Fedora Wiki Formatting Design Doc

Authors: Geidel Solivan Johnny Castillo Kevin Moreno

Geidel Solivan (geidelxavier@google.com)

Johnny Castillo (johnnycastillo@google.com)

Kevin Moreno (kjmtxep@google.com)

Background:

The issue on the table: the wiki page should be clearly formatted in such a way that users can easily access, read, and edit the contents on the page. We know this feature is vital because long paragraphs can promote page skimming and lead to misinformation.

Proposal:

To ensure that the user's need for readability is met, we will use markdown language to format the information clearly. More specifically, label the data as needed with header sections, paragraph forms, line breaks, subheadings, hyperlinks, etc. Using these categories will allow greater control over the details, furthermore, allowing an organized structure of the information so that the material displayed is comprehensible.

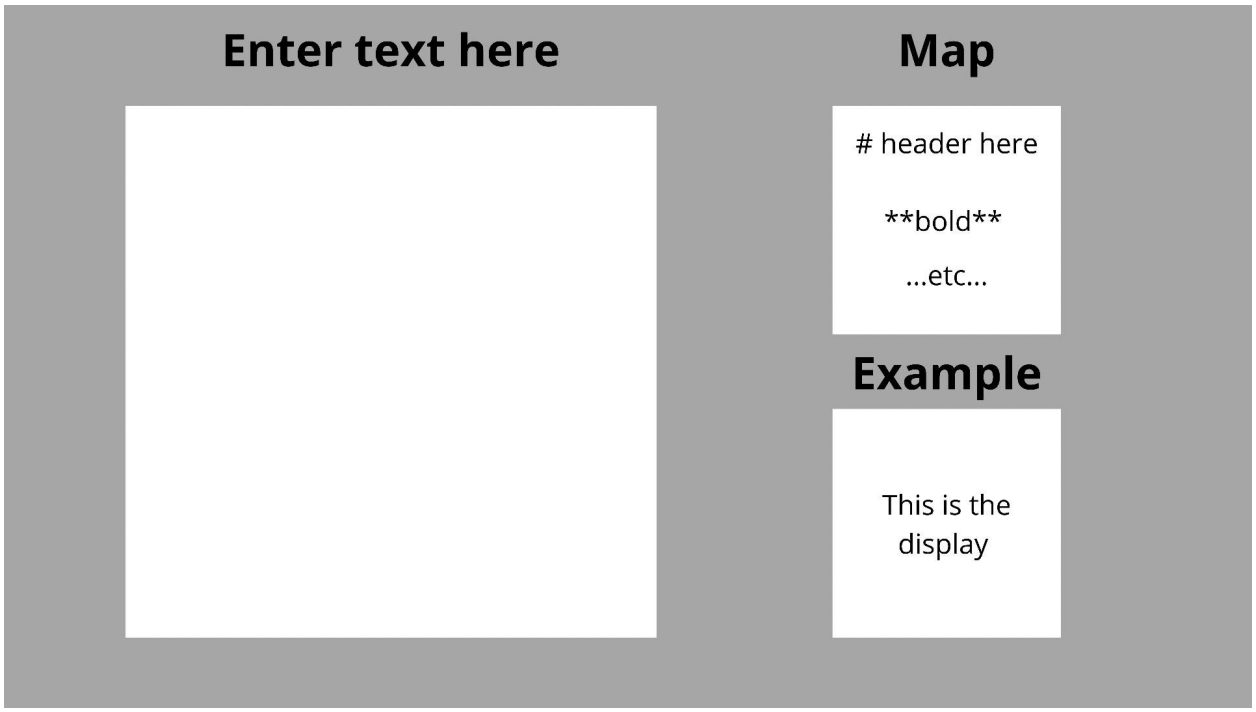
Beginning with the front page, users will be able to search for, edit or view featured googler information. Users will be able to also view information about the wiki itself, describing our intentions, favorite techx running gags and much more. This is achieved through the use of our templates. The idea being that templates can be reused and adapted to different page needs - not all pages may require the same modules.

Implementation details:

The general idea behind the implementation is to retrieve the information inputted by the user and how they want this to be shown. Once obtained, this data will be parsed to determine if it is valid and how it is intended to be displayed. Finally, the contents will be converted to the viewable format.

I. Data/Information retrieval (Input)

To obtain the contents that want to be added or edited into the page, we will use a text box for the user to input information which we will convert using a markdown language. This text box could potentially have some pre-written tags to make adding information easier. Initially, this input will be raw text data which will be passed to the parser. Additional to the text box, a map of tags and example box will be shown for users to know how to categorize their information. A header in markdown is denoted by "#Header" and converted to the proper <h><\h>, where the number of "pound" symbols indicates the level of header. For paragraphs the user simply separates them by breaking text with an empty line. Links are easily created in the form of [Click here](www.google.com) where the brackets indicate the text shown and parenthesis is the actual URL. Presented below is a general idea of how the page with all the input elements would look like.



II. Parsing (using Marko API)

Marko is an external library developed for converting markdown language to HTML. Since this library is external, the inner workings are a little bit of a mystery, however according to the documentation: Marko library parses through the given text and can then generate to an html, or print in console. The library also is designed around extensibility, allowing for us to extend and define behaviours for certain specific needs. This will provide us with the functionality to

read the raw inputted text and parse the markdown elements. Everything must be handled in the form of the Markdown language which the library then converts into the intended text.

III. HTML display (Output)

As a final step, we intend to take the generated parsed elements and turn them into html for display. When considering Marko, the library already has a CLI for creating an html file from the text. Nevertheless, suppose that marko for whatever reason ends up not being desirable. The idea would be to create a parser that can clearly go through text and add the appropriate tags for an html. Then it would be a matter of mapping that newly converted text into the flask templates and jinja format so that there are no issues. Otherwise, Marko allows us to simply create an html file of the desired text input.

Security Considerations:

One way we intend to mitigate injections is by looking for the specificity of injection attacks. The users are to strictly write in markdown so that our conversion happens smoothly. If the text input is not correctly formatted, we will omit whatever part of the file is not formatted by checking the content of the input as well. Essentially double checking to make sure that no dangerous script injections happen by simply ignoring inputs that may do that. We will need to check for script and html tags in user inputs to mitigate any injection attacks.

Alternatives Considered:

Two alternatives that we considered were using html parser and making our own parser. We decided not to use html parser and use markdown instead simply because of the fact that html parser required the user to know some html to implement certain tags whereas markdown used simpler notation. As for making our own parser, we didn't quite stick with that due to the time frame we have in the course. It would take us a bit longer to get the parser to work as we intended for it to.

Resources:

- Markdown and Marko
  - https://pypi.org/project/marko/
  - https://marko-py.readthedocs.io/en/latest/extend.html
  - https://marko-py.readthedocs.io/en/latest/api.html#module-marko.parser
  - https://www.markdowntutorial.com/lesson/1/

- HTML
    - https://docs.python.org/3/library/html.parser.html#module-html.parser
    - https://www.digitalocean.com/community/tutorials/how-to-use-python-markdown-to-convert-markdown-text-to-html
    - https://pythonbasics.org/flask-tutorial-templates/
    - https://www.vistainfosec.com/blog/comprehensive-guide-on-html-injection/