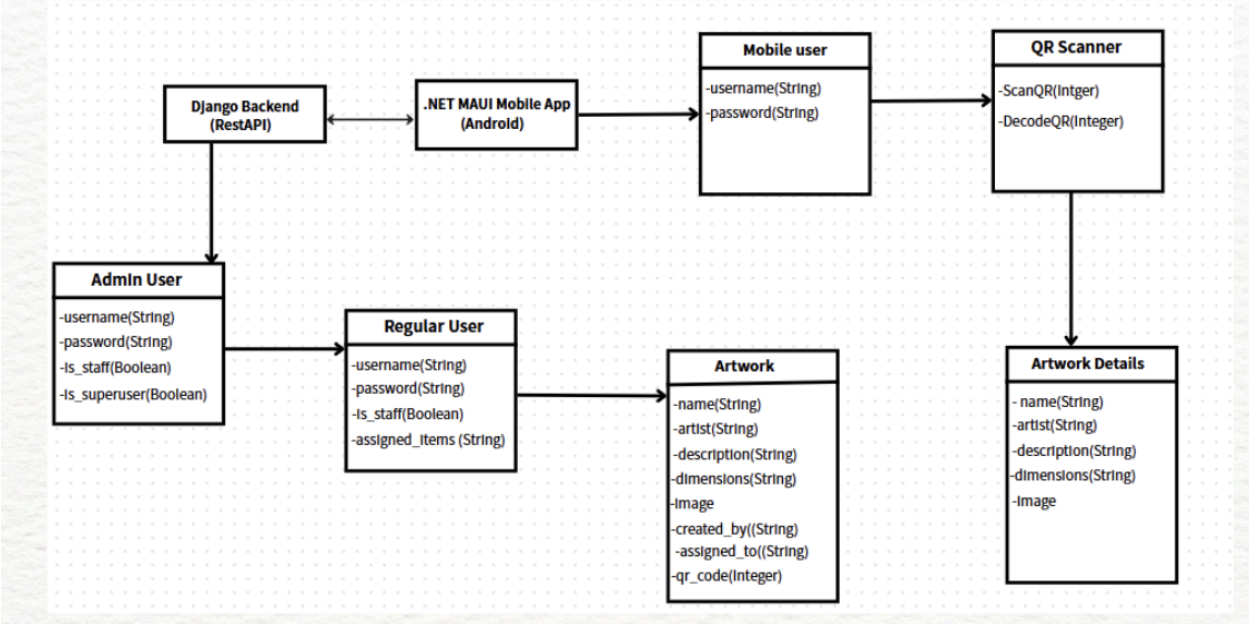


- **Backend Components**:
  - Django models (User, Artwork)
    - User Model: Stores information about both regular users and administrators. For regular users, it includes basic authentication details such as username, password (hashed), and email. Administrators have

additional permissions flags that determine their access to various administrative functions, like creating or modifying categories.
        - Artwork Model : it contains fields like name, description, artist, dimension and image.
    - Admin interface capabilities
        - User Management: Administrators can create, edit, and delete user accounts. They can assign roles and permissions to users, determining whether a user is a regular user or an administrator. Additionally, admins can view user activity logs, which record actions such as login times, artwork modifications, and inventory updates performed by each user.
        - Artwork Management: The admin interface provides a comprehensive dashboard for managing artworks. Admins can bulk-import artworks using CSV or other data formats, perform mass updates to artwork information, and manage the approval process for new artworks added by users (if applicable). They can also view detailed reports on artwork inventory, such as the number of artworks in each category, the average value of artworks, and the distribution of artworks by artist.
        - Category Management: Admins can create new categories, edit existing category names and descriptions, and delete categories that are no longer in use. They can also manage the hierarchical structure of categories, promoting or demoting categories as needed to organize the artwork collection effectively.
    - REST API endpoints (JSON responses)
        - Artwork Endpoints:
            - GET /artworks/: Retrieves a list of all artworks in the system. The response is paginated to manage large datasets, and each artwork entry in the list includes basic information such as name, artist, and a thumbnail image.
            - GET /artworks/{id}/: Fetches detailed information about a specific artwork, including all its fields such as description, dimension, category, and the full-sized image.
            - POST /artworks/: Allows authenticated users (usually with appropriate permissions) to create a new artwork. The request body should contain the artwork data in JSON format, and the API validates the input before creating the new record in the database.
            - PUT /artworks/{id}/: Enables users to update an existing artwork. Similar to the POST endpoint, the request body contains the updated data, and the API checks for data integrity and permissions before performing the update.

            - DELETE /artworks/{id}/: Deletes a specific artwork from the system. This action is restricted to users with the necessary permissions, and the API may perform additional checks, such as ensuring that the artwork is not associated with any ongoing transactions or exhibitions.
        - User Endpoints:
            - GET /users/: Returns a list of all users in the system, with basic user information like username and email.
            - GET /users/{id}/: Retrieves detailed information about a specific user, including their role, registration date, and a list of artworks in their collection.
            - POST /users/: Used for user registration. The request body contains user registration details, and the API validates the input, creates a new user account, and sends a confirmation email if configured.
            - PUT /users/{id}/: Allows users to update their own profile information (e.g., change password, update email) or enables administrators to modify user details.
            - DELETE /users/{id}/: Administrators can use this endpoint to delete a user account, along with any associated artwork records (depending on the system's configuration).
- **Mobile Components**:
    - Authentication flow
        - Initial Login: When the user first opens the mobile app, they are presented with a login screen. They can enter their username and password, which are then sent to the server-side API in a secure manner (using HTTPS). The API validates the credentials against the user database. If the credentials are correct, the API returns an authentication token (either a session token or a JSON Web Token, depending on the configuration).
        - Token Storage: The mobile app stores the authentication token securely on the device, typically in the device's keychain (for iOS) or the Android Keystore. This token is used in all subsequent API requests to authenticate the user.
        - Token Refresh: To handle scenarios where the authentication token expires, the app implements a token refresh mechanism. When the app detects an expired token during an API call, it sends a request to the server to refresh the token using a refresh token (if available). The server validates the refresh token and issues a new authentication token.
        - Logout: When the user logs out of the app, the stored authentication token is deleted from the device, and the user is redirected to the login screen.

    - Artwork detail views
        - Artwork name
        - Description
        - Artist
        - Dimensions
        - Image
- **Data Flow**:
    - How mobile app interacts with API
        - Calls API of specific ID number when pressing Find button
        - Uses user's input as ID number in the dynamic API call request
        - Catches error if item isn't found or user inputs invalid input to entry field

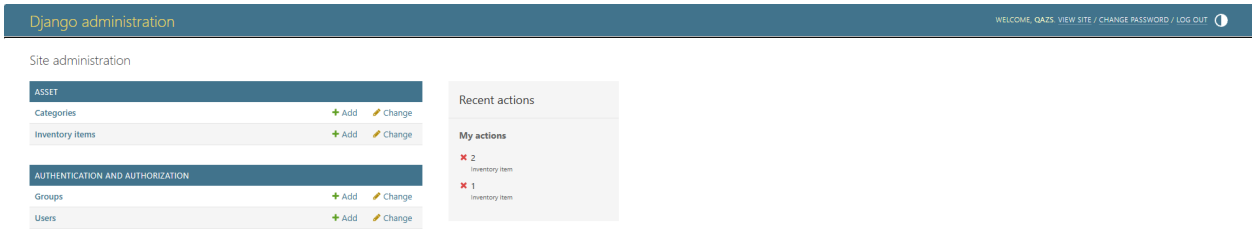


3. Key Features

There are 3 main key features in our system, which are user management, artwork management and Mobile Functionality.

For the user management, we separate between Admin and regular users. If the user is an admin user, then he is able to enter an admin page where the admin user can select a user or group to

add or change them. Admin users can also add or change the category or the inventory items.



Both admin and regular users can access the normal function, which is add, delete and edit a user's art collection. Users can control the name, category, description, artist, image and purchase Date of the art work.
For artwork management, our system can handle and store information about the artwork like image, text etc. We include four basic functions(CRUD Operations) to those data of artwork in our database or application:

1. **Create**: Add new artwork to the system.
2. **Read**: Retrieve or view existing artwork from the system.
3. **Update**: Modify or edit existing artwork in the system.
4. **Delete**: Remove artwork from the system.

For the Mobile Functionality, users can use the scan method to find artwork. User need to enters artwork id to find the artwork. And the artwork information appear

4. Technical Implementation

4.1 - Django Application

We use Django for the back end and .NET MAUI for the front end. The backend of this project was developed using the Django web framework alongside the Django REST Framework (DRF) to provide a robust API for managing and retrieving artwork data. A single core model, was implemented with the following fields:

- name: The title of the artwork
- description: A brief description of the piece
- artist: The name of the artist
- dimension: The size of the artwork
- category: A foreign key to a separate Category model
- image: The artwork’s uploaded image
- purchase_date: The date the item was purchased

The Category model was designed such that only admin users could create or modify categories. Regular users were limited to selecting from existing categories, ensuring standardization and control over the dataset.

Additionally, we used dynamic urls to create different pages for different functions. For example, in order to add new artwork, we provide a form to add a new inventory item.

```
path('add-item/', AddItem.as_view(), name='add-item'),
```

User authentication was also implemented so that it was required to access most of the backend functionality; unregistered or unauthenticated users were restricted from viewing or modifying the data.

The API was structured using DRF, exposing RESTful endpoints for CRUD operations on artwork records. Serializer classes were used to enforce data validation rules and ensure consistent formatting. We used Cross-Origin Resource Sharing (CORS) configuration as a security consideration to allow more secure communication between the mobile frontend and the backend.

4.2 - .NET MAUI App

The frontend was developed using .NET MAUI, targeting Android devices. The application consists of a simple and focused navigation structure with a primary home screen divided into two functional areas:

1. Artwork ID Entry: Users may manually enter an artwork ID
2. Information Display: Upon submitting a valid ID, the application fetches and displays relevant artwork information from the backend

A dedicated API service layer was implemented within the frontend to manage HTTP requests to the Django backend. This layer handled asynchronous network calls and deserialization of JSON responses for use within the UI. Platform-specific considerations for Android were taken into account. In particular, since the Django backend was hosted locally during development, it was necessary to route API requests through the special emulator IP address 10.0.2.2 to connect to the host machine. Additional permissions were configured in the AndroidManifest to allow internet access, as well as on the Django application to allow all hosts to connect to the API to ensure compatibility between the devices.

5. Challenges & Solutions

One of the main challenges was connecting the cross-platform API requests. Contrary to previous implementation of HTTP requests, where the endpoint is an open source API, the Django rest API had to be hosted locally. This meant that there were a lot of additional permissions that needed to be changed, for example the ALLOWED_HOST in our Django's settings.py, as well as understanding the Android emulator's IP address in order to successfully reach the API.

Since using android emulators were quite unfamiliar, it took some time to understand the different tweaks needed to connect the Django server to the emulator. Misuse of HTTPS for local HTTP endpoints caused silent failures until the protocol mismatch was identified and corrected. Debugging was made more difficult by the lack of clear error messages, which was improved by adding explicit error handling in the code. These challenges required a combination of backend configuration, network understanding, and debugging techniques, ultimately resulting in a reliable full-stack system.

6. Job Allocation

The project was a collaborative effort where each team member contributed to different aspects of the development process. The setup of the Django REST API and the core Django application was completed collectively by the entire team. Serena focused on implementing API requests on the mobile app and adjusting the Django rest framework's permissions to be compatible with android, and also contributed to the final presentation outline. Serena and Yuan worked together on Android app development, as well as the user interface. Zhehao was responsible for documenting the system by writing the developer guide and creating the UML diagram. Matthew led the creation of the presentation slides and assisted in designing the asset management user interface. The final report was a joint effort, with all members contributing to its writing and refinement.

7. Future Enhancements

- Barcode support addition
    - Functionality Expansion: Supporting multiple barcode formats such as QR codes, Code 128, and Data Matrix.
    - Barcode Generation for New Artworks: Admin users should be able to generate barcodes for newly added artworks. These barcodes can be printed and attached to the physical art pieces, facilitating easier tracking and management.
- Augmented Reality previews

    - Artwork Visualization: Implement AR technology to provide users with a more immersive experience. When a user scans an artwork's barcode or selects an artwork in the app, they can view an AR preview of how the artwork would look in a real - world setting. For example, they can place a virtual representation of a painting on their living room wall.
    - Interactive AR Features: Allow users to interact with the AR artworks. They could rotate the artwork, zoom in on details, or even see animations related to the art. For instance, a static sculpture could be shown with its original design process in an animated AR sequence.
- Offline synchronization
    - Data Caching: Implement a data caching mechanism in the mobile app so that users can access previously viewed artwork information even when they are offline. This can be achieved by storing the JSON responses from the API in a local database on the device.
    - Offline Editing and Sync: Enable users to perform basic operations such as adding, editing, or deleting artworks from their collection while offline. When the device regains an internet connection, the app can automatically synchronize these changes with the backend server.
- Multi-language support
    - Language Selection: Add a language selection feature in the app's settings, allowing users to switch between languages easily. The app should remember the user's language preference across sessions.
    - Artwork Information Translation: For the artwork information (name, description, artist bio, etc.), use machine translation services or hire professional translators to provide translations. This will make the system accessible to a global audience.
- Advanced search/filtering
    - Faceted Search: Implement faceted search, where users can filter search results by different dimensions such as artist, category, purchase date, and location. This will help users narrow down their search results more efficiently.
    - Multi - Criteria Search: Enhance the search functionality to allow users to search for artworks based on multiple criteria simultaneously. For example, users could search for paintings by a specific artist, within a certain time period, and in a particular category.

8. Conclusion

In this group project, we archive an Artwork Android app that connects to the Django framework. By conducting searches within the app, we can retrieve data from the Django backend.

During this time of project, the lesson we learnt is the importance of communication and standardization. It is very easy to confuse others if we do not standardize the code. For example, in our case the artwork folder was put under the static folder, but the static folder was not put under the artwork folder. It would confuse others but it can be fixed by communication and standardization.

Our system can be used as inventory management for galleries and museums, enhanced security and provenance tracking.

For inventory management, the system enables galleries and museums to efficiently track and manage their artwork collections digitally, which is clear, easy to sort and easy to handle large amounts of data. It is a real time update. The REST API allows the .NET MAUI mobile app to sync with the Django backend, enabling curators to update inventory details quickly. For example, A museum curator uses our mobile app to scan a QR code of a artwork, updating its data to the Django database and easily manage all the artwork.

For enhanced security and provenance tracking, the system ensures the authenticity and security of digital artworks by tracking their provenance and monitoring their status. For example, gallery worker can check the artwork gallery follow the list of artwork in the database every day.