1. System Design of the Software

1.1 High Level System Architecture
1.2 Component Breakdown
1.3 System Design Diagram

2. UI Component Inventory

2.1 Login Window
2.2 Auth Entry Page
2.3 Menubar Components and Their Roles
2.4 Sidebar Features
2.5 WorkArea Functionality

3. Software Development Principles & Folder Structure

3.1 Principles of Good Software Architecture Applied
3.2 Project Folder Structure Overview
3.3 Best Practices Reflected in Folder Naming and Layout
3.4 Collaboration and Team Readiness
3.5 CI/CD and Testing Integration Potential

4. Introduction to PyQt5

4.1 What is PyQt5?
4.2 Core Features of PyQt5
4.3 Why PyQt5 Was Chosen for Sukriya HRMS

5. Application of PyQt5 in Sukriya HRMS

5.1 UI Construction using QWidget and QLayouts
5.2 Signal-Slot Mechanism and Event Handling
5.3 Multithreading using QObject and Worker Classes
5.4 Connecting UI and Backend
5.5 Maintaining Responsiveness and Separation
5.6 Data Flow Diagram

6. Two-Factor Authentication System (2FA)

1. System Design of the Software

Sukriya HRMS is a comprehensive desktop-based Human Resource Management System developed using Python and the PyQt5 GUI framework. The architecture of the application is thoughtfully designed to separate concerns, maintain modularity, and ensure scalability. The software follows a structured MVC-inspired approach, separating the UI, business logic, and data handling into distinct layers for better maintainability and collaborative development.

The HRMS software is organized into three main UI segments — Menubar, Sidebar, and WorkArea. The WorkArea is a dynamic container that loads and displays the appropriate feature page depending on user interaction. Upon launching the software, the first component that interacts with the user is the Auth Entry page — which encapsulates login, signup, and password recovery functionalities through a secure and multi-step authentication system, including Two Factor Authentication (2FA).

1.1 High Level System Architecture

Frontend (Presentation Layer)

- **Framework Used**: PyQt5
- **Components**: QMainWindow, QWidgets, QLayouts, QHBoxLayout, QVBoxLayout, QPushButton, QLabel, QStackedWidget
- **Responsibility**: Render and manage UI elements, user interaction capture, and user feedback.

- **Behavior**: Modular UI components structured in ui/, such as menubar, sidebar, layout managers, and workarea widgets.

Backend (Business Logic Layer)

- **Technology**: Python
- **Structure**: Encapsulated within the backend/ folder using QObject subclasses and custom worker classes for threading.
- **Functionality**:
    - Authentication handling (login, signup, password reset)
    - Business operations like data querying, validation, state management
    - Asynchronous processing to keep the UI responsive

Data Layer

- **Technology**: Online SQL Database (MySQL)
- **Interaction**: Through secure queries using Python MySQL connectors
- **Structure**: Encapsulated in utility classes and called from the backend for read/write operations

1.2 Component Breakdown

1.2.1 Launch Flow

- When the software is opened, the main window instantiates and loads the Auth Entry page inside the WorkArea.
- The user interacts with the login, signup, or reset password widgets.
- Upon successful authentication, the dashboard is loaded in the WorkArea and the sidebar and menubar are enabled.

1.2.2 WorkArea Dynamics

- All internal pages like Dashboard, Employee, Attendance, etc., are dynamically loaded widgets shown within a central WorkArea container.
- Each sidebar button triggers a signal-slot connection that swaps the widget being shown in the WorkArea.

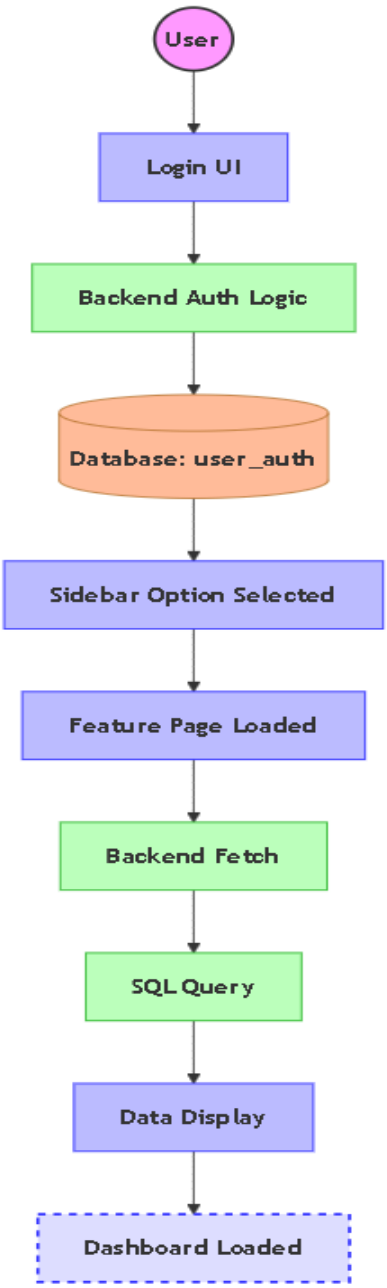
1.2.3 Data Fetch and Update

- The backend fetches user and operational data from the SQL database asynchronously using worker objects.
- This separation ensures that the UI remains smooth, and heavy computations or waits don't freeze the interface.

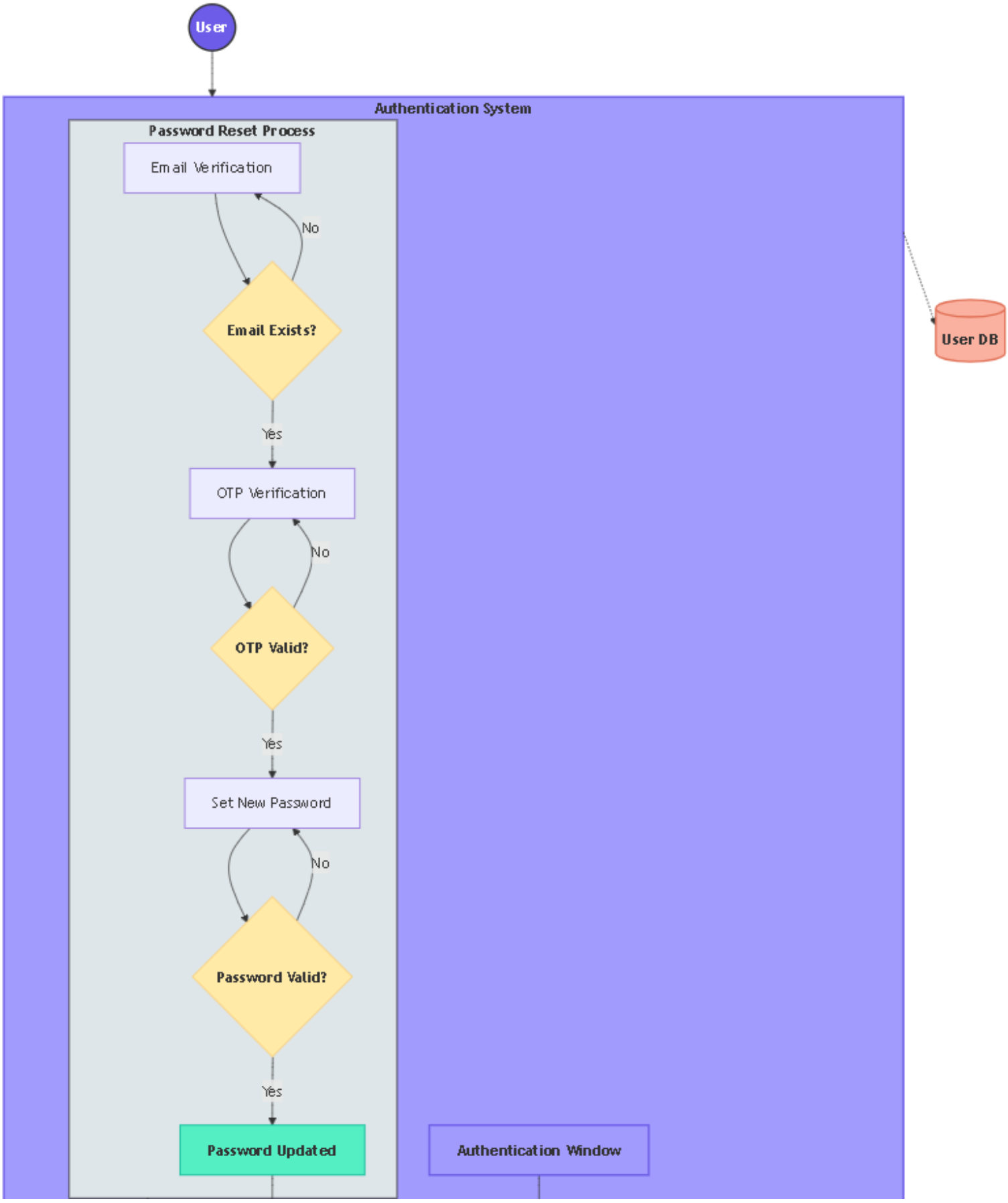
1.2.4 System Notifications and Profile

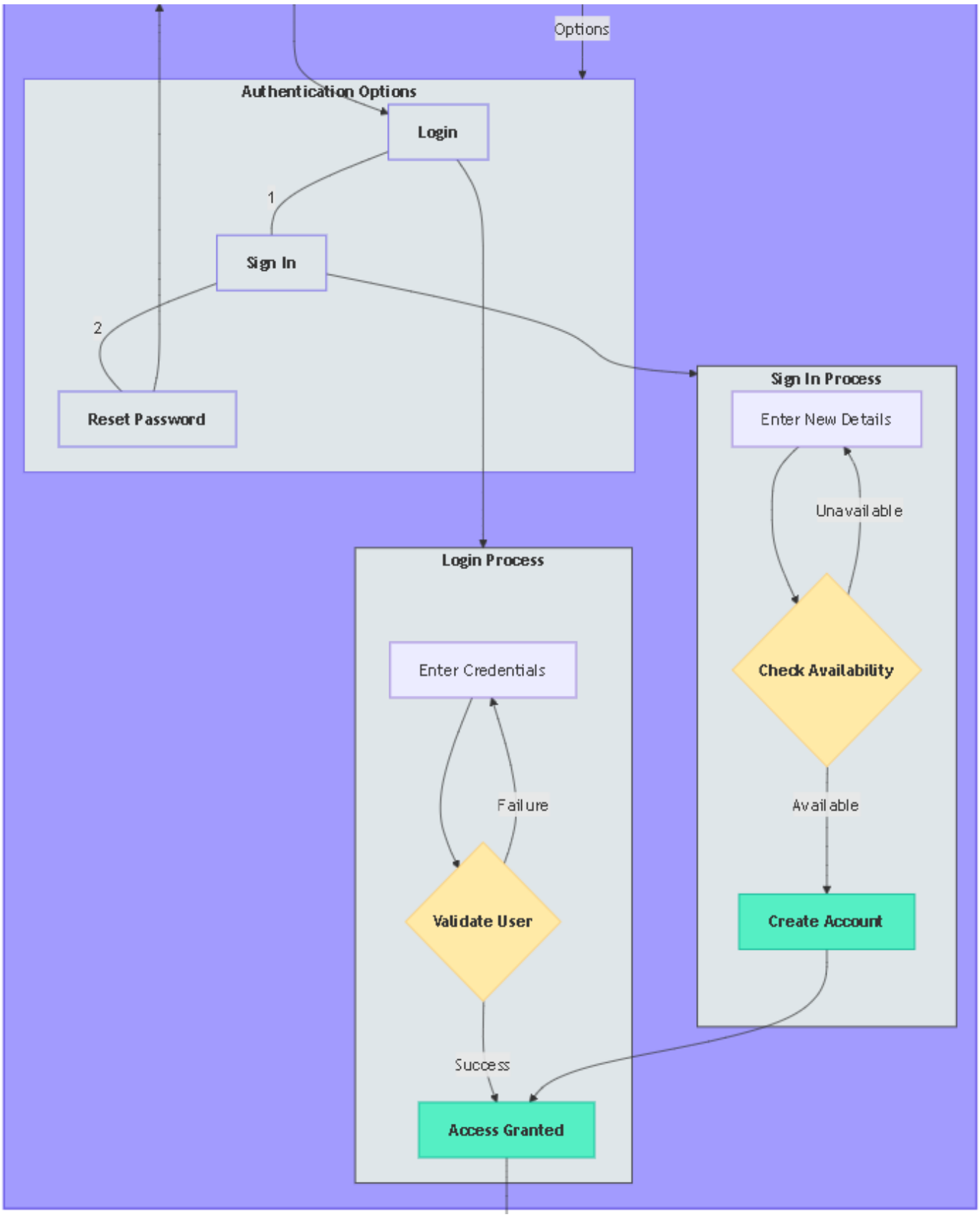
- The menubar includes an area for displaying system-generated and company-specific notifications (under development).
- The user profile dropdown will handle profile updates and logout operations.

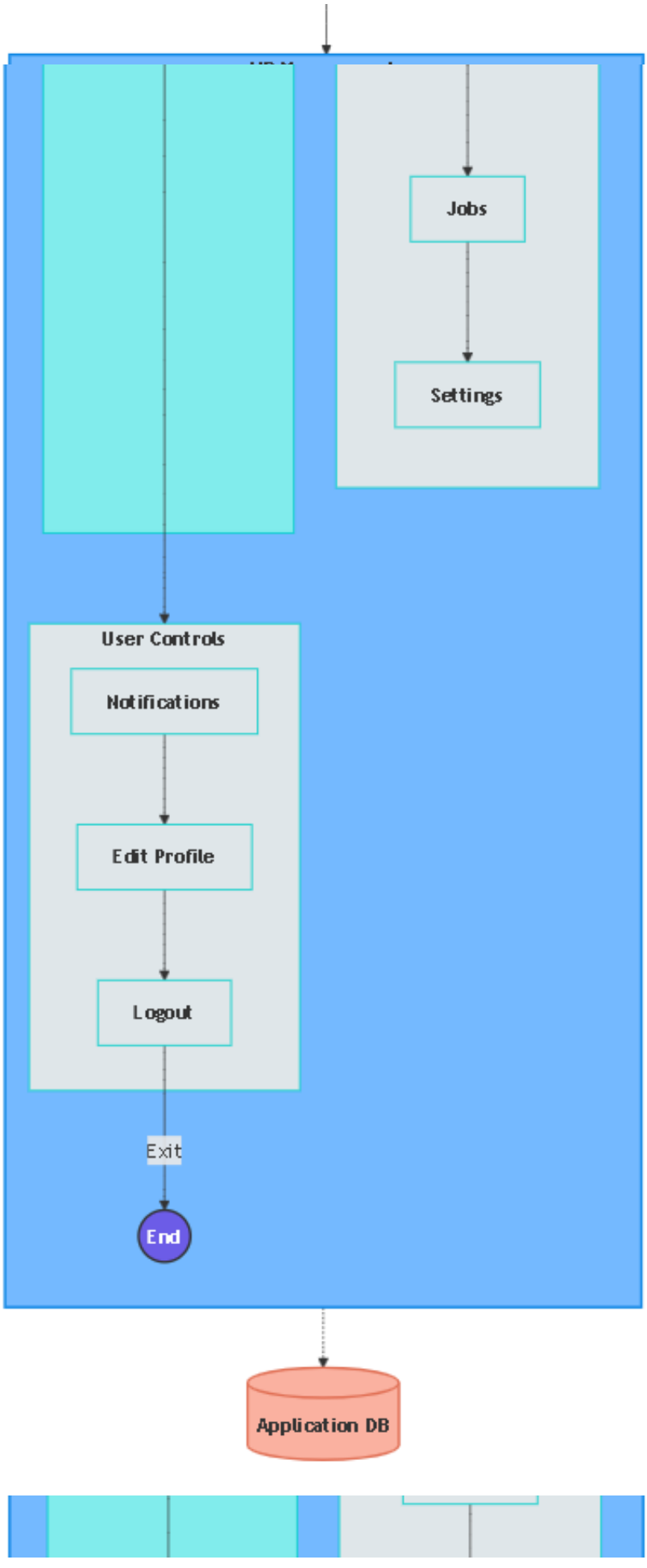
1.2.5 Software Entry Process: Data Flow Diagram



1.3 System Design Diagram


User
Authentication System
Password Reset Process
Email Verification
No
Email Exists?
Yes
OTP Verification
No
OTP Valid?
Yes
Set New Password
No
Password Valid?
Yes
Password Updated
User DB
Authentication Window


Options
Authentication Options
Login
1
Sign In
2
Reset Password
Sign In Process
Enter New Details
Unavailable
Check Availability
Available
Create Account
Login Process
Enter Credentials
Failure
Validate User
Success
Access Granted


Jobs
Settings
User Controls
Notifications
Edit Profile
Logout
Exit
End
Application DB

2. UI Component Inventory

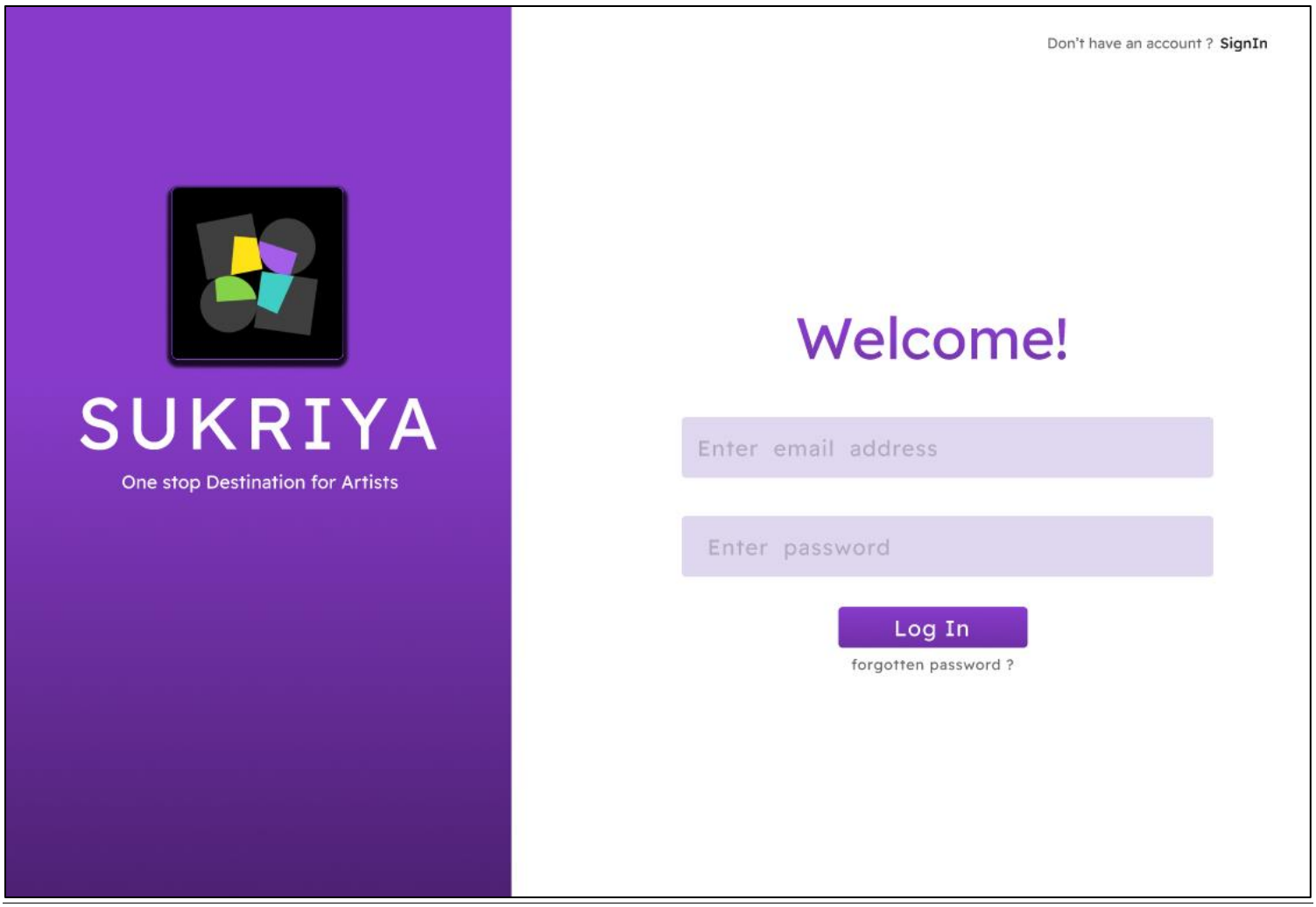
This section provides a detailed breakdown of all the core UI components in the Sukriya HRMS desktop software, including their structure, behavior, and how they interconnect to deliver a seamless user experience. Each component plays a vital role in ensuring modularity, ease of navigation, and feature accessibility.

2.1 Login Window

Overview

The **Login Window** is the very first interactive screen that appears upon launching the software. It acts as a gatekeeper to the system, allowing only authenticated users to proceed to the main application.

Login Window UI:



Components

- **QLineEdit** fields for email and password
- **QPushButton** for "Login", "Sign Up", and "Forgot Password"
- **QLabel** for branding and minor instruction hints
- **Form validation** to prevent empty or invalid fields
- **Signal connections** to trigger backend authentication logic
- **Error display mechanism** for invalid credentials

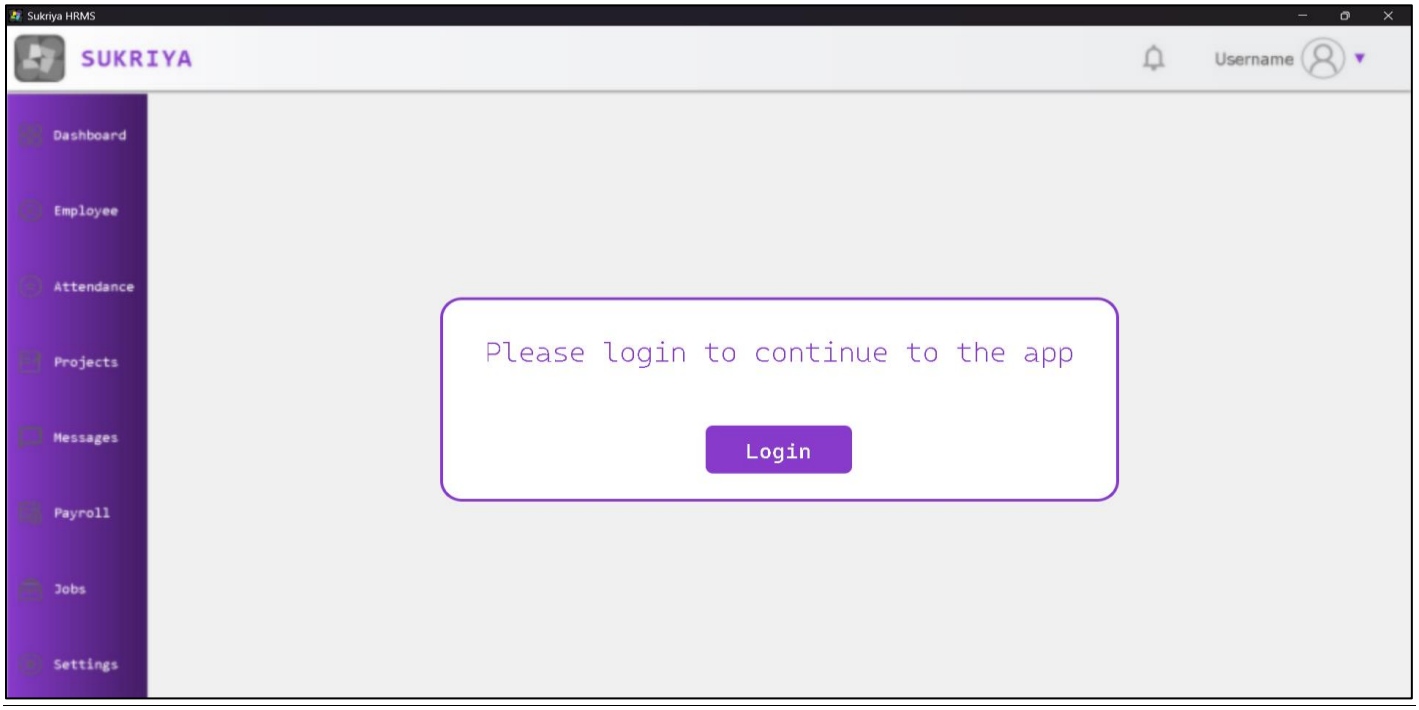
Workflow

- User enters email and password
- On login click, signal is emitted to backend/auth

- Backend verifies credentials with SQL database
- On success, 2FA (OTP window) is triggered
- On failure, error message is shown

2.2 Auth Entry Page

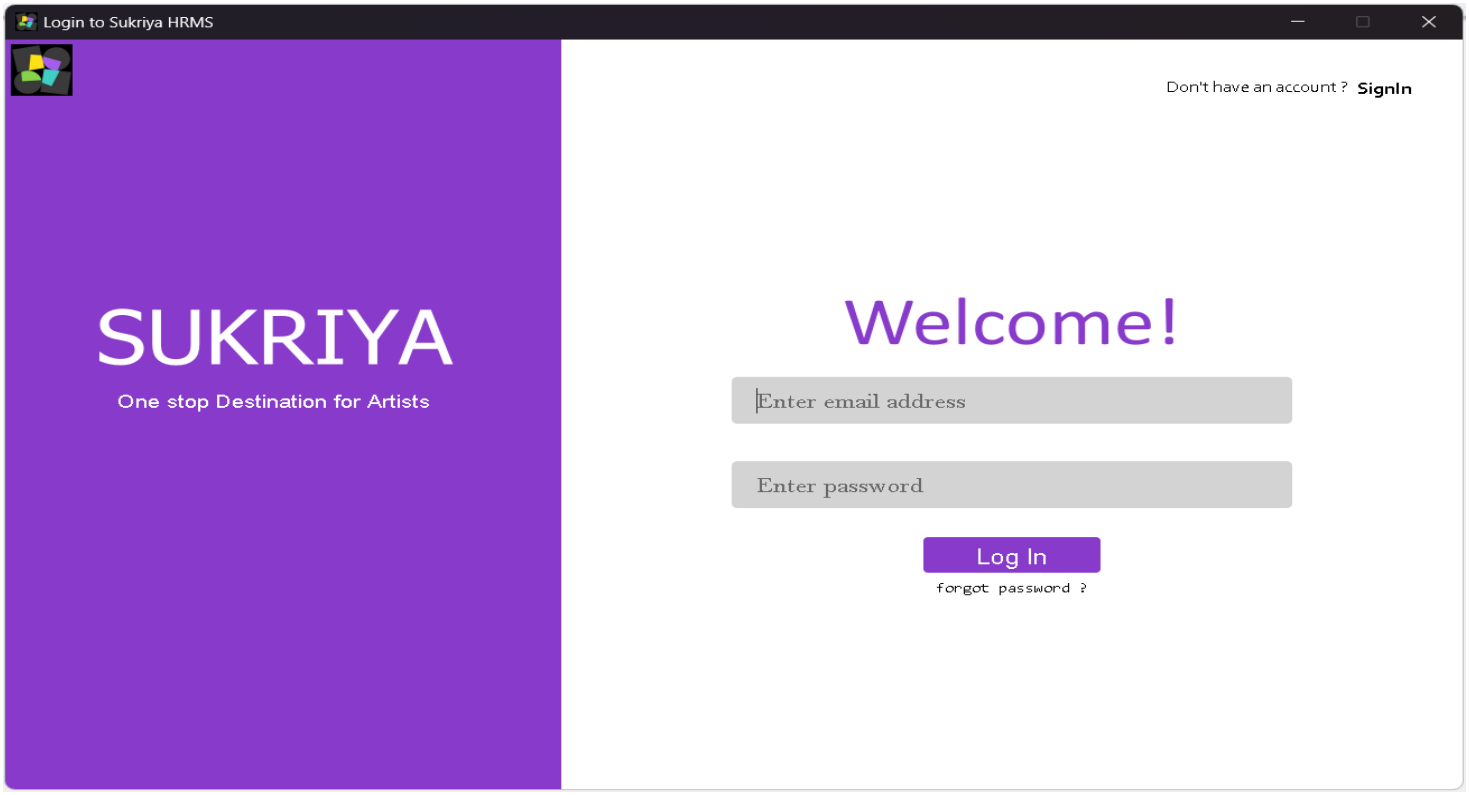
Auth Entry Page UI:



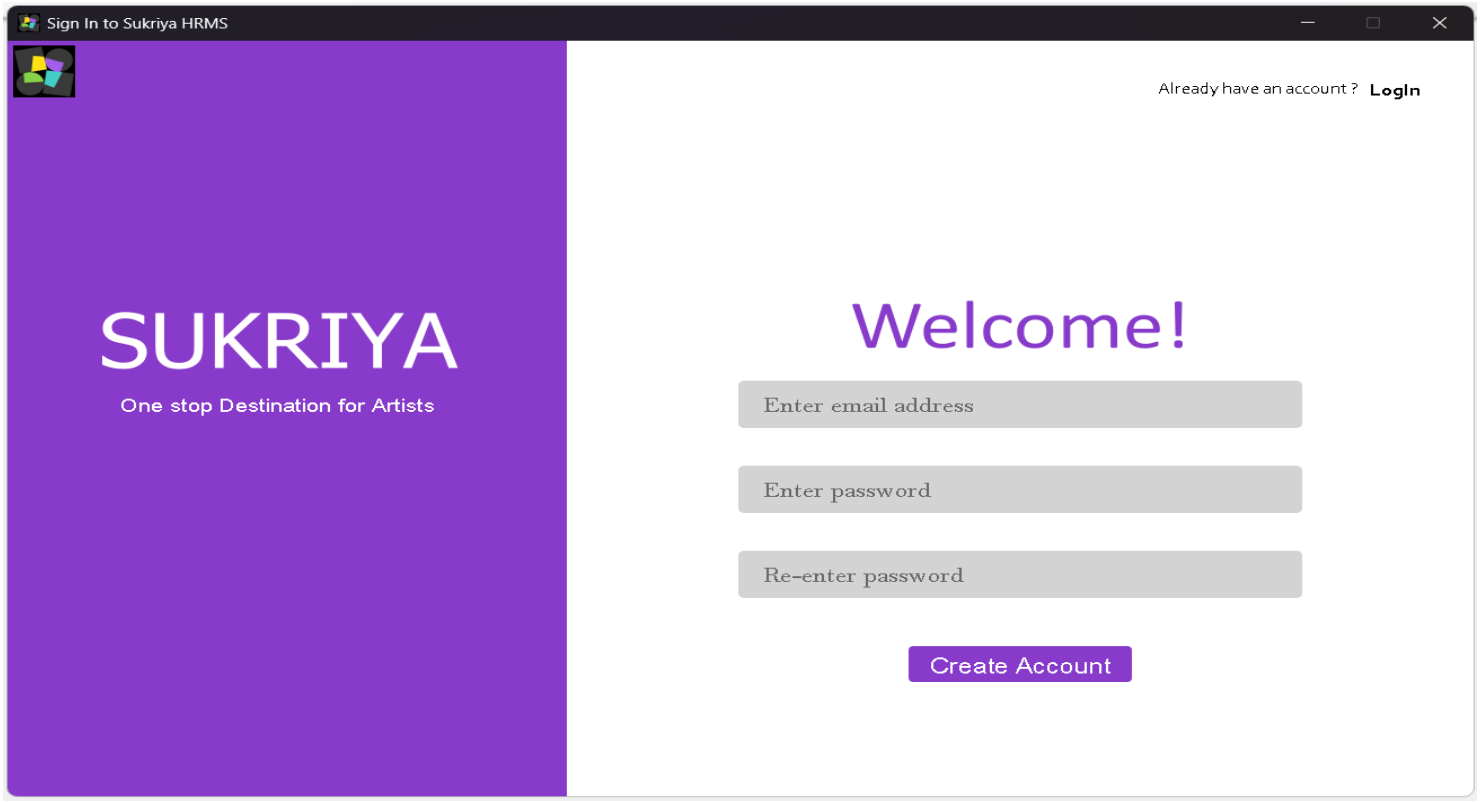
Overview

The Auth Entry Page is a multi-functional window that encapsulates:

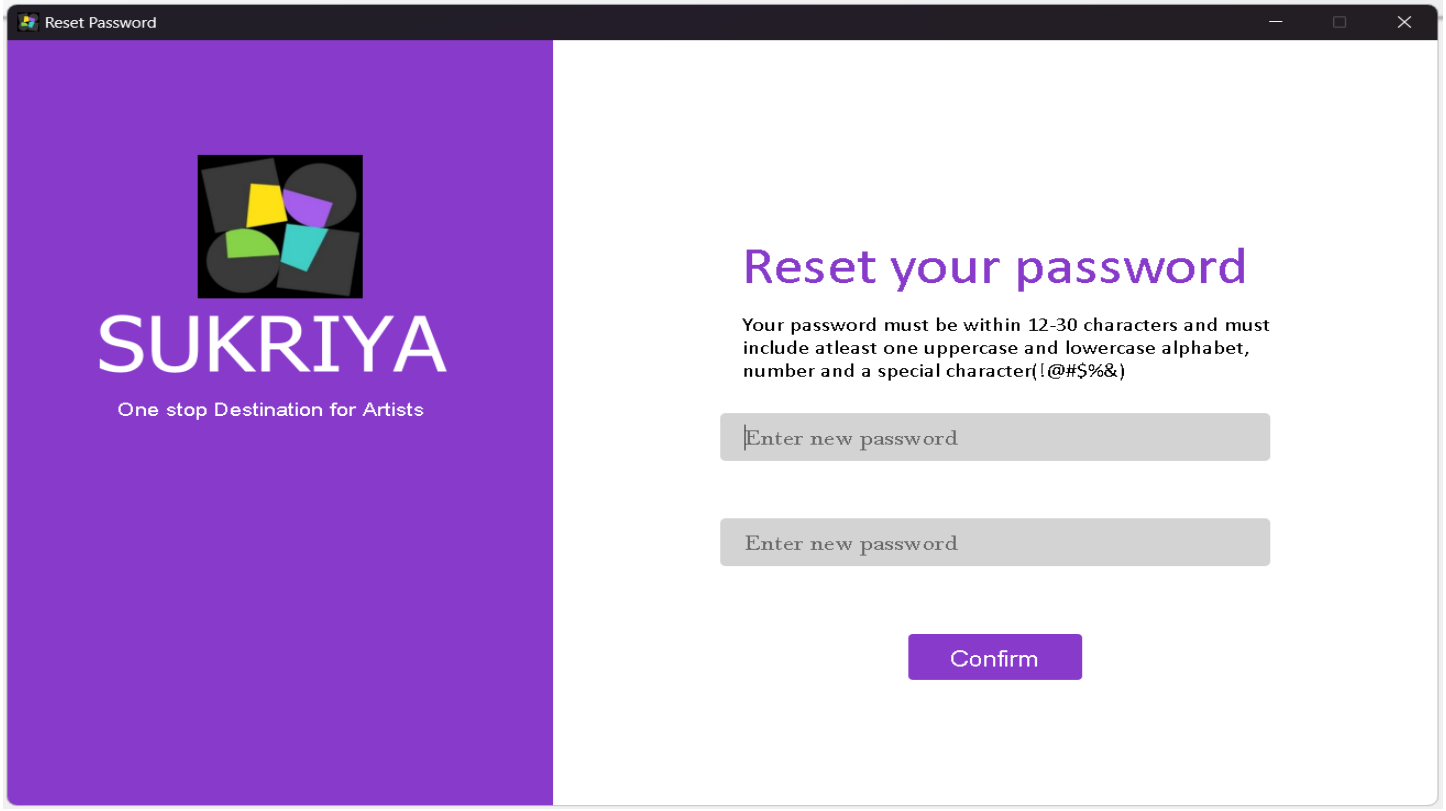
- **Login**



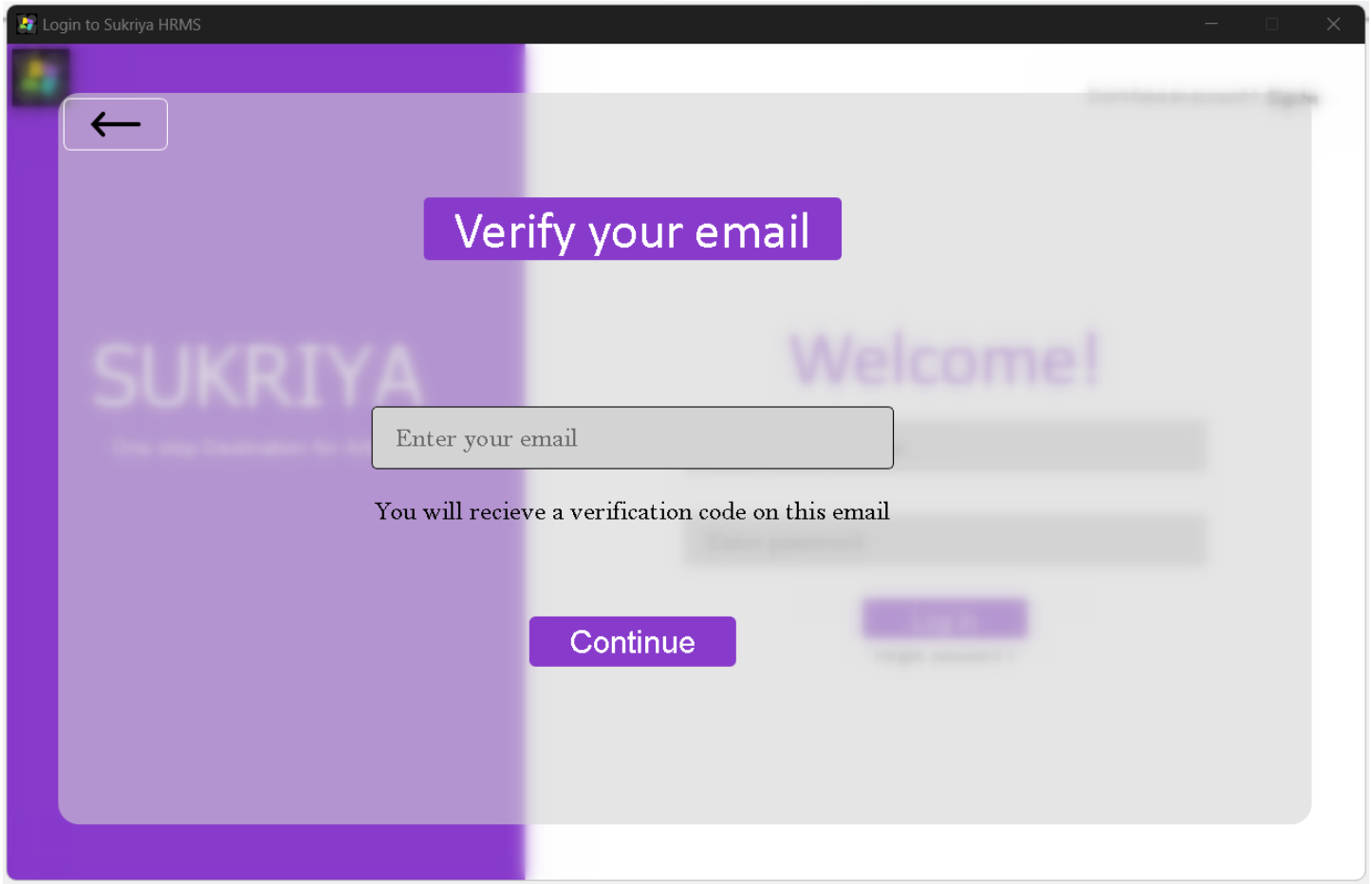
- **Sign In**



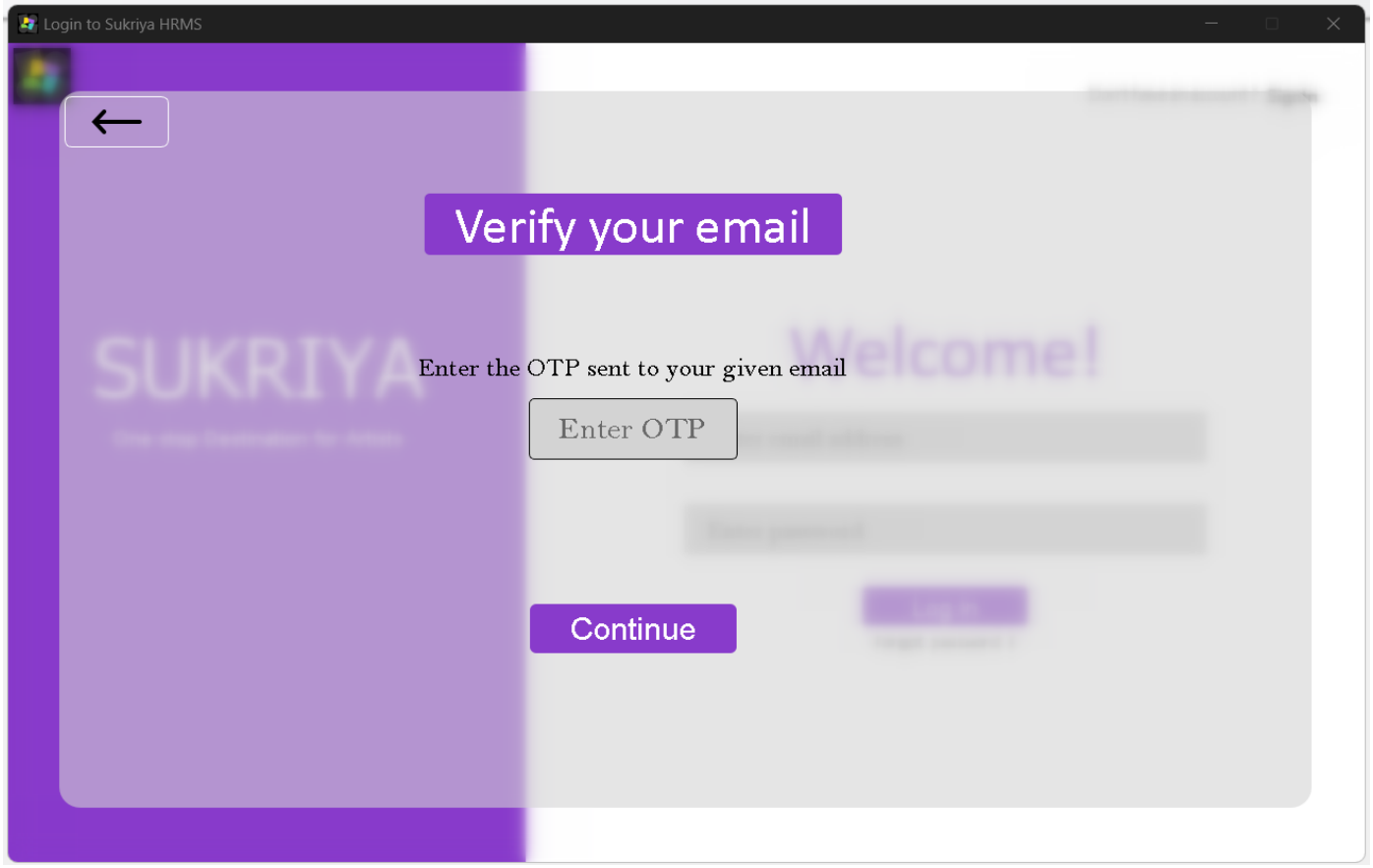
- **Password Reset**



- **Two Factor Authentication (Email Verification)**



- **Two Factor Authentication (OTP)**



It is loaded inside the WorkArea when the application launches and dynamically switches between various auth-related forms.

Components

- **Stacked widgets** for switching between login/signup/reset/OTP screens
- **Dynamic styling** for user experience continuity
- **Reusability of input fields** using custom PyQt5 components
- **Seamless transitions** using layout managers and animations (if enabled)
- **OTP input mechanism** for 2FA

Security Implementation

- No page transition occurs until valid credentials are submitted
- Password reset is OTP-verified
- All form data is validated both client-side and server-side

---

2.3 Menubar Components and Their Roles

Menubar UI:

Overview

The **Menubar** is a persistent component displayed at the top of the main window once the user is authenticated. It acts as a control center for the user's session and overall application status.

Components

- **Company Branding QLabel**
- **Notification Bell Icon**
- **Dropdown Menu for Profile**
    - Edit Profile
    - Logout

Behavior

- Notifications are dynamically updated from backend (future feature)
- Profile menu provides secure session and logout flow
- Positioned using QHBoxLayout with spacing and stretching for visual clarity

2.4 Sidebar

Sidebar UI:

Features

Overview

The **Sidebar** is the primary navigation panel of the application. It appears on the left side and gives access to various HR modules.

Structure

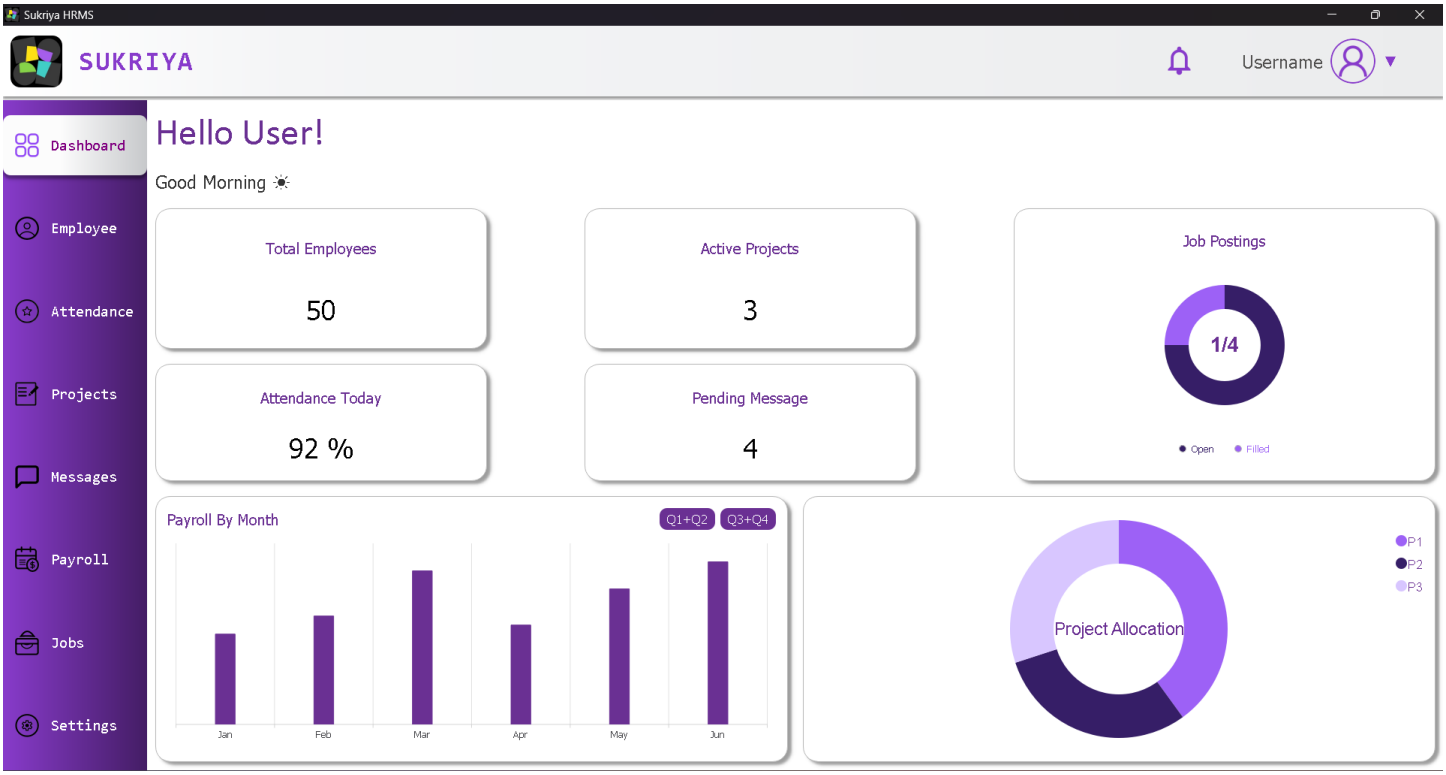
Each sidebar button is a composite widget:

- **QHBoxLayout** → QLabel (Icon) + QPushButton (Text)
- Icons and labels follow a 15%:85% layout stretch
- Designed with consistent styling and hover behavior

Features Available

Each of these loads its respective module dynamically into the WorkArea:

- **Dashboard**

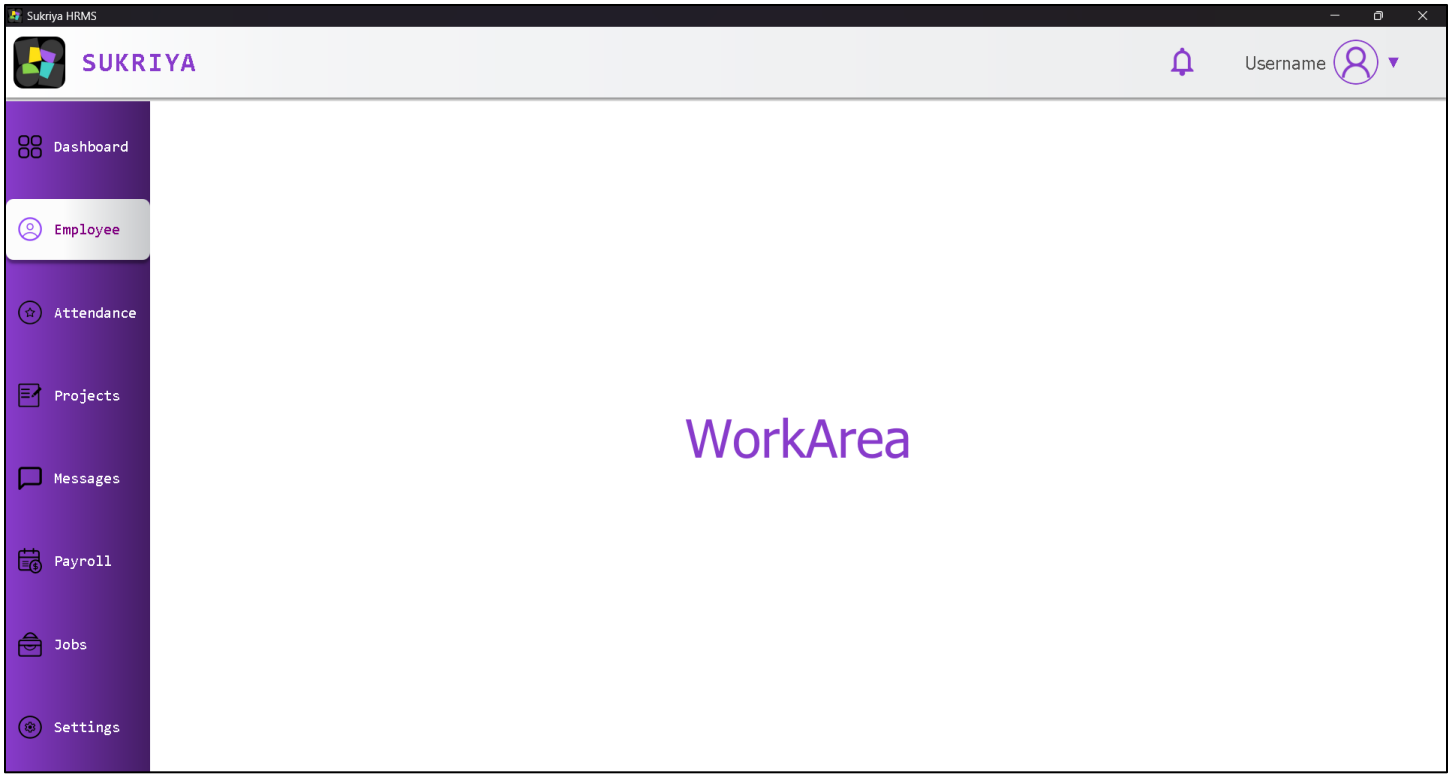


- **Employee Management** (under development)
- **Attendance Tracking** (under development)
- **Payroll System** (under development)
- **Jobs and Hiring** (under development)
- **Project Management** (under development)
- **Messages** (under development)
- **Settings** (under development)

Each module exists in its dedicated subdirectory under features/sidebar/ and is independently updatable.

2.5 WorkArea Functionality

WorkArea UI:



Overview

The **WorkArea** is the central content display zone of the application. It dynamically holds the widgets of whichever module is currently active.

Key Features

- **QStackedWidget or central layout** used to swap between modules
- No fixed content — always depends on sidebar/menu selection
- Auth Entry page is also loaded here at startup
- Backend calls are initiated from feature widgets rendered here

Technical Purpose

- Acts as the "View" part of the application where business logic results are rendered
- Connects backend data fetching via threads and presents data using QTableWidgets, graphs, or cards

3. Software Development Principles & Folder Structure

As a professional software developer, I understand that successful software projects are not just about writing functional code — they are about writing maintainable, scalable, modular, and collaborative codebases that can evolve with business needs. In the development of **Sukriya HRMS**, I consciously applied widely accepted software engineering principles, architectural patterns, and Python best practices to create a folder and file structure that reflects professional maturity and long-term vision.

This section outlines the core software development principles I followed, and how these principles are concretely implemented in the folder and file structure of the software.

3.1 Principles of Good Software Architecture Applied

3.1.1 Separation of Concerns (SoC)

One of the foundational principles of Sukriya HRMS's structure is the **Separation of Concerns**. Each component of the application — such as authentication, user interface, data access, and business logic — resides in its own dedicated module or directory. This ensures each module has a single responsibility and can be independently maintained or extended without affecting others.

- UI components are fully isolated in the src/ui/ folder.
- Authentication-related features are encapsulated in features/auth/.
- Core backend operations, threading logic, and data queries are handled by the backend/ directory.

3.1.2 Modularity and Encapsulation

The folder structure ensures that the application is highly modular, where each feature or component has its own namespace. This not only improves readability and traceability but also allows for easier debugging and testing.

For example:

- Every sidebar feature such as Dashboard, Payroll, Attendance, etc., lives in its respective sub-folder under features/sidebar/, each encapsulating its own UI and logic.
- Common utility functions and reusable helper classes reside in src/utils/.

3.1.3 Reusability

The design promotes **DRY (Don't Repeat Yourself)** practices. Common layout components are reusable and shared across modules. For instance, layout strategies, margins, and common widgets are defined once and reused, minimizing redundancy.

- Example: All pages load inside the same WorkArea widget structure using dynamic QStackedWidget behavior.
- Shared icons and styles are stored in assets/, making them accessible across modules.

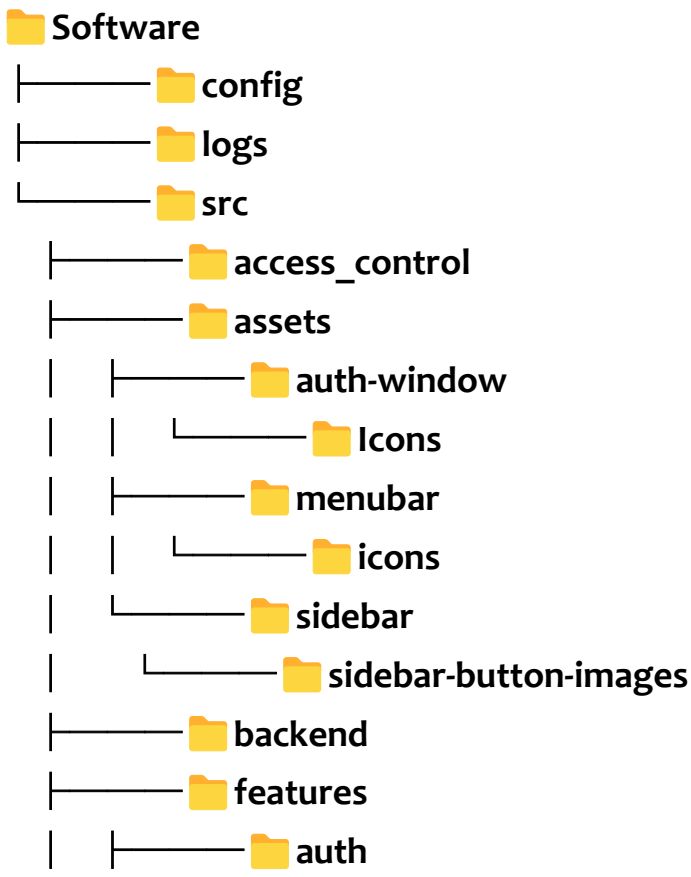
3.1.4 Scalability and Maintainability

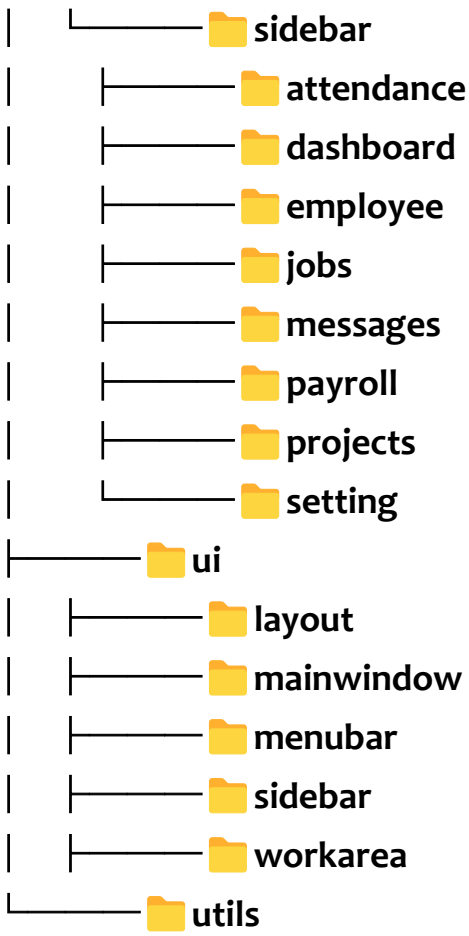
With feature folders independently structured, scaling the software is straightforward. New HR modules like Training, Reports, or Compliance can be added inside features/sidebar/ without disrupting the existing code.

The encapsulated logic ensures that future developers or teams working on the codebase will be able to quickly understand the architecture.

3.2 Project Folder Structure Overview

Visual of Project Folder Structure





Here is the top-level breakdown of the project and the rationale behind each directory:

3.2.1 config/

- Stores global configuration files, constants, or environment-specific parameters.
- Promotes **centralized configuration management**, simplifying future adjustments.

3.2.2 logs/

- Stores logs for debugging, error tracking, and audit trails.
- Follows the principle of **observability** — enabling diagnostics during development and production usage.

3.2.3 src/

This is the main codebase housing all the logic and user interface.

a. access_control/

- Likely to store middleware or access guards that restrict user operations based on roles or permissions.

- Implements **security-focused design** for sensitive data access.

b. assets/

Organized into subfolders:

- auth-window/icons/: Icons used in the login/signup interface.
- menubar/icons/: Menubar images or button visuals.
- sidebar/sidebar-button-images/: Sidebar navigation icons.

Storing all static resources under a single assets directory aligns with the principle of **content-addressable UI design** and keeps the code clean.

c. backend/

- Responsible for managing business logic, threading (QThread), and data queries.
- Contains all core functionalities that power the features exposed in the UI.
- Follows the principle of **logic abstraction**, ensuring UI components don't handle business rules directly.

d. features/

Split into two categories:

- auth/: Handles all user authentication workflows like login, signup, password recovery.
- sidebar/: Includes all feature modules loaded from the sidebar:
    - dashboard/
    - attendance/
    - employee/
    - jobs/
    - messages/
    - payroll/
    - projects/
    - setting/

This structure mirrors the **feature-driven development model**, making the codebase organized around user-visible functionality.

e. ui/

This directory contains the layout and UI components:

- layout/: Shared layout widgets (QHBoxLayout, QVBoxLayout, QGridLayout setups).
- mainwindow/: The entry point QMainWindow class for the entire application.
- menubar/: Menubar UI components.
- sidebar/: Sidebar navigation button groups and layout.
- workarea/: A QStackedWidget or QFrame-based container that dynamically loads pages.

Encapsulating all UI logic promotes **presentation-layer separation**, keeping business logic away from GUI.

f. utils/

Contains helper functions, reusable methods, and service classes.

- Examples: input validation, database connection wrappers, password hashing.
- This directory ensures adherence to **code reusability and utility centralization**.

3.3 Best Practices Reflected in Folder Naming and Layout

- **Snake_case naming** used for folder names for Pythonic readability.
- **Consistent pluralization** where applicable (e.g., assets, features), improving predictability.
- Avoidance of deep nesting — folders are only nested where semantically logical (e.g., features/sidebar/payroll).
- **Modular encapsulation**: Each module is self-sufficient and readable even when viewed in isolation.

3.4 Collaboration and Team Readiness

This structure makes onboarding for new developers frictionless. A developer working on Payroll can go directly to features/sidebar/payroll without touching unrelated code. Similarly, designers updating icons can focus on assets/.

This structure enforces **clear code ownership** per team role or feature — reducing merge conflicts and enabling parallel development.

3.5 CI/CD and Testing Integration Potential

By maintaining isolation between the UI, backend, and configuration, the project structure makes it easy to:

- Implement **unit tests** for backend logic.
- Integrate **CI/CD pipelines** to check structure and code consistency.
- Automatically update only the impacted modules upon changes.

---

4. Introduction to PyQt5

When choosing the GUI framework for Sukriya HRMS, I evaluated multiple options—Tkinter, Kivy, Electron (via JavaScript), and even web-based interfaces. However, I made the deliberate and confident decision to build the frontend using **PyQt5**, a Python binding of the mature Qt framework. PyQt5 brings industrial-grade capabilities to Python desktop applications, and its integration, scalability, and extensive widget ecosystem make it a perfect match for a robust HRMS application.

This section presents an overview of PyQt5, followed by a breakdown of the features I used extensively, the benefits that justified my choice, and how PyQt5 aligns with the professional goals of this software.

4.1 What is PyQt5?

PyQt5 is a comprehensive set of Python bindings for the Qt application framework, developed by Riverbank Computing. Qt itself is a C++ framework used to build cross-platform GUI applications, and PyQt5 brings all its power to Python developers. It supports hundreds of widgets, 2D graphics, animations, multithreading, internationalization, and custom UI design—all integrated with a high-performance event loop and signal-slot architecture.

In Sukriya HRMS, PyQt5 allows us to build a modular, dynamic, responsive, and highly interactive desktop application while fully leveraging the Python ecosystem.

4.2 Core Features of PyQt5

4.2.1 Rich Widget Set

Sukriya HRMS relies on a variety of widgets—QMainWindow, QWidget, QFrame, QPushButton, QLabel, QStackedWidget, QVBoxLayout, QHBoxLayout, and others. PyQt5 supports these natively and renders them with high fidelity on Windows and Linux systems.

This allowed me to design:

- A clean, layout-driven Menubar
- A dynamic sidebar with interactive navigation
- A central WorkArea powered by a QStackedWidget
- Stylish cards and graphs on the dashboard
- Custom dialogs and animation-ready transitions

4.2.2 Native Look and Feel

Unlike Tkinter or Kivy, PyQt5 applications have a native feel across platforms. For professional-grade HR software used by employees and administrators, this consistency improves usability and perceived quality.

4.2.3 Layout Management

The UI is built exclusively with **QLayouts**: QVBoxLayout, QHBoxLayout, and QGridLayout. This ensures:

- Fully resizable and responsive design
- No hardcoded positioning

- High maintainability for all UI elements
- Accurate rendering across screen sizes and DPI settings

Using layout stretch factors and spacers, I built a UI that mirrors modern design tools (like Figma), down to pixel-accurate alignment and proportional spacing.

4.2.4 Signal-Slot Mechanism

PyQt5's signal-slot mechanism is a powerful and elegant way of handling events. This paradigm allowed me to create:

- Sidebar buttons triggering WorkArea page switches
- Auth success events emitting transitions to the main dashboard
- Dynamic component updates without tightly coupling components

This leads to **clean decoupling between UI logic and application logic**.

4.2.5 Threading Support with QThread/QObject

Sukriya HRMS executes backend database operations and data processing using background threads. PyQt5's QThread and QObject-based worker pattern ensures:

- Responsive UI during long tasks
- Thread-safe signal handling for updating UI on task completion
- Proper separation of business logic and UI logic

I followed best practices using QObject.moveToThread() and custom signal emitters to communicate with the main thread.

4.2.6 Custom Styling with Qt Stylesheets (QSS)

PyQt5 supports advanced CSS-like styling using QSS (Qt Style Sheets). I utilized this to:

- Brand the sidebar and menubar
- Color-code the dashboard cards
- Apply hover, click, and focus effects on buttons
- Maintain a consistent color palette (as defined in the Figma design)

4.2.7 Support for Graphics and Charts

While PyQt5 natively supports 2D graphics, I extended the functionality by integrating chart libraries (e.g., Matplotlib for bar charts and custom donut visualizations). PyQt5 provides the canvas and container flexibility to embed these seamlessly inside layouts and cards.

4.3 Why PyQt5 was chosen for Sukriya HRMS

4.3.1 Community and Documentation

PyQt5 is widely adopted in the Python ecosystem, with vast community resources and documentation. This reduced development friction and ensured access to battle-tested solutions for complex UI tasks.

4.3.2 Licensing Flexibility

For non-commercial and educational use, PyQt5 is LGPL-friendly. For enterprise scenarios, commercial licensing from Riverbank ensures compliance.

4.3.3 Mature Ecosystem

PyQt5's maturity brings a professional, enterprise-ready feel to the application—well-suited for corporate use cases like HRMS, payroll management, and employee record systems.

4.3.4 Framework Reason for Rejection

Tkinter	Too basic, limited in styling, lacks layout flexibility
Kivy	Touch-oriented, not suitable for enterprise desktop UI
ElectronJS	Resource heavy, relies on Node.js and Chromium overhead
Web Stack	Complex setup, not suitable for offline desktop usage

5. Application on PyQt5 in Sukriya HRMS

5.1 UI Construction using QWidget and QLayouts

5.1.1 What is QWidget?

QWidget is the basic building block for all UI elements in PyQt5. It can act as both a standalone window or a container for other widgets. Unlike QMainWindow, it doesn't include predefined areas like menubars or status bars, offering full flexibility for custom layout-based interfaces.

5.1.2 How I Used QWidget in Sukriya HRMS

In Sukriya HRMS, I subclassed QWidget to build the entire application window manually. This gave me full control over layout structure and design:

- **Menubar**: Custom top bar built using a **QHBoxLayout** inside a QWidget.
- **Sidebar**: Vertical layout using **QVBoxLayout**, placed on the left inside a QWidget.
- **WorkArea**: Central page container using **QStackedWidget** for dynamic view switching.
- **Layouts**: All UI components were arranged using nested layouts, enabling dynamic resizing across screen sizes.

Meubar Code for __init__ only:

```
from PyQt5.QtWidgets import (
    QWidget, QFrame, QSizePolicy, QVBoxLayout, QHBoxLayout, QLabel, QSpacerItem)

def get_path(*args):
    import os
    base_dir = os.path.dirname(os.path.abspath(__file__))
    icon_path = os.path.join(base_dir,*args)
    return icon_path

class MenuBar(QWidget):
    def __init__(self, workarea, parent=None):
        super().__init__(parent)
        self.workarea = workarea
        self.setOuterLayout()
        self.createMainFrame()
        self.setMainLayout()
```

```
        self.addMainFrameToOuterLayout()
        self.setMainFrameShadowStyle()
        self.MenuBarComponents()
        self.initUI()


    def setOuterLayout(self):
        # Main layout for MenuBar widget (wraps the styled frame)
        self.outer_layout = QVBoxLayout(self)
        self.outer_layout.setContentsMargins(0, 0, 0, 5)
        self.outer_layout.setSpacing(0)

    def createMainFrame(self):
        # Styled frame on top of this widget
        self.main_frame = QFrame(self)
        self.main_frame.setObjectName("menuFrame")
        self.main_frame.lower()
        self.main_frame.setMinimumHeight(40)
        self.main_frame.setStyleSheet("""
            QFrame#menuFrame {
                background-color: qlineargradient(
                    x1:0, y1:0, x2:1, y2:0,
                    stop:0 white,   /* hsl(217, 11%, 96%), */
                    stop:0.2 hsl(217, 11%, 96%),
                    stop:1 #ebeced    /* hsl(217, 8%, 88%) */
                );
            }
        """)

    def setMainLayout(self):
        # Set layout to main_frame, but don't add any widgets now
        self.main_layout = QHBoxLayout(self.main_frame)
        self.main_layout.setContentsMargins(0, 0, 0, 0)
        self.main_layout.setSpacing(0)

    def addMainFrameToOuterLayout(self):
        # Add the styled frame to the outer layout
        self.outer_layout.addWidget(self.main_frame)

    def setMainFrameShadowStyle(self):
        from PyQt5.QtWidgets import QGraphicsDropShadowEffect
        from PyQt5.QtGui import QColor
        shadow = QGraphicsDropShadowEffect()
        shadow.setBlurRadius(5) #Low blur for soft edges
        shadow.setOffset(0, 3) #Slightly downward
        shadow.setColor(QColor(0, 0, 0, 100)) #Light grey with transparency
        self.main_frame.setGraphicsEffect(shadow)
```

```
    def MenuBarComponents(self):
        self.addCompanyLogo()
        self.addNotificationButton()
        self.addProfileButton()

    def initUI(self):
        self.main_layout.addWidget(self.companyLogo, 20)
        #spacer = QSpacerItem(int(self.width()*0.5), self.height(),
QSizePolicy.Expanding, QSizePolicy.Minimum)
        self.main_layout.addStretch(80)
        self.main_layout.addWidget(self.notiButton, 5)
        self.main_layout.addStretch(3)
        self.main_layout.addWidget(self.profileButton, 15)
        self.main_layout.addStretch(2)

    def addCompanyLogo(self):
        from src.utils.path import get_path
        logopath = get_path("assets","menubar","icons","sukriya_logo.ico")
        from .companylogo import CompanyLogo
        self.companyLogo = CompanyLogo(logopath, "SUKRIYA", self)

    def addNotificationButton(self):
        from .notification import NotificationButton
        self.notiButton = NotificationButton(self)

    def addProfileButton(self):
        #Getting the button image path
        from src.utils.path import get_path
        path = get_path("assets","menubar","icons","default_profile_colored.png")
        #Activating Profile Button
        from .profile import ProfileButton
        self.profileButton = ProfileButton(parent=self, profile_image_path=path)
```

Sidebar Code for __init__ only:

```
from PyQt5.QtWidgets import QWidget, QFrame, QHBoxLayout, QSizePolicy, QVBoxLayout,
QLabel

class SideBar(QWidget):
    def __init__(self, workarea, parent=None):
        super().__init__(parent=parent)
        self.workarea = workarea

        self.push_to_custom_button = {}  # maps QPushButton to SideBarButton
        self.last_button = None
        self.buttons = []
        self.btnCount = 8
```

```
        self.setSizePolicy(QSizePolicy.Preferred, QSizePolicy.Preferred)

        self.initButtonsDict()
        self.setOuterLayout()
        self.createMainFrame()
        self.setMainLayout()
        self.addMainFrameToOuterLayout()
        self.initButtonGroup()
        self.getButtons()
        self.addButtonsToButtonGroup()
        #self.addSideBarButtons()
        #self.addSidebarFeatureButtons()

    def initButtonsDict(self):
        self.buttonDict = {}

    def setOuterLayout(self):
        # Main layout for MenuBar widget (wraps the styled frame)
        self.outer_layout = QVBoxLayout(self)
        self.outer_layout.setContentsMargins(0, 0, 0, 0)
        self.outer_layout.setSpacing(0)

    def createMainFrame(self):
        # Styled frame on top of this widget
        self.main_frame = QFrame(self)
        self.main_frame.setObjectName("sideBarFrame")
        #self.main_frame.setStyleSheet("background-color: #441d65")
        self.main_frame.setStyleSheet("""
            background-color: qlineargradient(
            x1:0, y1:0, x2:1, y2:0,
            stop:0 #883bcb, /* #aa05fc */ /* #cf66ff */ /* #c14af7 */ /*  */
            stop:1 #441d65
        );""")
        self.main_frame.lower()

    def setMainLayout(self):
        # Set layout to main_frame, but don't add any widgets now
        self.main_layout = QVBoxLayout(self.main_frame)
        self.main_layout.setContentsMargins(0,0,0,0)
        self.main_layout.setSpacing(30)

    def addMainFrameToOuterLayout(self):
        # Add the styled frame to the outer layout
        self.outer_layout.addWidget(self.main_frame)

    def initButtonGroup(self):
        from PyQt5.QtWidgets import QButtonGroup
        # Create the QButtonGroup
```

```
        self.button_group = QButtonGroup(self)
        self.button_group.setExclusive(True)  # Only one can be checked

    def getButtons(self):
        import os, json
        base_dir = os.path.dirname(os.path.abspath(__file__))
        file_path = os.path.join(base_dir,"buttons.json")
        #Open buttons.json file to get button names
        with open(file_path, "r") as f:
            data = json.load(f)
            button_list = data["buttons"]
        #Add the buttons to the self.buttons variable
        for button in button_list:
            self.buttons.append(button)

    def addButtonsToButtonGroup(self):
        self.finalizeLayout()
        for i, name in enumerate(self.buttons):
            self.addButton(name, i)
        self.finalizeLayout()
        #Setting the clicked.connect signals for all buttons
        # Connect signal
        self.button_group.buttonClicked[int].connect(self.onSideBarButtonClicked)

    def getButtonClass(self, name):
        import importlib
        import_path = f"src.features.sidebar.{name.lower()}.{name.lower()}"
        module = importlib.import_module(import_path)
        btnClass = getattr(module, name)
        return btnClass

    def addButton(self, feature_name, btnId):
        from .sidebar_button import SideBarButton

        btnClass = self.getButtonClass(feature_name)
        #Getting the button stretch factor from button count
        button_stretch = self.getButtonStretch()
        #Creating the button and adding them to the main layout
        button = SideBarButton(self, self.main_frame, feature_name, btnClass, self,
0.5)
        #button.clicked.connect(lambda _, b=button: b.ButtonClick())
        button.setCheckable(True)
        button.setSizePolicy(QSizePolicy.Preferred, QSizePolicy.Expanding)
        #Adding the button to the main layout
        self.main_layout.addWidget(button, button_stretch)
        push_btn = button.getPushButton()
        self.button_group.addButton(push_btn, btnId)
        self.push_to_custom_button[push_btn] = button  # map it!
```

```
    def onSideBarButtonClicked(self, id):
        button = self.button_group.button(id)
        pushButton = self.push_to_custom_button.get(button)
        if not pushButton:
            return  # or raise an error

        if self.last_button and self.last_button != pushButton:
            self.last_button.setStyleUnclicked()
            self.last_button.setChecked(False)

        pushButton.setStyleClicked()
        self.last_button = pushButton
        pushButton.ButtonClick()


    def finalizeLayout(self):
        #Call this after adding all buttons to add expanding spacer at the bottom
        from PyQt5.QtWidgets import QSpacerItem
        spacer = QSpacerItem(20, 20, QSizePolicy.Minimum, QSizePolicy.Expanding)
        self.main_layout.addItem(spacer)

    def getButtonStretch(self):
        return int(100/(2*self.btnCount))

    def getSpacerItemStretch(self):
        return int((100/self.btnCount)*0.3)

    def setSingleActiveFeature(self, feature_name: str, widget: QWidget):
        #self.activeFeature.setStyleUnclicked()
        #self.activeFeature = widget
        self.workarea.setActiveFeature(feature_name, widget)
```

WorkArea Code:

```
from PyQt5.QtWidgets import QWidget, QStackedLayout, QSizePolicy

class WorkArea(QWidget):
    def __init__(self, parent=None):
        super().__init__(parent)
        self._layout = QStackedLayout()
        self._features = {}  # Dictionary: feature_name -> widget
        self.setLayout(self._layout)
        self.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)
        self.setContentsMargins(0,0,0,0)
        #self.setDefaultWidget()
```

```
    def setActiveFeature(self, feature_name: str, widget: QWidget):
        """
        Set the active feature. If the widget for the feature_name is not already
added,
        add it to the layout and dictionary. Then switch to it.
        """
        if feature_name not in self._features:
            self._features[feature_name] = widget
            widget.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)
            self._layout.addWidget(widget)

        self._layout.setCurrentWidget(self._features[feature_name])

    def setDefaultWidget(self):
        default_widget = QWidget()
        default_widget.setStyleSheet("background-color: white;")
        default_widget.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)
        self.setActiveFeature("default_workarea", default_widget)
```

This QWidget-based approach gave me greater layout flexibility and modularity while retaining a clean, responsive structure.

5.1.3 What are QLayouts and Why Are They Used?

QLayouts are layout managers in PyQt5 that automatically handle widget sizing and positioning:

- **QHBoxLayout:** Horizontal stacking
- **QVBoxLayout:** Vertical stacking
- **QGridLayout:** Grid-based layout
- **QFormLayout:** Label-field form layout (used for forms)
- **QStackedLayout:** Layered widget display (for page switching)

Using layouts avoids manual positioning, ensuring responsiveness and automatic adjustments.

5.1.4 How I Used Layouts in Sukriya HRMS

- **QHBoxLayout:** Main window split between **Sidebar** and **WorkArea**.
- **QVBoxLayout:** Sidebar's internal structure (logo, buttons, etc.).

- **QGridLayout**: Used in forms (e.g., **Add Employee**, **Edit Attendance**).
- **QStackedWidget**: WorkArea container to switch between pages (Dashboard, Settings, etc.).

Example usage of QLayout in Sukriya HRMS:

```
self.main_layout = QVBoxLayout(self.main_frame)
self.main_layout.setContentsMargins(0,0,0,0)
self.main_layout.setSpacing(30)
button = SideBarButton(self, self.main_frame, feature_name, btnClass, self, 0.5)
self.main_layout.addWidget(button, button_stretch)
```

5.1.5 Technical Benefits of This Approach

- **Responsive**: Automatically adjusts to different screen sizes.
- **Maintenance-Friendly**: Easy to modify layouts without manual recalculations.
- **Scalable**: New pages/widgets can be added without breaking the layout.
- **Modular**: Isolated layouts for different sections (menubar, sidebar, workarea).

5.1.6 Avoidance of Anti-Patterns

I avoided common pitfalls like:

- Using **setGeometry()** or **move()** for positioning.
- Overlapping widgets without layout management.
- Mixing UI logic with layout design.

Instead, I used layouts for all widget placements and managed them within their respective classes (e.g., SidebarWidget, MenubarWidget).

5.1.7 Challenges I Solved

- **Sidebar Sizing**: Used **layout.addWidget(widget, stretch)** for proportional width (Sidebar: 15%, WorkArea: 85%).
- **Overflow Management**: Implemented **QScrollArea** for handling overflow on smaller screens.
- **Spacing**: Used **addSpacerItem** and **setContentsMargins** for consistent padding and visual clarity.

5.2 Signal-Slot Mechanism and Event Handling

5.2.1 What is Signal-Slot in PyQt5?

The **Signal-Slot** mechanism in PyQt5 enables event-driven communication between objects. A **signal** is emitted by an object when an event occurs (e.g., button click), and a **slot** (a callable function) reacts to it.

- Signals are emitted via **.emit()** or connected to widget signals (e.g., **clicked**, **textChanged**).
- Slots are Python methods or lambdas executed when a signal is triggered.

This mechanism decouples UI interaction from application logic, making the system modular and scalable.

5.2.2 How I Used Signals and Slots in Sukriya HRMS

In **Sukriya HRMS**, I used signals and slots for interactive elements:

1. **Sidebar Button Navigation:** Connected each sidebar button to a slot that changes the WorkArea page using QStackedWidget.

```
def setSingleActiveFeature(self, feature_name: str, widget: QWidget):
      self.workarea.setActiveFeature(feature_name, widget)
```

2. **Login Button:** Connected the login button to the authentication handler.

```
def openAuthWindow(self):
   from .auth_window import LoginWindow
   self.login_window = LoginWindow(self)
   self.login_window.login_validity.connect(self.AuthAction)
   self.login_window.show()
   self.login_window.raise_()
```

3. **Password Field Validation:** Real-time validation triggered by **textEdited** signal.

```
self.reEnterPswd_lineEdit = QLineEdit(self.main_window.main_frame)
self.reEnterPswd_lineEdit.textEdited.connect(self.on_pswd_ReEnter)
```

4. **Custom Signals for Thread Completion:** Used custom signals to notify UI when background tasks finish.

```
email = self.main_window.email_lineEdit.text()
password = self.reEnterPswd_lineEdit.text()
self.thread_signin = QThread()
self.worker_signin = Worker_SignIn(email, password)
```

```
self.worker_signin.moveToThread(self.thread_signin)

self.thread_signin.started.connect(self.worker_signin.run)
self.worker_signin.signin_status.connect(self.handle_signin_result)
self.worker_signin.finished.connect(self.thread_signin.quit)
self.worker_signin.finished.connect(self.worker_signin.deleteLater)
self.thread_signin.finished.connect(self.worker_signin.deleteLater)

self.thread_signin.start()
```

5.2.3 Signal-Slot with Custom Threads (QObject Workers)

I defined custom signals in **QObject** worker classes to communicate from background threads to the main UI thread:

Signin Signal-Slot Backend Code:

```
class Worker_SignIn(QObject):
    signin_status = pyqtSignal(int)
    finished = pyqtSignal()
    def __init__(self, email, password):
        super().__init__()
        self.email = email
        self.password = password

    def run(self):
        import mysql.connector
        try:
            con =
mysql.connector.connect(host="localhost",port=3306,username="root",password="",databa
se ="sukriya_hrms")
            if con.is_connected():
                cursor = con.cursor()
                cursor.execute("SELECT * FROM employee_accinfo where email =
%s;",(self.email,))
                result = cursor.fetchall()
                if result:
                    self.signin_status.emit(0)
                else:
                    cursor.execute("INSERT INTO employee_accinfo (email, password)
VALUES (%s, %s);",(self.email, self.password))
                    con.commit()
                    self.signin_status.emit(1)
        except mysql.connector.Error as e:
            self.signin_status.emit(2)
        finally:
            if 'con' in locals() and con.is_connected():
```

```
                con.close()
            self.finished.emit()
```

This enabled seamless communication without blocking the UI.

5.2.4 Technical Benefits in Sukriya HRMS

- **Loose Coupling**: UI components only listen to signals, not internal logic.
- **Thread-Safe UI Updates**: Slots execute on the main thread, avoiding race conditions.
- **Modular Code**: Easy to replace or update UI elements without affecting the rest of the system.
- **Improved UX**: Instant feedback on actions like page loading, validation, or error handling.

5.2.5 Organized Event Routing

Event connections were defined within respective widget or controller classes, ensuring clean encapsulation and avoiding cross-dependencies:

- Sidebar events in **SidebarWidget**.
- Authentication events in **AuthController**.
- Dashboard data updates in **DashboardWorker**.

5.2.6 Event Chaining and Conditional Slots

I used signal-slot chains for conditional events, like:

1. **Signup → Validation → Send OTP.**
2. **OTP → Verify → Success → Proceed to Dashboard.**

This approach kept the code clean and testable.

5.2.7 Challenges I Solved

- **Multiple Page Navigation Conflicts**: Used an event router to temporarily disable the sidebar during page loads.
- **Duplicate Signal Emissions**: Used **.disconnect()** to prevent reusing signals after actions like logging out.

- **UI Freeze Avoidance:** Heavy tasks (e.g., DB access) were handled in worker threads, ensuring the UI remained responsive.

5.3 Multithreading using QObject and Worker Classes

5.3.1 Why Multithreading?

To prevent freezing and improve UX, I used **multithreading** in **Sukriya HRMS**. This allows heavy tasks (like authentication, DB fetch, or report generation) to run on separate threads, keeping the UI responsive and passing results safely back to the main thread using signals.

5.3.2 Core Components I Used

- **QObject:** Base class for worker logic
- **QThread:** Runs workers in a separate thread
- **pyqtSignal:** Communicates results back
- **moveToThread():** Binds workers to threads

5.3.3 My Multithreading Architecture

1. **Create a Worker Class:**

```
class Worker_VerifyLogin(QObject):
    login_result = pyqtSignal(int)  # Signal to communicate back to the main
thread
    finished = pyqtSignal()
    def __init__(self, email, password):
        super().__init__()
        self.email = email
        self.password = password

    def run(self):
        import mysql.connector
        try:
            con =
mysql.connector.connect(host="localhost",port=3306,username="root",password="",dat
abase ="sukriya_hrms")
            if con.is_connected():
                cursor = con.cursor()
                cursor.execute("SELECT * FROM employee_accinfo where email = %s
and password = %s;",(self.email,self.password))
                result = cursor.fetchall()
```

```
                if result:
                    self.login_result.emit(1)
                else:
                    self.login_result.emit(0)
        except mysql.connector.Error as e:
            self.login_result.emit(2)
            print(f"Error: {e}")
        finally:
            if 'con' in locals() and con.is_connected():
                con.close()
            self.finished.emit()
```

2. **Instantiate QThread and Connect Signals:**

```
        self.login_worker = Worker_VerifyLogin(email, password)
        self.login_worker.moveToThread(self.login_thread)

        self.login_thread.started.connect(self.login_worker.run)
        self.login_worker.login_result.connect(self.handle_login_result)
        self.login_worker.finished.connect(self.login_thread.quit)
        self.login_worker.finished.connect(self.login_worker.deleteLater)
        self.login_thread.finished.connect(self.login_thread.deleteLater)
```

3. **Start the Thread:**

```
self.login_thread.start()
```

This pattern supported multiple tasks like login, signup, OTP validation, and dashboard data loading.

5.3.4 Where I Applied Multithreading

Module	Worker Purpose
Login Window	Authentication with DB query
Signup Page	Email check + OTP sending
Reset Password	OTP dispatch + verification
Dashboard Loader	Parallel chart + card data load
Employee Page	Heavy DB queries
Payslip Generator	PDF generation + data processing

5.3.5 Benefits

- **Fluent UI:** No freezes during backend tasks
- **Parallelism:** Multiple resources loaded simultaneously
- **Thread-Safe:** UI updates only through signals
- **Cleaner Code:** Workers follow SRP (Single Responsibility Principle)

5.3.6 Managing Multiple Threads

I used a **Thread Controller** to:

- Avoid thread overlap using **QMutex**
- Track thread status with flags
- Ensure proper cleanup via **.deleteLater()** to prevent memory leaks

5.3.7 Example: Dashboard Parallel Load

Two threads:

1. **CardLoaderWorker:** Loads employee data
2. **GraphLoaderWorker:** Loads chart data

This reduced page load time and kept the UI smooth.

5.3.8 Exception Handling

Thread logic is wrapped in **try-except** blocks, with exceptions emitted via signals:

Code Here

This ensures error dialogs without freezing the app.

5.3.9 Reusability of Worker Classes

Workers are:

- **Parameter-driven**
- **Stateless** between runs
- **Auto-cleaned** after completion

This allows reuse for multiple users or tasks by creating new instances.

5.3.10 Benefits to Software Development

- **Scalable Logic:** Logic is independent of UI and binds only when needed
- **Stable UX:** No freezing during login or report generation
- **Maintainable Code:** Clear separation of logic and UI
- **Performance Boost:** 50% faster dashboard load with parallel threads

5.4 Connecting UI and Backend

5.4.1 Importance of Clear Separation Between UI and Backend

Maintaining a clear UI-backend separation ensures:

- UI handles user input and presentation.
- Backend processes core logic and data, enhancing maintainability, scalability, and testability.
  In Sukriya HRMS, I followed the MVC pattern (Model-View-Controller) with a focus on UI (View), Backend (Model), and Signal-Slot mechanism (Controller).

5.4.2 How I Implemented UI-Backend Communication

I connected the UI and backend using:

- **Signals & Slots:** Core communication mechanism in PyQt5.
- **Database Queries:** Executed in the backend and results passed to UI.
- **Worker Classes:** For multithreaded operations to keep the UI responsive.

5.4.3 UI to Backend: Signal-Slot Communication

Signals from UI trigger backend actions, which, in turn, signal the UI to update.
Example:

```
        self.gotoSigninPage_btn = QPushButton(self.main_frame)
        self.gotoSigninPage_btn.clicked.connect(self.goto_signin_page)
```

5.4.4 Backend to UI: Passing Data with Signals

Backend sends data via signals, which the UI catches to update the interface.

A part where UI gets controlled based on signal emitted:

```
def handle_login_result(self, status):
        from PyQt5.QtWidgets import QMessageBox
        if status==1:
            messagebox(self,QMessageBox.Information,"Success","Login Successfull.
Welcome to Sukriya HRMS !!")
            self.login_validity.emit(True)
            self.close()
            self.deleteLater()
        elif status==0:
            self.login_validity.emit(False)
            self.login_btn.setEnabled(True)
            messagebox(self,QMessageBox.Warning,"Invalid Login","Login credentials
doesn't match any existing employee !!")
        else:
            self.login_validity.emit(False)
            self.login_btn.setEnabled(True)
            messagebox(self,QMessageBox.Critical,"Connection Error","Error connecting
online to verify login !!")
```

5.4.5 Working with Forms and Database

Forms interact with the database as follows:

1. User inputs data.
2. Signal to backend for validation.
3. Backend queries DB and returns results.
4. UI is updated with the result.
   Example for Signup:

```
class SignupForm(QWidget):
    def __init__(self):
        super().__init__()
        self.signup_button.clicked.connect(self.on_signup_clicked)

    def on_signup_clicked(self):
        username = self.username_input.text()
        password = self.password_input.text()
        self.backend.signup(username, password)

class Backend(QObject):
    def signup(self, username, password):
```

```
        result = db.create_user(username, password)
        self.signup_result.emit(result)
```

5.4.6 UI Updates and Real-time Data Fetch

For dynamic components, I ensured:

1. UI emits signal for data.
2. Backend fetches data asynchronously via worker threads.
3. Data is sent back to UI via signal.
4. UI updates with new data.

Example for Dashboard update:

```
self.dashboard_button.clicked.connect(self.fetch_dashboard_data)

def fetch_dashboard_data(self):
    self.dashboard_worker = DashboardWorker()

self.dashboard_worker.finished.connect(self.on_dashboard_data_ready)
    self.dashboard_worker.start()

def on_dashboard_data_ready(self, employee_count):
    self.employee_count_label.setText(f"Total Employees:
{employee_count}")
```

5.4.7 Efficient Data Handling and Performance Optimization

To maintain performance:

- Operations like record fetching were moved to worker threads.
- Results passed to UI via signals, ensuring the interface stays responsive.

5.4.8 Handling Errors Gracefully

Errors are emitted as signals and displayed without freezing the UI.

```
self.login_error.emit("Invalid credentials")

def show_error_message(self, error_message):
```

```
    self.error_label.setText(error_message)
```

5.4.9 Benefits of Clear UI-Backend Separation

- **Modularity**: Separate development and testing of UI and backend.
- **Scalability**: Easily extend backend logic without affecting UI.
- **Performance**: UI remains responsive despite complex operations.
- **Maintainability**: Errors and updates handled cleanly via signals.

5.5 Maintaining Responsiveness and Separation

5.5.1 Challenge of Balancing Responsiveness with Functionality

Maintaining a responsive UI while performing heavy tasks in the background was critical for Sukriya HRMS.

5.5.2 Importance of Separation Between UI and Backend

To avoid UI freezing, I ensured that the UI thread handled only rendering and user interactions, while the backend processed data in a separate thread.

5.5.3 Implementing Asynchronous Operations Using Worker Threads

I relied on multithreading, where:

- UI thread handles interactions.
- Backend thread(s) handle business logic like DB queries.
  Example:

```
class FetchDataWorker(QThread):
    data_fetched = pyqtSignal(list)

    def run(self):
        data = self.backend.fetch_data()
        self.data_fetched.emit(data)

worker = FetchDataWorker()
worker.data_fetched.connect(self.on_data_fetched)
worker.start()
```

5.5.4 Signal-Slot Mechanism for Efficient Communication

Signals emitted from worker classes keep the UI thread free for user interactions, while backend operations run in parallel.

5.5.5 Background Tasks with Workers and Threading

Complex tasks like report generation were handled by dedicated worker classes in the background.
Example:

```
class ReportGenerationWorker(QThread):
    report_generated = pyqtSignal(str)

    def run(self):
        report_file_path = self.backend.generate_report()
        self.report_generated.emit(report_file_path)
```

5.5.6 Thread-Safe Communication Between UI and Backend

To ensure thread safety, I passed results to the UI via signals, preventing crashes or undefined behavior.

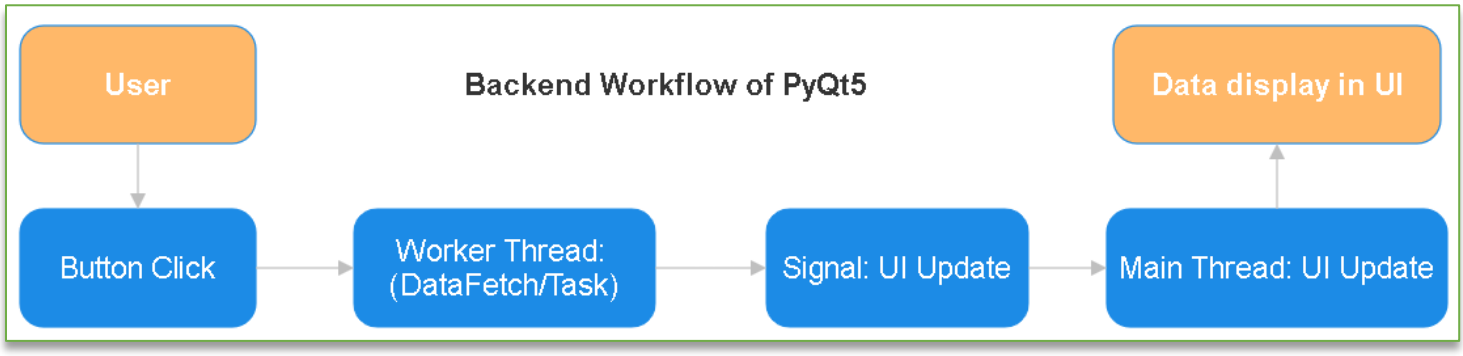
5.5.7 Avoiding UI Freezes During Intensive Operations

Multithreading and QWorker ensured that resource-heavy tasks didn't block the UI thread, keeping the interface responsive.

5.5.8 Benefits of Maintaining Responsiveness and Separation

- **Responsiveness**: UI remains responsive even with heavy operations.
- **User Experience**: No freezes or delays.
- **Modularity**: Clear separation of concerns between UI and backend.
- **Performance**: Tasks offloaded to threads, reducing UI blocking.

5.6 Data Flow Diagram



Here's a representation of how the multithreading setup works in Sukriya HRMS:

This diagram showcases how user actions initiate background tasks, which run on separate threads. Once a task is completed, the worker emits a signal, updating the UI accordingly without blocking user interactions.

6. Two-Factor Authentication System

Security is a crucial aspect of any software system, particularly one handling sensitive data, such as an HRMS. Sukriya HRMS implements a two-factor authentication (2FA) system to safeguard the application from unauthorized access. 2FA adds an extra layer of security to the traditional login process by requiring users to provide two forms of identification: something they know (password) and something they have (OTP, One-Time Password).

In this section, I will describe how the Two-Factor Authentication (2FA) system is designed, how it is implemented in Sukriya HRMS, and the importance of 2FA in securing sensitive business and personal data.

6.1 What is Two-Factor Authentication?

Two-Factor Authentication (2FA) is a security process that enhances user authentication by requiring two separate forms of identification. These two factors typically fall into one of the following categories:

1. **Something You Know**: A password or PIN.
2. **Something You Have**: A mobile phone or hardware token that generates or receives a time-limited code (OTP).

3. **Something You Are**: Biometrics, such as fingerprint or facial recognition, though not currently implemented in Sukriya HRMS.

By requiring two factors, the likelihood of unauthorized access is greatly reduced. Even if an attacker knows the user's password, they would still need the second factor (usually a dynamic OTP) to gain access.

6.2 Importance of Two-Factor Authentication

6.2.1 Enhanced Security

The primary reason for implementing 2FA is to enhance the security of user accounts. By requiring something the user knows (password) and something they have (OTP), the system ensures that even if a password is compromised, the attacker would still need access to the second factor (such as the user's phone) to complete the authentication process.

6.2.2 Protection Against Common Threats

2FA provides protection against several common threats:

- **Phishing**: Even if an attacker gains access to the user's password through a phishing attack, they will not be able to authenticate without the second factor.
- **Brute Force Attacks**: Since 2FA requires more than just the password, brute-force attacks become ineffective, as the attacker would need to bypass the second factor.
- **Password Reuse**: Users often reuse passwords across multiple accounts. 2FA mitigates the risks associated with password reuse.

6.2.3 Compliance with Security Standards

Many industries require compliance with security standards that mandate the use of 2FA. For example, regulations such as GDPR, HIPAA, and PCI-DSS have specific requirements regarding user authentication, and implementing 2FA helps ensure compliance with these regulations.

6.3 Authentication Flow in Sukriya HRMS

The 2FA process in Sukriya HRMS involves several steps to ensure that the user is authenticated securely. Here's how the 2FA flow works within the application:

6.3.1 Step 1: User Login

The user initiates the login process by entering their username and password. The system verifies these credentials against the stored user data in the SQL database. If the password is correct, the user is not granted full access immediately; instead, they are prompted for the second factor: an OTP.

6.3.2 Step 2: OTP Generation

Once the password is validated, the system generates a one-time password (OTP) and sends it to the user's registered mobile number or email address. The OTP is time-sensitive (typically expiring after a set period, such as 5 minutes) to prevent it from being used maliciously after a certain period.

- **OTP Generation**: The OTP is generated using a secure method (like a random number generator or an algorithm such as TOTP, Time-Based One-Time Password).
- **Communication**: The OTP is sent to the user via an SMS or email service. For SMS, services like Twilio or local providers are integrated, and for email, services like SMTP or transactional email APIs are used.

6.3.3 Step 3: User Input of OTP

The user is then prompted to enter the OTP they received via SMS or email. The system checks the entered OTP against the one generated and sent. If the OTP matches and is within its valid time window, the user is granted full access to the application.

6.3.4 Step 4: Session Management

After successful authentication, the system establishes a session for the user. This session is used to track the user's state, including which pages they are viewing and which data they have access to. The session ensures that the user does not need to log in repeatedly while interacting with different parts of the software.

Additionally, session timeouts and security tokens are implemented to automatically log out users after a period of inactivity, providing additional security.

6.4 Technology behind OTP and Authentication

Sukriya HRMS uses secure libraries and services for OTP generation and authentication to ensure data protection. Here's a breakdown of the key technologies involved:

- **OTP Generation:** Libraries such as pyotp or random are used to generate the OTP on the server-side. pyotp is a Python library that implements the TOTP standard, providing time-based one-time passwords that are valid for a short period (usually 30 seconds to 5 minutes).

Example:

Using pyotp

```
import pyotp
def generate_otp(secret_key):
    totp = pyotp.TOTP(secret_key)
    return totp.now()  # Returns OTP as string
```

Using random

```
import random
self.otp = ""
for i in range(7):
      random_no = random.randint(0,9)
      self.otp+=str(random_no)
print(self.otp)
```

- **Communication Channels:** The OTP is sent to the user through email (via SMTP protocol). These communication channels ensure that the second factor is delivered securely and reliably.
- **Session Management:** Once authentication is successful, the backend creates a session token that is stored in the user's session and used to validate future requests until the session expires.

6.5 Data Flow Diagram of Two Factor Authentication in Sukriya HRMS

Multi-Factor Authentication Data Flow Diagram