Project Report and Technical Road Map

Team 8 - Ebook Perpetual Access Tracking

CS-4820 Software System Project

Submitted by:

Beakal Begashaw

Cian Bottomley-Mason

Dilraj Singh Gill

Ethan Penney

Isaac Wolters

Jasleen Kaur

ABSTRACT

The ebook perpetual access tracking application will assist librarians with the tracking of perpetual access rights for ebooks available for their library or institution. Libraries have a collection of ebooks from various publishers that they purchase the perpetual access rights to. Librarians track these access rights, as occasionally, access rights may be incorrectly revoked by the publishers or simply to keep a log of all of the ebooks they have access to. However, there is no standardized way to do this. The Canadian Research Knowledge Network (CRKN) maintains a collection of spreadsheets that contain the ebooks that all Canadian institutions have access to from those publishers. Individual institutions can also purchase ebooks, so they also maintain their own collection of spreadsheets. As a result, searching for a particular ebook access rights is inconvenient and takes place across a number of spreadsheets in various locations. The ebook perpetual access tracking application will provide a centralized location for librarians to verify whether or not they have the rights to access a particular ebook or collection of ebooks at their particular library or institution.

ROAD MAP

- **resources**

    1. *Icons*

- **src**

    1. *data_processing*
        - database.py
        - scraping.py

    2. *user_interface*
        - manageDatabase.py
        - manageInstitutions.py
        - scraping_ui.py
        - searchDisplay.py
        - settingsPage.py
        - startScreen.py
        - welcomeScreen.py

    3. *utility*
        - ebook_database.db
        - export.py
        - logger.py
        - message_boxes.py
        - settings.json
        - settings_manager.py
        - upload.py
        - utils.py

- **testing**

    1. *data_processing_test*
        - database_test.py
        - scraping_test.py

    2. *utility_testing*
        - settings_manager_test.py

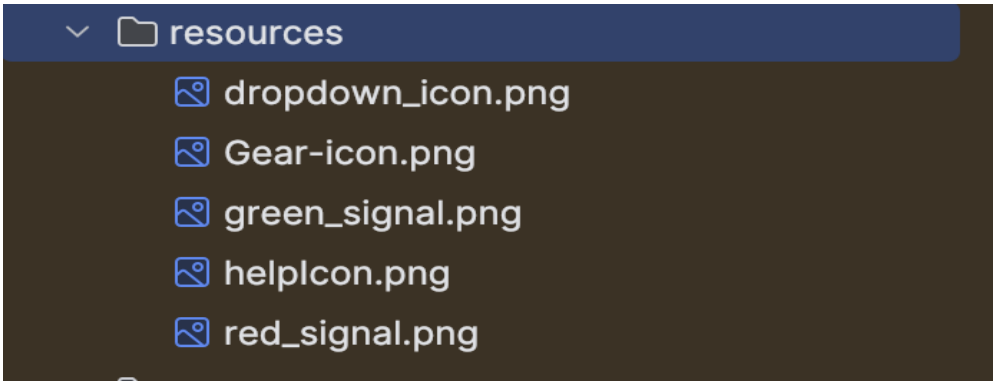
- **main**

Technical Details

- **resources**

    1. *Icons*

This project file in the project tool window is the “resources” file, and it contains the symbols used throughout the application's UI.



- **src**

    1. *data_processing*

- database.py

This part of the project file works and interacts with the SQLite database.

Database Connection Functions:

- connect_to_database(): Establishes a connection to a local SQLite database.

- close_database(connection): Closes the connection to the database.

Table Retrieval Functions:

- get_CRKN_tables(connection): Retrieves a list of CRKN table names from the database.
- get_local_tables(connection): Retrieves a list of local table names from the database.
- get_tables(connection): Retrieves the names of all tables by combining CRKN and local table names based on settings.

Database Table Creation:

- create_file_name_tables(connection): Creates default database tables (CRKN_file_names and local_file_names) if they do not exist.

Database Searching Functionality:

- search_database(connection, query, terms, searchTypes): Performs a search across all tables in the database based on the provided search terms and types.

```python
    # Searches for matching items through each table one by one and adds any matches to the list
    for table in list_of_tables:
        # Get institutions from each table
        institutions = cursor.execute(f'select * from [{table}]')
        institutions = [description[0] for description in institutions.description[8:-2]]

        # Only search table if it has the institution
        if settings_manager.get_setting("institution") in institutions:
            formatted_query = query.replace("table_name", f"[{table}]")
            # executes the final fully-formatted query
            cursor.execute(formatted_query, terms)

            results.extend(cursor.fetchall())
    return results
```

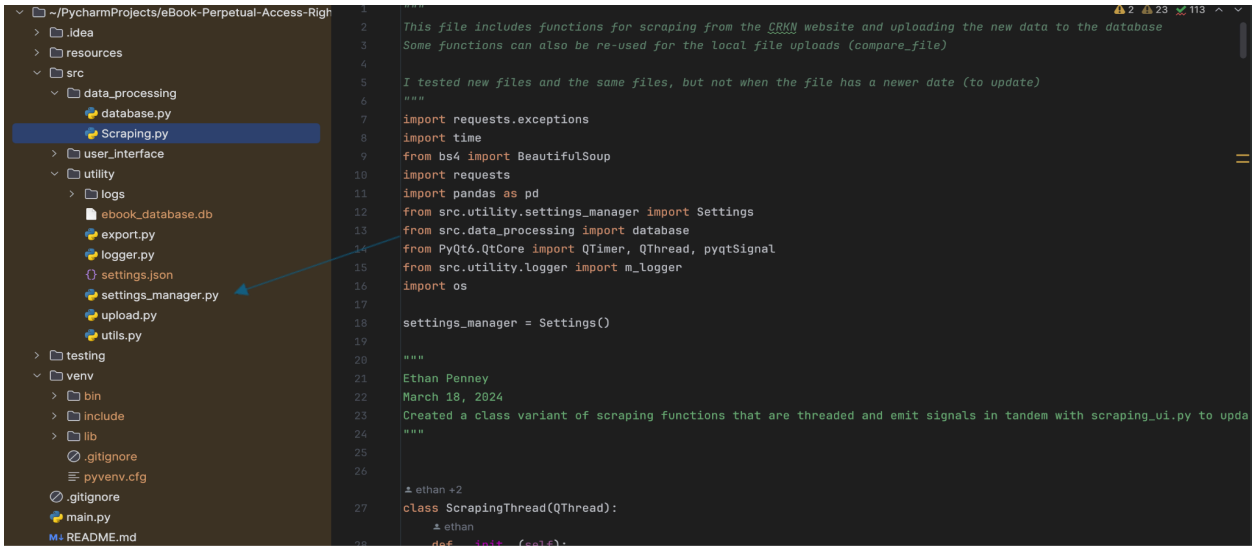
The script is modular and configurable, allowing for easy integration with different databases and setups. It also includes logging capabilities to track the execution flow and potential errors.

- scraping.py

This part of the project file deals with scraping the CRKN website, processing it and then uploading it to the local database.

Imports and Initial Setup:

- Various modules such as requests, pandas, BeautifulSoup, and PyQt6-related components are imported.
- Settings are accessed from a settings manager.

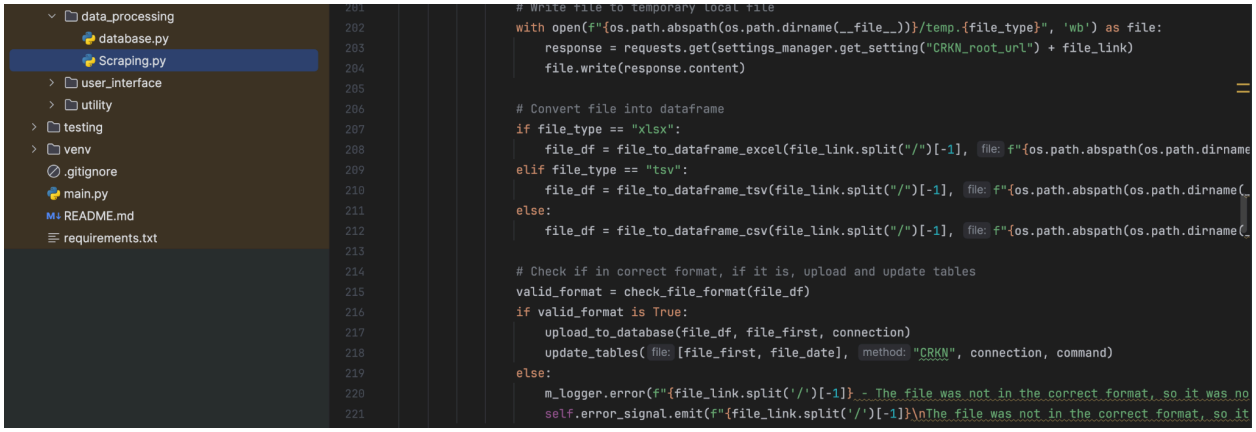


ScrapingThread Class:

- A custom class, ScrapingThread is defined and inherited from QThread, which allows for asynchronous scraping.
- It defines methods to run the scraping process (scrapeCRKN), handle retries, and emit signals for UI updates.
- Methods are provided to handle scraping errors and retries (retry_scrape) and to wait for user responses (wait_for_response).

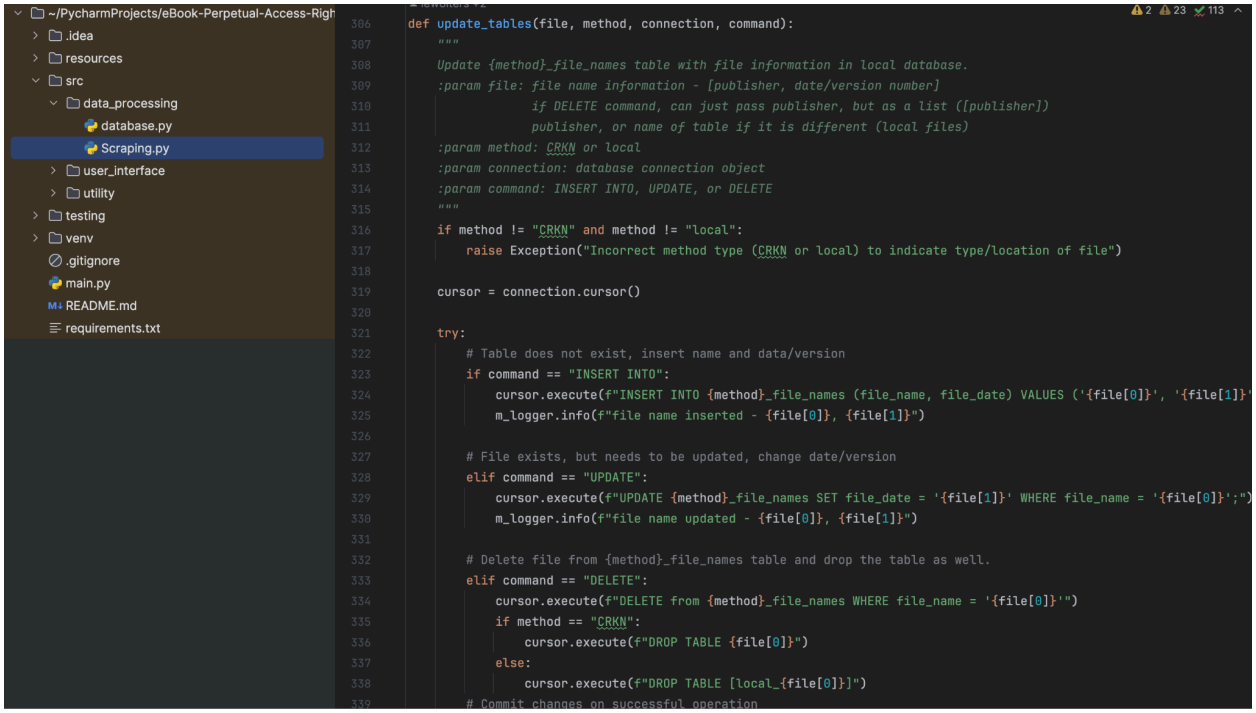
Scraping CRKN Website:

- The scrapeCRKN method is the core scraping function.
- It attempts to scrape the CRKN website multiple times, handling different types of exceptions.
- Upon successful scraping, it retrieves links to files (.xlsx, .csv, .tsv) from the website.
- It then compares these files with existing files in the local database to determine updates.
- If updates are found, it prompts the user to confirm before downloading and updating.



File Processing and Database Updates:

- Once updates are confirmed, files are downloaded and processed.
- File formats (.xlsx, .csv, .tsv) are converted to pandas data frames for manipulation.
- File formats are checked for correctness using check_file_format.
- Dataframes are uploaded to the database, and tables are updated accordingly.
- The method upload_to_database handles uploading data to the database.
- File and table names are formatted and updated accordingly.



Helper Functions:

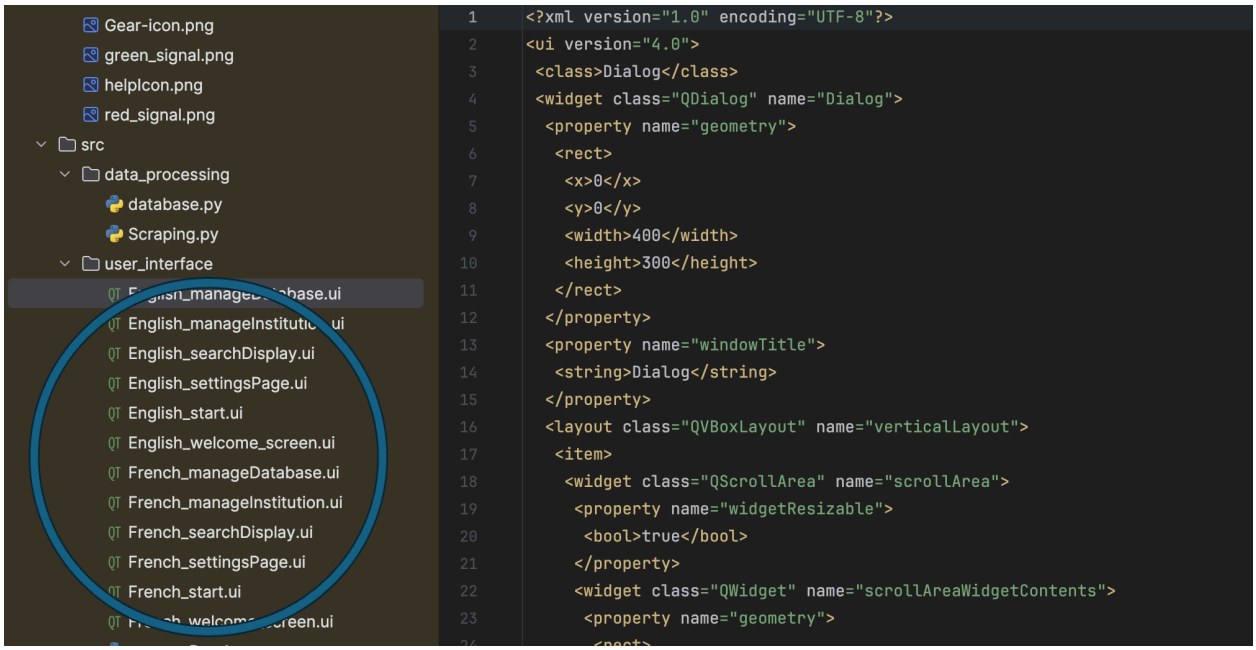
- Several helper functions like compare_file, update_tables, split_CRKN_file_name, and check_file_format are defined to facilitate scraping, comparison, and database updates.

Overall, this part of the code orchestrates the process of scraping data from the CRKN website, checking for updates, processing files, and updating the local database accordingly. It also handles errors and provides user feedback during the process.

2. *user_interface*

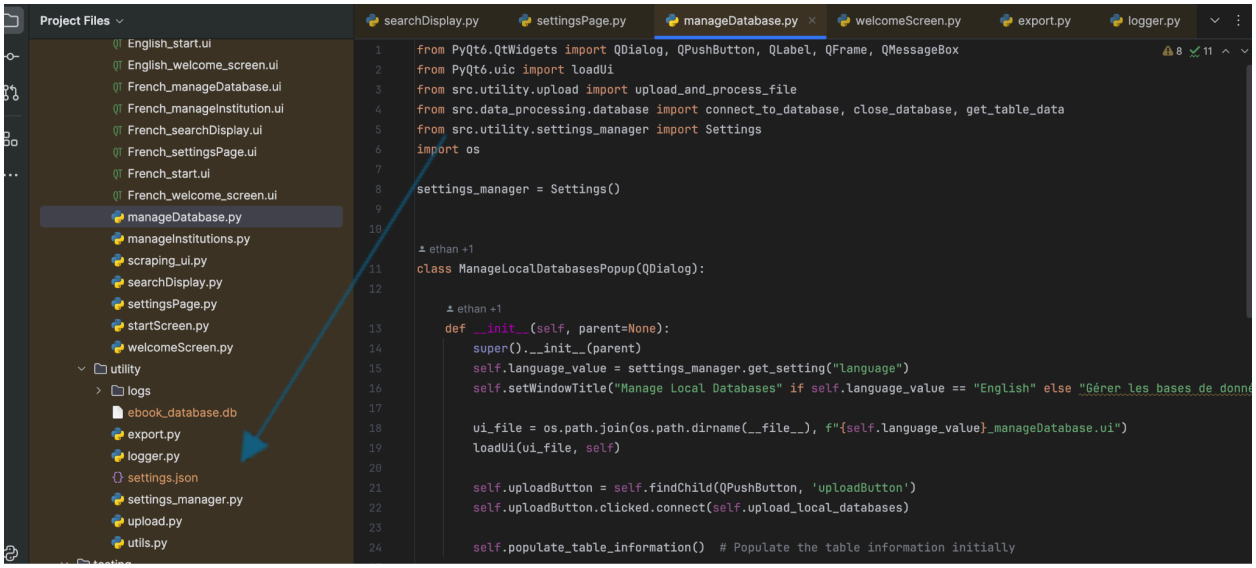
QT:
In PyQt, the GUI can either be hand-coded or made using PyQt's QT designer.
We have used QT Designer to design the GUI for our application.
QT designer converts the designed GUI into xml code for making it available to use in Python.

All the files below (circled) are GUIs designed by QT Designer.



- manageDatabase.py

- Imports:
  - It imports necessary classes and functions from PyQt6 and other modules such as upload_and_process_file() from src.utility.upload, database related functions from src.data_processing.database, and settings manager from src.utility.settings_manager.
  - os module is imported for handling file paths.



- Class Definition - ManageLocalDatabasesPopup:
    - This class inherits from QDialog, which is a PyQt class for creating dialog windows.
    - In the constructor __init__(), it sets the window title based on the language settings obtained from settings_manager.
    - It loads the UI file for the dialog window using loadUi() function.
    - Connects the uploadButton to the upload_local_databases method.
    - Calls populate_table_information() to initially populate the table information in the dialog window.

```python
    def populate_table_information(self):
        self.deleteTableData()

        # Get those tables
        connection = connect_to_database()
        local_table_data = get_table_data(connection, "local_file_names")

        # Populate the scroll area with table information
        for table_data in local_table_data:
            table_label = QLabel(f"{table_data[0]}, \n{'Date Added' if self.language_value == 'English' else 'Date ajou

            remove_button = QPushButton("Remove" if self.language_value == "English" else "Retirer")
            remove_button.clicked.connect(lambda checked, table=table_data[0]: self.remove_table(table))

            line = QFrame()
            line.setFrameShape(QFrame.Shape.HLine)
            line.setFrameShadow(QFrame.Shadow.Sunken)

            # Create a horizontal layout for each row
            # row_layout = QVBoxLayout()
            # row_layout.addWidget(table_label)
            # row_layout.addWidget(remove_button)
            # row_layout.addWidget(line)

            # Add the horizontal layout to the main vertical layout
            self.scrollLayout.addWidget(table_label)
            self.scrollLayout.addWidget(remove_button)
            self.scrollLayout.addWidget(line)

        close_database(connection)
```

- Methods:
    - populate_table_information(): Populates the table information in the dialog window by fetching data from the local database (local_file_names table), creating labels and buttons dynamically for each table, and adding them to the scroll area.
    - remove_table(): Removes the selected table from the local database. It prompts the user for confirmation using a QMessageBox and then calls remove_local_file() from src.utility.upload to remove the table from the database. After successful removal, it calls populate_table_information() again to refresh the table view and shows a success message.
    - deleteTableData(): Deletes the existing table data in the scroll area before populating it with updated data. This ensures that the scroll area is refreshed properly without duplicates.
    - upload_local_databases(): Triggers the process of uploading and processing local database files by calling upload_and_process_file() from src.utility.upload. After the upload process, it refreshes the table view using populate_table_information().
- Internationalization:

    - The dialog's title and messages are displayed based on the language settings obtained from settings_manager.
    - Buttons and messages are provided in English and French, depending on the selected language.
- Widgets:
    - It finds and connects the uploadButton from the UI file to the upload_local_databases method.
    - Labels, buttons, and frames are dynamically created and added to the scroll area layout (scrollLayout).

Overall, this code defines a dialog window for managing local databases, allowing users to upload new files, remove existing files, and view the list of available local databases. The UI is localized based on the selected language.

- manageInstitutions.py

This code defines another dialog window, ManageInstitutionsPopup, for managing institutions.

Imports:

- It imports necessary classes and functions from PyQt6 and other modules such as connect_to_database() and close_database() from src.data_processing.database, and settings manager from src.utility.settings_manager.
- os module is imported for handling file paths.

Class Definition - ManageInstitutionsPopup:

- Similar to the previous class, it inherits from QDialog.
- In the constructor __init__(), it sets the window title based on the language settings obtained from settings_manager.
- It loads the UI file for the dialog window using loadUi() function.
- Connects the uploadButton to the upload_local_institution method.
- Calls populate_table_information() to initially populate the table information in the dialog window.

Methods:

- populate_table_information(): Populates the table information in the dialog window by fetching institutions from the settings manager and creating labels and remove buttons dynamically for each institution. It then adds them to the scroll area.

```python
    def populate_table_information(self):
        self.deleteTableData()

        # Get those tables
        connection = connect_to_database()
        institutions = settings_manager.get_setting("local_institutions")

        # Populate the scroll area with table information
        for institution in institutions:
            table_label = QLabel(f"{institution}")

            remove_button = QPushButton("Remove" if self.language_value == "English" else "Retirer")
            remove_button.clicked.connect(lambda checked, institution=institution: self.remove_institution(institution))

            line = QFrame()
            line.setFrameShape(QFrame.Shape.HLine)
            line.setFrameShadow(QFrame.Shadow.Sunken)

            # Add the horizontal layout to the main vertical layout
            self.scrollLayout.addWidget(table_label)
            self.scrollLayout.addWidget(remove_button)
            self.scrollLayout.addWidget(line)

        close_database(connection)
```

- remove_institution(): Removes the selected institution from the settings manager. It prompts the user for confirmation using a QMessageBox and then calls remove_local_institution() from the settings manager to remove the institution. After successful removal, it calls populate_table_information() again to refresh the table view and shows a success message.
- deleteTableData(): Deletes the existing table data in the scroll area before populating it with updated data. This ensures that the scroll area is refreshed properly without duplicates.
- upload_local_institution(): Prompts the user to enter a new institution name using a QInputDialog. If the input is valid (not empty and not already in the list of institutions), it adds the new institution to the settings manager and refreshes the table view. If the input is empty or already exists, it shows a warning message using QMessageBox.

Internationalization:

- The dialog's title and messages are displayed based on the language settings obtained from settings_manager.
- Buttons and messages are provided in English and French, depending on the selected language.

```
# Populate the scroll area with table information
for institution in institutions:
    table_label = QLabel(f"{institution}")

    remove_button = QPushButton("Remove" if self.language_value == "English" else "Retirer")
    remove_button.clicked.connect(lambda checked, institution=institution: self.remove_institution(institutior

    line = QFrame()
    line.setFrameShape(QFrame.Shape.HLine)
    line.setFrameShadow(QFrame.Shadow.Sunken)

    # Add the horizontal layout to the main vertical layout
    self.scrollLayout.addWidget(table_label)
    self.scrollLayout.addWidget(remove_button)
    self.scrollLayout.addWidget(line)

close_database(connection)
```

Widgets:

- It finds and connects the uploadButton from the UI file to the upload_local_institution method.
- Labels, buttons, and frames are dynamically created and added to the scroll area layout (scrollLayout).

Overall, this code defines a dialog window for managing institutions, allowing users to add new institutions, remove existing ones, and view the list of available institutions. The UI is localized based on the selected language.

- scraping_ui.py

This code defines the functionality for updating the CRKN database through scraping.

Imports:

- It imports necessary classes and functions from PyQt6 and other modules such as ScrapingThread from src.data_processing.Scraping and settings manager from src.utility.settings_manager.

Global Variables:

- language is retrieved from the settings manager initially.

Function - scrapeCRKN():

```
def scrapeCRKN():
    global language
    language = settings_manager.get_setting("language")
    loading_popup = LoadingPopup()
    loading_popup.exec()
```

- This function is called to initiate the scraping process for updating the CRKN database.
- It creates an instance of LoadingPopup and executes it, which will show a loading dialog window.

Class Definition - LoadingPopup:

- This class inherits from QDialog and represents the loading dialog window during the scraping process.
- In the constructor __init__(), it sets the window title based on the language settings and specifies window flags.
- It sets up a vertical layout and adds a progress bar to it.
- Initializes a ScrapingThread instance for handling the scraping process in a separate thread.
- Sets up a timer to start the scraping thread.
- Defines slots to handle progress updates, file changes, errors, and completion of the scraping process.
- update_progress(): Updates the progress bar value and closes the dialog window when the progress reaches 100%.

```
ethan
43      def update_progress(self, value):
44          self.progress_bar.setValue(value)
45          if value == 100 and not self.finished:
46              self.finished = True
47              self.loading_thread = None
48              self.show_popup_once()
49              self.close()
50
ethan +1
```

- handle_file_changes(): Stops the timer and prompts the user with a QMessageBox to confirm whether to update the database with the detected file changes.
- handle_error(): Stops the timer, displays an error message with a QMessageBox, and handles the user response to continue or end the scraping process.
- show_popup_once(): Displays a QMessageBox indicating the completion of the scraping process.

Internationalization:

- Dialog titles, messages, and buttons are displayed based on the language settings obtained from settings_manager.

Overall, this code sets up a loading dialog window to show the progress of the CRKN database update process through scraping. It handles user interactions such as confirming file changes and displaying error messages appropriately. The UI language is determined based on the settings manager.

- searchDisplay.py

This code defines the searchDisplay class, which represents the search page where search results are displayed.

Imports:
- Necessary classes and functions are imported from PyQt6, including loadUi for loading UI files, various widget classes, and other modules such as export_data from src.utility.export and settings manager from src.utility.settings_manager.

Global Variables:
- settings_manager is used to access settings like language and institution name.

Class Definition - searchDisplay:
- This class inherits from QDialog and represents the search display page.
- It provides class methods get_instance and replace_instance for managing the instance of the class.
- The __init__ method initializes the search display UI by loading the corresponding .ui file and connecting signal handlers for buttons.

```
def __init__(self, widget, results):
    super(searchDisplay, self).__init__()
    language_value = settings_manager.get_setting("language")
    ui_file = os.path.join(os.path.dirname(__file__), f"{language_value}_searchDisplay.ui")
    loadUi(ui_file, self)

    # this is the back button that will take to the startscreen from the searchdisplay
    self.backButton.clicked.connect(self.backToStartScreen)
    self.exportButton.clicked.connect(self.export_data_handler)
    self.institutionName = self.findChild(QLabel, "institutionName")
    self.widget = widget
    self.results = results
    self.original_widget_values = None
    self.column_labels = ["Access", "File_Name", "Platform", "Title", "Publisher", "Platform_YOP", "Platform_eISBN

    self.tableWidget = self.findChild(QTableWidget, 'tableWidget')
    self.tableWidget.itemSelectionChanged.connect(self.updateCellNameDisplay)

    self.displayInstitutionName()
    self.display_results_in_table()
```

- It contains methods for updating the display of cell names when clicked, returning to the start screen, displaying search results in a table, displaying the institution name, and handling data export.
- update_all_sizes method adjusts the size and font of widgets dynamically based on the window size.
- resizeEvent method overrides the default resize event handler to call update_all_sizes when the window is resized.
- keyPressEvent method overrides the default key press event handler to ignore the Escape key event.

```
147     def keyPressEvent(self, event):
148         # Override keyPressEvent method to ignore Escape key event
149         if event.key() == Qt.Key.Key_Escape:
150             event.ignore()  # Ignore the Escape key event
151         else:
152             super().keyPressEvent(event)
153
```

Internationalization:

- Labels and messages are displayed based on the language settings obtained from settings_manager.

Exporting Data:

- The export_data_handler method is called when the export button is clicked. It invokes the export_data function to export search results.

```
91
    ethan
92  def export_data_handler(self):
93      export_data(self.results, self.column_labels)
94
    ethan
```

Overall, this class manages the search display page, including loading UI, displaying search results, handling user interactions, adjusting widget sizes dynamically, and exporting data.

- settingsPage.py

This code defines the settingsPage class, which represents the settings page of the application.

Imports:

- Necessary classes and functions are imported from PyQt6, including various widget classes, loadUi for loading UI files, and modules such as upload_and_process_file and settings manager.

Global Variables:

- settings_manager is used to access settings like language, institution name, and URLs.

Class Definition - settingsPage:

- This class inherits from QDialog and represents the settings page.
- It provides class methods get_instance and replace_instance for managing the instance of the class.
- The __init__ method initializes the settings page UI by loading the corresponding .ui file and connecting signal handlers for buttons and input fields.

```
def __init__(self, widget):
    super(settingsPage, self).__init__()
    self.language_value = settings_manager.get_setting("language")
    ui_file = os.path.join(os.path.dirname(__file__), f"{self.language_value}_settingsPage.ui")
    loadUi(ui_file, self)

    self.backButton2 = self.findChild(QPushButton, 'backButton')  # finding child pushButton from the parent class
    self.backButton2.clicked.connect(self.backToStartScreen2)
    self.widget = widget
    self.original_widget_values = None

    # Upload Button
    self.uploadButton = self.findChild(QPushButton, 'uploadButton')
    self.uploadButton.clicked.connect(self.upload_button_clicked)

    # Update Button
    self.updateButton = self.findChild(QPushButton, "updateCRKN")
    self.updateButton.clicked.connect(self.update_button_clicked)

    self.update_CRKN_button()
    self.update_CRKN_URL()

    # Finding the combobox for the institution
    self.institutionSelection = self.findChild(QComboBox, 'institutionSelection')
    self.institutionSelection.activated.connect(self.save_institution)
    self.populate_institutions()
    self.set_institution(settings_manager.get_setting("institution"))

    # Find the Push Button for manage local database
    self.manageDatabaseButton = self.findChild(QPushButton, 'manageDatabase')
    if self.language_value == "English":
        self.manageDatabaseButton.setToolTip("View, add, or remove local databases")
    elif self.language_value == "French":
        self.manageDatabaseButton.setToolTip("Afficher, ajouter ou supprimer les bases de données locales")
```

- It contains methods for updating settings like language, institution, CRKN URL, and help URL, handling user interactions, and resetting the application after changes.
- Methods like update_all_sizes and resizeEvent handle dynamic resizing of widgets based on window size changes.
- upload_button_clicked method is called when the upload button is clicked. It invokes the upload_and_process_file function and resets the application afterward.
- update_button_clicked method is called when the update CRKN button is clicked. It invokes the scrapeCRKN function and resets the application afterward.
- show_manage_local_databases_popup and show_manage_institutions_popup methods are called to show pop-ups for managing local databases and institutions, respectively.

Internationalization:

- Labels, tooltips, and messages are displayed based on the language settings obtained from settings_manager.

Handling User Input:

- Methods like save_language, save_institution, save_CRKN_URL, save_help_url, and save_allow_CRKN handle user input for changing settings and prompt for confirmation before applying changes.

```
157     def save_language(self):
158         current_language = settings_manager.get_setting("language")
159         selected_language = self.languageSelection.currentIndex()
160         if current_language == ("English" if selected_language == 0 else "French"):
161             return
162         reply = question_yes_no_box("Language Change" if current_language == "English" else "Changement de langue",
163                                     "Are you sure you want to change your language setting?" if current_language == "E
164         if reply:
165             settings_manager.set_language("English" if selected_language == 0 else "French")
166         self.reset_app()
167 
        ethan +1
168     def save_institution(self):
169         selected_institution = self.institutionSelection.currentText()
170         if selected_institution == settings_manager.get_setting("institution"):
171             return
172         response = question_yes_no_box("Change Institution" if self.language_value == "English" else "Changer d'établi
173                                        "Are you sure you want to change the institution?" if self.language_value == "E
174         if response:
175             if selected_institution.strip():
176                 settings_manager.set_institution(selected_institution)
177             else:
178                 settings_manager.set_institution("")
179         self.reset_app()
180 
        ethan +2
181     def save_CRKN_URL(self):
182         crkn_url = self.crknURL.text()
183         if crkn_url == settings_manager.get_setting("CRKN_url"):
184             return
185         if not (crkn_url.startswith("https://") or crkn_url.startswith("http://")):
186             information_box("Incorrect URL format" if self.language_value == "English" else "Format d'URL incorrect"
```

Overall, this class manages the settings page, including loading UI, updating settings, handling user interactions, and resetting the application after changes.

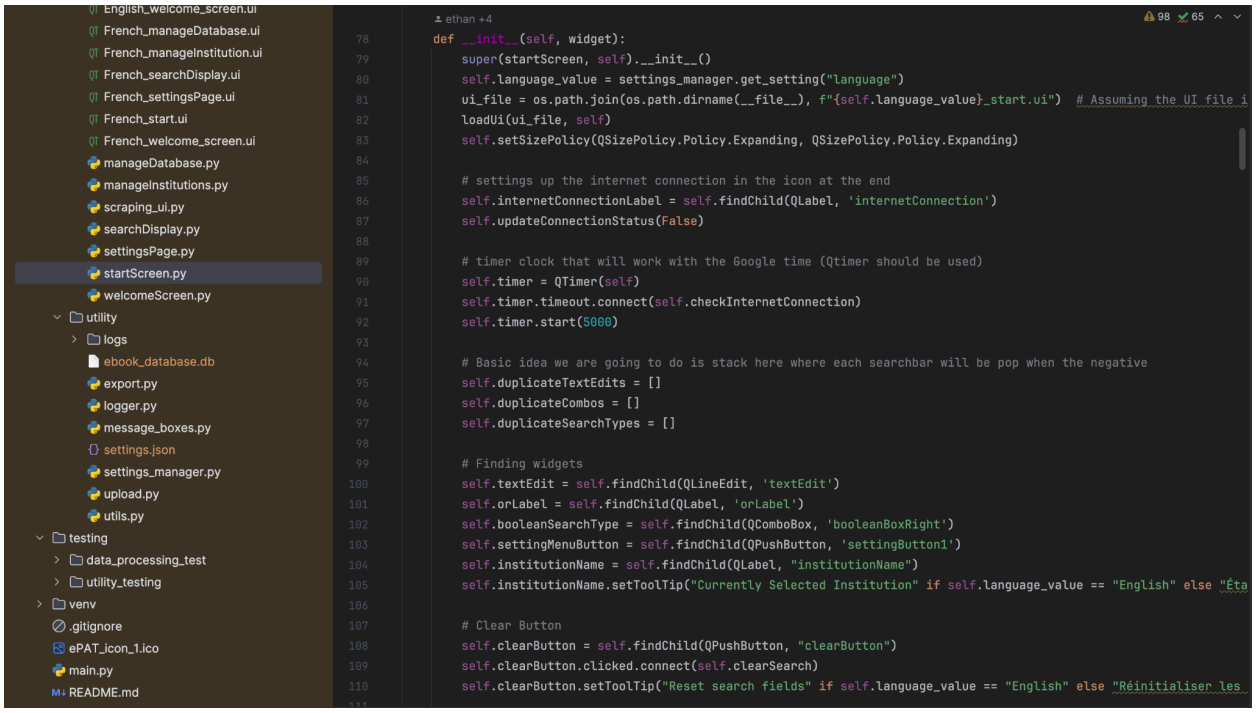
- startScreen.py

This Python code defines a PyQt6-based GUI application for search functionality.

Imports: The necessary modules and classes are imported. These include PyQt6 modules for GUI development, as well as some custom modules and classes from the application itself.
Custom RotatableButton Class: This class inherits from QPushButton and provides functionality to rotate an icon when the mouse hovers over it.

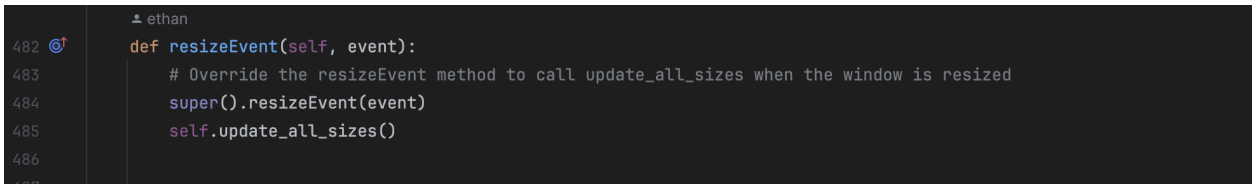
get_image_path Function: This function is used to get the path of an image file, handling differences between running the script normally and running it as an executable.
startScreen Class: This class represents the main screen of the application where users can initiate searches. It inherits from QDialog.

- get_instance Method: Returns an instance of the class. If an instance already exists, it returns the existing instance.
- replace_instance Method: Replaces the existing instance with a new one.
- __init__ Method: Initializes the start screen. It loads the UI file, sets up various widgets and functionality, and handles resizing of widgets based on window size changes.



- Methods for Handling Search Widgets: These methods handle adding and removing search text fields dynamically, as well as adjusting their sizes and positions.
- Methods for Handling Internet Connection: These methods check the internet connection status and update the UI accordingly.
- search_button_clicked Method: Handles the event when the search button is clicked. It constructs a search query based on user input and sends it to the backend for processing.
- settingsDisplay Method: Displays the settings page when the settings button is clicked.

- open_link Method: Opens a link (GitHub link in this case) in the default web browser.
- resizeEvent Method: Overrides the resizeEvent method to handle resizing of widgets when the window size changes.
- update_all_sizes Method: Updates the sizes of all widgets based on the window size.



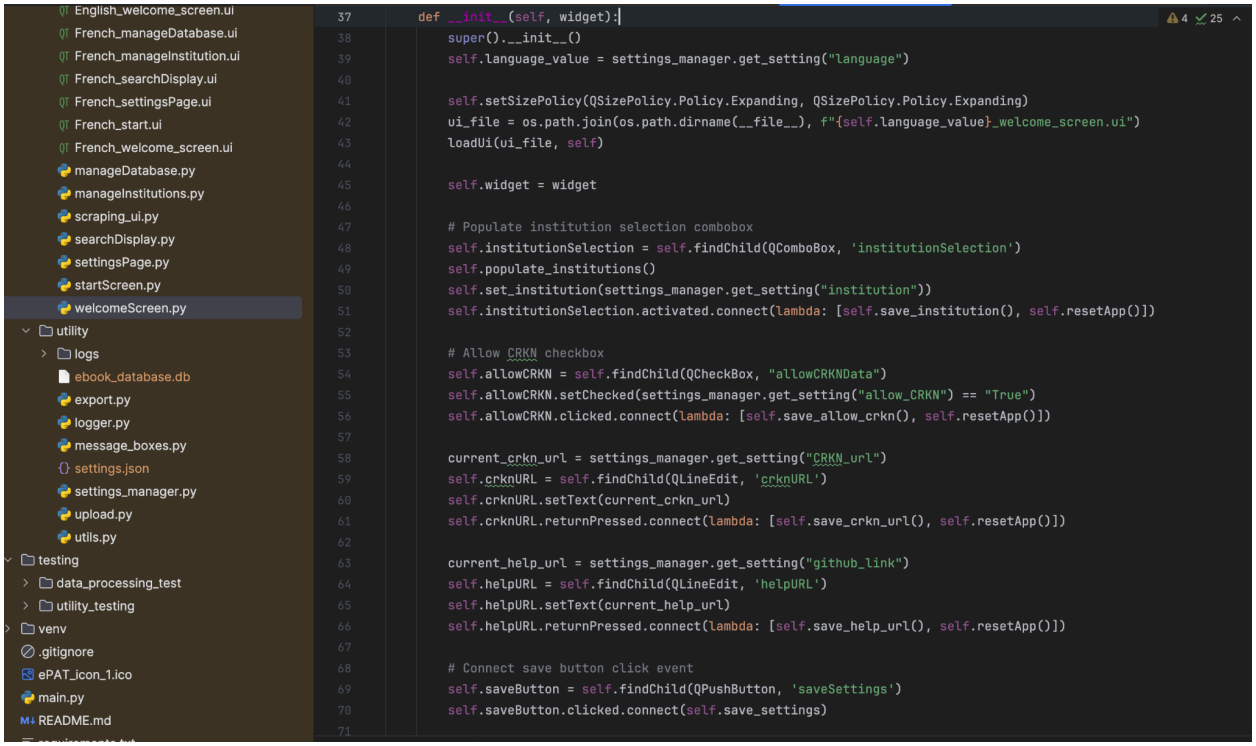
Other Helper Functions and Variables: These include helper functions and variables used within the class.

This code represents the main functionality of the start screen of the application, including search functionality and handling of user interactions.

- welcomeScreen.py

This code defines a WelcomePage class that serves as the initial page of an application.

- The class inherits from QDialog, which is a dialog window in PyQt.
- It imports necessary modules from PyQt6 and the application's internal modules.
- An instance of the Settings class is created from src.utility.settings_manager.
- The WelcomePage class implements several methods for initializing the page, handling user interactions, and saving settings.
- The __init__ method sets up the UI elements, connects signals to slots (functions), and initializes settings based on saved values.



- The populate_institutions method populates a combo box with institution names retrieved from the settings manager.
- Methods like save_crkn_url, save_help_url, save_institution, save_language, and save_allow_crkn handle saving user inputs to settings.
- The save_settings method saves all settings and updates the UI accordingly.
- The set_current_settings_values method sets UI elements to reflect the current settings.
- The update_all_sizes method adjusts the size and geometry of UI elements when the window is resized.
- The resizeEvent method overrides the default resize behavior to call update_all_sizes.
- The resetApp method resets the application by replacing the current widget instance with a new one.

```
def resetApp(self):
    widget_count = self.widget.count()
    for i in range(widget_count):
        current_widget = self.widget.widget(i)
        new_widget_instance = type(current_widget).replace_instance(self.widget)
        self.widget.insertWidget(i, new_widget_instance)
        self.widget.removeWidget(current_widget)
        current_widget.deleteLater()

    # Set the current index to the last widget added
    self.widget.setCurrentIndex(widget_count - 1)
```

Overall, this class manages the welcome page UI, user interactions, and settings management for the application.

3. *utility*

- export.py

This code segment deals with exporting data to a TSV (Tab-Separated Values) file using PyQt6 and pandas.

- It imports necessary modules from PyQt6 (QFileDialog, QApplication, QMessageBox) and pandas (pd) as well as the sys module and some custom modules from the application (logger and settings_manager).
- The export_data function takes two parameters: data (the data to export in the form of a list) and headers (the headers of the columns in the form of a list).
- Inside the export_data function:
    - It retrieves the current language setting from the settings_manager.
    - It checks if there is an existing instance of QApplication. If not, it creates a new instance.
    - It converts the input data and headers into a pandas DataFrame.
    - It gets the save path for the TSV file using the get_save_path function.
    - If a save path is selected by the user:
        - It appends ".tsv" to the file path if it doesn't already end with that extension.
        - It saves the DataFrame to a TSV file at the specified path.
        - It logs the export operation and shows an information message box confirming the export operation to the user.
- The get_save_path function opens a file dialog to allow the user to select the save path for the TSV file. It returns the selected file path.

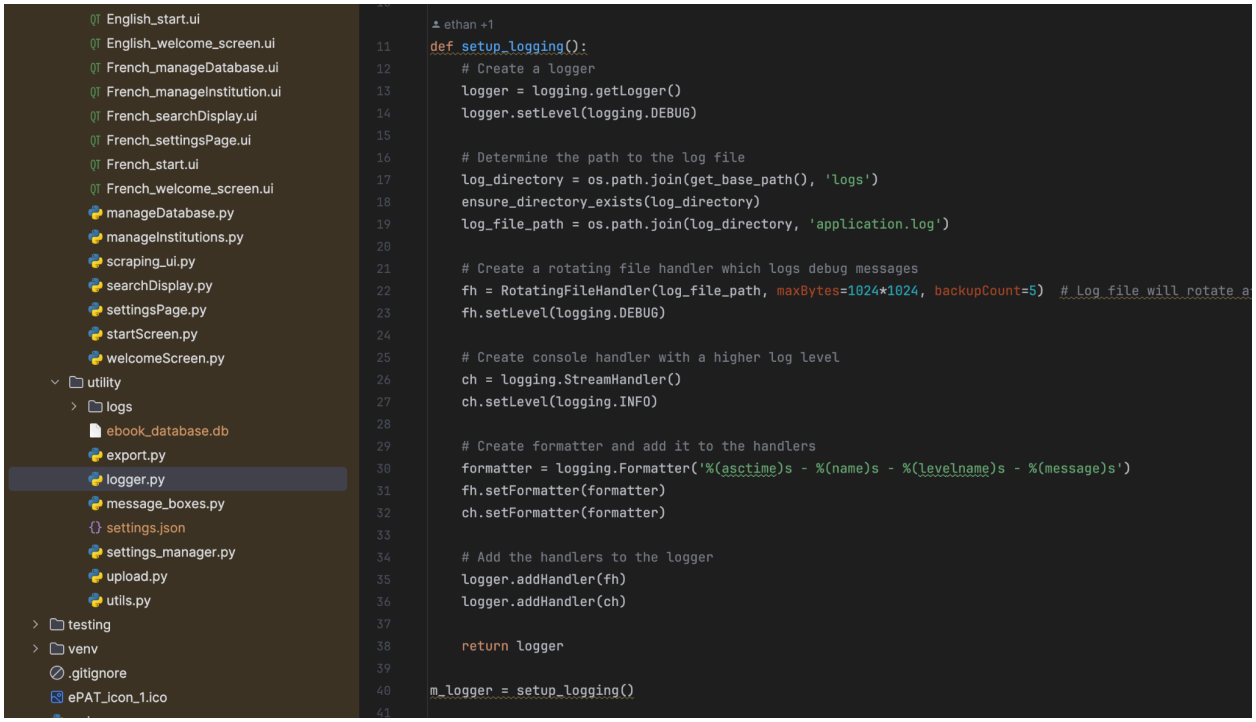
Overall, this code segment provides functionality to export data to a TSV file with user interaction for selecting the save path using PyQt6 dialogs.

- logger.py

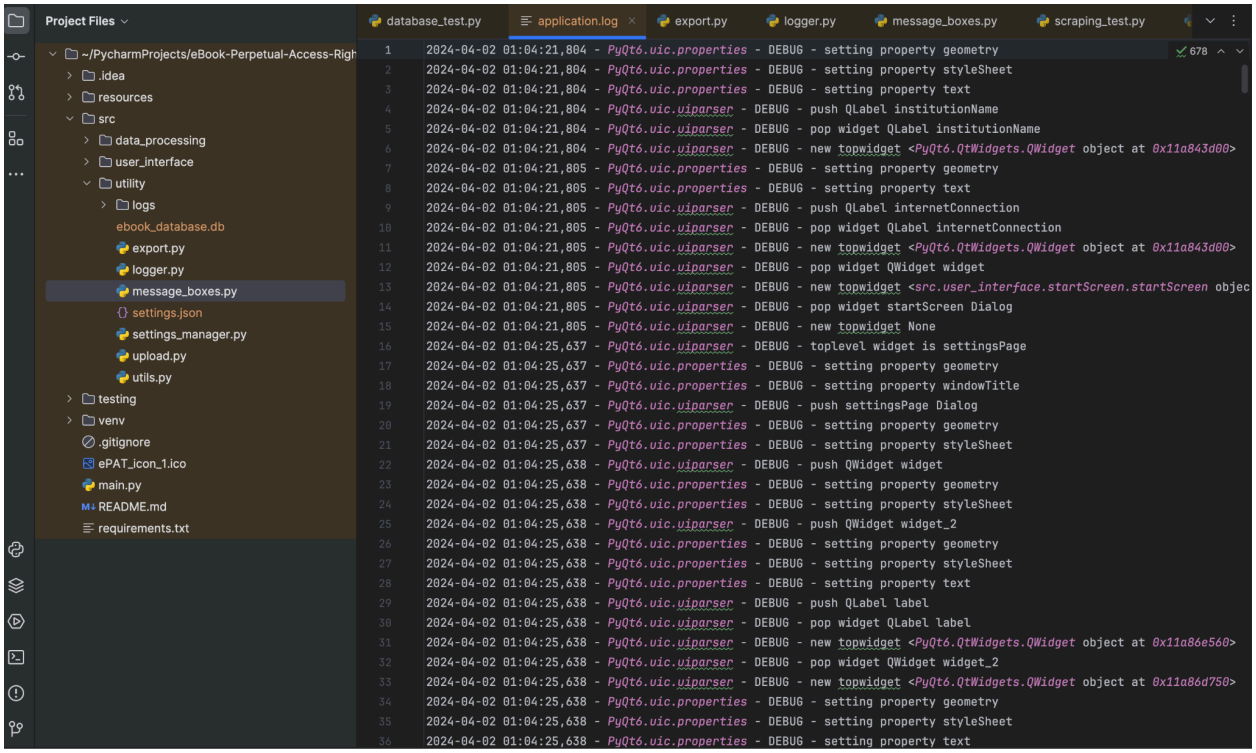
This code segment sets up logging for the application.

- It imports necessary modules: logging and RotatingFileHandler from the logging.handlers module, and os.
- The ensure_directory_exists function is defined to ensure that a directory exists at the given path. If the directory doesn't exist, it creates it.
- The setup_logging function is defined to configure the logging settings:
    - It creates a logger with the root logger name.
    - The logger level is set to DEBUG, meaning it will capture all levels of log messages.
    - It determines the path to the log directory using the get_base_path function from utils module and creates the directory if it doesn't exist.
    - It specifies the path for the log file (application.log) within the log directory.
    - It creates a RotatingFileHandler to handle logging to the file. This handler rotates the log file after it reaches a certain size (1MB in this case) and keeps up to 5 backup files.
    - It sets the level of the file handler to DEBUG.
    - It creates a console handler to log messages to the console with a level of INFO.
    - It creates a formatter for log messages with a specific format including timestamp, logger name, log level, and message.
    - It sets the formatter for both file and console handlers.
    - Finally, it adds both the file handler and console handler to the logger.

  The m_logger variable is assigned the logger object returned by the setup_logging function. This m_logger object can be used throughout the application to log messages.



Overall, this code segment sets up logging to capture messages of various levels and store them in a rotating log file (application.log) while also printing them to the console. It provides a standardized logging setup for the application, allowing for easy debugging and monitoring.



- message_boxes.py

This code segment deals with PyQt6, a Python binding for the Qt application framework. It provides functionality to display various types of message boxes and input dialogs to interact with the user.

- Imports:
    - from PyQt6.QtWidgets import QMessageBox, QInputDialog: This imports the QMessageBox and QInputDialog classes from the PyQt6.QtWidgets module. These classes are used to create message boxes and input dialogs, respectively.
    - from src.utility.logger import m_logger: This imports a logger object named m_logger from a custom utility module for logging purposes.
    - from src.utility.settings_manager import Settings: This imports a custom Settings class from a utility module for managing application settings.
- question_yes_no_box Function:
    - This function displays a message box with Yes and No buttons.
    - Parameters:
        - title: The title of the message box.
        - body: The message body to display.
        - icon: (Optional) The icon to display in the message box. Default is a question mark icon.

```python
def question_yes_no_box(title, body, icon = QMessageBox.Icon.Question):
    """
    Display a message box with Yes and No buttons.

    Parameters:
    - title (str): The title of the message box.
    - body (str): The message body to display.
    - icon (QMessageBox.Icon, optional): The icon to display in the message box. Defaults to QMessageBox.Icon.Question

    Returns:
    - bool: True if the Yes button is clicked, False otherwise.
    """
    language = settings_manager.get_setting("language")
    msg_box = QMessageBox()
    msg_box.setIcon(icon)
    msg_box.setWindowTitle(title)
    msg_box.setText(body)

    if language == "French":
        yes_button = msg_box.addButton("Oui", QMessageBox.ButtonRole.YesRole)
        no_button = msg_box.addButton("Non", QMessageBox.ButtonRole.NoRole)
    else:
        yes_button = msg_box.addButton(QMessageBox.StandardButton.Yes)
        no_button = msg_box.addButton(QMessageBox.StandardButton.No)

    msg_box.exec()

    return msg_box.clickedButton() == yes_button
```

- The function checks the current language setting from the application settings (settings_manager.get_setting("language")) and adjusts the button labels accordingly.
- It returns True if the Yes button is clicked, and False otherwise.

- information_box Function:
  - This function displays an information message box with an OK button.
  - Parameters:
    - title: The title of the message box.
    - body: The message body to display.
    - icon: (Optional) The icon to display in the message box. Default is an information icon.
  - Similar to the question_yes_no_box function, it adjusts the button labels based on the current language setting.
  - It doesn't return any value.
- input_dialog_ok_cancel Function:
  - This function displays an input dialog with OK and Cancel buttons.
  - Parameters:
    - title: The title of the input dialog.
    - body: The prompt text for the input dialog.

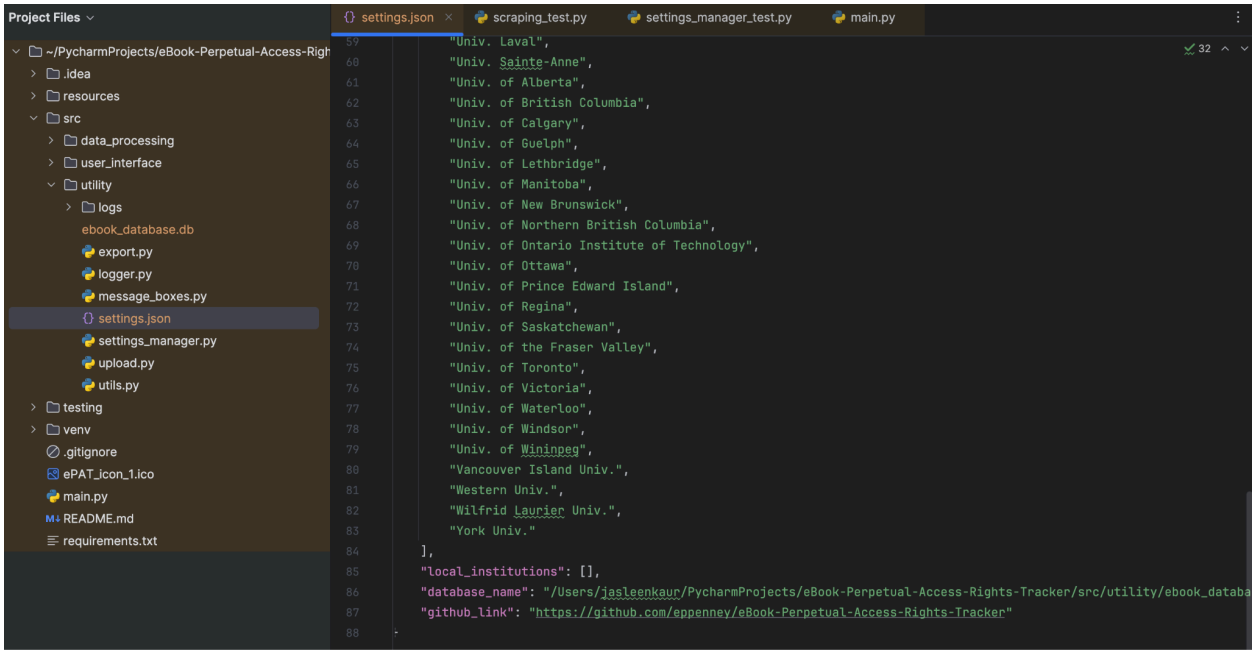
        - icon: (Optional) The icon to display in the input dialog. Default is a question mark icon.
    - It sets the button labels according to the current language setting.
    - It returns a tuple containing the entered text and a boolean indicating whether the OK button was clicked.

These functions encapsulate the functionality to interact with the user through message boxes and input dialogs, providing a convenient way to display information, ask questions, and get user input in PyQt6 applications.

- settings.json

This JSON data represents a configuration file or settings for the application.
This JSON data represents a configuration file or settings for the application. Here's a breakdown of its contents:

- "language": "English": Specifies the language setting for the application. It indicates that the application is currently set to use English.
- "allow_CRKN": "True": Indicates whether the CRKN (Canadian Research Knowledge Network) access is allowed. In this case, it is set to True, meaning CRKN access is allowed.
- "institution": "Univ. of Prince Edward Island": Specifies the default institution for the application. In this case, it is set to the University of Prince Edward Island.
- "CRKN_url": "https://library.upei.ca/test-page-ebooks-perpetual-access-project": Provides the URL for accessing CRKN resources.
- "CRKN_root_url": "https://library.upei.ca": Specifies the root URL for CRKN resources.
- "CRKN_institutions": [...]: This is a list of institutions that are part of the CRKN network. Each institution is listed with its name. These institutions include various universities and educational institutions from Canada.
- "local_institutions": []: This is an empty list, indicating that there are no local institutions specified in the configuration.
- "database_name": "/Users/jasleenkaur/PycharmProjects/eBook-Perpetual-Access-Rights-Tracker/src/utility/ebook_database.db": Specifies the file path for the database used by the application. It seems to be a SQLite database located at the specified path.
- "github_link": "https://github.com/eppenney/eBook-Perpetual-Access-Rights-Tracker": Provides the link to the GitHub repository for the application.



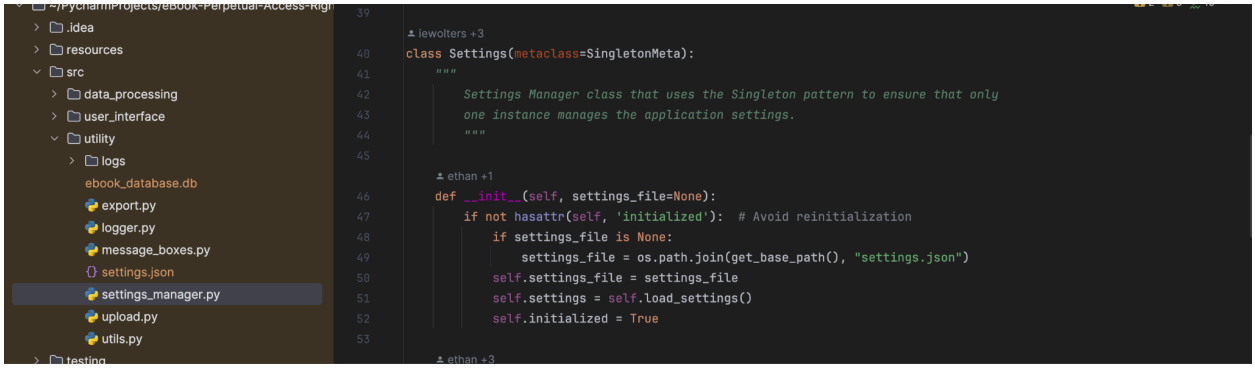
Overall, this JSON data contains various settings and configuration options that are used by the application, including language settings, CRKN access, default institution, database path, and GitHub repository link.

- settings_manager.py

This Python code defines a Settings class responsible for managing application settings.

This Python code defines a Settings class responsible for managing application settings. Here's a breakdown of its key components:

- SingletonMeta Metaclass: This metaclass ensures that only one instance of the Settings class is created throughout the application's lifecycle. It achieves this by storing instances in a dictionary and checking for existing instances before creating a new one.
- Settings Class:
    - The __init__ method initializes the Settings object by loading settings from a JSON file. If the file doesn't exist, default settings are used.



- load_settings method reads the settings from the JSON file or creates default settings if the file doesn't exist.
- save_settings method saves the current settings back to the JSON file.
- update_setting method updates a specific setting with a new value and saves the changes.
- get_setting method retrieves the value of a specific setting.
- Methods like set_language, set_allow_CRKN, set_crkn_url, set_github_url, set_institution, add_local_institution, remove_local_institution, set_CRKN_institutions, get_CRKN_institutions, set_local_institutions, and get_institutions are provided to update, retrieve, and manage different settings.
- The get_institutions method retrieves a combined list of CRKN and local institutions by querying a database.

- JSON File Handling: The settings are stored in a JSON file, and methods are provided to load and save settings from/to this file.
- Interaction Instructions: Comments in the code provide instructions on how to interact with the Settings class, such as creating an instance and using its methods to get or update settings.

Overall, this class facilitates the management of application settings, allowing for easy retrieval, updating, and saving of configuration options.

- upload.py

This code segment is responsible for uploading and processing local files into a local database.

upload_and_process_file Function:

- This function is called to initiate the process of uploading and processing local files.
- It opens a file dialog for the user to select one or more files.

- Upon file selection, it creates an instance of the UploadUI class to handle the file upload process.

UploadUI Class:

- This class represents a dialog window for monitoring the progress of file uploads.
- It displays a progress bar indicating the progress of the file upload process.
- It creates an instance of the UploadThread class to perform the file processing in a separate thread.
- It connects signals emitted by the UploadThread class to slots for updating the progress bar and handling errors or user responses.
- It closes itself automatically when the file upload process is completed.

UploadThread Class:

- This class represents a separate thread for processing files.
- It receives a list of file paths as input.
- It emits signals to update the progress bar, handle errors, and interact with the user.
- It iterates through each file path, processing each file one by one.
- For each file, it checks if it already exists in the database and prompts the user for action if necessary.
- It processes each file by converting it to a DataFrame, validating its format, and uploading it to the local database.
- It adds any new institutions found in the file to the local institutions list.
- It handles errors that may occur during the file processing.

```python
class UploadThread(QThread):
    ethan
    def __init__(self, file_paths):
        super().__init__()
        self.file_paths = file_paths
        self.file_length = len(file_paths)
        self.currentValue = 0
        self.one_file_progress_value = (1 / self.file_length) * 100

    progress_update = pyqtSignal(int)
    error_signal = pyqtSignal( *types: str, str)
    get_answer_yes_no = pyqtSignal( *types: str, str)
    get_okay = pyqtSignal( *types: str, str)

    ethan
    def run(self):
        self.process_files()

    ethan
    def process_files(self):
        for i in range(self.file_length):
            self.currentValue = i * self.one_file_progress_value
            self.progress_update.emit(int(self.currentValue))
            self.process_file(self.file_paths[i])
        self.progress_update.emit(100)

    ethan +4
    def process_file(self, file_path):
        """
        Process file and store in local database - similar to Scraping.download_files, but for local files
        :param file_path: string containing the path to the file
        """
        app = QApplication.instance()  # Try to get the existing application instance
        if app is None:  # If no instance exists, create a new one
```

Other Helper Functions:

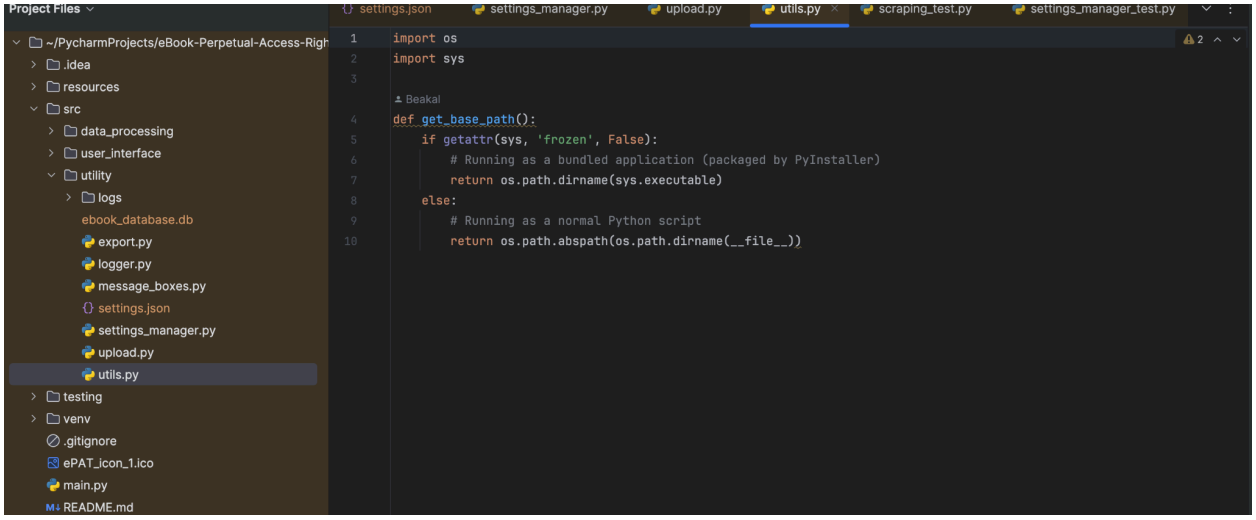
- get_new_institutions: Retrieves a list of institutions that are not in either the CRKN or local list from a new file DataFrame.
- file_to_df: Converts a file to a DataFrame based on its extension (csv, xlsx, or tsv).
- remove_local_file: Removes a local file from the database.

Overall, this code segment provides functionality for users to upload local files, process them, and store them in a local database, while also handling various aspects such as progress monitoring, error handling, and user interactions.

- utils.py

This code defines a function get_base_path() that determines the base directory path of the application.

- The function checks if the script is running as a bundled application, packaged by PyInstaller, or as a normal Python script.
- If the script is running as a bundled application (i.e., frozen), it returns the directory path of the executable file using sys.executable.
- If the script is running as a normal Python script, it returns the absolute directory path of the current script file using os.path.dirname(__file__).



This function is useful for obtaining the base directory path of the application regardless of how it is executed. It ensures that the application can locate its resources and files reliably, regardless of its execution context.

- **testing**

1. *data_processing_test*

- database_test.py

This code is a set of unit tests for testing database operations in a Python application.

This code is a set of unit tests for testing database operations in a Python application. Let's break down the key aspects:

- Imports:
  - unittest: This module provides a set of tools for constructing and running tests.
  - MagicMock, patch, call: These are utilities from the unittest.mock module used for mocking objects and patching.
  - sqlite3: This module provides a SQL database interface.
- Test Class TestDatabaseOperations:
  - This class inherits from unittest.TestCase and contains multiple test methods for database operations.
- Setup Method setUp:
  - This method is called before each test method and is used to set up the test environment.
  - It creates mock objects for the database connection and cursor.
- Test Methods:
  - test_connect_to_database: Tests the connect_to_database function by mocking the database connection and asserting the correct behavior.
  - test_close_database: Tests the close_database function by mocking the database connection and asserting the correct method calls.
  - test_get_CRKN_tables_allow_true and test_get_CRKN_tables_allow_false: Test the get_CRKN_tables function for different settings.
  - test_get_local_tables: Tests the get_local_tables function by mocking the database cursor and asserting the correct behavior.
  - test_get_tables_integration: Tests the get_tables function by mocking its dependencies and asserting the correct behavior.

```
Beakal
136 ▷   def test_get_table_data(self):
137         self.mock_cursor.fetchall.return_value = [('data1', 'data2')]
138         data = database.get_table_data(self.mock_connection, 'test_table')
139         self.assertEqual(data, second: [('data1', 'data2')])
140
        Beakal
141 ▷   def test_get_table_data_with_error(self):
142         self.mock_cursor.execute.side_effect = sqlite3.Error("Test Error")
143         with self.assertLogs(level='ERROR') as log:
144             data = database.get_table_data(self.mock_connection, 'bad_table')
145             self.assertEqual(data, second: [])
146             self.assertIn( member: 'An error occurred while fetching data from the table: Test Error', log.output[0])
147
```

- test_tables_need_creation and test_tables_already_exist: Test the create_file_name_tables function for different scenarios by mocking database interactions.
- test_search_by_title_with_wildcards: Tests the search_database function for searching by title with wildcards by mocking database interactions.
- test_get_table_data and test_get_table_data_with_error: Test the get_table_data function for retrieving table data with and without errors by mocking database interactions.

- Mocking:
  - @patch: This decorator is used to patch the specified functions or objects during the test.
  - It helps isolate the code under test from its dependencies, ensuring that tests focus on specific functionality.

Overall, these tests verify the correctness of database operations under different conditions and ensure robustness and reliability of the database-related functions in the application.

- scraping_test.py

  This code is a set of unit tests for testing various functionalities related to web scraping, data processing, and database operations.

  - Imports:
    - unittest: This module provides a set of tools for constructing and running tests.
    - patch, MagicMock: These are utilities from the unittest.mock module used for mocking objects and patching.
    - pd: This is the pandas library used for data manipulation.
    - HTTPError: This is an exception class from the requests.exceptions module.
  - Test Class TestScrapingThread:

- This class inherits from unittest.TestCase and contains multiple test methods for scraping related functionalities.

- Setup Method setUp:
  - This method is called before each test method and is used to set up the test environment.
  - It creates an instance of ScrapingThread and sets up mock signals and a mock database connection.

- Test Methods:
  - test_scrapeCRKN_success and test_scrapeCRKN_fail_http_error: These methods test the scrapeCRKN function for successful and failed HTTP requests, respectively.
  - test_compare_file_insert and test_compare_file_upadate: These methods test the compare_file function for different scenarios (insert and update).
  - test_update_tables_insert and test_update_tables_update: These methods test the update_tables function for different scenarios (insert and update).

```
150     def test_update_tables_update(self):
151
152         file = ['test_publisher', '2024-03-30']
153         method = 'local'
154         command = 'UPDATE'
155
156         # Execute the function with the mock objects
157         update_tables(file, method, self.mock_connection, command)
158
159         # Verify
160         self.mock_cursor.execute.assert_called_once_with(
161             f"UPDATE {method}_file_names SET file_date = '{file[1]}' WHERE file_name = '{file[0]}';")
162         self.mock_connection.commit.assert_called_once()
163
        Beakal +1
164     def test_check_file_format_correct(self):
165         # Create a dataframe with the correct header row and some data
166         correct_df = pd.DataFrame({
167             "Title": ["Book Title 1"],
168             "Publisher": ["Publisher 1"],
169             "Platform_YOP": [2020],
170             "Platform_eISBN": ["1234567890123"],
171             "OCN": [12345678],
172             "agreement_code": ["Agreement 1"],
173             "collection_name": ["Collection 1"],
174             "title_metadata_last_modified": ["2021-01-01"],
175         })
176         assert check_file_format(correct_df, language="English") is True, "The function should return True for correct
177
```

- test_check_file_format_correct, test_check_file_format_incorrect_header, and test_check_file_format_missing_columns: These methods test the check_file_format function for different scenarios (correct format, incorrect header, and missing columns).

- test_upload_to_database_success and test_upload_to_database_failure: These methods test the upload_to_database function for success and failure scenarios.

- Mocking:
  - @patch: This decorator is used to patch the specified functions or objects during the test.
  - It helps isolate the code under test from its dependencies, ensuring that tests focus on specific functionality.

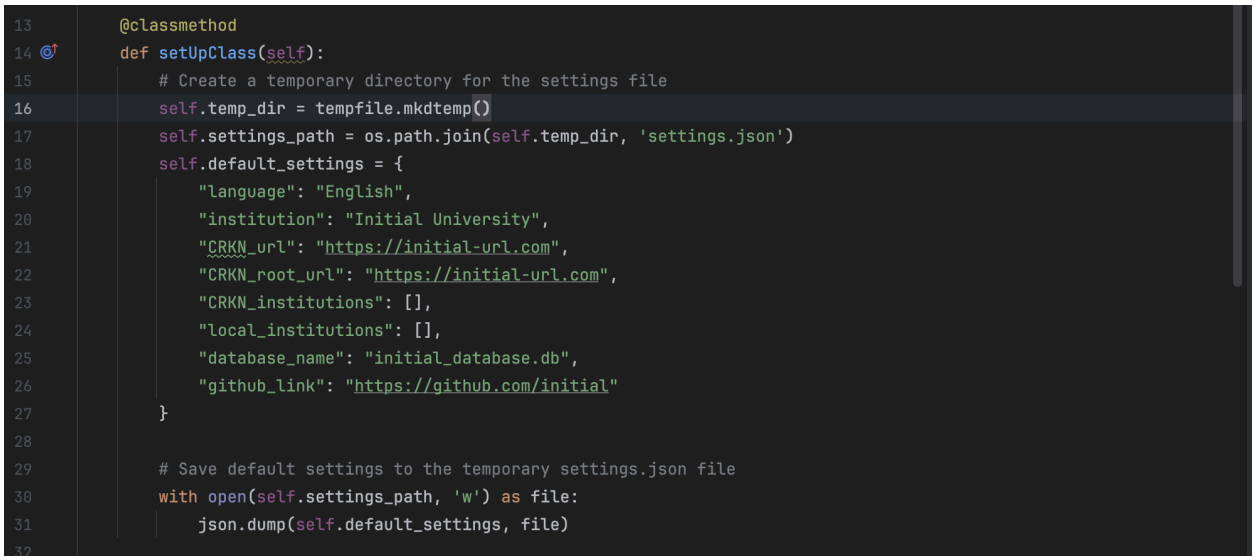
```
Beakal
@patch('src.data_processing.Scraping.ScrapingThread.wait_for_response')
@patch('src.data_processing.Scraping.ScrapingThread.download_files')
@patch('src.data_processing.Scraping.update_tables')
@patch('src.data_processing.Scraping.compare_file')
@patch('src.utility.settings_manager.Settings.set_CRKN_institutions')
@patch('src.data_processing.database.connect_to_database')
@patch( target: 'src.utility.settings_manager.Settings.get_setting', side_effect=settings_side_effect)
@patch('requests.get')
def test_scrapeCRKN_fail_http_error(self, mock_get, mock_get_setting, mock_connect,
                                    mock_set_crkn_inst, mock_compare, mock_update_tables, mock_download_files,
                                    mock_wait_res):
    """Successful HTTP request and scrape"""
    # Mocking the requests.get to return a successful response
    mock_response = Response()
    mock_response._content = bytes(mock_html_content, 'utf-8')  # Simulating byte content of the HTML response
    mock_get.return_value = HTTPError

    mock_connect.return_value = self.mock_connection
```

Overall, these tests ensure the correctness and robustness of web scraping, data processing, and database-related functionalities in the application. They cover various scenarios to ensure proper handling of different situations.

2. *utility_testing*

- settings_manager_test.py
  This test code is for testing the SettingsManager class, which is responsible for managing application settings stored in a JSON file.

  - Imports:
    - unittest: This module provides a set of tools for constructing and running tests.
    - tempfile, shutil, os: These modules are used for creating temporary directories, file operations, and cleanup after tests.
    - json: This module is used for JSON file operations.
    - Settings: This is the class being tested, imported from src.utility.settings_manager.
  - Test Class TestSettingsManager:
    - This class inherits from unittest.TestCase and contains test methods for testing the SettingsManager class.
  - Class Attributes:
    - temp_dir: Stores the path of the temporary directory created for the settings file.
    - default_settings: Stores the default settings in a dictionary format.
    - settings_path: Stores the path of the settings JSON file within the temporary directory.
  - Class Methods:
    - setUpClass(cls): This method is called before any test methods in the class are executed.
      - It creates a temporary directory and sets up the default settings, saving them to a JSON file.



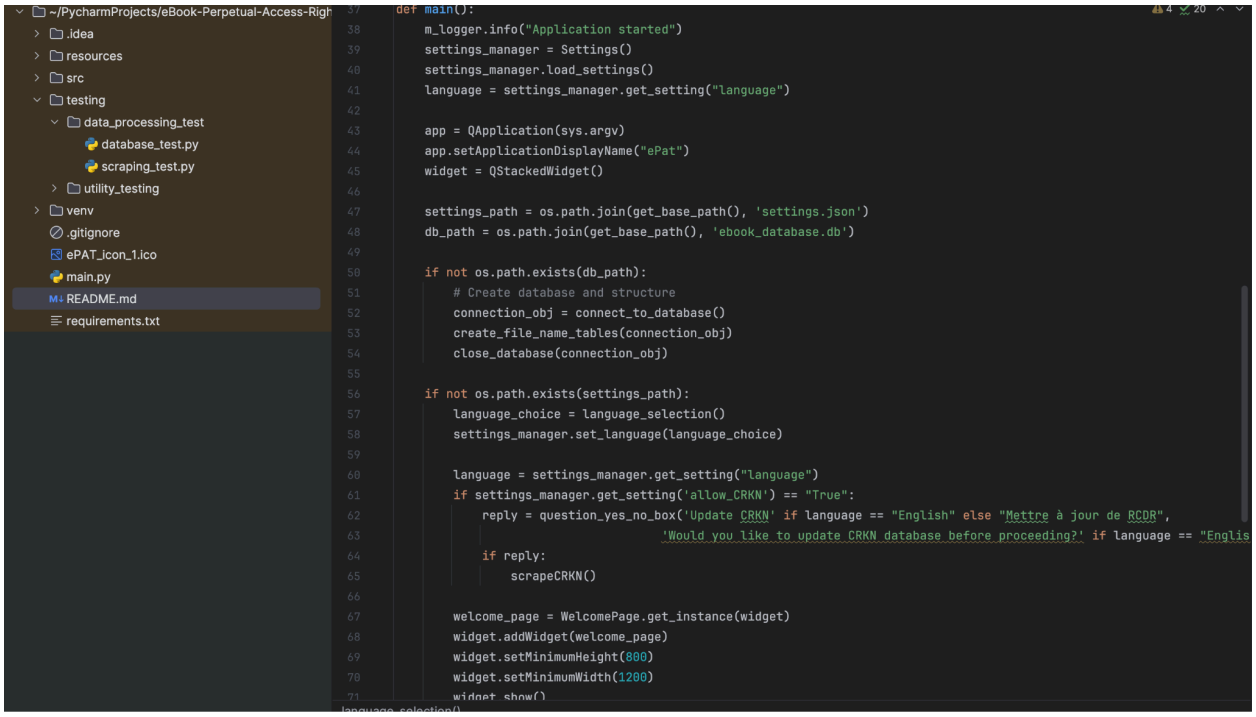
```
13      @classmethod
14      def setUpClass(self):
15          # Create a temporary directory for the settings file
16          self.temp_dir = tempfile.mkdtemp()
17          self.settings_path = os.path.join(self.temp_dir, 'settings.json')
18          self.default_settings = {
19              "language": "English",
20              "institution": "Initial University",
21              "CRKN_url": "https://initial-url.com",
22              "CRKN_root_url": "https://initial-url.com",
23              "CRKN_institutions": [],
24              "local_institutions": [],
25              "database_name": "initial_database.db",
26              "github_link": "https://github.com/initial"
27          }
28
29          # Save default settings to the temporary settings.json file
30          with open(self.settings_path, 'w') as file:
31              json.dump(self.default_settings, file)
```

    - tearDownClass(cls): This method is called after all test methods in the class have been executed.
        - It removes the temporary directory and its contents.
- Test Methods:
    - test_singleton_behavior: Tests whether the SettingsManager class enforces singleton behaviour, ensuring that only one instance is created for a given settings file path.
    - test_load_settings_successfully: Tests whether the settings are loaded successfully from the JSON file, comparing each setting value with the default settings.
    - test_handle_missing_settings_json: Tests whether the SettingsManager handles the case where the settings JSON file is missing.
    - test_update_setting: Tests whether the update_setting method updates a setting value correctly in the settings JSON file.
    - test_get_setting_value: Tests whether the get_setting method retrieves the correct value for a given setting key.
- Assertions:
    - Various assertions are used in each test method to verify the expected behavior of the SettingsManager class.

Overall, these test methods cover different scenarios to ensure the correctness and robustness of the SettingsManager class, including loading settings, handling missing settings files, updating settings, and enforcing singleton behaviour.

- main

This code is a Python script that serves as the entry point for an application. Let's break it down step by step:



Imports:

- sys: Provides access to some variables used or maintained by the Python interpreter and to functions that interact with the interpreter.
- QApplication, QMessageBox, QStackedWidget: Classes from PyQt6.QtWidgets module for building GUI applications.
- Various modules and classes from the application's own source files such as startScreen, connect_to_database, Settings, m_logger, get_base_path, and question_yes_no_box.
- os: Provides a portable way of using operating system-dependent functionality.

language_selection Function:

- Presents a message box for selecting the language between English and French.
- Returns the selected language or None if no selection is made.

```python
def language_selection():
    msg_box = QMessageBox()
    msg_box.setWindowTitle("Language Selection")
    msg_box.setText("Please select your language / Veuillez sélectionner votre langue")
    msg_box.setStandardButtons(QMessageBox.StandardButton.Yes | QMessageBox.StandardButton.No)
    msg_box.setDefaultButton(QMessageBox.StandardButton.Yes)

    button_en = msg_box.button(QMessageBox.StandardButton.Yes)
    button_en.setText("English")

    button_fr = msg_box.button(QMessageBox.StandardButton.No)
    button_fr.setText("Français")

    msg_box.exec()

    if msg_box.clickedButton() == button_en:
        return "English"
    elif msg_box.clickedButton() == button_fr:
        return "French"
    else:
        return None
```

main Function:

- Starts by logging that the application has started.
- Initializes the Settings manager to manage application settings.
- Loads settings from the settings file.
- Creates an instance of QApplication.
- Sets the application display name to "ePat".
- Creates a QStackedWidget to manage a stack of widgets.
- Constructs paths for the settings file and the database file.
- Checks if the database file exists, and if not, creates the necessary database structure.
- Checks if the settings file exists.
- If the settings file doesn't exist:
    - Asks the user to select a language.
    - Sets the selected language in the settings.
    - Checks if updating the CRKN database is allowed, and if so, asks the user if they want to update it.
    - Displays the welcome page widget.
- If the settings file exists:
    - Displays the start screen widget.
    - Checks if updating the CRKN database is allowed, and if so, asks the user if they want to update it.

Execution:

- If the script is run directly (not imported as a module), it calls the main function.

Overall, this script initializes the application, manages settings and GUI components, and interacts with the user to set preferences and update data.