SmartAgile Documentation

WEB

OVERVIEW

The Smart Agile Project Management web application is engineered to enhance the management of organizational projects through a user-friendly web interface. Users can efficiently update project statuses, access employee details including status and experiences, and handle tasks effectively. This application utilizes React JS for its development,ensuring a dynamic and responsive user experience. It incorporates sophisticated UI features styled with Tailwind CSS , which provides utility-first CSS classes that speed up the design process and maintain consistency across the application.

PURPOSE

This web application is designed to streamline the complexities of project management in corporate environments. It offers project managers and team members real-time access to vital project information, thereby enhancing decision-making processes and productivity. The user interface, developed using modern web technologies, provides an intuitive and responsive experience that simplifies navigation and interaction with project data.

INSTALLED PACKAGES

"@emotion/react": "^11.11.4",
"@emotion/styled": "^11.11.5",
"@fortawesome/fontawesome-free": "^6.5.2",
"@fortawesome/free-solid-svg-icons": "^6.5.2",
"@fortawesome/react-fontawesome": "^0.2.0",
"@mui/icons-material": "^5.15.15",
"@mui/material": "^5.15.15",
"chart.js": "^4.1.1",
"react": "^18.2.0",
"react-chartjs-2": "^5.2.0",
"react-dom": "^18.2.0",
"react-router-dom": "^6.22.3"

PROJECT STRUCTURE

public/
src/
|--assets
|--components
|--EmployeeComponents
|--NavBar.jsx
|--Sidebar.jsx
|--SupervisorComponents
|--pages
|--EmployeePages
|--Dashboard.jsx
|--Projects.jsx
|--Taskhub.jsx
|--Chat.jsx
|--Settings.jsx
|--SupervisorPages
|--Dashboard.jsx
|--Projects.jsx

```
            |--EmployeeView.jsx
            |--Settings.jsx
        |--LandingPage.jsx
        |--Login.jsx
        |--Organization.jsx
  |--App.jsx
  |--App.css
  |--index.css
  |--main.jsx
.eslintrc.cjs
.gitignore
README.md
package-lock.json
package.json
postcss.config.js
tailwind.config.js
vite.config.js
```

LANDING PAGE

Purpose:

The Landing Page component serves as the initial page for the application. It presents users with options to choose between organization and personal use, and upon selection, redirects them to the respective login/register pages.

Imported Dependencies:

- **React:** The Landing page component is built using React.
- **react-router-dom:** Utilizes useNavigate hook for navigation within the application.

Functions:

- **handleOrganizationClick:** Function to navigate to the login page for organization users.

Buttons:

- **"Organization" button:** Navigates to the login page for organization users.
- **"Personal" button:** Placeholder for personal user functionality (currently inactive).

LOGIN PAGE

Purpose:

The Login component serves as the user interface for user authentication within the application. It provides a form for users to input their credentials (username and password), validates the input, and handles form submission for authentication. Upon successful authentication, the user is redirected to the dashboard.

Imported Dependencies:

- **React:** The Login component is built using React.
- **react-router-dom:** Utilizes useNavigate hook for navigation within the application.
- **Fetch API:** Used for making asynchronous HTTP requests to the backend server for user authentication.

State:

- **username:** Manages the username input field.
- **password:** Manages the password input field.
- **usernameError:** Manages error messages related to the username input field.
- **passwordError:** Manages error messages related to the password input field.

Functions:

- **handleOrganizationalClick:** Function to navigate to the dashboard upon successful authentication.
- **handleSubmit:** Function to handle form submission, validate user input, and perform authentication.
- **handleRegisterClick:** Function to navigate to the organization registration page.

Form Validation:

- **Username Validation:** Checks if the username field is empty.
- **Password Validation:** Checks if the password field is empty.

Authentication:

- Uses the Fetch API to send a POST request to the backend server's login endpoint with the username and password.
- **Handles the server response:**
    - If authentication is successful (HTTP status code 200), redirects the user to the dashboard.
    - If authentication fails, display an error message to the user.

Components:

- Inputs:
    - Username input field.
    - Password input field.
- Buttons:
    - "Sign In" button: Submits the login form.
    - "Register" button: Navigates to the organization registration page.
- Links:
    - "Forgot Password?" link: Placeholder for password recovery functionality.

ORGANIZATION PAGE

Purpose:

The Organization component serves as the user interface for organization registration within the application. It provides a form for organizations to input their details such as organization name, email, website, address, owner details, and

password. It also performs client-side validation of the form fields and allows users to navigate back to the login page or submit the form.

Imported Dependencies:

- **React:** The **Organization** component is built using React.
- **react-router-dom:** Utilizes **useNavigate** hook for navigation within the application.
- **useState:** Utilizes the **useState** hook for managing component state.

State:

- **`formData`:** Manages the form data entered by the user.
- **`errors`:** Manages error messages for form validation.

Functions:

- **handleInputChange:** Function to handle changes in form input fields and update the formData state accordingly.
- **handleBackClick:** Function to navigate back to the login page.
- **handleSubmit:** Function to handle form submission, perform client-side form validation, and submit the form data.

Form Validation:

- Validates various form fields such as organization name, email, website, address, owner details, password, and confirm password.
- Displays error messages for invalid input.

Components:

- **Inputs:**
    - Organization name
    - Organization email
    - Organization website
    - Organization address

    - Owner name
    - Owner email
    - Owner mobile number
    - Password
    - Confirm password

- **Buttons:**
    - "Back" button: Navigates back to the login page.
    - "Save" button: Submits the organization registration form.

NAVBAR

Purpose:

The Navbar component serves as the navigation bar at the top of the application layout. It typically contains the application logo, user profile information, and notification indicators.

Imported Dependencies:

- **React:** The Navbar component is built using React.
- **FontAwesomeIcon:** Utilizes FontAwesome for displaying icons in the navigation bar.

State:

- **userData:** Manages user data fetched from the backend.
- **notificationCount:** Manages the count of notifications for the current user.

Management Hook:

- **useEffect:** Fetches user data from the backend when the component mounts.

Functions:

- **fetchUserData:** Asynchronous function to fetch user data from the backend API.

Components:

- **Logo:** Displays the application logo.
- **User Profile:** Displays the user's profile picture and username.
- **Notification Bell Icon:** Displays a bell icon with a notification count.

SIDEBAR

Description:

- The Sidebar component is a React functional component responsible for rendering the navigation sidebar in the user interface.
- It displays a list of navigation items, each represented by an icon and a label.
- Users can click on these items to navigate to different sections of the application.

Imported Dependencies:

- **React:** Imported to define React components.
- **useState:** Imported from React to manage component state.
- **NavLink:** Imported from `**react-router-dom**` to create navigational links.
- **FontAwesomeIcon:** Imported from `**@fortawesome/react-fontawesome**` to render Font Awesome icons.
- **faChartBar, faBriefcase, faTasks, faComments, faCog:** Specific icons imported from `**@fortawesome/free-solid-svg-icons**`.

State:

- **active:** State variable used to track the currently active navigation item.

Variables:

- **navItems:** An array containing objects representing navigation items. Each object contains the following properties:
    - **name:** The name of the navigation item.
    - **icon:** The icon associated with the navigation item.
    - **path:** The path to navigate when the item is clicked.

Component Structure:

- The Sidebar component is wrapped inside a **div** element with specific styling for width, height, background color, and font.
- It maps over the **navItems** array to render **NavLink** components for each navigation item.
- Each NavLink component represents a navigation link with the following characteristics:
    - It has a unique **key** attribute based on the index of the **navItems** array.
    - It links to the specified **path** using the **to** prop.
    - It dynamically applies styles based on whether it is active or not.
    - It updates the **active** state when clicked to reflect the currently active navigation item.
    - It displays the icon and name of the navigation item.

Event Handling:

- The **onClick** event handler is attached to each NavLink component to update the **active** state when a navigation item is clicked.

Return Statement:

- Returns the JSX markup representing the sidebar component.
- Each navigation item is represented by a NavLink component with its icon and label.
- The **active** class is conditionally applied to the NavLink component based on its active state.

EMPLOYEE DASHBOARD:

Description:

The Dashboard component is a React functional component that displays a dashboard interface. It includes a vertical bar chart showing productive and unproductive time, a calendar displaying tasks, and information about productive and unproductive time.

Modules and Tools Used:

- **React:** A JavaScript library for building user interfaces.
- **Chart.js**: A JavaScript library for creating charts.
- **FullCalendar:** A customizable calendar library for displaying events.
- **Axios:** A promise-based HTTP client for making HTTP requests.

Props:

- **tasks:** An array of tasks.
- **projects:** An array of projects.

State Variables:

- **productiveTime:** State variable to store the total productive time.
- **unproductiveTime:** State variable to store the total unproductive time.

useEffect Hook for Calculating Time:

- The useEffect hook is used to calculate productive and unproductive time based on the tasks passed as props.
- It runs whenever the tasks array changes.
- The calculateTime function iterates through the tasks and updates the state variables accordingly.

useEffect Hook for Drawing Bar Chart:

- Another useEffect hook is used to draw the vertical bar chart using Chart.js.
- It runs whenever productiveTime or unproductiveTime changes.
- It initializes a Chart.js instance with the provided data and options to draw the bar chart.

Return:

- The component returns a JSX structure representing the dashboard interface.
- It includes a title, a vertical bar chart canvas, a FullCalendar component for displaying tasks, and information about productive and unproductive time.

EMPLOYEE PROJECT PAGE:

Description:

The Projects component is a React functional component responsible for fetching and displaying a list of projects. It retrieves project data from a backend API and renders each project as a card containing information such as name, deadline, description, and an image representing the project.

State Variables:

- projects: State variable to store the list of projects fetched from the backend.

useEffect Hook for Fetching Projects:

- The useEffect hook is used to fetch projects from the backend API when the component mounts.
- It triggers the fetchProjects function to perform the API request and update the project's state accordingly.

fetchProjects Function:

- The fetchProjects function is an asynchronous function that makes a GET request to the backend API endpoint (/projects) to fetch project data.
- Upon receiving a successful response, it parses the JSON data and updates the project's state with the retrieved data.
- If an error occurs during the fetch operation, it logs the error to the console.

Return:

- The component returns a JSX structure representing the list of projects displayed as cards.
- It maps through the projects array and renders a card for each project.
- Each card includes information such as project name, deadline, tagline, description, and an image representing the project.

EMPLOYEE TASKHUB PAGE:

Description:

The TaskHub component is a React functional component responsible for displaying a task management interface. It includes sections for different task

statuses such as "To-do", "In Progress", and "Completed". Each section displays tasks along with their deadlines, team members, and additional details.

State Variables:

- userData: State variable to store user data fetched from the backend.
- notificationCount: State variable to store the count of notifications.

useEffect Hook for Fetching User Data:

- The useEffect hook is used to fetch user data when the component mounts.
- It triggers the fetchUserData function to perform the API request and update the userData and notificationCount state accordingly.

fetchUserData Function:

- The fetchUserData function is an asynchronous function that makes a GET request to the backend API endpoint (/api/user) to fetch user data.
- Upon receiving a successful response, it parses the JSON data and updates the userData state with the retrieved data.
- It also extracts the notificationCount from the user data and updates the corresponding state variable.

Return:

- The component returns a JSX structure representing the task management interface.
- It includes sections for different task statuses, each containing cards representing individual tasks.
- Each task card displays information such as task name, deadline, team members, and additional details.

EMPLOYEE CHAT PAGE:

TEAM PAGE

Description:

The Chat component is a React functional component that represents a chat interface. It displays a list of groups along with their respective team members.

Return:

- The component returns a JSX structure representing the chat interface.
- It includes a container with flex layout and vertical alignment.
- Within the container, there's a list of groups displayed as cards.
- Each group card contains information such as the group name and team members.
- The team members are represented by avatars or icons.
- Styling is applied to enhance the appearance of the interface, including background colors, shadows, and borders.

CHAT PAGE

Description:

The Chat component is a functional component responsible for implementing a real-time chat interface using WebSockets. It allows users to send and receive messages in real-time.

State Variables:

- messages: State variable to store an array of messages received from the WebSocket server.
- inputMessage: State variable to store the message entered by the user in the input field.

useEffect Hook:

- The useEffect hook is used to establish a WebSocket connection when the component mounts.
- It sets up event listeners for handling WebSocket events such as opening, receiving messages, and closing the connection.
- When a new message is received, it updates the messages state accordingly.

sendMessage Function:

- The sendMessage function is called when the user clicks the send button.
- It checks if the input message is not empty, then creates a new WebSocket connection and sends the message to the server.
- After sending the message, it clears the input field.

Render Method:

- The render() method returns a JSX structure representing the chat interface.
- It includes a heading <h1> indicating the chat title, a list <ul> to display messages, an input field for typing messages, and a send button.
- Messages are displayed as list items <li> within the unordered list using the map() function.

SETTINGS PAGE:

Description:

The PersonalSettings component is a React functional component used to display personal information fetched from a backend API. It fetches data from the backend and renders the personal information in a UI.

Modules and Tools Used:

- **React:** A JavaScript library for building user interfaces.
- **useState Hook:** A React hook that allows functional components to manage state.
- **useEffect Hook:** A React hook that performs side effects in functional components.
- **Fetch API:** A modern interface for fetching resources asynchronously across the network.
- **JSX:** A syntax extension for JavaScript used with React to describe what the UI should look like.

Component Structure:

- **useState:** Used to manage the state of personalData and loading.

- **useEffect:** Used to fetch personal data from the backend API when the component mounts.
- **fetchPersonalData:** Asynchronous function to fetch personal data from the backend API using the Fetch API.
- **Rendering Logic:** Conditional rendering based on the loading state and availability of personal data.
- **Loading Indicator:** Displays a loading message while data is being fetched.
- **Error Handling:** Displays an error message if fetching personal data fails.

Component Flow:

- The component initially sets loading to true.
- When the component mounts, useEffect is triggered, which calls the fetchPersonalData function.
- The fetchPersonalData function makes a GET request to the backend API endpoint /api/personal-info.
- If the request is successful (response.ok), the received JSON data is stored in the personalData state.
- If the request fails, an error message is logged, and loading is set to false.
- The component renders based on the state:
- If loading is true, a loading message is displayed.
- If personalData exists, the personal information is rendered.
- If no personal data is available or fetching fails, an error message is displayed.

SUPERVISOR DASHBOARD PAGE:

Description:

The SupervisorDashboard component is a React component designed to display various data related to projects, tasks, and team members in a supervisor's dashboard. It includes functionality to fetch data from an API, calculate performance metrics, and visualize them using Chart.js library.

Modules and Tools Used:

- **useState:** React hook for managing state variables.
- **useEffect:** React hook for performing side effects in function components.
- **Chart.js:** A JavaScript library for creating charts and visualizations.
- **fetch:** A modern API for making asynchronous HTTP requests in the browser.

Component Features:

- **State Management:** Utilizes the useState hook to manage state variables for projects, tasks, and team members fetched from the backend API.
- **Data Fetching:** Uses the useEffect hook to fetch data from the backend API when the component mounts.
- **Performance Metrics Calculation:** Calculates performance metrics such as overall project progress and employee task distribution.
- **Chart Visualization:** Utilizes Chart.js library to create visual representations of performance metrics.
- **Vertical Bar Chart:** Displays the company's overall progress as a vertical bar chart.
- **Horizontal Bar Chart:** Visualizes employee performance in terms of task distribution using a horizontal bar chart.
- **API Integration:** Fetches project, task, and team member data from the backend API endpoints /api/projects, /api/tasks, and /api/team-members respectively.

Code Documentation:

- **useState and useEffect Hooks:** Used to manage component state and perform side effects such as data fetching.
- **Fetch Data:** Fetches project, task, and team member data from the backend API using the fetch function.
- **Calculate Metrics:** Calculates performance metrics like overall project progress and employee task distribution.
- **Draw Charts:** Uses Chart.js library to draw vertical and horizontal bar charts for visualizing performance metrics.

- **Canvas Elements:** Creates canvas elements with unique IDs to render the charts.
- **Component Structure:** Divides the dashboard into sections for displaying charts, project lists, task lists, and team member lists.

SUPERVISOR PROJECT PAGE:

Description:

The SupervisorProjectPage component is responsible for displaying project details, including tasks associated with the project and team members involved.

Props:

- **projectId:** A unique identifier for the project whose details are to be displayed.

State Variables:

- **project:** Stores the details of the project fetched from the API.
- **tasks:** Stores the list of tasks associated with the project.
- **teamMembers:** Stores the list of team members associated with the project.

Hooks Used:

- **useState:** Used to define state variables for project, tasks, and teamMembers.
- **useEffect:** Used to fetch project details, tasks, and team members from the API when the component mounts or when the projectId prop changes.

Dependencies:

- **React:** The core library for building user interfaces in React.js.
- **useState:** A React hook for managing state within functional components.
- **useEffect:** A React hook for performing side effects in functional components.

- **fetch:** A built-in JavaScript function for making HTTP requests to fetch data from APIs.

Component Structure:

- **Project Name Header:** Displays the name of the project.
- **Tasks Section:** Displays a list of tasks associated with the project.
- **Team Members Section:** Displays a list of team members associated with the project.

Implementation Details:

- The component fetches project details, tasks, and team members from the backend API using the fetch function.
- Data fetching is triggered when the component mounts or when the projectId prop changes.
- Fetched data is stored in state variables (project, tasks, teamMembers) and rendered in the component.

SUPERVISOR EMPLOYEE VIEW

Description:

The SupervisorEmployeeViewPage component is responsible for displaying information about a specific employee, including the projects they are assigned to and the tasks they are responsible for.

Props:

- employeeId: A unique identifier for the employee whose details are to be displayed.

State Variables:

- **employee:** Stores the details of the employee fetched from the API.
- **projects:** Stores the list of projects associated with the employee.
- **tasks:** Stores the list of tasks assigned to the employee.

Hooks Used:

- **useState:** Used to define state variables for employee, projects, and tasks.
- **useEffect:** Used to fetch employee details, projects, and tasks from the API when the component mounts or when the employeeId prop changes.

Dependencies:

- **React:** The core library for building user interfaces in React.js.
- **useState:** A React hook for managing state within functional components.
- **useEffect:** A React hook for performing side effects in functional components.
- **fetch:** A built-in JavaScript function for making HTTP requests to fetch data from APIs.

Component Structure:

- **Employee Name Header:** Displays the name of the employee.
- **Projects Section:** Displays a list of projects associated with the employee.
- **Tasks Section:** Displays a list of tasks assigned to the employee.

Implementation Details:

- The component fetches employee details, projects, and tasks from the backend API using the fetch function.
- Data fetching is triggered when the component mounts or when the employeeId prop changes.
- Fetched data is stored in state variables (employee, projects, tasks) and rendered in the component.

SUPERVISOR SETTING PAGE

Description:

The SupervisorSettingPage component allows supervisors to toggle between light and dark themes. It also updates the theme setting in the backend API.

State Variables:

- **theme:** Stores the current theme mode, either 'light' or 'dark'.

Hooks Used:

- **useState:** Used to define state variable theme.
- **useEffect:** Used to perform side effects, specifically to update the theme setting in the backend API when the theme state changes.

Dependencies:

- **React:** The core library for building user interfaces in React.js.
- **useState:** A React hook for managing state within functional components.
- **useEffect:** A React hook for performing side effects in functional components.
- **fetch:** A built-in JavaScript function for making HTTP requests to interact with backend APIs.

Component Structure:

- **Settings Header:** Displays a heading for the settings page.
- **Toggle Theme Button:** Allows the user to toggle between light and dark themes.
- **Current Theme Display:** Shows the currently selected theme ('light' or 'dark').

Implementation Details:

- The component uses the useState hook to manage the current theme state (theme) and toggles between 'light' and 'dark' themes.
- When the theme changes, the component updates the theme setting in the backend API using the fetch function within the useEffect hook.
- The toggleTheme function is called when the user clicks the "Toggle Theme" button, which updates the theme state accordingly.
- The useEffect hook ensures that the theme setting is updated in the API whenever the theme state changes.

BACKEND

env/
  |--Lib/
  |--Scripts/
  |--pyvenv.cfg
Organization/
  |--__pycache__/
  |--migrations/
  |--__init__.py
  |--admin.py
  |--apps.py
  |--models.py
  |--serializers.py
  |--tests.py
  |--urls.py
  |--views.py
Projects/
  |--__pycache__/
  |--migrations/
  |--__init__.py
  |--admin.py
  |--apps.py
  |--models.py
  |--serializers.py
  |--tests.py
  |--urls.py
  |--views.py
SmartAgile_Backend/
  |--__pycache__/
  |--__init__.py
  |--asgi.py
  |--settings.py
  |--urls.py

```
   |--wsgi.py
Tasks/
   |--__pycache__/
   |--migrations/
   |--__init__.py
   |--admin.py
   |--apps.py
   |--models.py
   |--serializers.py
   |--tests.py
   |--urls.py
   |--views.py
Users/
   |--__pycache__/
   |--migrations/
   |--__init__.py
   |--admin.py
   |--apps.py
   |--backends.py
   |--forgot_password_views.py
   |--models.py
   |--serializers.py
   |--tests.py
   |--urls.py
   |--utils.py
   |--views.py
|--db.sqlite3
|--manage.py
```

APPS:

1.Organization

OVERVIEW:

The Organization is an app which holds data of all the Organizations for the SmartAgile. This app will make the organization to register and will authenticate each Organization.

models.py

Class

1.Organization: The Organization model represents entities in the application that hold information about different organizations.

Fields:

org_id - It represents the unique identifier for each instance of the model.Setting "primary_key=True" ensures that this field serves as the primary key for the corresponding database table.
org_name - This is the field where the name of the organization is stored
org_mail - This is the field where organization email id is stored
org_website - This is the field where the URL of the organization website is stored
Owner - This field establishes a relationship with instances of the User model, representing that each instance of this model is owned by a single user

serializers.py

Class

1. **OrganizationSerializer**
    - **Meta Inner Class**
      The Meta inner class within the OrganizationSerializer class is used to provide metadata to the serializer.
      model: This attribute specifies the Django model class that the serializer is based on the model Organization.
      fields: This attribute specifies which fields from the model should be included in the serialized output. In this case, '__all__' is used.

views.py

Class

1. **OrganizationView -** The OrganizationView class handles various HTTP methods (GET, POST, PUT, DELETE) for managing organizations.

Methods:

1. **get Method**

Retrieving Organizations:

- If org_id is not provided (i.e., None), the method retrieves all organizations from the database using "Organization.objects.all()".
- It then serializes the queryset of organizations using OrganizationSerializer with "many=True" to serialize multiple objects.
- Finally, it returns a response with the serialized data.

Retrieving a Specific Organization:

- If org_id is provided, it attempts to retrieve the organization with that ID using "Organization.objects.get(org_id=org_id)".
- If the organization exists, it serializes the organization using OrganizationSerializer.
- If the organization does not exist, it returns a 404 Not Found response with an error message.

2. **post Method**

Data Validation:

- It attempts to deserialize the request data using "OrganizationSerializer(data=request.data)" .
- If the data is valid, it proceeds to save the new organization to the database using "new_organization.save()" .
- If the data is invalid, it returns a 400 Bad Request response with the validation errors.

3. **put Method**

Retrieving and Updating Organization:

- It retrieves the existing organization with the provided org_id using “Organization.objects.get(org_id=org_id)” .
- It attempts to update the organization instance with the request data using “OrganizationSerializer(organization_update, data=request.data)” .
- If the data is valid, it saves the updated organization using “new_update.save()”.
- If the organization does not exist, it returns a 404 Not Found response with an error message.
- If the data is invalid, it returns a 400 Bad Request response with the validation errors.

4. **delete Method**

Deleting Organization:

- It retrieves the organization with the provided org_id using “Organization.objects.get(org_id=org_id)” .
- It attempts to delete the organization using “organization_delete.delete()” .
- If the organization is successfully deleted, it returns a 200 OK response with a success message.
- If the organization does not exist, it returns a 404 Not Found response with an error message.

urls.py

Path: "
View: OrganizationView
HTTP Methods Supported:

- **GET:** Retrieves details of all organizations.
- **POST:** Creates a new organization.

Name: 'organization_view'
Usage:

- **GET:** /organizations/

- **POST:** /organizations/

Path: '<int:org_id>/'
View: OrganizationView
HTTP Methods Supported:
- **GET:** Retrieves details of a specific organization by ID.
- **PUT:** Updates an existing organization by ID.
- **DELETE:** Deletes an organization by ID.

Name: 'organization_view_id'
Usage:
- **GET:** /organizations/<org_id>/
- **PUT:** /organizations/<org_id>/
- **DELETE:** /organizations/<org_id>/

2.Users

OVERVIEW:
The Users is an app which holds the data of all the users in the SmartAgile. This app contains the models to create users and maps to the organization using Foreign key and the app provides token authentication for each user in the SmartAgile.

models.py

Class

1. **UserProfile :** The UserProfile model represents additional information associated with a user, including their organization affiliation.

   Fields:
   - **user -** This field represents a one-to-one relationship with the User model provided by Django's authentication system. It establishes a

link between a user and their profile. Each user can have only one profile, and vice versa

- **organization** - This field represents a many-to-one relationship with the Organization model. It links the user profile to the organization the user belongs to. Each user profile can be associated with one organization, while an organization can have multiple user profiles.

2. **User** : The User model is a custom user model that extends Django's built-in AbstractUser model. It represents users in the system and includes additional fields and functionalities tailored to your application's requirements. Username , password , email , first_name , last_name all these fields are Inherits from AbstractUser

   Fields:

   - **Otp -** This field represents the one-time password (OTP) associated with each user. It's a character field with a maximum length of 6 characters. It's nullable and blank, allowing users to have no OTP initially.
   - **groups** - This field represents a many-to-many relationship with the Group model provided by Django's authentication system. It allows users to belong to multiple groups, granting them access to group-specific permissions and functionalities.
   - **user_permissions** - This field represents a many-to-many relationship with the Permission model provided by Django's authentication system. It allows users to have specific permissions granted to them individually, regardless of their group memberships.

serializers.py

Class

1. **UserSerializer:** It inherits from serializers.ModelSerializer to simplify the serialization process of user data.
   - **Meta Inner Class**

The Meta inner class within the UserSerializer class is used to provide metadata to the serializer.
model: This attribute specifies the Django model class that the serializer is based on the model User.
fields: This attribute specifies which fields from the model should be included in the serialized output. In this case, 'username, email, password' is used.

Create Method

Functionality:

- It accepts validated data, likely from a user serializer, to create new user accounts.
- It first creates a new user object using User.objects.create, populating the username and email from the validated data.
- Importantly, it retrieves the password but doesn't store it plainly.
- Instead, it utilizes the user model's set_password method to securely hash the password before saving the user object.
- Finally, it persists the user with the hashed password to the database using user.save and returns the newly created user object, likely for further processing or inclusion in a user response.

Validate_email Method

Functionality:

- It takes the provided email (value) and uses User.objects.filter(email = value) to query the database for existing users with that email.
- If any users are found (indicating a duplicate), it raises a ValidationError with a message indicating an existing account with that email, preventing duplicate accounts.
- Conversely, if the email filter returns nothing (meaning the email is unique), the function simply returns the original email (value) unchanged, allowing it to be used for creating the new user account.

2. **LoginSerializer:** It inherits from serializers.Serializer to simplify the serialization process of User login.

    - The LoginSerializer class defines a schema for validating user login credentials in a Django REST framework application.
    - Inheriting from serializers.Serializer, it specifies two fields: an email field using serializers.EmailField to ensure a valid email address is provided, and a password field created using serializers.CharField.
    - This password field has a special configuration (style={'input_type': 'password'}) that renders it as a password input field in the user interface, masking the entered password for security.

3. **PasswordResetRequestSerializer:**

    Validate Method

    Functionality:
    - It extracts the email from the data dictionary and attempts to find a matching user in the database using User.objects.get(email=email).
    - If the email is invalid (causing a User.DoesNotExist exception), it raises a serializers.ValidationError to prevent sending password resets to non-existent users.
    - Conversely, if a valid user is found with the email, the function simply returns the entire validated data (likely assuming only email needs validation), allowing the password reset process to proceed with the verified email address.

4. **PasswordResetConfirmOtpSerializer:**

    Validate Method

    Functionality:
    - The validate function in a serializer works during password reset confirmation by verifying the provided one-time password (OTP).

- It extracts the OTP from the data and attempts to find a matching user using User.objects.get(otp=otp).
- If no user is found with that specific OTP (indicating an invalid OTP), it raises a serializers.ValidationError with an "Invalid OTP" message to prevent unauthorized resets.
- Conversely, if a valid user is found with the matching OTP, the function simply returns the entire validated data (likely assuming only OTP needs validation), allowing the password reset process to proceed with the verified user.

5. **PasswordResetConfirmSerializer:**

   Validate Method

   Functionality:
   - It retrieves the OTP, new password, and confirm password from the data.
   - It then ensures both passwords are provided and rejects missing entries with a "Passwords are required" error.
   - Next, it enforces a minimum password length of 8 characters, raising an "Password must be at least 8 characters long" error for weak passwords.
   - It also verifies that the new password and confirm password match, raising a "Passwords do not match" error to prevent typos. Finally, it validates the OTP by looking up a user with the provided OTP.
   - If the OTP is invalid (causing a User.DoesNotExist exception), it raises an "Invalid OTP" error.
   - If all checks pass, the function simply returns the entire validated data, all necessary validations are complete for the password reset process to proceed with the verified user.

   Update Method

   Functionality:

- The update function serves as the final step in password reset. The data is already validated, it retrieves the user associated with the provided OTP.
- It then securely sets the user's new password using user.set_password. Since the OTP is no longer needed, it's invalidated by setting the user's otp field to None.
- Finally, it persists these changes (new password and invalidated OTP) to the database using user.save and returns the updated user object, likely for further use in the password reset workflow.

6. **SuperuserSerializer:** It inherits from serializers.ModelSerializer to simplify the serialization process of superuser data.
   - **Meta Inner Class**
     The Meta inner class within the SuperuserSerializer class is used to provide metadata to the serializer.
     model: This attribute specifies the Django model class that the serializer is based on the model User.
     fields: This attribute specifies which fields from the model should be included in the serialized output. In this case, 'id, username, email, password' is used.

   Create Method

   Functionality:
   - It accepts validated data, likely from a user serializer, to create new user accounts.
   - It first creates a new user object using User.objects.create, populating the username and email from the validated data.
   - Importantly, it retrieves the password but doesn't store it plainly.
   - Instead, it utilizes the user model's set_password method to securely hash the password before saving the user object.
   - Finally, it persists the superuser with the hashed password to the database using user.save and returns the newly created superuser object, likely for further processing or inclusion in a user response.

Validate_email Method

Functionality:

- It takes the provided email (value) and uses User.objects.filter(email = value) to query the database for existing users with that email.
- If any users are found (indicating a duplicate), it raises a ValidationError with a message indicating an existing account with that email, preventing duplicate accounts.
- Conversely, if the email filter returns nothing (meaning the email is unique), the function simply returns the original email (value) unchanged, allowing it to be used for creating the new superuser account.

7. **UserProfileSerializer:** It inherits from serializers.ModelSerializer to simplify the serialization process of user data.
   - **Meta Inner Class**
     The Meta inner class within the UserProfileSerializer class is used to provide metadata to the serializer.
     model: This attribute specifies the Django model class that the serializer is based on the model UserProfile.
     fields: This attribute specifies which fields from the model should be included in the serialized output. In this case, '__all__' is used.

views.py

Class

1. **MyTokenObtainPairSerializer**

- Inheriting from TokenObtainPairSerializer within a Django REST framework application, MyTokenObtainPairSerializer customizes token generation during user login.
- It overrides the get_token method to leverage the parent class's functionality of retrieving a standard token response with access and refresh token information.

- However, it goes a step further by enhancing the token response. It adds the user's username to the token dictionary, although it includes the username twice likely due to a bug.
- This customization allows the token response to include not only standard token data but also the user's username, potentially for further use within the application logic.

2. **MyTokenObtainPairView**

- MyTokenObtainPairView class specializes in user login and token generation.
- It inherits from TokenObtainPairView, gaining the core functionalities for handling token requests. The key customization lies in setting serializer_class to MyTokenObtainPairSerializer.
- This instructs the view to use a custom serializer during login for handling token data.

3. **UsersView**

- The UsersView class acts as an API endpoint for managing user data. Inheriting from ListCreateAPIView, it offers functionalities for both listing and creating users.
- The view fetches all user data from the database using “queryset = User.objects.all()”, providing access to a complete list of users. It then utilizes UserSerializer defined in the serializer_class attribute.
- This serializer plays a key role in formatting user data during API interactions, ensuring proper data exchange between the application and the users.
- Overall, UsersView empowers users to interact with user data through API calls, potentially allowing them to retrieve user lists or create new users based on the permissions set within the application.

4. **UserEditUpdateDeleteView**

- The UserEditUpdateDeleteView class acts as an API endpoint for managing individual user data. Inheriting from RetrieveUpdateDestroyAPIView, it provides functionalities for CRUD (Create, Read, Update, Delete) operations on user objects.
- Similar to UsersView, it retrieves all user data using queryset = User.objects.all() for comprehensive access.
- It then utilizes the UserSerializer defined in serializer_class to format user data during API interactions. To distinguish between specific users, it sets lookup_field = 'id'.
- This means the API endpoint expects a user ID to be included in the URL, allowing for targeted actions.
- Essentially, this view empowers users to not only retrieve a complete user list but also to edit specific user details, update their data, or even delete users entirely, based on the assigned permissions within the Django REST framework application.

5. LoginView

post Method

Functionality:

- The LoginView acts as a login endpoint for users. It extracts email and password from the request data.
- It then attempts to authenticate the user using Django's built-in mechanism.
- If the credentials are invalid, it returns an "Invalid Credentials" error. Upon successful authentication, it retrieves the user's profile along with organization details in an optimized way.
- It checks for missing profiles and returns an appropriate error if none exists. If the profile is found, it generates access and refresh tokens.
- Finally, it returns a comprehensive response containing tokens, user information, organization data, and success messages upon successful login.

6. SuperuserCreate

- The SuperuserCreate class acts as an API endpoint for managing user data. Inheriting from ListCreateAPIView, it offers functionalities for both listing and creating superusers.
- The view fetches all user data from the database using “queryset = User.objects.all()”, providing access to a complete list of superusers. It then utilizes SuperuserSerializer defined in the serializer_class attribute.
- This serializer plays a key role in formatting user data during API interactions, ensuring proper data exchange between the application and the superusers.
- Overall, UsersView empowers users to interact with superuser data through API calls, potentially allowing them to retrieve user lists or create new users based on the permissions set within the application.

7. **SuperuserViewEditDelete**

- The SuperuserViewEditDelete class acts as an API endpoint for managing individual superuser data. Inheriting from RetrieveUpdateDestroyAPIView, it provides functionalities for CRUD (Create, Read, Update, Delete) operations on superser objects.
- Similar to SuperuserCreate, it retrieves all superuser data using queryset = User.objects.all() for comprehensive access.
- It then utilizes the SuperuserSerializer defined in serializer_class to format user data during API interactions. To distinguish between specific superusers, it sets lookup_field = 'id'.
- This means the API endpoint expects a user ID to be included in the URL, allowing for targeted actions.
- Essentially, this view empowers superusers to not only retrieve a complete superuser list but also to edit specific superuser details, update their data, or even delete users entirely, based on the assigned permissions.

8. **UserProfileCreate-** The UserProfileCreate class handles various HTTP methods (GET, POST, PUT, DELETE) for managing organizations.

Get Method

Functionality:

- It handles both fetching a specific profile and retrieving a list of all profiles.
- If an ID is provided in the URL, it attempts to find a matching user profile using that ID.
- Upon success, it creates a serializer to format the retrieved data and returns it within a response. However, the error message for missing profiles ("Invalid Credentials") could be improved for clarity.
- If no ID is included in the URL, it fetches all user profiles and utilizes a serializer set to handle multiple instances to format the data for all profiles before returning them in a single response.

Post Method

Functionality:

- It starts by deserializing the request data into a UserProfileSerializer object, attempting to map the provided data to the serializer's defined fields.
- Next, it performs validation to ensure the data adheres to the serializer's field definitions and any custom validations.
- If the data is valid, it persists the data to the database, creating a new user profile.
- Upon successful creation, it returns a success message with a 201 Created status code.
- However, if the data is invalid during deserialization or validation, it returns a response containing the specific errors associated with the serializer, potentially helping the user rectify the issues before submitting the data for profile creation.

Put Method

Functionality:

- It retrieves the profile to be updated based on the ID provided in the URL.
- If the profile doesn't exist, it returns an error message indicating a missing profile with a more specific message (not "Invalid Credentials").
- It then merges the retrieved profile data with the update data from the user's request.
- This merged data goes through validation to ensure it adheres to the serializer's rules.
- Upon successful validation, the changes are saved to the database, updating the user profile. The function responds with a success message and the updated profile information.
- However, if the data is invalid during processing, it returns a response containing the specific errors associated with the update, allowing the user to fix the issues and retry the update process.

Delete Method

Functionality:

- It retrieves the profile to be deleted based on the ID provided in the URL.
- If the profile doesn't exist, it returns an error message indicating a missing profile, although a more specific message would be clearer (not "Invalid Credentials").
- Upon finding the profile, it proceeds with deletion from the database. If the deletion is successful, the function responds with a "Successfully deleted" message.
- Overall, it offers an endpoint for users to delete their profiles through a DELETE request.

backends.py

Class

8. **EmailBackend :** The EmailBackend class extends ModelBackend, which is a built-in authentication backend provided by Django. It's responsible for authenticating users based on their email addresses.
   authenticate Method

   Purpose : This method is called by Django's authentication system to authenticate a user based on the provided email address and password.

   Functionality:
   - It first checks if the username is None. If so, it attempts to retrieve the username from the keyword arguments using "UserModel.USERNAME_FIELD" .
   - This allows flexibility in specifying the username field (e.g., username or email) based on the custom user model.
   - It then attempts to retrieve a user object from the database using the provided email address (username).
   - If a user with the provided email address exists, it checks if the provided password matches the user's stored password using "user.check_password(password)" .
   - If the password matches, it returns the authenticated user.
   - If the user does not exist or the password does not match, it returns None, indicating authentication failure.

   get_user Method

   Purpose : This method is called by Django's authentication system to retrieve a user object based on the user ID stored in the session.

   Functionality:
   - It attempts to retrieve a user object from the database using the provided user ID (user_id).
   - If a user with the provided ID exists, it returns the user object.
   - If the user does not exist, it returns None.

forgot_password_views.py

Class

1. **PasswordResetRequestView:** This PasswordResetRequestView class is designed to handle password reset requests within a Django web application. It inherits from Django REST framework's GenericAPIView for simplified API view creation.

   Post Method

   Functionality:

   - It takes in the user's POST request data, which likely contains an email address.
   - This data is then validated using a serializer to ensure it's a valid email format.
   - If the validation passes, the function attempts to find a user in the database matching that email.
   - Upon finding the user, it generates a unique token and a one-time password (OTP) specifically for password reset. These are then saved to the user's model in the database.
   - The function doesn't stop there; it likely calls another function (send_password_reset_email) to send an email to the user containing a password reset link or instructions on how to reset their password using the generated token.
   - Finally, depending on whether the validation or user lookup was successful, the function returns a corresponding response.
   - It returns a success message with a 200 (OK) status code if everything went smoothly.
   - On the other hand, if the data validation fails (e.g., invalid email), it returns the validation errors along with a 400 (Bad Request) status code.

2. **PasswordResetConfirmOtpView:** The PasswordResetConfirmOtpView deals with confirming a one-time password (OTP) for password resets. It inherits from UpdateAPIView to handle updates on user data

   Post Method

   Functionality:
   - In the post function of the PasswordResetConfirmOtpView class, users confirm a one-time password (OTP) used for password resets.
   - It first validates the user's POST request data using a PasswordResetConfirmOtpSerializer, likely checking for a valid OTP format.
   - If valid, it attempts to find a matching user in the database based on the provided OTP using User.objects.get(otp=otp).
   - However, the user lookup can fail due to an invalid OTP (e.g., expired or not matching a user) or no user being found with that specific OTP.
   - In either case, a serializers.ValidationError with an "Invalid OTP" message is raised.
   - If a user is successfully found with the matching OTP, the function bypasses the error handling and returns a success message indicating verified OTP and the ability to proceed with password reset.
   - This success response is accompanied by an HTTP status code of 200 (OK).

3. **PasswordResetConfirmView:** The PasswordResetConfirmView class, deals with finalizing password resets after OTP confirmation. It inherits from UpdateAPIView to handle user data updates.

   Post Method

   Functionality:
   - In the post function of the PasswordResetConfirmView class, the password reset process is finalized after a confirmed OTP.

- It first validates the user's POST request data using a PasswordResetConfirmSerializer, likely checking for password requirements.
- If valid, it attempts to retrieve the user associated with the confirmed OTP.
- If successful, the function uses set_password to securely update the user's password with the provided new password. It then clears the used OTP and saves the changes to the database.
- Finally, it returns a success message with a 200 (OK) status code upon reset completion.
- If the user lookup fails (likely due to an invalid OTP), it returns the validation errors and a 400 (Bad Request) status code.

utils.py

Generate_password_reset_otp Method

Functionality:

- The generate_password_reset_otp function creates a random 6-digit numeric code specifically for password resets.
- In a single line using list comprehension, it selects a random digit from the string '0123456789' six times using the secrets module's choice function.
- These six random digits are then converted to strings and joined together using ".join to form the final OTP string.

Send_password_reset_email Method

Functionality:

- The send_password_reset_email function facilitates sending an email to a user requesting a password reset.
- It generates a random OTP using generate_password_reset_otp and assigns it to the user's model field.

- It then constructs a password reset confirmation link using the base URL, a path indicating confirmation with OTP, user ID, and a token.
- The email body is prepared with string formatting, including the generated OTP, instructions mentioning it, and the clickable reset link.
- Django's send_mail function transmits the email with a "Password Reset Request" subject, the constructed body, the user's email as recipient, and the sender email from Django settings.
- Finally, the user model is saved to the database, and the generated OTP is used for verification during password reset confirmation.

urls.py

Path: 'employees/ '
View: UsersView
HTTP Methods Supported:

- **GET:** Retrieves details of all users.
- **POST:** Creates a new user.

Usage:

- **GET:** /employees/
- **POST:** /employees/

Path: 'employees/<int:id>/'
View: UserEditUpdateDeleteView
HTTP Methods Supported:

- **GET:** Retrieves details of a specific user by ID.
- **PUT:** Updates an existing user by ID.
- **DELETE:** Deletes an user by ID.

Usage:

- **GET:** /employees/<int:id>/
- **PUT:** /employees/<int:id>/
- **DELETE:** /employees/<int:id>/

Path: 'login//'
View: LoginView
HTTP Methods Supported:

- **POST:** Authenticates a user and generated tokens upon successful login.

Usage:

- **POST:** /login/

Path: 'employees/superuser/'
View: SuperuserCreate
HTTP Methods Supported:

- **POST:** Creates a new superuser account.

Usage:

- **POST:** /employees/superuser/

Path: 'employees/superuser/<int:id>'
View: SuperuserViewEditDelete
HTTP Methods Supported:

- **GET:** Retrieves details of a specific superuser by ID.
- **PUT:** Updates an existing superuser by ID.
- **DELETE:** Deletes a superuser by ID.

Usage:

- **GET:** /employees/superuser/<int:id>/
- **PUT:** /employees/superuser/<int:id>/
- **DELETE:** /employees/superuser/<int:id>/

Path: 'employees/profile'
View: UserProfileCreate
HTTP Methods Supported:

- **GET:** Retrieves a list of all user profiles.
- **POST:** Creates a new user profile.

Usage:

- **GET:** /employees/profile/
- **POST:** /employees/profile/

Path: 'employees/profile/<int:id>'
View: UserProfileCreate
HTTP Methods Supported:

- **GET:** Retrieves details of a specific user profile by its ID.
- **PUT:** Updates an existing user profile.
- **DELETE:** Deletes a user profile by its ID.

Usage:

- **GET:** /employees/profile/<int:id>/
- **PUT:** /employees/profile/<int:id>/
- **DELETE:** /employees/profile/<int:id>/

Path: 'auth/password_reset/'
View: PasswordResetRequestView
HTTP Methods Supported:

- **POST:** Initiates a password reset process by sending an email with a reset link.

Usage:

- **POST:** /auth/password_reset/

Path: 'auth/password/reset/confirm/otp/<int:pk>/<str:code>/'
View: PasswordResetConfirmOtpView
HTTP Methods Supported:

- **POST:** Confirms a password reset using a one-time password (OTP) received via email.

Usage:

- **POST:** /auth/password/reset/confirm/otp/<int:pk>/<str:code>/

Path: 'auth/password_reset/confirm/password/<int:pk>/<str:code>/'

View: PasswordResetConfirmView
HTTP Methods Supported:

- **POST:** Sets a new password after email confirmation.

Usage:

- **POST:** /auth/password_reset/confirm/password/<int:pk>/<str:code>/

Path: 'token/'
View: MyTokenObtainPairView
HTTP Methods Supported:

- **POST:** Obtains access and refresh tokens for user authentication.

Usage:

- **POST:** /token/

Path: 'token/refresh/'
View: TokenRefreshView
HTTP Methods Supported:

- **POST:** Refreshes an expired access token using a valid refresh token.

Usage:

- **POST:** /token/refresh/

3. Projects

OVERVIEW:
The Projects is an app which holds data of all the Projects in an Organization for the SmartAgile.

models.py

Class
1.Projects: The Projects model represents about each project associated in an Organization

Fields:

proj_id- It represents the unique identifier for each instance of the model.Setting "primary_key=True" ensures that this field serves as the primary key for the corresponding database table.
proj_name - This is the field where the name of the project is stored
proj_deadline - This is the field where deadline for the project is stored
proj_desc- This is the field where the detailed description about the project is stored
proj_membets- This is the field to mention the team members in an organization to work in the particular project.
organization - This field establishes a relationship with instances of the Organization model, to represent that the projects belong to the specific organization.
proj_prd- This field is used to get the prd document or the abstract of the project while creating a new project.

serializers.py

Class

ProjectSerializer

- **Meta Inner Class**
  The Meta inner class within the ProjectSerializer class is used to provide metadata to the serializer.
  model: This attribute specifies the Django model class that the serializer is based on the model Projects.
  fields: This attribute specifies which fields from the model should be included in the serialized output. In this case, '__all__' is used.

views.py

Class

ProjectView - The ProjectView class handles various HTTP methods (GET, POST, PUT, DELETE) for managing projects in an organization.

Methods:

get Method

Retrieving Projects:

- If proj_id is not provided (i.e., None), the method retrieves all projects in an organization from the database using "Projects.objects.all()".
- It then serializes the queryset of projects using ProjectSerializer with "many=True" to serialize multiple objects.
- Finally, it returns a response with the serialized data.

Retrieving a Specific Project:

- If proj_id is provided, it attempts to retrieve the specific project in an organization with that ID using "Projects.objects.get(proj_id=proj_id)".
- If the project exists, it serializes the project using ProjectSerializer.
- If the project does not exist, it returns a 404 Not Found response with an error message.

post Method

Data Validation:

- It attempts to deserialize the request data using "ProjectSerializer(data=request.data)" .
- If the data is valid, it proceeds to save the new project to the database using "new_project.save()" .
- If the data is invalid, it returns a 400 Bad Request response with the validation errors.

put Method

Retrieving and Updating Project:

- It retrieves the existing project with the provided proj_id using "Projects.objects.get(proj_id=proj_id)" .
- It attempts to update the project instance with the request data using "ProjectSerializer(project_update, data=request.data)" .
- If the data is valid, it saves the updated project using "new_update.save()".
- If the project does not exist, it returns a 404 Not Found response with an error message.

- If the data is invalid, it returns a 400 Bad Request response with the validation errors.

delete Method

Deleting Project:

- It retrieves the project with the provided proj_id using "Projects.objects.get(proj_id=proj_id)" .
- It attempts to delete the project using "project_delete.delete()" .
- If the project is successfully deleted, it returns a 200 OK response with a success message.
- If the project does not exist, it returns a 404 Not Found response with an error message.

urls.py

Path: 'projects/'
View: ProjectView
HTTP Methods Supported:

- **GET:** Retrieves details of all projects.
- **POST:** Creates a new project.

Usage:

- **GET:** /projects/
- **POST:** /projects/

Path: 'projects/<int:proj_id>/'
View: ProjectView
HTTP Methods Supported:

- **GET:** Retrieves details of a specific project by ID.
- **PUT:** Updates an existing project by ID.
- **DELETE:** Deletes a project by ID.

Usage:

- **GET:** /projects/<proj_id>/

- **PUT:** /projects/<proj_id>/
- **DELETE:** /projects/<proj_id>/

4. Tasks

OVERVIEW:
The Tasks is an app which holds data of all the tasks related to a specific project in an Organization for the SmartAgile.

models.py

Class
1.Tasks: The Tasks model represents each task associated with a project in an organization.

Fields:
task_id- It represents the unique identifier for each instance of the model.Setting "primary_key=True" ensures that this field serves as the primary key for the corresponding database table.
task_name- This is the field where the name of the task is stored
task_deadline - This is the field where deadline for the task is stored
task_desc- This is the field where the detailed description about the task is stored
assigned_to- This is the field that mentions the team member assigned to complete the task.
task_priority- This is the field where the priority of the task is stored
task_attachments- This field used to get any attachments associated with the tak.
project- This field establishes a relationship with instances of the Project model, to represent that the task belongs to the specific project in an organization.

serializers.py

Class
TaskSerializer
- **Meta Inner Class**

The Meta inner class within the TaskSerializer class is used to provide metadata to the serializer.
model: This attribute specifies the Django model class that the serializer is based on the model Tasks.
fields: This attribute specifies which fields from the model should be included in the serialized output. In this case, '__all__' is used.

views.py

Class

TaskView- The TaskView class handles various HTTP methods (GET, POST, PUT, DELETE) for managing tasks in an organization.

Methods:

get Method

Retrieving Tasks:

- If task_id is not provided (i.e., None), the method retrieves all tasks in a project from the database using "Tasks.objects.all()".
- It then serializes the queryset of tasks using TaskSerializer with "many=True" to serialize multiple objects.
- Finally, it returns a response with the serialized data.

Retrieving a Specific Task:

- If task_id is provided, it attempts to retrieve the specific task from a project with that ID using "Task.objects.get(task_id=task_id)".
- If the task exists, it serializes the project using TaskSerializer.
- If the task does not exist, it returns a 404 Not Found response with an error message.

post Method

Data Validation:

- It attempts to deserialize the request data using "TaskSerializer(data=request.data)" .

- If the data is valid, it proceeds to save the new task to the database using “new_task.save()” .
- If the data is invalid, it returns a 400 Bad Request response with the validation errors.

put Method

Retrieving and Updating Task:

- It retrieves the existing task in a project with the provided task_id using “Tasks.objects.get(task_id=task_id)” .
- It attempts to update the task instance with the request data using “TaskSerializer(task_update, data=request.data)” .
- If the data is valid, it saves the updated task using “new_update.save()”.
- If the task does not exist, it returns a 404 Not Found response with an error message.
- If the data is invalid, it returns a 400 Bad Request response with the validation errors.

delete Method

Deleting Task:

- It retrieves the task with the provided task_id using “Tasks.objects.get(task_id=task_id)” .
- It attempts to delete the task using “task_delete.delete()” .
- If the task is successfully deleted, it returns a 200 OK response with a success message.
- If the task does not exist, it returns a 404 Not Found response with an error message.

urls.py

Path: 'tasks/'
View: TaskView
HTTP Methods Supported:

- **GET:** Retrieves details of all tasks from a project.
- **POST:** Creates a new task.

Usage:

- **GET:** /tasks/
- **POST:** /tasks/

Path: 'tasks/<int:task_id>/'
View: TaskView
HTTP Methods Supported:

- **GET:** Retrieves details of a specific task by ID.
- **PUT:** Updates an existing task by ID.
- **DELETE:** Deletes a task by ID.

Usage:

- **GET:** /tasks/<task_id>/
- **PUT:** /tasks/<task_id>/
- **DELETE:** /tasks/<task_id>/

MOBILE

Related Resources and Links
React Native Documentation https://reactnative.dev/docs/getting-started
Redux Documentation https://redux.js.org/introduction/getting-started

INSTALLED PACKAGES

"@react-native-async-storage/async-storage": "^1.23.1",
"@react-navigation/bottom-tabs": "^6.5.20",
"@react-navigation/native": "^6.1.17",
"@react-navigation/native-stack": "^6.9.26",
"@react-navigation/stack": "^6.3.29",
"lottie-react-native": "^6.7.0",
"react": "^18.2.0",
"react-native": "^0.73.6",
"react-native-config": "^1.5.1",
"react-native-linear-gradient": "^2.8.3",
"react-native-safe-area-context": "^4.9.0",
"react-native-screens": "^3.30.1",
"react-native-svg": "^15.1.0",
"react-native-vector-icons": "^10.0.3",
"react-redux": "^9.1.1",
"redux": "^5.0.1"

CODEBASE STRUCTURE

```
android/
  |-- app/
    |-- src/
      |-- main/
        |-- java/com/smartagile_mobile/
            |-- MainActivity.kt/
            |-- MainApplication.kt/
            |-- MyModule.kt/
            |-- ToastService.kt/

assets/
  |-- fonts/
    |-- Poppins-Regular.ttf/
  |-- hello-bot-theme.json/
  |-- hello-bot.json/
  |-- robot.json/
  |-- t-logo.png/
  |-- t-logo.svg/
  |-- welcoming.json/

Navigation/
  |-- Navigation.tsx/
  |-- NavigationTypes.tsx/
  |-- SupervisorTabBar.tsx/

redux/
  |-- actions.ts
  |-- reducers.ts
  |-- store.ts
screens/
  |-- Supervisor/
      |-- EmployeeView.tsx/
      |-- SupervisorDashboard.tsx/
      |-- SupervisorProfile.tsx/
      |-- SupervisorProjectScreen.tsx/
```

```
      |-- SupervisorSettings.tsx/
  |-- EmployeeDashboard.tsx
  |-- NewOrganization.tsx
  |-- OrganizationLogin.tsx
  |-- PersonalLogin.tsx
  |-- Welcome.tsx

styles/
  |-- Colors.ts/
  |-- GlobalStyles.ts/
.eslintric.js/
.gitignore/
.prettierrc.js/
.watchmanconfig/
app.json/
App.tsx/
babel.config.js/
env.js/
index.js/
package-lock.json/
package.json/
tsconfig.json/
```

1) ASSETS

They include various file types that the application uses to display content dynamically and statically.

Proper management and documentation of these assets are essential for efficient collaboration among developers and for maintaining the quality and performance of the app. This folder consists of font folders , images (png , jpg etc,.) and lottie json files

2) ANDROID

The android folder in a React Native project contains all the native Android code and configuration files necessary to build and run your application on Android devices. This folder is a critical part of the project because it allows for deep customization and integration of native functionalities that are not covered by the React Native framework alone.

i) ToastService.kt

android\app\src\main\java\com\smartagile_mobile\ToastService.kt

ToastService is a Java class that extends ReactContextBaseJavaModule, making it a module that can be accessed from JavaScript code in a React Native application. This class provides a method to show Android toast notifications from the React Native JavaScript environment.

Components Breakdown

1. Package Declaration:

   package com.smartagile_mobile;: Defines the package name of the Java class, which is typically used to organize the code files within larger projects.

2. Imports:

   android.view.Gravity: Used to specify the placement of the toast message on the screen.

   android.widget.Toast: Required to create the Android toast message.

   com.facebook.react.bridge.ReactApplicationContext: Provides access to the React application context, allowing the module to interact with various aspects of the React Native environment.

com.facebook.react.bridge.ReactContextBaseJavaModule: The base class for all native modules in React Native.

com.facebook.react.bridge.ReactMethod: Annotation used to expose a Java method as a JavaScript method to the React Native environment.

3. Class Definition:

   class ToastService(private val reactContext: ReactApplicationContext): Defines the ToastService class that takes a ReactApplicationContext as a constructor parameter. The private val declaration makes reactContext a private immutable variable, accessible only within this class.

4. Module Name Override:

   override fun getName(): String: This method is overridden from ReactContextBaseJavaModule and specifies the name of the module as it will be exposed to JavaScript. Here, it returns "ToastModule", which is how you will refer to it from JavaScript.

5. Toast Display Method:

   @ReactMethod: This annotation makes the following method available in JavaScript.

   fun showToast(message: String): Defines a method that takes a string message as an argument. This method creates and displays a toast with the specified message.

6. Inside the method:
   Toast.makeText(reactContext, message, Toast.LENGTH_LONG): Creates a toast message that will display for a long duration.
   toast.setGravity(Gravity.CENTER, 0, 0): Sets the position of the toast to the center of the screen.
   toast.show(): Displays the toast on the device's screen.

How to use in javascript code?

```
import { NativeModules } from 'react-native';
const { ToastModule } = NativeModules;

ToastModule.showToast('Hello from Android');
```

ii) MyModule.kt

android\app\src\main\java\com\smartagile_mobile\MyModule.kt

MyModule is a class that implements the ReactPackage interface. React Native uses packages to include native modules or view managers that expose functionality to JavaScript. In this case, MyModule facilitates the inclusion of a custom native module (i.e, ToastService) into the React Native application.

Class Definition:

class MyModule() : ReactPackage: Defines MyModule as an implementation of the ReactPackage interface. The class is responsible for registering native modules and view managers with the React Native app.

Methods:

createNativeModules:

- Takes ReactApplicationContext as a parameter, providing the necessary context for module creation.
- Initializes an ArrayList of NativeModule.
- Adds an instance of ToastService, initialized with the provided ReactApplicationContext, to this list.
- Returns the list of native modules, making ToastService available to the React Native JavaScript environment.

createViewManagers:

Returns an empty list of ViewManager, indicating that this package does not provide any custom views. If you were to extend this application with custom native views, they would be initialized and returned in this method.

Integrating custom modules:

```
@Override
protected List<ReactPackage> getPackages() {
   return Arrays.<ReactPackage>asList(
      new MainReactPackage(),
      new MyModule() // Registering MyModule with the app
   );
}
```

iii) MainApplication.kt

MainApplication is a pivotal class in a React Native application for Android. It initializes and configures the React Native environment, manages native modules, packages, and includes settings for developer tools like Flipper.

iii) MainActivity.kt

MainActivity serves as the main Android activity for a React Native application named "SmartAgile_Mobile". This class extends ReactActivity, which is specifically designed to host a React application within an Android activity. The key functionalities provided include specifying the main React component and configuring the React Native environment with the option to use the new architecture.

3) NAVIGATION

This folder consists all the setup for navigation among screens of the application with conditional rendering for filtering the users according to their positions in the organization

i) Navigation.tsx

It uses the createNativeStackNavigator to manage stack navigation.

Navigation Tags

1. NavigationContainer - serves as the root component of the navigation system. It maintains the navigation state and provides essential context for navigation actions throughout the app.

2. Stack.Navigator - This component sets up a stack-based navigation model, allowing users to transition between screens where each new screen is placed on top of a stack.

3. Conditional Rendering in NavigationContainer -
   Authenticated Users: If the user state is present (indicating authentication), the Stack.Navigator starts with SupervisorTabBar and includes routes for SupervisorProfile.

   Unauthenticated Users: If no user state is found, navigation begins with the Welcome screen, followed by OrganizationLogin, NewOrganization, and PersonalLogin.

ii) NavigationTypes.tsx

TypeDefinitions

RootStackParamList

Defines the navigation state parameters for all routes within the application. This object specifies all valid route names and the types of parameters they accept. In this setup, all routes are defined with undefined as their parameter, indicating they do not expect any parameters.

NavigationType<T extends keyof RootStackParamList>

This generic type is used to provide type-safe navigation props throughout the application. It utilizes TypeScript's utility type NativeStackNavigationProp from @react-navigation/native-stack to enforce the correct navigation parameters and methods based on the specified route key.

RouteType<T extends keyof RootStackParamList>

A generic type defining route props for a given screen, leveraging RouteProp from @react-navigation/native. It ensures that each screen receives the appropriate route parameters as defined in RootStackParamList, providing type safety for route parameter access within screen components.

iii) SupervisorTabBar.tsx

This component configures the bottom tab navigation for supervisors using createBottomTabNavigator from @react-navigation/bottom-tabs.

Tab Screens Defined:

SupervisorDashboard: Main dashboard view with a custom dashboard icon.
SupervisorProjectScreen: Access to project-related information with a project icon.
EmployeeView: Manages employee information with a users icon.
SupervisorSettings: Settings screen for supervisors, with settings icon and custom header title.

4) REDUX

i) actions.ts

This file defines the actions and action creators related to user data management in the Redux store.

SET_USER Action Constant

Purpose: Defines a constant for the action type used in Redux actions and reducers. This constant standardizes the action type name, reducing errors and improving code readability.

Type: string

Value: 'SET_USER'

Usage: Used to identify the action for setting user data in the Redux store. It is referenced in both action creators and reducers to ensure that the correct action is processed.

UserData Interface

Purpose: Specifies the structure of user data that is handled within the application. This TypeScript interface enforces type safety by defining the expected properties and their types for user data objects.

Properties:
username: string - The user's username.
email: string - The user's email address.
organization: number - An identifier (likely numeric) for the organization to which the user belongs.
is_owner: boolean - A flag indicating whether the user has owner privileges.
is_staff: boolean - Indicates whether the user is a staff member.

setUser Action Creator

Purpose: Creates and returns a Redux action object for setting user data in the global state. This function is crucial for updating the state when user information is fetched, updated, or needs to be initialized at login.

Parameters:
userData (UserData): The user data to be set in the Redux store.

Returns: An object representing a Redux action.

type: A string corresponding to SET_USER, identifying the action's purpose.
payload: The user data (UserData), which will be used by the reducer to update the state.

Functionality
The setUser action creator simplifies dispatching actions to update the user data in the Redux store. When called, it constructs an action object with a predefined type (SET_USER) and the passed userData as its payload. This object is then dispatched to the Redux store, where a reducer processes it to update the state.

ii) reducers.ts

Describes the shape of user data with properties like username, email, organization, is_owner, and is_staff.

State Interface

Purpose: Describes the structure of the state concerning the user within the application.

Properties:
user: Can be null or an object containing user data. The object includes:
username: string - The user's username.
email: string - The user's email address.
organization: number - Represents the organization ID to which the user belongs.
is_owner: boolean - Indicates if the user has owner-level permissions.
is_staff: boolean - Indicates if the user is a staff member.

initialState

Purpose: Defines the initial state used by the reducer upon initialization or when the state is reset.

Structure:
user: Initially set to null, indicating no user is logged in or user data is not yet loaded.

reducers Function

Functionality: This function is the reducer that manages changes to the state based on actions received. It ensures that updates to the state are handled immutably.

Parameters:

- state: State - The current state of the application. Defaults to initialState if not provided.
- action: Action - The action dispatched to the reducer.
- Return Value: State - The new state after the action is processed.

Action Handling:

SET_USER:
When this action is received, the reducer updates the user property of the state with the payload of the action, which should contain new or updated user data.
Uses the spread operator (...state) to create a new state object, ensuring that updates are immutable.

Default Case:
If the action type does not match any known cases, the current state is returned unchanged.

iii) store.ts

initializeUser Function

Purpose: This asynchronous function retrieves the user data stored in AsyncStorage and initializes the Redux store's state with this data.

Process:

- AsyncStorage Retrieval: Attempts to fetch the stored user data with the key 'user'.
- State Initialization: If user data exists, it dispatches the SET_USER action to update the Redux state with the retrieved data.

Error Handling
Catches and logs errors related to data retrieval from AsyncStorage, such as permissions issues or data corruption.

Store Creation
Store Variable: store
Type: Store<State>
Initialization: The Redux store is created using the createStore function, which takes the root reducer (reducers) and an initial state.
Initial State: The initial state is potentially set by the initializeUser function. This function does not directly set the initial state during store creation but dispatches an action to populate the state once the store is created and the data is retrieved.

5) SCREENS

The screens folder contains all the primary components that serve as the main views users interact with in the application. Each screen component corresponds to a specific function or feature within the app. It is the individual pages that shown to the user when they navigates from one screen to other.

i) Supervisor/EmployeeView.tsx

ii) Supervisor/SupervisorDashboard.tsx

The SupervisorDashboard serves as the primary interface for supervisors to view critical data and actions related to the organization's operations. It combines data visualization, event notifications, and quick access to management tools, enabling supervisors to effectively oversee and respond to organizational needs.

Props
navigation: Allows navigation to other screens within the app, facilitating deep linking to detailed views or management screens.

Component Functionality

- Dashboard Widgets:

  Progress Overview: Cards or tiles that display key performance indicators such as project completion rates, upcoming deadlines, or budget usage.

  Event Feed: A list or feed of important events or notifications that require supervisor attention, allowing for quick overview and action.

  Task Summary: Displays a summary of tasks assigned to various teams or individuals, including status updates to monitor progress in real-time.

- Interaction:

  Drill-Down Capability: Each widget or card on the dashboard provides interaction options like tapping to view detailed reports or swiping to see additional data.

  Quick Actions: Buttons or links to perform common tasks like scheduling meetings, approving requests, or updating task statuses directly from the dashboard.

iii) Supervisor/SupervisorProfile.tsx

The SupervisorProfile screen is a dedicated area for supervisors to manage their personal and professional details within the application. It facilitates easy access to profile editing, password changes, and viewing of personal performance metrics.

Props

navigation: This prop allows for navigation to other screens for deeper interactions, such as updating credentials or accessing detailed settings.

Component Functionality

<u>Profile Information Display:</u>

Displays the supervisor's name, photo, and basic contact information.
Provides a summary of their role and responsibilities within the organization.

Editable Fields:
Allows supervisors to update their personal information such as phone number, email address, and password.
Offers interactive elements like text inputs for editing and buttons for submitting changes.

iv) Supervisor/SupervisorProjectScreen.tsx

The SupervisorProjectScreen is designed to enable supervisors to monitor ongoing projects, manage tasks, and delegate responsibilities effectively. It includes features for viewing project details, updating task statuses, and assigning tasks to team members.

Props

navigation: Facilitates navigation to other screens, such as detailed task views or user profiles, enhancing the app's interactivity and functional depth.

Component Functionality

Project Details Display:
Showcases key project metrics such as deadlines, current status, and overall progress.
Offers a summarized view of the project's scope and objectives.

Task Management:
Displays a list of tasks associated with the project using a FlatList or similar component.
Each task can be tapped to view detailed information or to edit the task status and assignees.

Task Assignment:
Incorporates a modal or another UI element to assign tasks to team members.
Utilizes a UserSelector component to choose team members for specific tasks, allowing for dynamic assignment based on availability and skill set.

Interactive Updates:
Provides interactive elements like buttons or swipe gestures to update task statuses (e.g., from "In Progress" to "Completed").
Facilitates real-time updates to ensure all team members have the latest project information.

v) Supervisor/SupervisorSettings.tsx

The SupervisorSettings screen offers a collection of settings options for supervisors, including profile management, organizational information, help sections, and logout functionality. It uses a straightforward list-based navigation approach, enabling users to tap on options to navigate or perform actions like logging out.

Props

navigation: This prop allows navigation between different screens within the app using the type NavigationType<'SupervisorSettings'>, ensuring navigation actions are type-checked.

Component Functionality

Logout Function:

Purpose: To log out the user by removing user data from AsyncStorage and reloading the app to reset state.

Implementation: The function logout is async and performs AsyncStorage.removeItem('user') to clear user data, followed by DevSettings.reload() to restart the application.

Settings Options

Profile: Navigates to the SupervisorProfile screen.
About Organization: Placeholder for navigation to detailed organization info.
Help: Placeholder for help-related information.
FAQ: Placeholder for frequently asked questions.
Logout: Executes the logout function to clear user data and reload the app.

vi) NewOrganization.tsx

The NewOrganization screen provides an interface for creating a new organization along with its owner's account. The screen includes a series of input fields to capture relevant data such as organization name, email, and website, as well as the owner's name, email, and password.

Props
navigation: This prop is used to navigate between screens. It is typed with NavigationType<'NewOrganization'> ensuring that any navigation call adheres to the defined type for safety.

State Management

organization: A state object holding all input values related to the new organization and its owner. Managed via useState.

Functions
handleOrganizationDetails:

Purpose: To update the organization state with input values as they are typed.

Parameters: name (string) indicating the field to update, value (string) the new value for that field.
Implementation: Uses a functional state update to merge changes with existing state.

createOrganization:

Purpose: Handles the submission of the new organization form data to respective API endpoints to create a superuser and the organization itself.
Process:
Submits the superuser data.
On successful creation, submits the organization data linking it to the newly created superuser.
Optionally creates a user profile and gives feedback via toast messages.
API Interaction
Uses fetch to send POST requests to backend services defined in baseUrl. Manages both user and organization data submission through sequential API calls ensuring data integrity and proper linkage between user and organization.

vii) OrganizationLogin.tsx

The OrganizationLogin screen provides a login interface for organizational users. It features a form for email and password input, and options for navigating to account creation or password recovery.

Props
navigation: A NavigationType<'OrganizationLogin'> object enabling navigation to other screens.

Navigation

- Provides buttons for navigation to the NewOrganization screen for new account creation
- A placeholder for password recovery functionality.

Functionality

State Management:

email and password: Local state variables for storing user input.

Login Function:

Validates input fields and makes an API request to log in. On success, stores user data locally and updates Redux state, navigating the user to the dashboard or relevant post-login screen.

Functional Workflow

User Authentication:

The user submits their credentials (email and password), which are sent to the server via an HTTP POST request.

If the credentials are verified, the server responds with user data and a success message.

State Management and Data Persistence:

i) Redux Store Update:

The retrieved user data is dispatched to the Redux store using the setUser action. This action updates the Redux state, making the user data available globally across the application.

Syntax: dispatch(setUser(userData))

ii) AsyncStorage:

For data persistence across app launches, the user data is also stored in AsyncStorage. This allows the app to rehydrate the user state from local storage when the app is closed and reopened.

Syntax: AsyncStorage.setItem('user', JSON.stringify(userData))

iii) Feedback and Navigation:

A toast notification is displayed to inform the user of successful login using the native module ToastModule.

The user is then redirected to the appropriate screen based on their role or the intended post-login workflow.

viii) Welcome.tsx

The Welcome screen is the entry point for users accessing the Smart Agile app. It offers a visually engaging introduction to the app's capabilities and provides users with options to navigate to either personal or organizational login screens.

Props
navigation: A NavigationType object enabling navigation to other screens (OrganizationLogin and PersonalLogin).

Navigation - (with onPress event handler)
Organization Button: Navigates to OrganizationLogin when pressed.
Personal Button: Navigates to PersonalLogin when pressed.

6) STYLES

The styles folder organizes and centralizes CSS-like styles used throughout a React Native application. It helps maintain a cohesive design language and facilitates easy updates to the app's appearance. The folder typically contains:

- GlobalStyle.ts: Defines universal style properties that can be applied across multiple components.
- Colors.ts: Centralizes the color palette of the application to ensure consistency in the usage of colors.

i)GlobalStyle.ts

container
Purpose: Serves as the base container style for most screens or components, focusing on center alignment and flexibility.

Properties:
display: Set to 'flex' to enable flexible box layout.
flexGrow: Allows the container to expand to fill any available space.
alignItems: Centers children horizontally within the container.
justifyContent: Centers children vertically within the container.

authContainer

Purpose: Specifically designed for authentication-related screens where a centered alignment and specific background are needed.

Properties:
Inherits all properties from container.
backgroundColor: Set to Colors.background from the Colors module, providing a consistent background color for authentication screens.

scrollContainer

Purpose: Used for containers that need to support scrolling, typically used in forms or lists that might exceed the screen height.

Properties:
Includes all flex properties of the standard container.
width: Sets the width to 90% of the device screen to provide adequate padding on the sides.

scrollAuthContainer

Purpose: A variation of scrollContainer for authentication-related screens that may require scrolling.

Properties:
Inherits properties from scrollContainer but allows for additional customization if necessary in the future.

textStyle

Purpose: Defines a global text style to ensure consistency in typography across the app.

Properties:
color: Black, ensuring readability across different backgrounds.
fontFamily: 'Poppins', a modern sans-serif typeface that enhances the visual appeal.
fontSize: Set to 20, which is versatile for various text elements.

rowBetween

Purpose: Frequently used layout style for arranging items in a row with space distributed between them.

Properties:
display: Flex layout to utilize the row structure.
flexDirection: 'row' to organize children horizontally.
width: '100%' to span the full width of the parent container.
alignItems: Aligns children vertically in the center.
justifyContent: Distributes space between children evenly.

ii) Colors.ts

The `Colors` module in a React Native application is a centralized collection of color definitions used throughout the app to ensure consistency and ease of maintenance in the visual design. Each color is assigned a meaningful name (like `primary`, `secondary`, `background`, `White`, `text`) and a corresponding hexadecimal value, defining the application's color palette in one place. This approach facilitates straightforward updates to the app's color scheme and enhances uniformity across different components and screens.

7) APP.TSX

App is the main component of the application, serving as the entry point for integrating the navigation system and the Redux store, which are essential for managing the app's navigation and state.

App Component:

This is a functional component that renders the application's main structure.
It uses the Provider component to make the Redux store available to all nested components. This is crucial for managing global state across the app.
Inside the Provider, the Navigation component is rendered. This component likely encapsulates all the navigation logic, such as routing and screen transitions, using a navigation library (commonly React Navigation).

Purpose

The setup is designed to facilitate state management across the application with Redux, allowing components to connect to the Redux store and dispatch actions or subscribe to updates. The Navigation component manages where users can go within the app and how they get there, enhancing the app's interactivity and functional layout.

8) INDEX.JS

The primary purpose of this code is to set up the entry point for the React Native application. It tells the React Native system which component should be the root to bootstrap the whole application. The appName used here must match the application name specified in native configurations (like AndroidManifest.xml for Android and Info.plist for iOS).

- AppRegistry from react-native: This module is responsible for registering the root component of the app that will be run from the JavaScript side.

- App from ./App: This imports the App component, which is the root component of your application. It likely includes setup for navigation and state management (as seen in the previous snippet).

- {name as appName} from ./app.json: This imports the name property from the app.json configuration file and renames it to appName. This is typically used to define the name under which the app is registered, aligning with native application identifiers.

AI DEVELOPMENT:

PURPOSE:

The AI within SmartAgile enables real-time monitoring and analysis of employee activities, distinguishing between work-related and non-work-related tasks to enhance productivity. It processes collected data to identify patterns and trends, aiding in decision-making processes. Additionally, it automates task management within Agile workflows, facilitating smoother project execution. Finally, it generates detailed reports and metrics, providing insights for strategic planning and operational improvement.

INSTALLED PACKAGES:

jupyter_client	8.6.0
jupyter_core	5.7.1
keras	2.15.0
matplotlib	3.8.2
matplotlib-inline	0.1.6
numpy	1.23.5
opencv-python	4.9.0.80
pandas	2.2.0
pillow	10.2.0
psutil	5.9.8
PyAutoGUI	0.9.54
PyGetWindow	0.0.9
scikit-learn	1.4.0
scipy	1.12.0
seaborn	0.13.2
tensorflow	2.15.0
tf-keras	2.15.0
torch	2.2.1
torchvision	0.17.1

word2number 1.1

CODEBASE STRUCTURE:

DEEP LEARNING_MODEL - 1 :

Input Data Processing : The dataset is loaded from a CSV file and preprocessed. The 'desc' column is concatenated with the 'title' column and converted to lowercase. The text is tokenized into words, and a vocabulary of the most common words is created.

Data Encoding : Each word in the text is mapped to an integer index based on the vocabulary created earlier. Text sequences are then padded or truncated to a fixed length to ensure uniformity.
Label Encoding : The target labels ('category') are encoded using one-hot encoding to prepare them for classification.

Model Architecture : The CNN model is defined sequentially using Keras. It consists of an embedding layer followed by multiple convolutional layers with dropout regularization to prevent overfitting. Max pooling layers are added to reduce dimensionality. The flattened output is fed into densely connected layers, followed by a softmax layer for multi-class classification.

Model Compilation : The model is compiled with categorical cross-entropy loss function and Adam optimizer. Accuracy is chosen as the evaluation metric.

Model Training : The model is trained on the training data with a specified learning rate. The training process iterates through epochs, adjusting the model's weights to minimize the loss function.

SVM_MODEL - 2 :

Data Preparation : The dataset is loaded from a CSV file containing two columns: 'URL' and 'Category'.The dataset is split into individual categories, and a

subset of data is selected for each category to create a balanced training dataset. The remaining data is used for testing.

Data Visualization : The distribution of categories in both the original and the training/test datasets is visualized using bar plots and countplots.

Feature Engineering : The 'URL' column is used as the input feature (X), and the 'Category' column is the target variable (y).Text preprocessing techniques such as tokenization, stop-word removal, and stemming are applied using the NLTK library.

Model Definition : The model is defined using a pipeline consisting of a stemmed count vectorizer, TF-IDF transformer, and a linear Support Vector Machine (SVM) classifier (perceptron).

Model Training : The pipeline is trained on the training dataset using the fit() method.

Model Evaluation : The trained model is evaluated on the test dataset using classification metrics such as precision, recall, F1-score, and accuracy.
A confusion matrix is visualized to analyze the performance of the model across different categories.

Model Prediction : The trained model is used to make predictions on new URLs to assign them to appropriate categories.

BACKGROUND RUNNING APPS:

PSUTIL : The psutil library is a powerful tool for system monitoring and management in Python. It provides a high-level interface for retrieving information about running processes, system utilization, and various system resources. By abstracting the complexities of interacting with the operating system, psutil simplifies tasks such as process management, system monitoring, and performance analysis.

Function Definition: get_all_processes_cpu_times()

The get_all_processes_cpu_times() function serves as the entry point for retrieving CPU times for all running processes. This function encapsulates the logic necessary to interact with the psutil library and iterate over the list of running processes. It abstracts away the details of process enumeration and CPU time retrieval, providing a clean and reusable interface for accessing this information.

Retrieving CPU Times for Processes

Within the get_all_processes_cpu_times() function, a loop iterates over all running processes using psutil.process_iter(). For each process, essential information such as the process name, process ID (PID), and CPU times are retrieved. The CPU times typically include metrics such as user CPU time, system CPU time, and others, providing insights into how much CPU resources each process consumes.

Error Handling

To ensure robustness, the code includes error handling using a try-except block. If any exceptions occur during the process retrieval, such as permission errors or system limitations, the code gracefully handles them and prints an error message. This proactive approach enhances the reliability of the code, making it more resilient to potential issues that may arise during execution.

Leveraging the datetime Module for Time Tracking:

Python's datetime module provides a versatile toolkit for working with dates and times. By utilizing the capabilities of datetime, developers can accurately measure the time elapsed between different events, such as the start and end of application usage sessions. This precise time tracking enables the calculation of the duration spent on each background application.

Monitoring Background Apps:

The process of monitoring background applications involves continuously iterating over the list of running processes using psutil.process_iter(). During each iteration, relevant information about each process, such as its name and execution status, is extracted. By applying criteria to identify background applications, such as excluding system utilities or known foreground processes, the script can focus on monitoring user-centric applications.

Tracking Time Spent:

As the script iterates over background applications, it simultaneously calculates and updates the time spent since the monitoring started. By capturing the current time at regular intervals and calculating the difference from the start time, the script accurately tracks the cumulative time spent on each background application. This time tracking mechanism provides real-time insights into application usage patterns and resource utilization.

Customization and Adaptation:

The monitoring script can be customized and adapted to suit specific requirements and preferences. Developers can adjust the criteria for identifying background applications, fine-tune the frequency of time updates, and extend functionality to include additional metrics or logging capabilities. This flexibility enables the script to cater to diverse use cases and accommodate varying monitoring needs.