# Project Design Document

|  |  |
| --- | --- |
| Task | Responsible Team Member |
| System Overview | Tuna Yılmaz, Demir Doruk Dilek, Ahmet Kaan Tırhış |
| Implementation Details | Tuna Yılmaz |
| Use Case Support in Design | Tuna Yılmaz |
| Design Decisions | Tuna Yılmaz, Demir Doruk Dilek, Aahmet Kaan Tırhış |
| GitHub Commit Requirement | Tuna Yılmaz |

**1. System Overview**

**Project Description**

PlanEasy is a to-do list application designed to help users organize their daily tasks efficiently. Users can add, edit, delete tasks, and set reminders. The application operates independently, using its own data management system without relying on external APIs.

**System Architecture**

* **Layered architecture** (Frontend, Backend, Database) is used.
* **Data Management:** JPA and Spring Data JPA.
* **Frontend:** Java.
* **Version Control:** GitHub.

**2. Implementation Details**

**Codebase Structure**

* The project follows a modular structure under the com.todo package.
* Key modules include:
  + **UI Module:** Manages the user interface.
  + **Logic Module:** Handles business logic and data processing.
  + **Database Module:** Manages data storage and retrieval.

**Key Implementations**

* **Task Management Module:** Allows users to add, edit, and delete tasks.
* **Notification Module:** Sends reminders for upcoming tasks.
* **Data Storage Module:** Ensures persistent data storage for tasks.

**Component Interfaces**

* API endpoints and function signatures for key operations will be defined.
* Example:

java

public class Task {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    private String description;

    private Boolean completed;

    private LocalDateTime createdAt;

    private List<String> tags;

    private String color;

    private LocalDate deadline;

    public Task()

    public Task(String description, List<String> tags)

    public Task(String description, List<String> tags, LocalDate deadline)

}

**3. Use Case Support in Design**

This section will outline four key use cases based on the **Requirements Document** and describe how the system architecture supports them.

**1. Use Case Selection**

The following four use cases have been selected as critical functionalities for the PlanEasy application:

1. **Adding a New Task** (Corresponds to Functional Requirement #1)
2. **Editing an Existing Task** (Corresponds to Functional Requirement #2)
3. **Task Filtering by Category** (Corresponds to Functional Requirement #9)
4. **Task Reminder Notifications** (Corresponds to Functional Requirement #6)

**2. Requirement Mapping**

Each selected use case is mapped to the corresponding functional requirement:

| **Use Case** | **Functional Requirement** |
| --- | --- |
| **Adding a New Task** | Users should be able to add tasks. |
| **Editing an Existing Task** | Users should be able to edit tasks. |
| **Task Filtering by Category** | The application should allow users to list tasks by category. |
| **Task Reminder Notifications** | Users should be able to activate reminders. |

**3. Use Case Design**

**Use Case 1: Adding a New Task**

**System Architecture Support**

* **Frontend:**
  + The user interacts with the **Task Creation UI** to input task details (title, description, due date).
  + The UI validates the inputs before submission.
* **Backend:**
  + A request is sent to the **Task Management Module**.
  + The task data is stored in the **SQLite/Firebase database**.
* **Database:**
  + A new record is created in the tasks table.

**Data Flow**

1. User clicks "Add Task" in the UI.
2. The application collects input data and sends it to the backend.
3. The backend processes and stores the task in the database.
4. The updated task list is retrieved and displayed to the user.

**Use Case 2: Editing an Existing Task**

**System Architecture Support**

* **Frontend:**
  + Users select a task and modify its details in the **Task Editor UI**.
* **Backend:**
  + The request is processed by the **Task Management Module**.
  + The updated task details are stored in the database.
* **Database:**
  + The existing task entry is modified.

**State Changes**

* **Before Editing:**

{

"task\_id": 101,

"title": "Submit Assignment",

"due\_date": "2025-03-12"

}

* **After Editing:**

{

"task\_id": 101,

"title": "Submit Final Assignment",

"due\_date": "2025-03-14"

}

**Use Case 3: Task Filtering by Category**

**System Architecture Support**

* **Frontend:**
  + Users select a category filter in the **Task List UI**.
* **Backend:**
  + The **Filtering Module** queries the database for tasks under the selected category.
* **Database:**
  + The filtered task list is retrieved and sent back to the frontend.

**Interaction Example**

1. User selects "Work" from the filter options.
2. The application queries all tasks with category = "Work".
3. The task list updates to show only relevant tasks.

**Use Case 4: Task Reminder Notifications**

**System Architecture Support**

* **Frontend:**
  + Users enable reminders via the **Task Settings UI**.
* **Backend:**
  + The **Notification Module** schedules a notification at the task's due date.
* **Database:**
  + The reminder settings are stored with the task.

**Data Flow**

1. User sets a reminder for a task.
2. The application stores the reminder details in the database.
3. At the scheduled time, the notification service triggers an alert.

**4. Demo Requirement**

* These four use cases **must be fully implemented** and will be demonstrated during the final project presentation.
* The evaluation will include:
  + **Correctness:** Ensuring expected system behavior.
  + **Completeness:** Verifying all functionalities.
  + **Documentation adherence:** Checking if the implementation aligns with the design.

**4. Design Decisions**

**Technology Comparisons**

* **SQLite vs Firebase:**
  + SQLite is a local database suitable for offline access.
  + Firebase provides cloud-based data storage but requires internet connectivity.
  + **Decision:** SQLite is preferred for offline functionality.
* **Java vs Kotlin:**
  + Java is widely used and has robust documentation.
  + Kotlin is more concise and offers better null safety.
  + **Decision:** Java is chosen due to team familiarity and ease of debugging.

**5. GitHub Commit Requirement**

* **Code Implementations & Interfaces:**
  + All implemented features, including core functionalities, component interfaces, and UI designs, must be committed to the GitHub repository.
  + Each commit must include a clear and descriptive message explaining the changes made.
  + The repository should follow a structured branching strategy, such as **feature branches** for new implementations and **main/development branches** for stable versions.
* **Technology Comparisons:**
  + The source code snippets used in technology comparisons (e.g., SQLite vs. Firebase, Java vs. Kotlin) must be committed to the repository.
  + Each comparison should include a brief explanation in the commit message, clarifying the reasons behind the chosen technology.
  + The comparison results should also be documented in the repository’s README file or a separate documentation file.