Local Markdown wiki and editor: OwnWiki

RITTWICK BHABAK, Indian Institute of Technology, Delhi, India
SAGAR AGRAWAL, Indian Institute of Technology, Delhi, India
MANIK JAIN, Indian Institute of Technology, Delhi, India

This is a beginner system architecture project where a **Local Markdown Wiki and Editor** is built. The application is divided into components and the components are loosely coupled and there is a central authority which manage all of the on/off's of the components. In this application all of the markdown files are residing inside the application but it is so loosely coupled that this can be easily connected to some cloud database.

Additional Key Words and Phrases: System Architecture, GUI, Version Control, Project Documentation

1 INTRODUCTION

The application is built in python programming language as there are a lot of materials available how to build GUI applications in python. Although the underling architecture is independent of the programming language. The architecture which is followed can be implemented by using any programming language and it is also independent of the platform and operating system. The layers and the components inside the layers are loosely coupled. There is a central authority (here it is named 'state') keeps track of what is visible on the screen and also helps in data flow. When user changes from one screen to another screen the process happens through the state.

2 ARCHITECTURE AND DATA FLOW

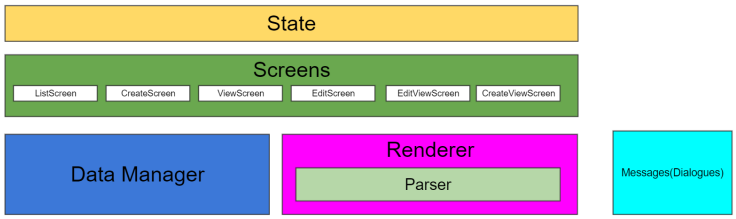

Fig. 1. System Architecture

In our application there are several screens namely: ListScreen, ViewScreen, CreateScreen, EditViewScreen and CreateViewScreen. As we see in the diagram, there is a central authority (here it is called 'State') keeps tracks of which screen is visible. And when some content has to be fetched it is done the 'Data Manager'. Any screen can directly call and receive result from data manager. And when a screen has to show some formatted content, it contacts with the Renderer which in turn contacts with the Parser.

Authors' addresses: Rittwick Bhabak, Indian Institute of Technology, Delhi, Delhi, India, mcs222054@iid.ac.in; Sagar Agrawal, Indian Institute of Technology, Delhi, Delhi, India, mcs222065@iid.ac.in; Manik Jain, Indian Institute of Technology, Delhi, Delhi, India, mcs222832@iid.ac.in.

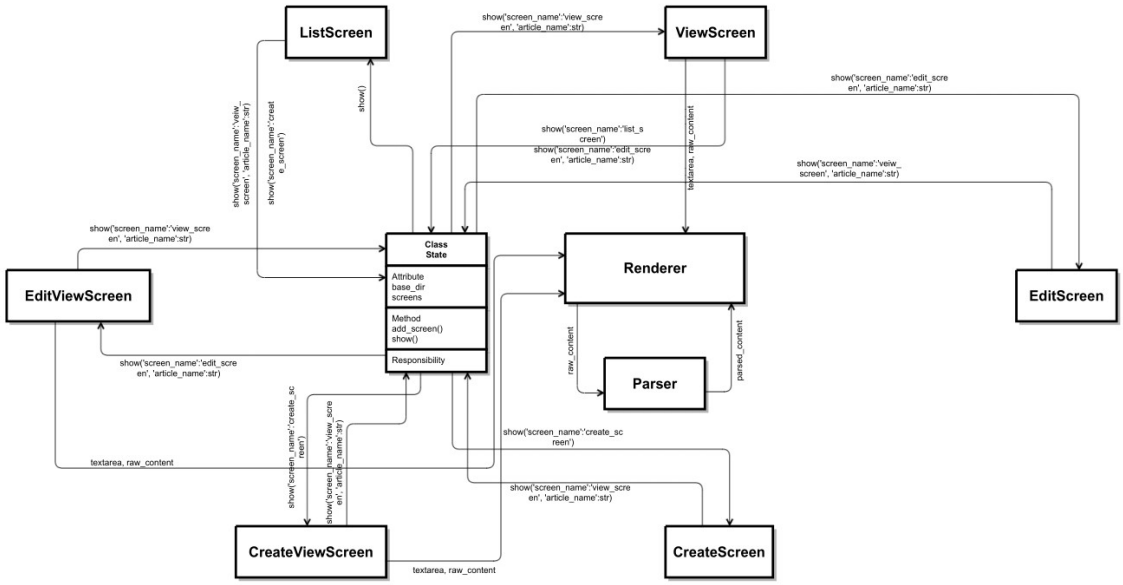

Fig. 2. Data flow

3 INTERNAL MODULES

There are mainly 7 internal modules in the application:

3.1 screens.py

Screen is a abstract base class. Every Screen has to implement this Screen abstract class. This class sets the basic element of a screen namely root, title, heading. Now from screen **ListScreen**, **CreateScreen** and **ViewScreen** is created. It is observed that **EditScreen** has a lot of things common with **CreateScreen** and has to implement a few things, that's why **EditScreen** is extended from **CreateScreen**.

As it is required to implement live preview which will contain similar functionalities with View Screen, to build **CreateViewScreen** and **EditVeiwScreen** multiple inheritance is used and so, **CreateViewScreen** is built using **CreateScreen** and **ViewScreen**, **EditViewScreen** is built using **EditScreen** and **ViewScreen**

- `List Screen`:It is the homepage that consists a list of articles.
- `View Screen`:It contains the complete articles as we click on any article.
- `Edit Screen`:This is the place where we edit our articles.

All of the three screens have the ability to-

- show themselves
- hide themselves
- can decide how the elements are placed in the screen (front-end)

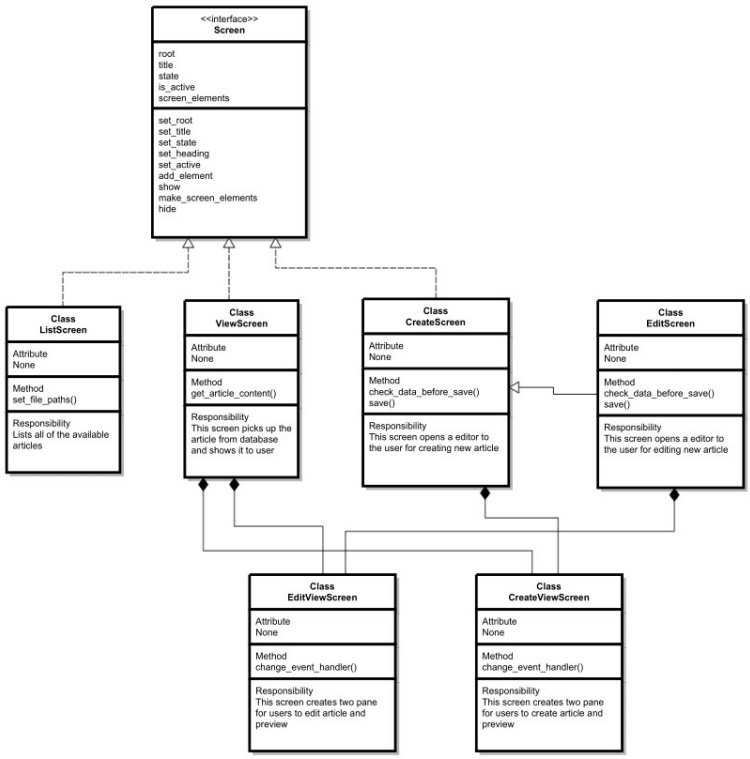

Fig. 3. Class Dependencies

3.2 state.py

The **State** class object holds all of the information and helps in data passing. For example in ListScreen if user clicks on the title of an article, then ListScreen sends the article name to state and tells that the particular article has to be shown.

3.3 renderer.py

In **ViewScreen** the markdown content has to be parsed and rendered. The Renderer class object does the job. It attaches the parsed content to the screen.

3.4 parsers.py

To show the markdown content in parsed text, the content has to be parsed. The different parser classes goes through the content several times and output the parsed content. For example,

- input: This is a **bold** text.
  output: [{'char': 'T'}, {'char': 'h'}, {'char': 'i'}, {'char': 's'}, {'char': ' '}, {'char': 'i'}, {'char': 's'}, {'char': ' '}, {'char': 'a'}, {'char': ' '}, **{'char': 'b', 'bold': True}, {'char': 'o', 'bold': True}, {'char': 'l', 'bold': True}, {'char': 'd', 'bold': True}**, {'char': ' '}, {'char': 't'}, {'char': 'e'}, {'char': 'x'}, {'char': 't'}, {'char': '.'}]

- input: This is _underlined_.
  output: [{'char': 'T'}, {'char': 'h'}, {'char': 'i'}, {'char': 's'}, {'char': ' '}, {'char': 'i'}, {'char': 's'}, {'char': ' '}, **{'char': 'u', 'underline': True}, {'char': 'n', 'underline': True}, {'char': 'd', 'underline': True}, {'char': 'e', 'underline': True}, {'char': 'r', 'underline': True}, {'char': 'l', 'underline': True}, {'char': 'i', 'underline': True}, {'char': 'n', 'underline': True}, {'char': 'e', 'underline': True}, {'char': 'd', 'underline': True}**, {'char': '.'}, {'char': ' '}]
- input: This is *italic*.
  output: [{'char': 'T'}, {'char': 'h'}, {'char': 'i'}, {'char': 's'}, {'char': ' '}, {'char': 'i'}, {'char': 's'}, {'char': ' '}, **{'char': 'i', 'italic': True}, {'char': 't', 'italic': True}, {'char': 'a', 'italic': True}, {'char': 'l', 'italic': True}, {'char': 'i', 'italic': True}, {'char': 'c', 'italic': True}**, {'char': '.'}]

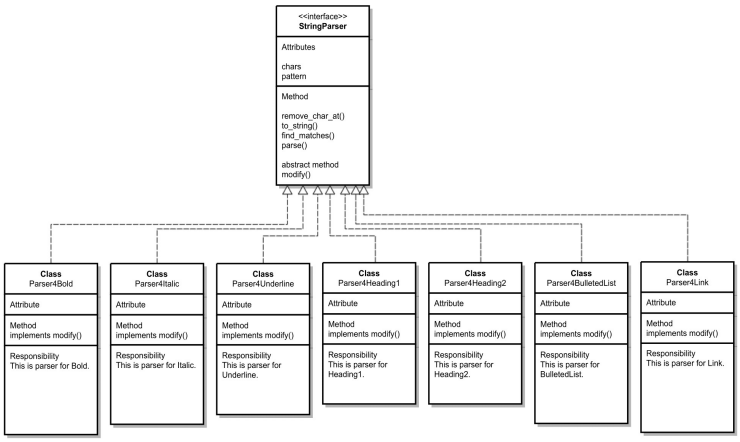

Fig. 4. Parsers

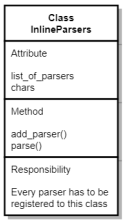

Fig. 5. Inline Parser Class

Now all of these parsers have to registered to **Inlineparsers** class. and the parse method of this module will build a inlineparser object and passes the sting through all of the registered parsers and returns the parsed list of characters.

3.5 messages.py

To show useful messages like 'an article name can not be blank', 'an article with the same name already exists', 'an article with the blank content is tried to be saved', dialogue boxes are used. This module helps to do that job.

3.6 data_manager.py

All the data read and writes are done by this module. Before taking any action this module also checks if the action can be preformed (if user tries to save two article with same name etc.). Currently all of the markdown files are residing inside the application but in future this can be easily integrated to some cloud storage. The other components of this application does not have any relation with how data read and write happens.

3.7 app.py

This is the entry point of the application. In this module all of the screens, state object are actually created and on startup of the application the listscreen is shown. From there user can navigate various screens.

4 EXTERNAL MODULES

Several external modules are used in the application:

- `os`: To get the articles and their content os is used.
- `tkinter`: Tkinter is used to make the GUI
- `sphinx`: Sphinx is used to make project documentation
- `re`: Python's Regular expression package is used to parse the content
- `datetime`: To create random id

5 SCREENSHOTS OF GUI

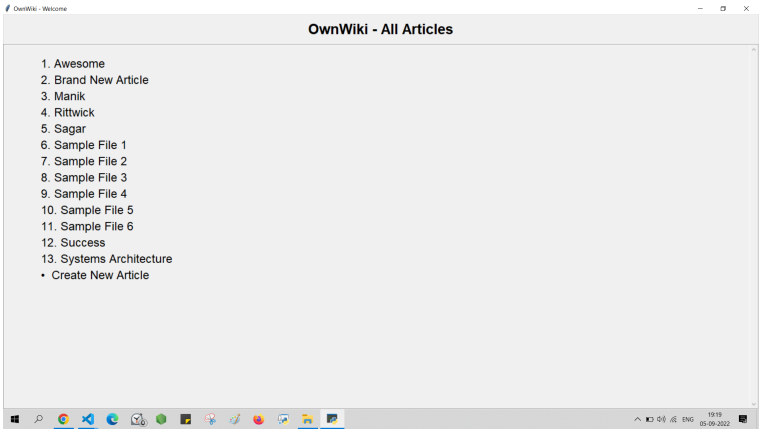

Fig. 6. Showing all available articles

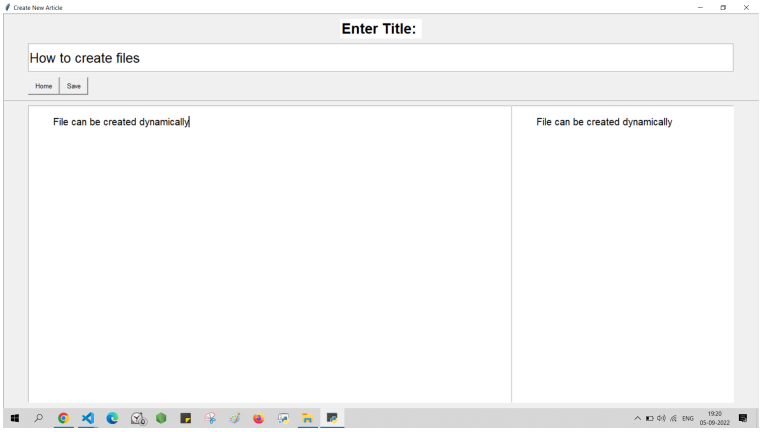

Fig. 7. Creating New Article with live preview

Fig. 8. Showing an article

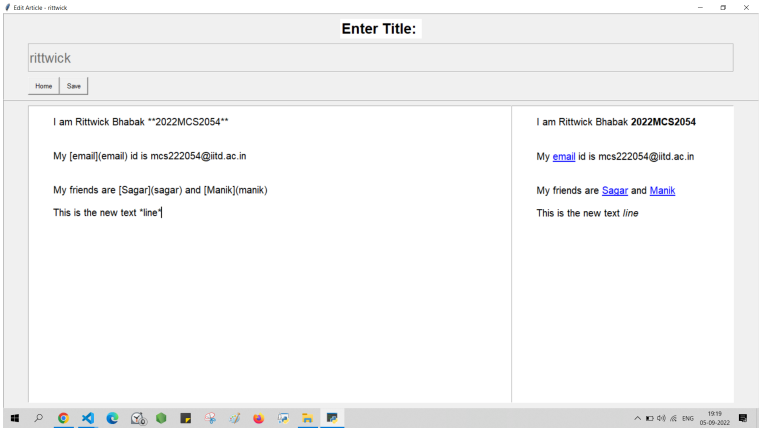

Fig. 9. Editing an Article with live preview

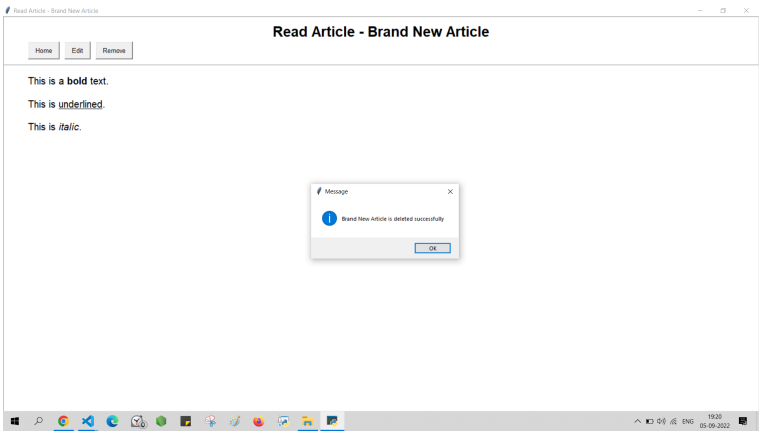

Fig. 10. Deleting an Article

6 REQUIREMENTS SATISFIED

- Version Control Using github.com. The link to git repository: https://github.com/rittwickBhabak/OwnWiki
- Project documentation using Sphinx Library

7 FURTHER SCOPE

- When user is using the live preview, then after user make some changes in the editor, after a couple of seconds the change is reflected in the preview pane, This lag can be reduced using some multi-threading.
- In the live preview screen, the editor width and the preview pane width is unequal, which does not looks great. This can be corrected.

8 CONTRIBUTION BY DIFFERENT GROUP MEMBERS

- `Rittwick Bhabak`: Contributed in making the architecture and has build the 7 internal modules and made the project report.
- `Sagar Agrawal`: Contributed in making the project documentation and helped in writing the project report.
- `Manik Jain`: Has been a supporting arm and during the project has learnt new concepts, which he will be utilising in forthcoming projects.

REFERENCES

[Chaa] Code With Harry Youtube Channel. *Python GUI: Tkinter Tutorial In Hindi 2019*. URL: https://www.youtube.com/playlist?list=PLu0W_9lII9ajLcqRcj4PoEihkukF_OTzA.

[Chab] Codemy.com YouTube Channel. *Python GUI's With TKinter*. URL: https://www.youtube.com/playlist?list=PLCC34OHNcOtoC6GglhF3ncJ5rLwQrLGnV.

[fig] Stackoverlow user figbeam. *stackoverflow.com*. URL: https://stackoverflow.com/questions/50327234/adding-link-to-text-in-text-widget-in-tkinter.

[Swe] Al Sweigart. *Automate the Boring Stuff with Python*. URL: https://automatetheboringstuff.com/.

ACKNOWLEDGMENTS

We would like to express our special thanks of gratitude to my teacher Rahul Narain sir as well as our TA and CodeWithHarry[Chaa] YouTube Channel, Codemy.com[Chab] YouTube channel, AI Swiget[Swe] and Bhagath Mamindlapelly YouTube channel who helped us to do this wonderful project on the topic Local Markdown wiki and editor, as a result we learnt so many new things. We are really thankful to them.

A ONLINE RESOURCES

To implement the interlinking fuctionality we have used a class 'HyperlinkManager' which we have taken from stackoverflow.com[fig]

Some articles like 'Awesome', 'System Architecture' are shown in the app are taken from wikipedia and populated after some minor changes in the application as dummy article.