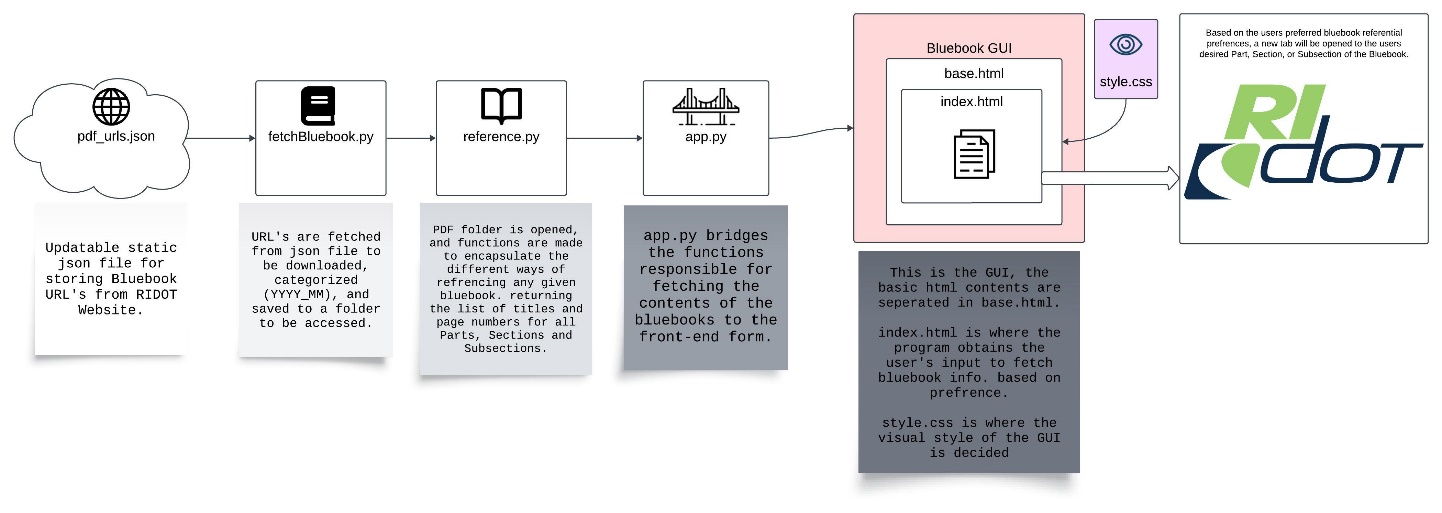
## **RIDOT BLUEBOOK INDEXING PROGRAM DOCUMENTATION PAGE**

## 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.
* **Referencing Functionality:**

*Running reference.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(pdf\_path)`, displaying the part titles, along with their corresponding page numbers extracted from the selected PDF.
    - Selects the part corresponding to the selected index and prints the return value of `extract\_section(doc, toc, section\_index)`, which lists the section titles and corresponding page numbers for that part (Filtering out the Sections with no content).
    - Repeats the same process to print the subtopics for the selected section index, printing out the subsection titles along with their corresponding page numbers.
    - 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(pdf\_path)*
      * **Purpose:** Extracts titles of parts from the PDF's table of contents (TOC).
      * **Arguments:**
        + `pdf\_path` (str): Path to the PDF file.
      * **Returns:**
        + **list:** A list of dictionaries containing part titles and their respective page numbers.
      * **Steps:**
        + Takes the path to the preferred PDF.
        + Creates a list to store dictionaries.
        + Opens the PDF with the `fitz` library and extracts the TOC.
        + Extracts strings from the TOC that begin with 'Part' followed by a number or letter.
        + Inserts matching string into the 'title' key in dictionaries.
        + Captures the page number from the TOC entry and stores it in the 'page\_number' value for each part.
        + Returns the list of dictionaries associating each part title with its corresponding page number.
    - **Section Title Extraction:**
      * *extract\_section(pdf\_path, part\_title)*
      * **Purpose:** Extracts section titles for a specified part from the PDF's table of contents (TOC).
      * **Arguments:**
        + `pdf\_path` (str): Path to the PDF file.
        + ‘part\_title` (str): The title of the part for which sections need to be extracted.
      * **Returns:**
        + **list:** A list of dictionaries containing section titles and their respective page numbers under the specified part.
      * **Steps:**
        + 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.
        + Identifies items starting with "SECTION" as section titles.
        + Calls `contains\_subsections(doc, toc, section\_index)` to check for subsections.
        + Adds section titles to the list, marking those without subsections as "[No Subsections]".
        + Returns the list of dictionaries containing section titles and their respective page numbers.
    - **Subsection Scanning:**
      * *contains\_subsections(doc, toc, section\_index)*
      * **Purpose**: Checks if a section has subsections by examining the table of contents (TOC) and PDF pages.
      * **Argument:**
        + `doc` (fitz.Document): The PDF document object.
        + `toc` (list): Table of contents of the PDF.
        + `section\_index` (int): Index of the current section in the TOC.
      * **Returns:**
        + **bool:** True if subsections are found, False otherwise.
      * **Steps:**
        + 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 (e.g., patterns like "1.1", "2.3", etc.).
        + Returns `True` if subsections are found within the section, otherwise returns `False`.
    - **Subsection Extraction:**
      * *Extract\_subsection(pdf\_path, section\_number)*
      * **Purpose**: Extracts the titles of subtopics for a specified section from the PDF.
      * **Arguments:**
        + `pdf\_path` (str): Path to the PDF file.
        + `section\_number` (str): The number of the section for which subtopic titles need to be extracted.
      * **Returns:**
        + **list:** A list of dictionaries containing subtopic titles and their page numbers.
      * **Steps:**
        + Opens the PDF file using `fitz`.
        + Iterates through each page of the PDF.
        + Loads text from each page and splits it into lines.
        + Combines lines where the section number and title are separated by a newline for better pattern matching.
        + Uses a regex pattern (`subsection\_pattern`) to match lines that represent subtopic titles under the specified section number.
        + Collects subtopic titles and their page numbers, starting from `.01` onwards where the title starts with at least two capital letters.
        + Returns a list of dictionaries containing subtopic titles and their corresponding page numbers.
* **User Form Functionality:**
  + **Application Routing (app.py)**
    - The `app.py` file defines routes and functionalities for a Flask web application that serves as a Bluebook reference tool. It includes:
    - **Setup Functions:** 
      * `check\_and\_fetch\_bluebooks()` ensures the existence of a directory for Bluebook PDFs and fetches them if necessary.
      * `setup\_application()` orchestrates this setup process.
    - **Routes:**
      * `/`: Renders an `index.html` template listing available Bluebook PDF files.
      * `/get\_part\_titles`: Retrieves and returns part titles from a selected PDF file using `extract\_part`.
      * `/get\_sections`: Fetches sections for a specified part in a PDF using `extract\_section`.
      * `/get\_subsections`: Retrieves subsections for a selected section in a PDF via `extract\_subsection`.
      * `/pdf\_urls.json`: Serves a JSON file (`pdf\_urls.json`) containing URLs to PDF files.
  + **HTML Structure** (index.html)
    - The `index.html` file is the main template for the Bluebook reference tool web application. It features:
    - **Dropdowns and Form:** Allows users to select a Bluebook PDF file and choose between referencing parts, sections, or subsections.
    - **Dynamic Content:** 
      * Lists available Bluebook PDF files retrieved from the server.
      * Utilizes JavaScript to populate dropdowns dynamically based on user selections.
  + **JavaScript Functionality (script.js)**
    - The `script.js` file provides client-side functionalities for the Bluebook reference tool:
    - **Event Handling:** 
      * Responds to user interactions such as selecting a PDF, choosing a reference type (part, section, or subsection), and submitting selections.
    - **AJAX Requests:** 
      * Retrieves data from the server asynchronously (`/get\_part\_titles`, `/get\_sections`, `/get\_subsections`, and `/pdf\_urls.json`) to populate dropdown menus dynamically.
    - **Form Validation:** 
      * Ensures that all necessary form fields are populated before enabling the submit button.
    - **PDF Link Handling:** 
      * Constructs and opens URLs to specific PDF pages based on user selections, utilizing data fetched from `pdf\_urls.json`.

These components collectively provide a user-friendly interface for accessing and navigating RIDOT Bluebook PDFs online.

## Constraints:

* Future Bluebook PDF Files must be formatted the same as the 2022-2024 Bluebook PDF’s. (This is due to the fact that the program heavily relies on the naming convention of the bluebook when fetching the information.)
* 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:**
  + This was the second version of the application. This version was where I chose which web stack to use and began implementing functionalities
    - **Made flow chart**
    - **Decided on web stack**
    - **Created user input form**
    - **Developed Developed processing functions**
* **RIDOT Bluebook Indexing Program Version 1:**
  + This was the program that would’ve been the final version of this program if I had gotten it to work, but it turned out later on that my original plan of scraping the PDF text directly and formatting it on the web page was very inefficient, and extremely hard to format, nonetheless, it was this version of the program that caused me to shift my approach and completely change how the finalized version of my app would function.
    - **Failed to finalize text scraping capabilities**
    - **Limited scalability**
    - **Failed to finalize text formatting capabilities**
    - **Unable to integrate compilations**
* **RIDOT Bluebook Indexing Program Version 2 (Final Version):**
  + It was this program that I was able to fully flesh out and scale up to standard, I recycled some of the work I did in my previous version that I found useful in order to optimize this program to maximum effeciency. In this program I was able to kill 5 birds with 1 stone by implementing the following functionalities:
    - **Accessibility to any Part, Section OR Subsection**
    - **Optimized code to reduce redundancy and increase speed**
    - **Resolved formatting issues by referencing the PDF file directly**
    - **Gave the program room for scalability to implement Compilations by directly referencing PDF Files.**
    - **Opened door for future bluebooks and compilations to be universally integratable.**

## Issues:

* **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.
* **Naming Constraints:**
  + **PDF Naming conventions**: when the PDF Link is added to the list of PDF File links, it has to follow the naming convention before it is uploaded to the RIDOT Website for it to be able to download the PDF to scrape the titles and know how to divide the PDF for referencing. If the Date is somewhere in the name of the file either in YYYY\_\_MM, MM\_YYYY or MM\_YY format it should still be able to work universally.

Improvements:

* **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.
* **Compilation Involvement**: The program would need to have a switch programmed somewhere that will allow the user to access future compilations. Since my program doesn’t take in any PDF Formatting considerations, this shouldn’t be difficult to do here is a basic concept:
  + **FEATURE IMPLEMENTATION SYNOPSIS:**  
    *if the year of the compilation matches the year of the bluebook, have an optional compilation checkbox appear next to the PDF Dropdown for that year, that when checked off, will replace the radio buttons and reference dropdowns with a single Compilation dropdown menu to select from, that upon submitting will open the link to that Compilation.*