|  |  |
| --- | --- |
|  |  |

Project Documentation

**Mail Management System**

**Documentation**

Maximilian Müller, D953

Dimitrios Tsigaropoulos, E098

Dr. Lamya Abdullah

Content

[1 Project 1](#_Toc185795223)

[1.1 Features 1](#_Toc185795224)

[1.2 Getting Started 1](#_Toc185795225)

[2 Code-Level Documentation 1](#_Toc185795226)

[2.1 File Structure 1](#_Toc185795227)

# Project

This Project is an assignment for the programming course.

The goal is a command-line based Mail-Management-System.

## Features

User creation, login, logout, folder-management, mail-management, contact-management, mail-writing.

## Getting Started

Download all components and make sure that they are arranged in the file structure shown below. Or clone the git project: <https://github.com/MaxNize/MMSAssignment.git> .

No further resources are needed then Python. Every library used is a standard library of Python. Please use Python 3.10.4 or above.

To run the program, run the ‘App.py’ file.

Your command line will be the in- and output panel.

# Code-Level Documentation

## File Structure

Src/

├─ .vscode

│ └─settings.json

├─Abstarcts

│ ├─ Programm.py

│ └─ System.py

├─ db

│ ├─ payslips

│ ├─ ├─ Payslip-01.01.2025-James Doe.csv

│ ├─ ├─ Payslip-01.01.2025-John Doe.csv

│ ├─ └─ Payslip-01.01.2025-Kamala Doe.csv

│ ├─ data.db

│ └─db.py

├─ Documentation

│  ├─ Documentation.docx

│  ├─ README.md

│  └─ToDo.txt

├─ Models

│  ├─ Contact.py

│  ├─ Employee.py

│  ├─ Folder.py

│  ├─ HoursAndSales.py

│  ├─ Mail.py

│  └─ User.py

├─ Systems

│  ├─ ContactManagementSystem.py

│  ├─ EmployeeInformationSystem.py

│  ├─ FolderManagementSystem.py

│  ├─ MailManagementSystem.py

│  ├─ MenuSystem.py

│  ├─ SearchManagementSystem.py

│  └─ UserManagementSystem.py

├─ Utils

│  ├─ FileHandling.py

│  └─ logging.py

├─ \_\_init\_\_.py

└─ App.py

## Usage of files

### .vscode

#### settings.json

The settings.json file configures database connections for the SQLTools extension within the development environment. It includes configurations for two SQLite connections. The first connection, named "data," uses the SQLite driver and accesses the database located at :”C:\Users\mamueller\Documents\Projekte\Priv\NextJS\test\db/sqlite.db”, with a query preview limit set to 50 rows. The second connection, named "mms," also employs the SQLite driver and references a database located at: “${workspaceFolder:MMSAssignment}/db/data.db”, similarly restricting query previews to 50 rows. These configurations enable SQLTools to manage interactions with the specified SQLite databases effectively.

### Abstarcts

#### Programm.py

The programm.py file defines the Programm class, which serves as the central manager for coordinating various systems within the application. It imports multiple classes from Models and Systems to facilitate structured interactions. These systems include UserManagementSystem, MailManagementSystem, FolderManagementSystem, ContactManagementSystem, SearchManagementSystem, EmployeeInformationSystem, and MenuSystem, along with associated models like Contact, Employee, and HoursAndSales.

Upon initialization, the Programm class sets up instances of the mentioned systems and links them together to ensure inter-system collaboration, such as assigning FolderManager to both MailManager and UserManager, and establishing connections between UserManager and ContactManager. The setup methods—setupUsers, setupMails, setupContacts, setupEmployees, and setupHoursAndSales—initialize the application state from the database, loading users, emails, contacts, employees, and sales data.

The mainloop() method activates the MenuSystem's mainloop, which manages user interactions, while the save() method commits changes back to the database, including users, mails, contacts, employees, and sales records.

#### System.py

The system.py file defines the abstract System class, which provides foundational functionality and patterns to be inherited by specific system classes within the application. It imports the `re` module to enable Regular Expression operations and utilizes the `abc` module to create abstract methods.

Upon initialization, the System class sets up common variables used across derived classes, such as text dividers, spacers, and regex patterns for email and date validation. It includes a variety of system-wide variables, including flags like `running` and `active`, as well as user interaction prompts such as `TEXTenterUserName` and `TEXTenterYourPw`.

System-wide methods like `safeQuestion()` facilitate safe input handling across several expected value types, ensuring consistent data entry and reducing crash risks. These types include `int`, `date`, `mail`, `hourType`, `float`, `decimalPercentage`, and confirmation checks.

Additional methods include `TEXTheading()` for formatted heading prints, `checkForMailpattern()` for validating email formats using regex, and the abstract `initing()` method intended for system-specific initialization processes.

The `baseQuestionnaire()` method initializes the questionnaire sequence, presenting user prompts for subsequent actions, while the abstract `specificQuestionnaire()` method is designed for system-specific question handling. The `mainloop()` function continuously runs the questionnaire until an exit condition is met, facilitating interactive system operations.

### db

#### payslips

##### Payslip-01.01.2025-James Doe.csv

The Payslip-01.01.2025-James Doe.csv file contains a detailed record of earnings for James Doe on January 1st, 2025. This CSV file includes the following data columns: `date`, `hours worked`, `sales`, `base salary`, `commission rate`, and `daily earning`. The entry for January 1st, 2025, reports 8 hours worked, with no sales recorded, a base salary of €30.0 per hour, and a commission rate of 0.09. The calculated daily earning, based solely on hours worked and base salary, totals €240.0 for the day. The file concludes with a `Total Overall Earnings` row, summing the earnings to €240.0, reflecting the complete earnings for the specified date.

##### Payslip-01.01.2025-John Doe.csv

The Payslip-01.01.2025-John Doe.csv file provides a record of earnings for John Doe on January 1st, 2025. This CSV includes columns for `date`, `hours worked`, `sales`, `base salary`, `commission rate`, and `daily earning`. On this day, John Doe worked 8 hours, achieved sales totaling €50,000.0, with a base salary of €12.0 per hour, and a commission rate of 0.005. The daily earning is calculated to be €346.0, taking into account both the base salary and commission from sales. The file concludes with a `Total Overall Earnings` row reflecting total earnings of €346.0 for the day.

##### Payslip-01.01.2025-Kamala Doe.csv

The Payslip-01.01.2025-Kamala Doe.csv file contains a record of earnings for Kamala Doe on January 1st, 2025. The CSV format includes columns for `date`, `hours worked`, `sales`, `base salary`, `commission rate`, and `daily earning`. On this date, Kamala Doe did not work any hours but achieved sales totaling €12,000.0. With a base salary of €30.0 per hour and a commission rate of 0.005, the calculated daily earning from commission is €60.0. The file finishes with a `Total Overall Earnings` row, summing up the earnings to €60.0 for the specified day.

#### data.db

#### db.py

The db.py file defines the structure and operations for managing a SQLite database used in the application. It imports sqlite3 to interact with the database and includes functions for initializing and manipulating several tables.

The `initing()` function establishes the database structure, creating tables such as `users`, `mails`, `contacts`, `employeeInfo`, `salesAndHours`, and `log` if they don't already exist.

Functions such as `getUsers()`, `setUsers(data)`, `deleteUsers()`, and similar ones for mails, contacts, employees, and hours and sales manage CRUD operations for their respective tables. Each `setXXX()` function uses executemany to efficiently insert multiple records, while `getXXX()` functions retrieve records, and `deleteXXX()` functions clear table contents.

Special query functions like `getMailsBySenderString(senderString, userName)`, `getMailsBySubjectString(subjectString, userName)`, and `getMailsByAttachmentString(contentString, userName)` enable retrieval of mail records based on specific criteria.

Logging actions are handled by the `logAction(action)` function, which inserts actions into the `log` table with a timestamp. `getLastEmployeeID()` helps fetch the ID of the last employee record, useful for managing new entries.

The file also includes `setupDBForTesting()`, which resets and populates the database with test data, facilitating development and testing processes.

### Documentation

#### Documentation.docx

#### README.md

#### ToDo.txt

### Models

#### Contact.py

The Contact.py file defines the Contact class, which encapsulates attributes related to an individual contact in the application. The class is initialized with six primary attributes: `name`, `firstName`, `lastName`, `mail`, `userName`, and `phone`. These attributes store essential information for each contact, supporting functionalities like communication and record-keeping within the application. The class provides a simple structure for creating contact objects, facilitating storage and retrieval of contact details associated with a user.

#### Employee.py

The Employee.py file defines the Employee class, capturing essential details about an employee within the application. The class is initialized with attributes including `id`, `name`, `birthdate`, `role`, `mail`, `hoursType`, `baseSalary`, and `comissionRate`. These attributes hold critical information such as identification, contact details, employment role, salary structure, and commission rate, enabling functionalities related to employee management and payroll processing. The `\_\_str\_\_` method provides a string representation of the Employee object, formatting its attributes neatly for display or logging purposes.

#### Folder.py

The Folder.py file defines the Folder class, designed to manage a collection of emails within named folders in the application. The class imports `Mail` from Models.Mail to facilitate the creation and manipulation of mail objects. Upon initialization, a Folder instance is characterized by its `name` attribute and a `mails` list, which stores Mail objects.

Key functions within the class include:

* `createMail(subject, to, sender, bcc, cc, content, attachmentsPath, timestamp)`: This function constructs a new Mail object with specified attributes and adds it to the `mails` list.
* `deleteMail(mailIndex)`: Accepts an index to remove the mail at the specified position in the `mails` list, provided that the index is valid.
* `getMailBySubject(subject)`: Searches through the `mails` list to find a mail matching the given subject, returning the Mail object if found, or `None` if no match is found.

The Folder class provides organized email management, allowing for easy addition, deletion, and retrieval of mail objects based on their subject.

#### HoursAndSales.py

The HoursAndSales.py file defines the HoursAndSales class, which encapsulates data related to an employee's working hours and sales activities on a specific date. Upon initialization, the class takes the following attributes: `id`, `employeeId`, `date`, `sales`, and `hoursWorked`. These attributes allow for tracking and analyzing an employee's productivity and sales achievements over time.

The `\_\_str\_\_` method offers a human-readable representation of a HoursAndSales object, displaying the `id`, associated `employeeId`, `date` of record, as well as the `sales` and `hoursWorked`. This string format is useful for logging, reporting, or visualizing the data in a user-friendly manner.

#### Mail.py

The Mail.py file defines the Mail class, which models an email with its relevant attributes. The class is initialized with several attributes: `subject`, `to`, `sender`, `bcc`, `cc`, `content`, `attachmentsPath`, and `timestamp`. These attributes represent key components of an email, including addressing information, message content, attachment paths, and the time of creation.

The `\_\_str\_\_` method provides a user-friendly string representation of the Mail object, detailing its main attributes in a structured format. This allows for convenient display or logging of email details, useful for debugging, visualization, or record-keeping within systems that manage email communications.

#### User.py

The User.py file defines the User class, encapsulating user-related attributes and functionalities within the application. It imports `Folder` from Models.Folder to manage the folders and mails associated with the user. When a User object is initialized, it takes `userName`, `firstName`, `lastName`, `mail`, `pw`, and `default` as attributes. The password attribute `pw` is privately stored as `\_\_pw`. Based on the `default` boolean, the user is assigned either default folders ("Inbox", "Sent", "Trash") or an empty list of folders.

The class includes several key functions:

* `sendMail(subject, to, sender, bcc, cc, content, attachmentsPath, timestamp)`: Facilitates sending emails by creating a mail object in the user's outbox folder.

Getter and checking methods provide structured access and manipulation of folder and password data:

* `getFolders()`: Returns a list of folder names associated with the user.
* `getFoldersForSave()`: Prepares folder names as a comma-separated string for saving.
* `checkForExistingFolder(folderName)`: Checks if a folder exists within the user's folders.
* `getFolder(folderName)`: Retrieves a specific folder object based on the name.
* `getMailsOfFolder(folderName)`: Returns all mails within the specified folder, although the implementation retrieves mails from the first folder and needs adjustment for folder-specific retrieval.
* `checkPw(pw)`: Checks the provided password against the stored password.
* `getPw()`: Returns the stored password.

### Systems

#### ContactManagementSystem.py

The ContactManagementSystem.py file implements the ContactManagementSystem class, which handles all functionalities related to contact management within the application. It extends the abstract System class from Abstracts.System and imports the Contact model and datetime for timestamping operations. Upon initialization, specific system variables such as `title` set to "Contacts" and base question prompts are defined.

The class includes various functions for managing contacts:

* `addNewContactQ()`: Prompts the user to add a new contact, ensuring unique names and valid email formats before adding the contact to the user's contact list.
* `deleteContactQ(name)`: Handles user confirmation for contact deletion and invokes `deleteContact(name)` to perform the removal.
* `editContactQ(name)`: Guides users through editing contact details, ensuring data validation before updating.
* `sendMailToContact(mail)`: Facilitates composing and sending an email to a specific contact, interacting with the user's mail system.

Additional helper methods include:

* `checkForExistingContactUsername(name)`: Checks if a contact name already exists within the user's contacts.
* `getContact(name)`: Retrieves a contact object by its name.
* `editContact(name, newName, newFirstName, newLastName, newPhoneNumber, newMail)`: Updates specified contact details.

The `updateBaseQuestion()` method adjusts the base question menu based on active contacts, while `openContact(contactIndex)` provides detailed interactions options for a selected contact, including delete, edit, or send mail operations.

System-specific operations are coordinated through `updateBasedOnActivity()` and `initing()` methods to synchronize user choices with contact data. The `specificQuestionnaire(answer)` method handles menu interactions, executing appropriate functions based on the user's choices.

#### EmployeeInformationSystem.py

The EmployeeInformationSystem.py file defines the EmployeeInformationSystem class, extending the System class to manage employee records and their associated hours and sales data. It imports models like Employee and HoursAndSales, utilizes logging from Utils.logging, and FileHandling for payslip operations. Upon initialization, the system-specific variables such as `title`, `baseQuestion`, lists for `employees` and `hoursSales`, and `lastId` are set up, along with a date format.

Core functionalities involve:

* Managing Employees:
  + `addNewEmployeeQ()`: Adds a new employee with attributes like name, birthdate, mail, role, hours type, base salary, and commission rate, updating the employee list.
  + `printEmployeeQ()`: Displays current employee information.
  + `editEmployee()`: Allows editing of employee details by ID.
  + `deleteEmployee()`: Marks an employee's details as `None` upon confirmation, logically deleting them from the list.
* Handling Hours and Sales:
  + `addHoursSales()`: Appends new hours and sales records for a specified employee ID.
  + `editHoursSales()`: Enables editing of hours and sales entries.
  + `printHoursSales()`: Lists all recorded hours and sales.
* Sending Payslips:
  + `sendPayslipQ()`: Collects data within a specified date range, organizes it by employee, and sends payslips.
  + `sendPayslip(data, employee)`: Creates payslip files and initiates mailing.
  + `mailPayslip(file, employee)`: Logs into the accounting user, sends payslips via email using the system's mail feature, then logs out.

The class executes different operations based on user input via `specificQuestionnaire(answer)`, ensuring efficient and structured handling of employee-related tasks. This comprehensive functionality supports employee management and payroll operations within the application.

#### FolderManagementSystem.py

The FolderManagementSystem.py file implements the FolderManagementSystem class, responsible for managing email folders within the application. This class inherits from the abstract System class and focuses on folder-specific operations, such as updating folder views, deleting folders, and activating mail managers.

Upon initialization, FolderManagementSystem sets system-specific variables like `title`, which indicates the currently active folder, and `baseQuestion`, providing options for user actions when interacting with folders. The `MailManager` and `UserManager` attributes facilitate interaction with other managers.

Key functionalities include:

* Folder Management:
  + `updateTitle()`: Updates the title based on whether a folder is active, reflecting the folder name.
  + `updateBaseQuestion()`: Changes the base question prompt to include options related to folder deletion and mail overview.
  + `deleteFolderQ()`: Prompts the user to confirm folder deletion, calling `deleteFolder()` upon confirmation.
  + `deleteFolder()`: Removes the active folder, adjusts settings if needed (ensuring default folders like inbox, outbox, or trash point to remaining folders), and requires at least one folder to exist.
* Interaction with MailManager:
  + `activateMailManager(mail)`: Sets active mail in the MailManager and initiates its main loop for mail-specific actions.

The `updateBasedOnActivity()` method synchronizes changes in active folder data to its title and options, while `initing()` and `specificQuestionnaire(answer)` methods handle system initialization and user-driven actions, respectively.

#### MailManagementSystem.py

The MailManagementSystem.py file defines the MailManagementSystem class, inheriting from the System abstract class. It facilitates the management of mails a user has received, providing functionalities to delete, move, answer, or forward emails.

Upon initialization, the class sets the `title` to "MAIL" and defines `baseQuestion` with options for mail-related actions. It incorporates flags for `FolderManager` and `UserManager`, enabling integration with folder and user management systems.

Key functionalities include:

* Mail Deletion:
  + `deleteMailQ()`: Prompts user confirmation for mail deletion, executing `deleteMail()` if confirmed.
  + `deleteMail()`: Removes the active mail from the current folder and stops its operations.
* Mail Movement:
  + `moveMailQ()`: Guides the user to move an email to another folder, calling `moveMail(destination)` to perform the move if the destination is valid and exists.
* Mail Response:
  + `answerMailQ()`: Handles composing and sending responses to a received email, appending original content and attachments.
  + `answerMail(content, attachmentsPath, time)`: Constructs and dispatches response email with appropriate headers and content.
* Mail Forwarding:
  + `forwardQ()`: Prompts user to enter the receiver, constructs, and sends the forwarded message.
  + `forwardMail(to, msg)`: Prepares and sends an email with forwarded content using established email attributes.

Supportive methods like `updateBasedOnActivity()`, `updadteBaseQuestion()`, and `updateTitle()` ensure UI consistency and state synchronization, reflecting current mail and folder operations. `initing()` and `specificQuestionnaire(answer)` facilitate initialization and handle user-selected actions.

#### MenuSystem.py

The MenuSystem.py file defines the MenuSystem class, serving as a navigational interface within the application. This class inherits from the System abstract class, providing users with a central menu to access different systems, specifically the Employee Information System and the Mail Management System.

In the `\_\_init\_\_` method, the class sets system-specific variables such as `title`, positioned as "SYSTEM MENU", and `baseQuestion`, offering two main choices for system entry: 1) Enter Employee System, and 2) Enter Mail System. It also receives instances of `mailManager` and `employeeManager` to handle specific operational transitions.

The `specificQuestionnaire(answer)` method processes user selections via a match-case statement:

* Selecting option 1 triggers the `mainloop()` method of `employeeManager`, transferring control to the Employee Information System.
* Selecting option 2 activates the `mainloop()` method of `mailManager`, allowing users to manage their mail communications.

#### SearchManagementSystem.py

The SearchManagementSystem.py file defines the SearchManagementSystem class, which provides functionalities for users to search received mails within the application. This class inherits from the System abstract class and works alongside the database module for querying mails based on specific criteria.

The `\_\_init\_\_` method sets system-specific variables such as `title`, initialized as "SEARCH", and `baseQuestion`, which offers users options to search mails by sender, subject, or attachment. The class maintains interaction flags with MailManager and UserManager systems for active mail operations.

Core functionalities for searching mails include:

* Search by Sender:
* `searchBySender(senderString)`: Queries the database to find mails matching the specified sender string using `getMailsBySenderString`. It then presents these matches, allowing the user to open a selected email within the MailManager's main loop.
* Search by Subject:
* `searchBySubject(subjectString)`: Behaves similarly, using `getMailsBySubjectString` to locate mails by subject. It also guides users in selecting and opening specific mail results.
* Search by Attachment:
* `searchByAttachment(attachmentString)`: Utilizes `getMailsByAttachmentString` for fetching mails associated with attachments. It facilitates user interactions to navigate results and open desired mails.

The `specificQuestionnaire(answer)` method executes the appropriate search function based on user input and handles string inputs required in each search type.

#### UserManagementSystem.py

The UserManagementSystem.py file defines the UserManagementSystem class, responsible for managing user accounts within the application. This class extends the System abstract class and interfaces with models such as User and Folder. It sets up system-specific variables like `title` ("MENU : No User logged in") and `baseQuestion`, providing options for user creation, login, and deletion.

The main functionalities include:

* User Creation and Deletion:
* `createUserQ()`: Prompts the user for details to create a new account, ensuring uniqueness of usernames and validity of email format.
* `deleteUserQ()`: Handles user account deletion with confirmation and password verification, aiming to permanently remove user data.
* Login Management:
* `loginQ()`: Facilitates user login by checking username existence and matching password credentials, setting the user active upon success.
* Folder and Mail Operations:
  + `writeMail()`: Initiates the email composition process for logged-in users.
  + `openFolder()`: Allows users to open existing folders or create new ones for mail organization.
  + `setupFoldersQ()`: Configures inbox, outbox, and trash folder settings after validating existence.
* Search and Contact Management:
  + `searchMails()`: Executes the SearchManager's main loop for mail search operations.
  + `contacts()`: Activates the ContactsManager to manage user contacts.

Supportive methods like `initing()`, `updateBasedOnActivity()`, `updateTitle()`, and `updateBaseQuestion()` maintain UI consistency and functionality synchronization for both logged-in and logged-out states.

The `specificQuestionnaire(answer)` method uses match-case statements to route user actions to corresponding functions, differentiating tasks based on user login status.

### Utils

#### FileHandling.py

The FileHandling.py file provides functionality for generating payslips in CSV format based on employee performance data. The primary function, `createPayslip(data, employee)`, handles the computation and output of daily earnings for each day of recorded activity for an employee, accounting for hours worked and sales commissions.

Main functionalities include:

* Calculation of Earnings:
  + The function iterates over the `data`, computing daily earnings by combining hourly wages and sales commissions, storing these results in the `daily\_data` list.
* CSV Generation:
  + It constructs a CSV file named with the format "Payslip-[date]-[employee.name].csv", saving it to the "db/payslips" directory.
  + Daily earnings records are written as rows, with a header row defining the columns: `date`, `hours worked`, `sales`, `base salary`, `commission rate`, and `daily earning`.
* Appending Total Earnings:
  + After listing daily records, the total overall earnings are appended as a separate entry to the CSV, showing the sum of all daily earnings.

The function returns the complete path to the generated payslip file, facilitating further operations like email attachment or record keeping.

#### logging.py

The logging.py file defines a logging utility designed to record application actions into the database. It includes a single function, `logger(func)`, which acts as a decorator to wrap other functions.

Key functionalities include:

* Decorator Functionality:
  + The `logger(func)` decorator intercepts the execution of functions passed to it, allowing actions to be logged.
  + Within the `wrapper(\*args, \*\*kwargs)`, the original function is invoked with its arguments, capturing the return result.
  + The `db.logAction(result)` call logs the returned action description or outcome to the database, enabling audit trails or action history maintenance.

This logging utility provides a straightforward mechanism to integrate logging capabilities throughout the application, ensuring that user or system activities are consistently recorded for monitoring, debugging, or compliance purposes.

### \_\_init\_\_.py

The `\_\_init\_\_.py` file marks the directory as a Python package, enabling imports of modules within the directory using package syntax. While empty, it serves primarily for backward compatibility with older Python versions where its presence was required to recognize a directory as a package. In cases where content is added, it can execute package initialization code and control module visibility within the package.

### App.py

The app.py file serves as the entry point for launching the application. It imports the `Programm` class from Abstracts.Programm, initializes an instance of this class, and invokes its `mainloop()` method to start the application's main event loop, facilitating user interactions and system operations.