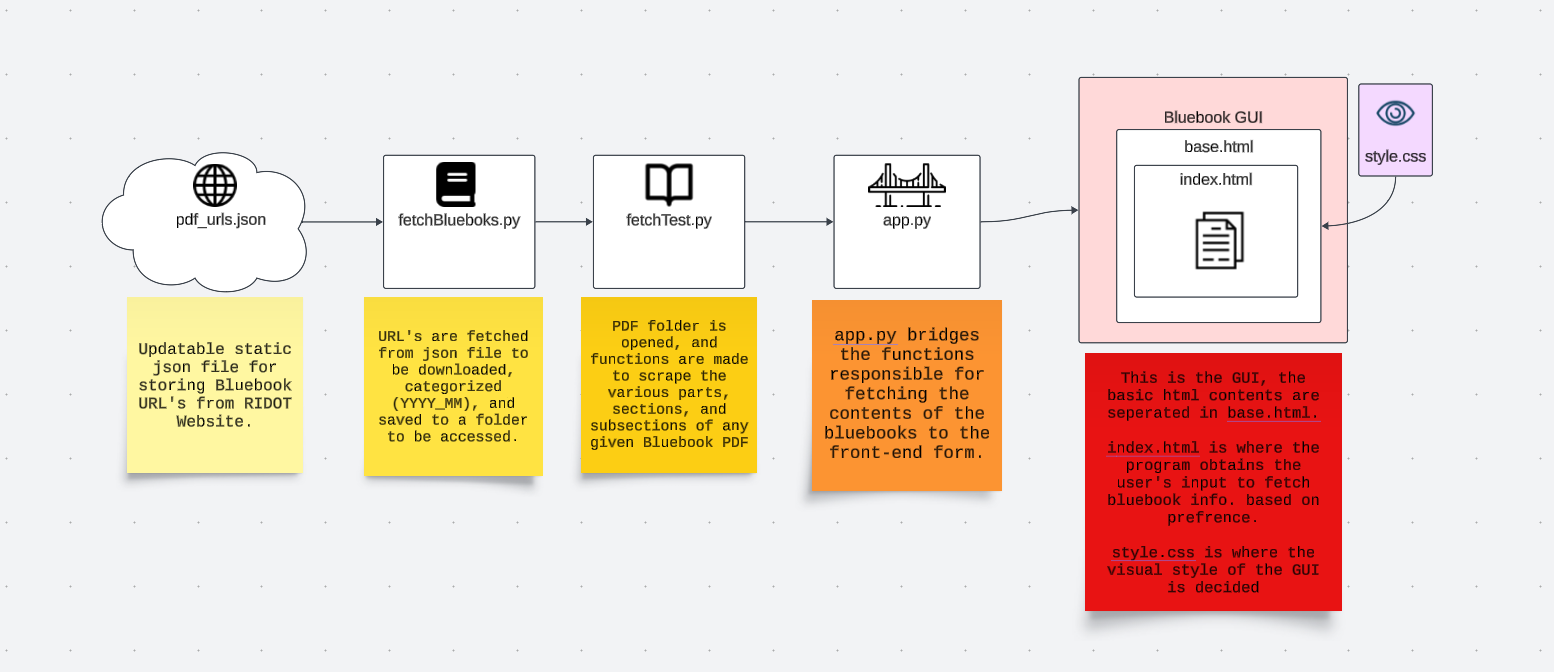
## **RIDOT BLUEBOOK INDEXING PROGRAM DOCUMENTATION PAGE**

## Main Description:

* The *RIDOT Bluebook Indexing Program* is designed to streamline access to both past and future editions of the RIDOT Bluebook PDFs. This program processes the information into an easy-to-use GUI, ensuring that retrieving data from any Bluebook year is convenient and efficient.

## Flowcharts:



**This Flowchart illustrates the purpose of each file when the application is in use.**

## Functionality:

* **Web Stack:** 
  + *Flask:*Provides web server and routing for application.
  + *Python:* Web functionalities are made with python scripts.
  + *HTML:* Front-End Templates used to structure GUI.
  + *CSS:* Front-End Templates are styled with CSS files.
  + *JavaScript:* client-side functionalities are coded in JavaScript.
* **PDF Fetch Functionality:**

*Running fetchBluebook.py:*

* **Script Structure:**
  + Loads PDF URLs from `pdf\_urls.json`
  + Creates a `bluebook\_pdfs` directory
  + Iterates through each URL in `pdf\_urls.json`
* **Script Functionality:**
  + **download\_pdf(url):** Sends a GET request to fetch the PDF. If successful, it extracts the filename from the URL and calls get\_year\_and\_month(filename) to get the year and month.
  + **get\_year\_and\_month(filename):** Splits the filename to return the year and month.
  + **download\_pdf(url):** Renames the file to `YYYY\_MM` format and saves it to the `bluebook\_pdfs` directory.
* **Content Extraction Functionality:**

*Running fetchTest.py:*

* + **Script Structure**
    - Determines the path for the `bluebook\_pdfs` directory on the local machine.
    - Checks if PDFs exist in the directory and begins execution if they do.
    - Opens the `bluebook\_pdfs` directory and selects a PDF file based on its index.
    - Prints the return value of `extract\_part\_titles(pdf\_path)`, displaying the part titles extracted from the selected PDF.
    - Selects the part corresponding to the selected index and prints the return value of `extract\_section\_titles(doc, toc, section\_index)`, which lists the section titles for that part.
    - Repeats the same process to print the subtopics for the selected section index.
    - Prints an error message if it can't find the PDFs or if it can't find subtopics for the specified section.
* **Script Functionality:**
  + - **Part Title Extraction:**
    - *extract\_part\_titles(pdf\_path)*
      * Take the path to the preferred PDF.
      * Creates an array to store part title strings.
      * Opens the PDF with the `fitz` library and extracts the table of contents (TOC).
      * Extracts strings from the TOC that begin with 'Part' followed by a number or letter.
      * Appends matching strings to the array.
      * Returns the array of part titles.
    - **Section Title Extraction**
    - *extract\_section\_titles(pdf\_path, part\_title)*
      * Takes in the PDF path and part title.
      * Initializes an empty list for section titles.
      * Opens the PDF using `fitz` and extracts the TOC.
      * Uses a regex pattern to match part titles.
      * Finds the index of the part title in the TOC.
      * Loops through TOC items until the next part is encountered.
      * If an item starts with "SECTION", it's considered a section title.
      * Calls `contains\_subsections(doc, toc, section\_index)` to check for subsections.
      * Adds section titles to the list, marking those without subsections as "[No Subsections]".
    - **Subsection Scanning:**
    - *contains\_subsections(doc, toc, section\_index)*
      * Takes the PDF document, TOC list, and section index.
      * Gets the section's starting page from the TOC.
      * Finds the last page of the section by checking the next part or section in the TOC.
      * Extracts text from each page within the section's range.
      * Uses a regex pattern to detect subsections.
      * Returns `True` if subsections are found, `False` otherwise.
* **Subtopic Title Extraction**
* *extract\_subtopic\_titles(pdf\_path, section\_number)*
  + Takes in the PDF path and section number.
  + Initializes an empty list for subtopic titles.
  + Opens the PDF using `fitz`.
  + Iterates through each page of the PDF:
    - *Extracts text from the current page and splits it into lines.*
    - *Combines lines where the section number and title are separated by a newline.*
    - *Checks each line against the subsection pattern (e.g., section number followed by a title starting with a capital letter).*
    - *Starts collecting subtopics from `.01` onwards where the title starts with at least two capital letters.*
    - *Continues collecting subtopics if the title starts with at least two capital letters.*
  + Adds matching subtopic titles to the list.
  + Returns the list of subtopic titles.
    - **Content Extraction:**
    - *extract\_subtopic\_content(pdf\_path, section\_number, subtopic\_number)*
      * Inputs: PDF path, section number (e.g., 101), and subtopic number (e.g., 01).
      * Initialization: An empty string for the text and two regex patterns for the start and stop subtopics.
      * Error Handling: Implement a try-catch block for processing errors.
      * Processing: Loop through each page, matching lines with the starting regex for the subtopic.
        + Start Match: Skip the title and save subsequent lines into the string.
        + Stop Match: Check if the next subtopic matches the stop regex; if matched, break the loop.
      * Output: Return the accumulated text string.
* **User Form Functionality:**
  + Application Routing:
  + Front-End Form Requests:

## Constraints:

* Future Bluebook PDF Files must be formatted the same as the 2022-2024 Bluebook PDF’s.
* Future Bluebook PDF Files must be named when put on the website in this format: “Blue\_Book\_MM\_YYYY” for the program to extract the month and year from the PDF.
* Future Bluebook renditions URL’s must be included in the pdf\_urls.json file for the program to know where to find it.

## Dependencies:

* **Libraries:**
  + *Fitz (PyMuPDF):* fetchTest.py specifically utilizes the PyMuPDF Library to carry out PDF scanning functionalities.

## Security:

* **Availability:** There isn't any sensitive information being used, as all the data comes from publicly available sources. The bluebooks are scraped from the public RIDOT website. Additionally, the inputs for the program are predefined in dropdown menus, which further minimizes security concerns.
* **Integrity:** Since the program runs on the local device, it reduces vulnerabilities and potential exploits associated with database connections. Running locally mitigates risks such as SQL Injection attacks and other exploits, ensuring a more secure execution environment.

## Testing:

* **PDF Fetch Testing**
  + *fetchBluebook.py:*
* **Content Extraction Testing:**
  + *fetchTest.py:* The \_\_main\_\_ conditional of this file is where the functions used to extract the PDF content can be directly referenced and tested. \*See in comments where to adjust indexes that correspond to Part, Section, Subtopic, and Content modification\*.
* **Unit Testing**
  + *unitTest.py*
* **Integration Testing**
  + *seleniumTest.py*

## History/Changes:

* ***Bluebook Info. App* (First Version)*:***
  + This version of the application is the most experimental version of the application
    - Made a flow chart of how I wanted the app to flow initially
    - Tried different web stacks
    - Outlined the inputs and outputs
    - Made a final blueprint of how to approach the final version
* **RIDOT Bluebook Index (Final Version)**
  + This was the second and final version of the application. This version was where I chose which web stack to use, and implemented the final functionality of the web app.
    - Finalized flow chart
    - Decided on web stack
    - Created user input form
    - Developed pdf processing functions

## Issues:

* **Future Bluebook PDF Formatting Changes:**
  + **Impact of Format Changes:** If future bluebook renditions added to the website are formatted differently, even with updated links in the `pdf\_urls.json` file, the program's ability to process data and generate form options might be affected.
    - *Example of Format Variation:* For instance, the 2018 Bluebook is formatted differently from the bluebooks made from 2022-2024. To parse it similarly, specific exceptions are needed. Any future changes in formatting will require program updates to handle the new structures.
* **Implementation Location:**
  + **Hosting Considerations:** Determining where the web app will be accessed is crucial. Will it be integrated into an existing webpage or hosted independently? Understanding the hosting environment will help address any potential issues and stipulations for making the web app accessible.
* **Backend Requirements:**
  + **Current Setup:** Currently, the program runs locally on the user's device, which minimizes potential exploits or vulnerabilities and keeps the program straightforward to manage.
  + **Future Database Integration:** If a database is integrated in the future, additional security measures will be needed to prevent potential exploits and vulnerabilities. This will involve implementing robust security protocols to protect the system and user data.
* **Managing New Bluebook Versions:**
  + **Updating Process:** Clear procedures need to be established for implementing new bluebook versions. This includes updating the `pdf\_urls.json` file with new links, ensuring the program can handle any new formatting, and testing to confirm the accuracy of data extraction and form generation.

## Improvements:

* **Making a database to store future Bluebook PDF’s:** To achieve any of the other Improvements, this would need to be a step that will eventually need to be taken. Without a database, improving the system will become increasingly difficult to automate certain procedures, or make certain tasks easier to perform.
* **Dynamicity:** Making the program able to update itself when a new bluebook version comes out would be ideal, though it would be difficult to achieve, I believe it is a possible improvement to be able to implement.
* **Security improvements:** Security improvements will eventually need to be implemented whether a database is going to be integrated in this program or not. Wherever this webapp is hosted whether it be a webpage, or linked somewhere to use, or even downloaded, Security should be the main priority before using it for professional use.
* **Potential style changes:** This is an optional improvement that wont effect how the application functions, but if the style of the application should be changed for whatever reason to make it easier to view or match the styling of wherever its going to be accessed from this is also a viable potential improvement.
* **Recursive Formatting:** Having the fetched content be formatted in a more organized way is a much bigger task to be able to handle, more specifically if there aren’t any noticeable text patterns that can be easily modified that need to be modified for special cases (tables, graphs, images, etc.), you would need to make a recursive formatting function that will find those patterns, and if its unable to find those patterns it would need to be coded in a way that will make the patterns to be noticed for it to format the entire document of any specific format preferences. And it would also need to be able to be applied across all PDF documents that it has as data, (Having an AI machine compare the formatting of the PDF to the string of text before its displayed could be a potential solution.)