**December 2024 Task Tracking App**

A white brain with arrows and dots

Description automatically generated

TRASKIO

**Task Tracking App Documentation**

# **Section 1: System Overview:**

The Task Tracking App is designed to help users efficiently manage and organize their tasks. It provides features for task creating and tracking progress. The app aims to enhance productivity by offering intuitive user interfaces and features such as progress visualization and collaboration tools.

# **Section 2: Functional Requirements:**

## **User Authentication & Account Management:**

* **Sign Up**: The user must be able to sign up using an **email address**, **password**, and **username**.
* **Login**: The user must be able to log in using either their **username** or **email** and **password**.

## **Homepage:**

* **Active Homepage**: The app should display a homepage that shows the **current time** and updates in real-time.
* The homepage should be the landing page after logging in.

## **Task Management:**

* **Add Task Card**: The user must be able to create a new **To-Do card** with a **title**.
* **Check Task Card**: The user should be able to check task card as done.
* **Delete Task Card**: The user should be able to delete any created To-Do card.
* **Add Mini Tasks**: The user should be able to add **mini tasks** within each To-Do card.

Each mini task should have the ability to be **checked** off as completed or **deleted**.

* **Progress Bar**: A progress bar should be displayed for each To-Do card that updates based on the completion of mini tasks.

## **Search and Filter:**

* **Search Bar**: The user should be able to search for **To-Do cards** or **mini tasks** by title or description.
* **Filtering**: The user can filter tasks based on their status:
* **Completed** tasks.
* **In Progress** tasks.
* **All tasks** (default view).

## **Calendar and Schedule:**

* **Task Scheduling**: The user can add **time** to any To-Do card or mini task.
* **Automatic Schedule Update**: When a time is assigned to a task, it is automatically added to the **calendar schedule**.
* **View Schedule**: The user should be able to view their tasks on the calendar page, where they can see the schedule in a calendar view.

# **Section 3: Non-Functional Requirements**

**3.1 Performance:**

* **Fast Loading**: The app should load within **2 seconds** when navigating between pages.
* **Instant Updates**: Any changes to tasks (like checking mini tasks or deleting cards) should be reflected instantly, without noticeable delay.

**3.2 Scalability:**

* The app should be able to handle up to **1000 users** simultaneously without a significant drop in performance.
* As the number of tasks increases, the app should still load and update efficiently.

**3.3 Usability:**

* **Simple and Intuitive UI**: The interface should be easy to navigate, with a clean layout and user-friendly features.
* **Accessibility**: The app should support basic accessibility features like **keyboard navigation** and **screen reader support**.

**3.4 Security:**

* **Password Encryption**: User passwords should be encrypted securely using industry-standard algorithms (e.g., bcrypt).
* **Session Management**: After successful login, a secure session should be created to maintain user state across pages.
* **Data Protection**: Sensitive user data, such as email and task details, should be protected with proper security measures like **SSL/TLS** encryption during transmission.

**3.5 Reliability:**

* The app should ensure **99.9% uptime** with minimal downtime for maintenance.
* Backup systems should be in place to ensure that task data is not lost in case of failure.

**3.6 Consistency:**

* The visual design should be consistent across all pages (e.g., consistent header and footer, colors, fonts).

**3.7 Maintainability:**

* Code should be modular, well-documented, and easy to maintain, with clear separation of concerns for components (e.g., components for task cards, progress bar, calendar).

# **Section 4: Tools and Technology**

## **4.1 Frontend:**

## **Framework****: React.js – React.js will be used to build the frontend, providing a responsive, interactive, and modern user interface.**

## **Language: JavaScript – JavaScript will be used for frontend logic and interaction with the backend APIs.**

## **4.2 Backend:**

### **Framework: Django** **– Django will power the backend, providing a robust framework for developing APIs and managing system logic.**

### **Language: Python** **will be used in conjunction with Django for server-side development.**

### **Database: PostgreSQL – PostgreSQL will be the relational database used for managing tasks, notes, and user information.**

### **Database Hosting: Neon Serverless PostgreSQL**

## **4.3 Version Control and Collaboration:**

### **Tool: Git**

**Git will be used for version control, tracking changes to the codebase and enabling collaboration**

## **4.4 Development Tools:**

### **Tool: Visual Studio Code (VS Code)**

**Visual Studio Code will be used as the primary code editor, providing a development environment with extensions for React.js, Python, Django, and Git integration.**

# **Section 5: Risk Analysis**

## **5.1 Risk Analysis Table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Risk | Description | Likelihood | Impact | Mitigation Strategy |
| Service Interruptions | The application may experience downtime due to issues with the server or database infrastructure. | Medium | Critical | Utilize cloud-based database solutions |
| Overloading System Resources | System performance degradation when handling a large number of concurrent users | Medium | High | Optimize codebase for better resource management and load testing |
| Integration Failures | Issues with connecting services like AI features could cause disruptions. | Low | Medium | Ensure testing of all integrations for third-party failures. |
| User Interface Issues | Poor user interface design or unexpected user behaviors might reduce the app's usability. | High | Moderate | Focus on conducting user testing, gathering feedback, and refining the design to ensure the interface is simple and intuitive. |
| User Experience Problems | Complex or unintuitive UI reducing app usability. | High | Moderate | Gather user feedback. |
| Version Control Conflicts | Development team members might face conflicts when merging changes | Medium | Low | Establish clear Git workflow guidelines, conduct frequent code reviews, |

# **Section 6: Test Cases**

## **6.1 Test Case Table:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Test Case ID** | **Test Description** | **Input Data** | **Expected Result** | **Result** | | **Status** |
| **T-001** | **Login with valid credentials** | **Valid email and password** | **User successfully logs in** | **Successfully logged in** | **Pass** | |
| **T-002** | **Login with invalid credentials** | **Invalid email and password** | **Display “Invalid credentials”** | **Displayed “Invalid Credentials”** | **Pass** | |
| **T-003** | **Create a new task** | **Create Task** | **Task successfully created** | **Task was successfully created** | **Pass** | |
| **T-004** | **Testing Task Completion** | **Completed Task** | **Task state is updated to “Complete”** | **Task state was successfully updated to complete** | **Pass** | |

# 

# **Section 7: Screenshots of Frontend**

**7.1 Homepage:**

A screenshot of a black and white clock

Description automatically generated

**7.2 Login/Sign-up Screen:**

**A screen shot of a login form

Description automatically generated**

**A screenshot of a login screen

Description automatically generated**

**7.3 ToDos Page:**

**A screenshot of a computer program

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

# **Section 8: Diagrams**

## **8.1 Class Diagram:**

A diagram of a computer

Description automatically generated

Figure 1 Class Diagram

## **8.2 Sequence Diagram:**

A screenshot of a graph

Description automatically generated

Figure 2 Sequence Diagram

## **8.3 Use Case Diagram:**

A diagram of a workflow

Description automatically generated

Figure 3 Use Case Diagram

## **8.4 Activity Diagram:**

A screenshot of a computer

Description automatically generated

Figure 4 Activity Diagram

## **8.5 State Diagram:**

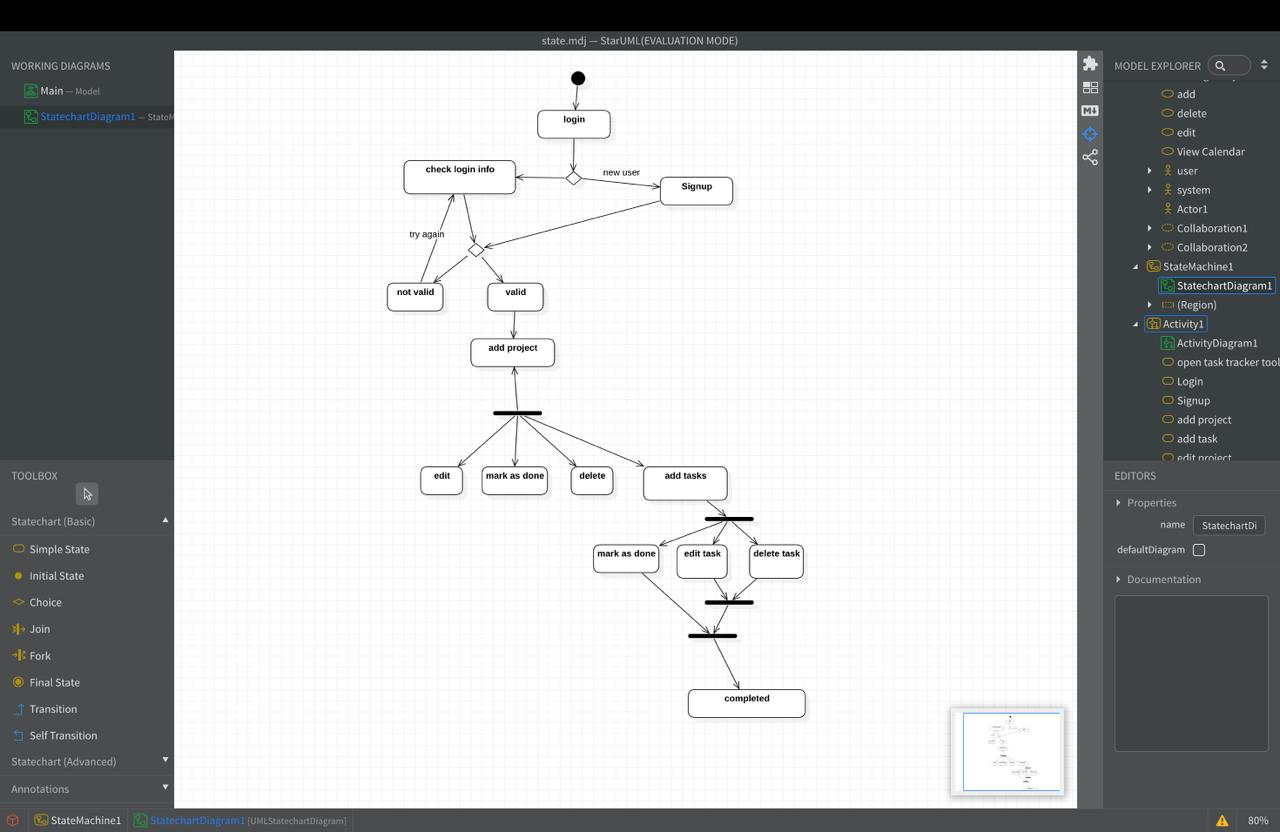


Figure 5 State Diagram

# **Section 9: Design Patterns**

## **9.1** **Factory Method Model**

### **Definition:** **The Factory Method pattern is used to create objects without specifying the exact class of the object that will be created.**

### **Responsibilities:**

## **Encapsulates the creation of objects.**

## **Allows the code to create different types of objects without knowing the exact class. Implementation:**

## **9.2 Observer Pattern**

## **The Observer pattern allows an object (subject) to notify other objects (observers) about changes in its state, without knowing who or what those objects are. Responsibilities:**

## **Notifies observers about state changes.**

## **Allows for real-time updates in response to changes in the system.**

## **9.3 Strategy Pattern**

### **Definition:** **The Strategy pattern allows a class to change its behavior based on different conditions or contexts by switching between strategies. Responsibilities:**

### **Defines a family of algorithms or behaviors.**

### **Allows for switching between algorithms based on the context.**

### **Implementation: In Django, the controllers (views) handle the API requests from the frontend, process them, and return the data to the View.**

## **9.4 Benefits of Patterns**

**- Separation of Concerns: Each pattern focuses on a specific aspect of the application, making the code easier to manage and extend.**

**- Flexibility: By using design patterns, the system becomes more flexible, allowing for easier changes and additions without breaking existing functionality.**

**- Scalability: Design patterns provide structured solutions that can grow with the application, making it easier to scale and add new features.**