
جامعة طرابلس

كليـة الهندسة

قسم الهندسة الكهربائية والإلكترونية


# EE432

## PROJECT TO DO LIST APPLICATION


| NAMES | ID NUMBER |
|---|---|
| Ahmed Al-Amare | 2190203870 |
| Abdelhaseeb Eljamal | 2190202771 |
| Ibrahim Benhalim | 2190203702 |

# ABSTRACT

This report presents the development of a Task Manager application utilizing PyQt5 for GUI design and Tkinter for additional interface elements. The system provides users with the ability to add, remove, filter, and manage tasks with priority-based sorting and status tracking. The application incorporates animations, a splash screen, and a persistent storage system using JSON.

# INTRODUCTION

Task management is an essential part of productivity. This project aims to create a user-friendly desktop application that allows users to efficiently manage tasks. The application provides a graphical interface for adding, organizing, and tracking tasks using PyQt5, ensuring a responsive and intuitive experience. Tkinter is also utilized to extend functionality.

# DATA STRUCTURES USED

1. Task Class (Custom Data Structure)

The Task class is a fundamental data structure that represents an individual task. Each task has the following attributes:

- description: A string that holds the task details.
- priority: A string representing the priority level (High, Medium, Low).
- status: A boolean indicating whether the task is completed (True) or incomplete (False).

```
class Task:
    def __init__(self, description, priority, status=False):
        self.description = description
        self.priority = priority
        self.status = status
```

This class provides a structured way to manage task-related data efficiently.

2. List Data Structure (Array)

The application stores all tasks in a **list** (self.all_tasks). This list is used for:

- Storing tasks dynamically.
- Iterating through tasks for searching and filtering.
- Updating and deleting tasks.

Example usage in the application:

```
self.all_tasks = self.load_tasks()
```

Each task in this list is an instance of the Task class.

---

3. JSON FILE FOR PERSISTENT STORAGE

The application uses a **JSON file (tasks.json)** to store tasks persistently.

- **When saving tasks**, the list of Task objects is converted into a list of dictionaries.
- **When loading tasks**, the JSON data is converted back into Task objects.

Example of saving tasks:

```
def save_tasks(self):
    data = [{'description': t.description, 'priority': t.priority, 'status': t.status}
            for t in self.all_tasks]
    with open('tasks.json', 'w') as f:
        json.dump(data, f, indent=4)
```

Here, JSON works like a **dictionary-based** structure, where each task is stored as a dictionary.

---

4. TASK TABLE MODEL

This class is responsible for **displaying tasks in a table**. It acts as a bridge between the data and the UI.

- The data is stored as a **list of Task objects**.
- The rowCount() and columnCount() methods define the **2D structure of the table**.
- The data() method fetches specific task attributes to display in the UI.

Example:

```
class TaskTableModel(QAbstractTableModel):
    def __init__(self, tasks=None):
        super(TaskTableModel, self).__init__()
        self.headers = ['Task', 'Priority', 'Status']
        self.tasks = tasks or []

    def data(self, index, role):
        if not index.isValid():
            return None

        task = self.tasks[index.row()]
        column = index.column()

        if role == Qt.DisplayRole:
            if column == 0:
                return task.description
            elif column == 1:
                return task.priority
            elif column == 2:
                return '✅ Completed' if task.status else '❌ Incomplete'
```

Here, the table model **manages tasks in a structured tabular format**.

---

5. Dictionary (Used for Priority Sorting)

A dictionary is used for **sorting tasks by priority** efficiently.

- Each priority level is assigned a numerical value to enable sorting.

Example:

```
priority_order = {'🚨 High': 0, '⚠️ Medium': 1, '✅ Low': 2}
tasks.sort(key=lambda x: priority_order[x.priority])
```

This ensures that tasks are sorted from **High → Medium → Low** priority.

The **combination of lists, dictionaries, JSON files, and PyQt's table model** allows the application to efficiently **store, modify, and display tasks** in a structured way.

# FUNCTIONALITY IMPLEMENTATION

## 1- ADDING A TASK

Functionality: Allows the user to add a new task with a description, priority level, and completion status.

Implementation:

- The user enters task details in the input fields.
- A new Task object is created and added to the self.all_tasks list.
- The table view updates automatically.
- Tasks are saved persistently in tasks.json.

```python
def add_task(self):
    description = self.task_input.text().strip()
    if not description:
        QMessageBox.warning(self, "Input Error", "Task description cannot be empty!")
        return

    priority = self.priority_combo.currentText()
    new_task = Task(description, priority)
    self.all_tasks.append(new_task)
    self.apply_filters()
    self.task_input.clear()
    self.add_anim.start()
```

## 2- DISPLAYING TASKS IN A TABLE VIEW

The TaskTableModel class is responsible for displaying tasks inside a QTableView. It formats how tasks appear, including their description, priority, and completion status.

```python
class TaskTableModel(QAbstractTableModel):
    def __init__(self, tasks=None):
        super(TaskTableModel, self).__init__()
        self.headers = ['Task', 'Priority', 'Status']
        self.tasks = tasks or []

    def data(self, index, role):
        if not index.isValid():
            return None

        task = self.tasks[index.row()]
        column = index.column()

        if role == Qt.DisplayRole:
            if column == 0:
                return task.description
            elif column == 1:
                return task.priority
            elif column == 2:
                return '✅ Completed' if task.status else '❌ Incomplete'

        if role == Qt.TextAlignmentRole:
            return int(Qt.AlignLeft | Qt.AlignVCenter) if column == 0 else int(Qt.AlignCenter)

        return None
```

### 3. Searching and Filtering Tasks

Users can search for tasks using a keyword or filter them based on priority and completion status.

```
def apply_filters(self):
    search_text = self.search_input.text().lower()
    status_filter = self.filter_status.currentIndex()
    tasks = [task for task in self.all_tasks
             if search_text in task.description.lower() and
             (status_filter == 0 or
              (status_filter == 1 and task.status) or
              (status_filter == 2 and not task.status))]

    if self.display_mode.currentIndex() == 1:
        priority_order = {'🚨 High': 0, '⚠️ Medium': 1, '✅ Low': 2}
        tasks.sort(key=lambda x: priority_order[x.priority])

    self.table_model.updateData(tasks)
    self.task_count_label.setText(f"Total Tasks: {len(tasks)}")
```

### 4. Marking a Task as Completed

Users can double-click a task in the table to toggle its completion status.

```
def toggle_task_status(self, index):
    if index.column() == 2:
        self.table_model.setData(index, value: None, Qt.EditRole)
```

### 5. Deleting Selected Tasks

Users can delete tasks by selecting them and clicking the delete button.

```
def delete_selected_tasks(self):
    selected = self.table_view.selectionModel().selectedRows()
    if not selected:
        QMessageBox.warning(self, "No Selection", "Please select tasks to delete!")
        return

    confirm = QMessageBox.question(
        self,
        "Confirm Deletion",
        f"Delete {len(selected)} selected task(s)?",
        QMessageBox.Yes | QMessageBox.No
    )

    if confirm == QMessageBox.Yes:
        for index in sorted(selected, reverse=True):
            task = self.table_model.tasks[index.row()]
            if task in self.all_tasks:
                self.all_tasks.remove(task)
        self.apply_filters()
```

# Framework Choice

For the development of **Task Manager Pro**, I chose **PyQt5**, a Python binding for the **Qt framework**, due to its rich set of features and powerful GUI capabilities. Below are the key reasons behind this choice:

## 1. Rich and Modern UI Components

**PyQt5** provides a vast collection of **widgets, layouts, and styling options**, making it ideal for developing an interactive and visually appealing **task management** application. It supports:
**QTableView** – For structured data display
**QComboBox, QLineEdit, QPushButton** – For user input and controls
**QMessageBox** – For alerts and confirmations

**Example:** The **task list** is implemented using a **QTableView**, allowing sorting, filtering, and selection.

## 2. Flexibility and Scalability

PyQt5 allows developers to create applications ranging from **simple to complex** while keeping the **code modular and reusable**. It also provides **custom styling using Qt Style Sheets (QSS)**, similar to CSS.

**Example:** The application UI can be customized with **dark mode themes, animations, and responsive layouts**.

## 3. MVC Architecture for Clean Code Structure

PyQt5 follows an **MVC (Model-View-Controller)** pattern, helping separate **data management (model)** from **UI rendering (view)**.

- **Model (TaskTableModel)** – Manages task data
- **View (QTableView)** – Displays tasks dynamically
- **Controller (Main Application Logic)** – Handles user interactions

**Example:** The **TaskTableModel** class handles how tasks are displayed, while the main application updates and filters them.

### 4. Cross-Platform Compatibility

PyQt5 applications run on **Windows, macOS, and Linux**, making it easy to distribute and use across different operating systems.

**Example:** The same Python code can be packaged as a standalone executable using **PyInstaller** for easy deployment.

### 5. Event-Driven Programming Model

PyQt5 uses **signals and slots**, enabling smooth user interactions and **real-time UI updates** without constant polling.

**Example:** When a user **adds a task**, the **table updates instantly** without requiring a page refresh.

### 6. Strong Documentation & Community Support

With extensive **official documentation** and an **active developer community**, PyQt5 offers ample learning resources and troubleshooting support.

**Example:** If an issue arises with **QTableView filtering**, solutions can be found in **Qt documentation or Stack Overflow**.

### Alternative Considerations

Other GUI frameworks were considered but had limitations:

- **Tkinter** – Simpler but lacks advanced widgets like **QTableView**.
- **Kivy** – Great for mobile apps but less suited for **desktop task management**.
- **PySide2** – Similar to PyQt5 but has licensing constraints.

### Final Decision

After evaluating these factors, **PyQt5 was the best choice** for building a **feature-rich, scalable, and cross-platform** task manager with a **modern UI** and **smooth interactions**.

# Team Contribution

| Name | Percentage | Work part |
|---|---|---|
| Ahmed Al-Amare | 50% | Main code |
| Abdelhaseeb Eljamal | 25% | GUI, README, Report |
| Ibrahim Benhalim | 25% | GUI, README, Report |

# CONCLUSION

The Task Manager application successfully provides an efficient, user-friendly interface for managing tasks. With its PyQt5-based design, JSON storage, and Tkinter integration, it offers an engaging experience for users. Future improvements could include cloud synchronization and mobile compatibility.