**Assignment 1B**

**Github\_repository\_link:** https://github.com/21341A05F1-RNareshReddy/-Team\_Charmers-EH-To\_Do\_App-

**Introduction:**

The To-Do List app is designed to help users manage their tasks efficiently using a mobile interface built with React Native. The app allows users to add, edit, delete, and view tasks. The frontend interacts with a backend service to handle data operations.

**FRONTEND**

**1. Project Structure**

The frontend of the project is organized as follows:

* src/: Contains the main source code for the React Native application.
  + components/: Contains reusable components for the app.
    - AddTask.js: Component to add new tasks.
    - EditTask.js: Component to edit existing tasks.
    - Task.js: Component to display individual tasks.
    - TaskList.js: Component to display a list of tasks.
  + screens/: Contains screen components representing different views of the application.
    - HomeScreen.js: Main screen displaying the task list and navigation to add/edit tasks.
    - TaskDetailScreen.js: Screen to view details of a specific task (optional).
    - LoginScreen.js: Screen for user login.
    - RegisterScreen.js: Screen for user registration.
  + App.js: The main application component that integrates all other components.
  + api.js: Contains functions to interact with the backend API.
  + AuthContext.js: Manages authentication state for the application.

**2. Components**

**2.1 AddTask**

* Purpose: Provides a user interface for adding new tasks to the list.
* Features:
  + Input field for entering the task title.
  + Button to submit the new task.
* Functionality:
  + Sends the task title to the backend and updates the task list once the task is added.

**2.2 EditTask**

* Purpose: Allows users to edit the title of an existing task.
* Features:
  + Input field pre-filled with the current task title.
  + Button to save the changes.
* Functionality:
  + Sends the updated task title to the backend and updates the task list upon successful edit.

**2.3 Task**

* Purpose: Displays a single task item with options to edit or delete it.
* Features:
  + Displays the task title.
  + Buttons to edit or delete the task.
* Functionality:
  + Handles user actions for editing or deleting a task. On deletion, it removes the task from the backend and updates the task list. On editing, it navigates to the EditTask component.

**2.4 TaskList**

* Purpose: Renders a list of tasks.
* Features:
  + Displays a scrollable list of Task components.
* Functionality:
  + Takes a list of tasks and maps over them to render each Task component. Provides handlers for deleting tasks and initiating editing.

**3. Screens**

**3.1 LoginScreen**

* Purpose: Provides a user interface for logging in.
* Features:
  + Input fields for username and password.
  + Button to submit login credentials.
* Functionality:
  + Handles user login by integrating with the backend authentication API. Manages login state and provides feedback on authentication status.

**3.2 RegisterScreen**

* Purpose: Provides a user interface for registering a new account.
* Features:
  + Input fields for username, email, password, and confirm password.
  + Button to submit registration details.
* Functionality:
  + Handles user registration by integrating with the backend authentication API. Manages registration state and provides feedback on registration status.

**4. App.js**

* Purpose: Serves as the main component for the application, managing overall state and integrating other components.
* Functionality:
  + Manages the state for tasks and the currently editing task.
  + Handles the logic for adding, editing, and deleting tasks.
  + Integrates HomeScreen, AddTask, and EditTask components.
  + Manages navigation between different screens using React Navigation.

**5. api.js**

* Purpose: Handles communication with the backend API.
* Functionality:
  + Contains functions to fetch tasks, add new tasks, update existing tasks, and delete tasks from the backend.
  + Each function interacts with specific endpoints on the backend to perform CRUD (Create, Read, Update, Delete) operations.

**6. AuthContext.js**

* Purpose: Manages the authentication state for the application.
* Functionality:
  + Provides context for authentication state management.
  + Includes methods for logging in and out.
  + Manages user authentication state across the application.

**7. Setup and Installation**

* Install Dependencies:
  + Run npm install in the project directory to install all necessary dependencies for the React Native project.
* Start Metro Bundler:
  + Use npx react-native start to start the Metro Bundler, which serves the JavaScript bundle to the React Native application.
* Run on Android Emulator or Device:
  + Use npx react-native run-android to build and run the application on an Android emulator or connected device.

**BACKEND**

Flask API provides a basic implementation for a To-Do application with user authentication. It includes endpoints for user registration, login, and task management. The application uses MongoDB for data storage, JWT for authentication, and Flask-CORS for handling cross-origin requests.

**Setup**

**Dependencies**

* Flask
* Flask-MongoEngine
* Flask-CORS
* Flask-JWT-Extended
* MongoDB

**Configuration**

1. **Database Configuration**

* Database Name: todo\_app
* Host: localhost
* Port: 27017

1. **JWT Configuration**

* Secret Key: your\_jwt\_secret\_key

**app.py**

**Endpoints**

**1. User Registration**

**Endpoint:** /register

**Method:** POST

**Description:** Registers a new user.

**Request Body:**

{

"username": "string",

"password": "string"

}

**Responses:**

* + - **201 Created**

{

"message": "User registered successfully"

}

* + - **400 Bad Request**

{

"error": "User already exists"

}

**2. User Login**

**Endpoint:** /login

**Method:** POST

**Description:** Logs in a user and returns a JWT token.

**Request Body:**

{

"username": "string",

"password": "string"

}

**Responses:**

* + - **200 OK**

{

"token": "string"

}

* + - **401 Unauthorized**

{

"error": "Invalid credentials"

}

**3. Get Tasks**

**Endpoint:** /tasks

**Method:** GET

**Description:** Retrieves all tasks for the authenticated user.

**Authentication:** Required (JWT Token)

**Responses:**

* + - **200 OK**

[

{

"id": "string",

"user": "string",

"title": "string",

"description": "string",

"completed": "boolean",

"created\_at": "string"

}

]

**4. Add Task**

**Endpoint:** /tasks

**Method:** POST

**Description:** Adds a new task for the authenticated user.

**Authentication:** Required (JWT Token)

**Request Body:**

{

"title": "string",

"description": "string"

}

**Responses:**

* + - **201 Created**

{

"id": "string",

"user": "string",

"title": "string",

"description": "string",

"completed": false,

"created\_at": "string"

}

**5. Edit Task**

**Endpoint:** /tasks/<task\_id>

**Method:** PUT

**Description:** Updates an existing task for the authenticated user.

**Authentication:** Required (JWT Token)

**Request Body:**

{

"title": "string",

"description": "string",

"completed": "boolean"

}

**Responses:**

* + - **200 OK**

{

"message": "Task updated successfully"

}

* + - **404 Not Found**

{

"error": "Task not found"

}

**6. Delete Task**

**Endpoint:** /tasks/<task\_id>

**Method:** DELETE

**Description:** Deletes an existing task for the authenticated user.

**Authentication:** Required (JWT Token)

**Responses:**

* + - **200 OK**

{

"message": "Task deleted successfully"

}

* + - **404 Not Found**

{

"error": "Task not found"

}

**Authentication**

**JWT Token:** Required for endpoints that modify or retrieve tasks. Include the token in the Authorization header as a Bearer token.

**Error Handling**

The API returns appropriate HTTP status codes and JSON responses for errors, including:

**400 Bad Request:** Invalid input or request format.

**401 Unauthorized:** Invalid credentials or missing token.

**404 Not Found:** Requested resource does not exist.

**Auth.py**

**1.Register User**

* Endpoint: /register
* Method: POST
* Description: Registers a new user by creating a user account with a hashed password.
* Request Body:
  + Content-Type: application/json
  + Body:

{

"username": "string",

"password": "string"

}

* + Parameters:
    - username: (string) The username for the new user.
    - password: (string) The password for the new user.
* Response:
  + Status Code: 201 Created
  + Body:

{

"message": "User registered successfully"

}

* Functionality:
  + Receives the username and password in the request body.
  + Hashes the password using SHA-256 encryption.
  + Creates a new user instance and saves it to the database.
  + Returns a success message upon successful registration.

**2. Login User**

* Endpoint: /login
* Method: POST
* Description: Authenticates a user and generates a JWT token if the credentials are valid.
* Request Body:
  + Content-Type: application/json
  + Body:

{

"username": "string",

"password": "string"

}

* + Parameters:
    - username: (string) The username of the user trying to log in.
    - password: (string) The password of the user trying to log in.
* Response:
  + Status Code: 200 OK
  + Body:

{

"token": "string"

}

**(OR)**

* + Status Code: 401 Unauthorized
  + Body:

{

"message": "Invalid credentials"

}

* Functionality:
  + Receives the username and password in the request body.
  + Fetches the user from the database based on the username.
  + Checks if the provided password matches the hashed password in the database.
  + If credentials are valid, generates a JWT token and returns it.
  + If credentials are invalid, returns an error message.

**Models.py**

**1. User Model**

* Class: User
* Purpose: Represents a user in the application, storing user credentials.

Fields:

* username (db.StringField):
  + Type: String
  + Required: Yes
  + Unique: Yes
  + Description: The unique username for the user. This field must be unique for each user.
* password (db.StringField):
  + Type: String
  + Required: Yes
  + Description: The password for the user. It is stored as plain text in this model; consider hashing passwords for security in a production environment.

**2. Task Model**

* Class: Task
* Purpose: Represents a task associated with a user.

Fields:

* user (db.StringField):
  + Type: String
  + Required: Yes
  + Description: The username of the user to whom the task belongs. This field associates a task with a specific user.
* title (db.StringField):
  + Type: String
  + Required: Yes
  + Description: The title of the task. This field is mandatory when creating a task.
* description (db.StringField):
  + Type: String
  + Description: A brief description of the task. This field is optional and can be left blank if not needed.
* completed (db.BooleanField):
  + Type: Boolean
  + Default: False
  + Description: Indicates whether the task is completed. Defaults to False when the task is created.
* created\_at (db.DateTimeField):
  + Type: DateTime
  + Description: The date and time when the task was created. This field is optional and should be set to the current date and time when creating a new task.