**Programmer Documentation**

* **Introduction**

This project aims to develop a website using Python-Wedgit and HTML, designed to manage OnShape projects efficiently. The website provides users with the ability to view statistics based on data retrieved from OnShape, with customizable filters according to user preferences. The data, formatted as JSON, is uploaded to the project and processed to generate meaningful insights.

The code is written, organized, and compiled using Google Colab, which provides an accessible and collaborative environment for development.

**Interesting Parts:**

**Files :** HW3\_Cloud.ipynb – Main Code File

The file splitted to multiple cells , each cell will explained ahead .

* **Establish firebase connection** and **Retrieve a json file from firebase**

This code connects to a Firebase Realtime Database to retrieve and filter JSON data based on a given file name. The `fetch\_data\_by\_file\_name` function extracts the relevant data associated with the file, removing any entries that do not contain the keyword 'Document'. The `prep\_data` function further processes this data by converting it into a Pandas DataFrame, specifically parsing the 'Time' field into a datetime format.

* **PageBase Class**

The `PageBase` class serves as a foundational template for creating pages within a user interface using widgets. It initializes a vertical box (`VBox`) container to hold the page's content and provides basic methods to show or hide this content. The `create\_widgets` method is meant to be implemented by subclasses, allowing each page to define its specific widgets. Initially, the content is hidden to allow for controlled display based on user interaction or other conditions.

* **Main Page class**

The `MainPage` class is an extension of the `PageBase` class and serves as a user interface page designed to display and interact with project-related data. Upon initialization, it sets up various widgets and layouts using the `create\_widgets()` method, including dropdowns for selecting years (`year\_dropdown` and `project\_dropdown`), a grid layout for organizing visual outputs, and an HTML widget for displaying a header. The `getEmployees()` method retrieves and formats employee statistics, such as the number of actions and projects associated with each employee, which are displayed on the page. The class provides methods like `GraphTotalAction()` and `GraphActionsPerProjects()` to generate and display graphs that visualize total actions per month and actions per project for a selected year. These graphs are dynamically updated based on user selections in the dropdown menus. The `update\_main\_info()` method populates the page with overall statistics, including the total number of projects, actions, and employees, giving users a comprehensive overview of the project data.

* **Project class**

The ProjectsPage class is designed to display detailed project information and visualize project-specific data through various interactive charts and widgets. Upon initialization, the create\_widgets() method sets up the interface components, including a dropdown menu for selecting a project (project\_dropdown), a title widget (project\_title), and a grid layout (grid) that organizes different output widgets for visual content.

When a project is selected from the dropdown, the on\_project\_change() method is triggered, updating the project title and calling update\_project\_stats() to display basic statistics like the number of employees and actions associated with the selected project. The method also dynamically generates and displays several charts, such as a bar chart (create\_chartjs\_bar\_chart()) showing the number of actions per tab, a pie chart (create\_chartjs\_pie\_chart()) illustrating the distribution of user actions, a line chart (create\_chartjs\_line\_chart()) that tracks modeling time over dates, and an operations bar chart (create\_operations\_bar\_chart()) that visualizes the frequency of various operations like delete, undo, and edit actions.

Data for these visualizations is fetched and processed through specific methods like get\_operations\_data(), which filters and groups the data based on the project name to calculate the occurrences of different actions. Labels and data for the charts are prepared using get\_labels() and get\_data() methods, which tailor the information to fit the specific type of chart being displayed. The resulting charts are embedded within styled HTML containers for a polished visual presentation.

* **Employee Page Class**

The `EmployeesPage` class is designed to display and analyze user-specific activity within OnShape, providing detailed insights through interactive widgets and visualizations. Upon initialization, the `create\_widgets()` method sets up the user interface, which includes dropdown menus for selecting the year, month, and user (`year\_dropdown`, `month\_dropdown`, `user\_dropdown`), as well as an output area for displaying results and visual content. The method also creates layout containers for data summary boxes and a heatmap, organizing the interface for easy navigation and interaction.

When a user, year, and month are selected, the `filter\_data()` method is triggered, filtering the data based on these selections. It then counts and categorizes user actions—such as creations, edits, and deletions—by project using a grouped DataFrame and displays the results in a table alongside a bar chart that shows the distribution of these actions by day. The method leverages specific functions like `countCreatForUser()`, `countDeletsForUser()`, and `countAddOrInsertForUser()` to tally various types of actions for the selected user.

The class also includes dynamic updates for the user-specific data and visual elements. The `update\_student\_name()` method adjusts the displayed name and calls `update\_data\_items()` to refresh the data summary boxes with metrics like the total number of projects (`countProjectsForUser()`) and total modeling time (`calcTime()`). Additionally, the `update\_heatmap()` method generates a heatmap to visualize the user’s activity patterns across different days of the week and hours of the day, providing a comprehensive view of when the user is most active. These components work together to offer a detailed, interactive analysis of user activity in a visually organized manner.

Also , this code visualizes user activity in the OnShape environment through a heatmap. It starts by converting raw data into a DataFrame and extracting the day of the week and hour of the day from timestamps. The data is grouped by user, day, and hour to count actions, and a pivot table is created for each user to prepare the data for visualization. The `plot\_heatmap()` function then uses seaborn to generate a heatmap, showing action frequency across days and hours. A dropdown widget allows users to select different users, updating the heatmap interactively. Another graph is The category graph provide a visual representation of user activity across three distinct categories: basic document operations, adding or deleting content, and editing or modifying content. The `count\_categories()` function is responsible for generating these graphs by first filtering the dataset for a specific user and then aggregating the number of actions performed in each category over time. The data is plotted on a time series graph, with each line representing one of the three categories. This allows users to easily observe trends and patterns in their activity, such as increases or decreases in certain types of tasks over a given period. The visual distinction between the categories, highlighted by different colors, offers a clear and insightful overview of the user's work habits, enabling a deeper understanding of their focus and productivity within each category.

* **Notification page class**

The `NotificationsPage` class is designed to analyze activity data and generate notifications based on specific scenarios to alert managers about potential issues. Upon initialization, the `create\_widgets()` method sets up the basic structure of the page and processes the activity data by converting string timestamps to datetime objects and sorting them chronologically.

The class includes several functions to detect specific cases:

- `detect\_case\_1` identifies when two different users are working simultaneously on the same project tab within a short time window, which might indicate a conflict.

- `detect\_case\_2` tracks when a user performs multiple undo/cancel/delete actions within five minutes, suggesting they might need assistance.

- `detect\_case\_3` checks for projects that haven't been worked on for more than two weeks, indicating possible neglect.

- `detect\_case\_4` monitors projects with frequent undo/cancel/delete actions, which could point to difficulties in the project.

The `combine\_notifications()` function aggregates notifications from all the cases, sorts them by time, and filters out redundant notifications occurring within an hour of each other. Finally, the notifications are displayed on the page as HTML widgets, each formatted to show the notification's title, description, and timestamp. This approach allows for real-time monitoring and timely intervention to address potential project management issues.

* **Upload Json page class**

The `UploadJsonPage` class facilitates the uploading and management of JSON files within the application, integrating with Firebase for file storage and retrieval. The `create\_widgets()` method sets up the page layout, including an upload widget for new JSON files, a dropdown for selecting existing files from Firebase, a button to set the chosen file as the current dataset, and an output area for messages. When a user uploads a JSON file, the `on\_upload\_change()` method validates the content and handles updating or uploading the file to Firebase, with options to update if the file already exists. The `on\_set\_button\_click()` method sets the selected file from the dropdown as the active dataset, loading it into global variables and refreshing the dashboard display. The `get\_existing\_json\_files()` method retrieves the list of available JSON files from Firebase, ensuring efficient data management by allowing users to easily upload new datasets or update and select existing ones. This class ensures that the application's data remains current and accessible.

* **ChatBot page class**

**This Part splitted to two parts (Logic and front end)**

**ChatBot Logic** : The chatbot logic is designed to handle specific user queries related to project data and provide relevant responses. It uses the Natural Language Toolkit (NLTK) for text processing and pattern matching. The `generate\_user\_actions\_response()` function identifies the top 5 actions performed by a specified user, while `generate\_documents\_accessed\_response()` lists all documents accessed by that user. The `generate\_tab\_access\_response()` function determines which user accessed a specified tab the most, and `generate\_least\_used\_tab\_response()` identifies the least used tab by a specific user. The `custom\_chatbot()` function integrates these responses with a default fallback for unsupported questions. The chatbot also processes data to calculate statistics like the total number of actions, unique users, and frequently accessed documents. It utilizes predefined patterns and responses to handle common queries and dynamically processes specific requests based on user input.

**Chatbot Frontend** : The `ChatBotPage` class constructs the user interface for the chatbot. It includes a title, a dropdown menu with suggested queries, a text input field for user messages, and a button to send messages. The `create\_widgets()` method initializes these components and arranges them vertically. The `on\_suggestion\_select()` method allows users to select suggestions from the dropdown, automatically populating the chat input field with the chosen suggestion. The `on\_send\_click()` method processes the user's message, generates a response using the `process\_input()` function, and displays both the user message and the chatbot response in a styled chat bubble format. The `display\_message()` method handles the visual presentation of messages, with different styles for user and bot messages. The `show()` and `hide()` methods control the visibility of the chat interface, making it adaptable for integration into the larger project dashboard.

* **Collaborations Page Class**

The `CollaborationsPage` class is designed to visualize student contributions to different documents through interactive charts. Upon initialization, it creates a dropdown menu populated with unique document names and an output area for displaying a stacked bar graph. The `on\_document\_change()` method is triggered when a user selects a document from the dropdown, which in turn calls `update\_graph()` to filter data based on the selected document and prepare it for visualization. In `update\_graph()`, data is processed to count contributions per tab and user, normalized to show percentages, and plotted as a stacked bar graph, showcasing each student's contributions across different tabs in the selected document.

* **Main Wrapper Class**

The `DashboardWidget` class serves as the main controller for the project's graphical user interface, managing the navigation and display of various pages. It initializes essential widgets, including navigation buttons and page instances, and applies custom CSS for styling. The navigation buttons (`ToggleButtons`) facilitate switching between different pages such as "Main," "Projects," "Employees," "Notifications," "Upload json," "ChatBot," and "Collaborations." Each page is an instance of its corresponding class (e.g., `MainPage`, `ProjectsPage`). The `create\_header()` method dynamically generates a header with the current file title and user information. The `layout\_widgets()` method organizes the navigation buttons and page content vertically, with a top bar for navigation and a main content area for displaying the selected page. The `setup\_navigation()` method configures event handling for navigation, while `on\_nav\_change()` determines which page to display based on user selection. The `show\_page()` method hides all pages and displays only the selected one. The `display()` method renders the dashboard layout, and `refresh\_dashboard()` updates it by clearing the output and reinitializing the widget. This class effectively manages and controls the overall GUI of the project, making it the central hub for user interaction.

**Design pattern**

1. **Model-View-Controller (MVC) Pattern**

* **Controller:** The DashboardWidget class acts as a controller that manages the navigation between different pages (Main, Projects, Employees, etc.). It handles the logic for which page to display based on user input.
* **View:** Each page, like MainPage, ProjectsPage, and ChatBotPage, acts as a view that manages its own widgets and layout. These views are responsible for rendering the user interface components.
* **Model:** Although not explicitly shown in the provided code, the data structure (which seems to be a collection of user actions, documents, etc.) represents the model. The model would be the source of truth for data that the views and controllers work with.

1. **Factory Method Pattern**

The create\_widgets() method in the DashboardWidget class and its related page classes (MainPage, ProjectsPage, etc.) can be seen as following a Factory Method pattern. Each of these methods is responsible for instantiating and configuring the widgets needed for each specific page. This pattern is beneficial for separating the creation logic of objects from their usage.