***SYNOPSIS***



***SKYLON***

*BRINGING PYTHON TO THE WEB*

*BY ~ SUJAL SINGH*

**ACKNOWLEDGEMENT**

The success and final outcome of this project required a lot of

guidance and assistance from many people and we are privileged to

have got this all along the completion of our project. All that we

have done is only due to such supervision and assistance and we

would not forget to thank them.

We owe our deep gratitude to our Computer Science teacher Miss

Simrat Kaur, for believing in our capabilities and guided us all

along, till the completion of our project work by providing all the

necessary information for developing a good system. We take this

opportunity to express our deep sense of gratitude for her

invaluable guidance and constant encouragement, immense

motivation, which has sustained our efforts at all stages of this

project work.

**Certificate**

This is to certify that the project entitled

**SKYLON**

is the work of **SUJAL SINGH** in class **XII-D**

Roll No. **25**, school **St. Xavier’s School, Rohini** during the academic year 21/22, is prepared under the supervision and guidance of

**MRS. SIMRAT KAUR**

**⠀⠀⠀⠀⠀⠀⠀⠀⠀⠀⠀⠀⠀⠀⠀** **⠀⠀⠀⠀⠀⠀⠀⠀⠀⠀⠀⠀⠀⠀**

**Teacher’s Signature Student’s Signature**

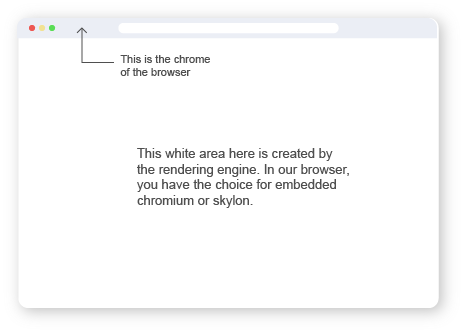
**INDEX**

|  |  |
| --- | --- |
| S. No | TOPIC |
|  | *Acknowledgement* |
|  | *Certificate* |
|  | *Introduction* |
|  | *Objective* |
|  | *Working* |
|  | *System Requirements* |
|  | *Python as front-end (WHY?)* |
|  | *Backend as MySQL (WHY?)* |
|  | *Dependencies* |
|  | *UDF* |
|  | *Glossary* |
|  | *Future Aspects* |
|  | *Bibliography* |

**INTRODUCTION**

The **World Wide Web**, developed by **Tim Berners Lee** in 1989 while working at **CERN**. Initially the web was meant to share documents between scientists of various universities. One couldn’t have imagined how much the web has evolved today. A software that is used to interact with the web is the **web browser**.

Web browsers are complex machinery, one could argue modern web browser’s development is more complex than an entire operating system. **Chromium** an open-source browser, has 35 million lines of code in it, compared with **Linux** kernel having 15 million lines combined with **GNU** operating system having another 14 adding up to only 29 million lines.



A web browser has primarily two components, one being the **chrome** of the browser that is tiny search bar and other related GUI components that sit on top of your browser covering less than 4-5% of the available screen area. The other part where the actual web pages are displayed is created by the second component called the **rendering engine**. The later component is the one that is difficult to develop.

The chrome of our browser has been developed with **PyQT5**, a cross platform GUI framework. The engine is a little more complicated, building an entire engine would be impossible, here’s a proof for this statement:

*Firefox has 21 million lines of code, if on an average each line has 20 words and the programmer was to type non-stop 24/7 at a constant speed of 50 words per minute, it would take 16 years to make a browser: (21,000,000 \* 20) / (50 \* 60 \* 24 \* 365) = ~16 years*

To solve this problem, we have created a new engine but maintained backwards compatibility by using the **chromium embedded framework**.

We provide the user with the choice to select whether they would like to use embedded chromium or our new engine. **CEF Python** was required to use the chromium embedded framework with python.

**OBJECTIVE**

The objective of this project is to create a web browser which replaces JavaScript with Python (it still maintains JavaScript support through CEF).

The objective might seem simple but the applications of such a project are quite a few. One could utilize this as a GUI framework for python, like Electron JS for JavaScript. Such an application allows a python developer to write code once and package and distribute the same code base to all desktop platforms and on the web too. The set of all use cases for the web is a subset of the use cases of our application.

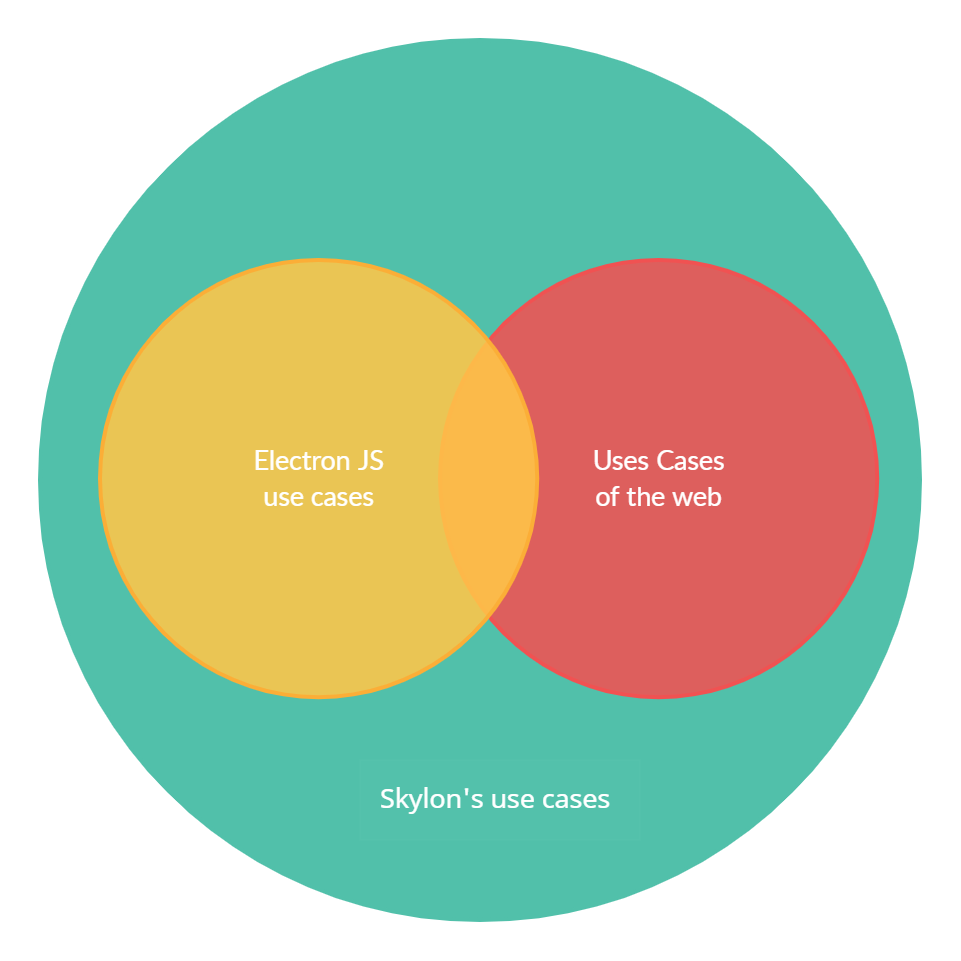


Figure 1: Use cases of Skylon

**WORKING**

Our browser has two operation modes one for chromium embedded and one for Skylon. The browser’s launch process is explained with a flow chart below:

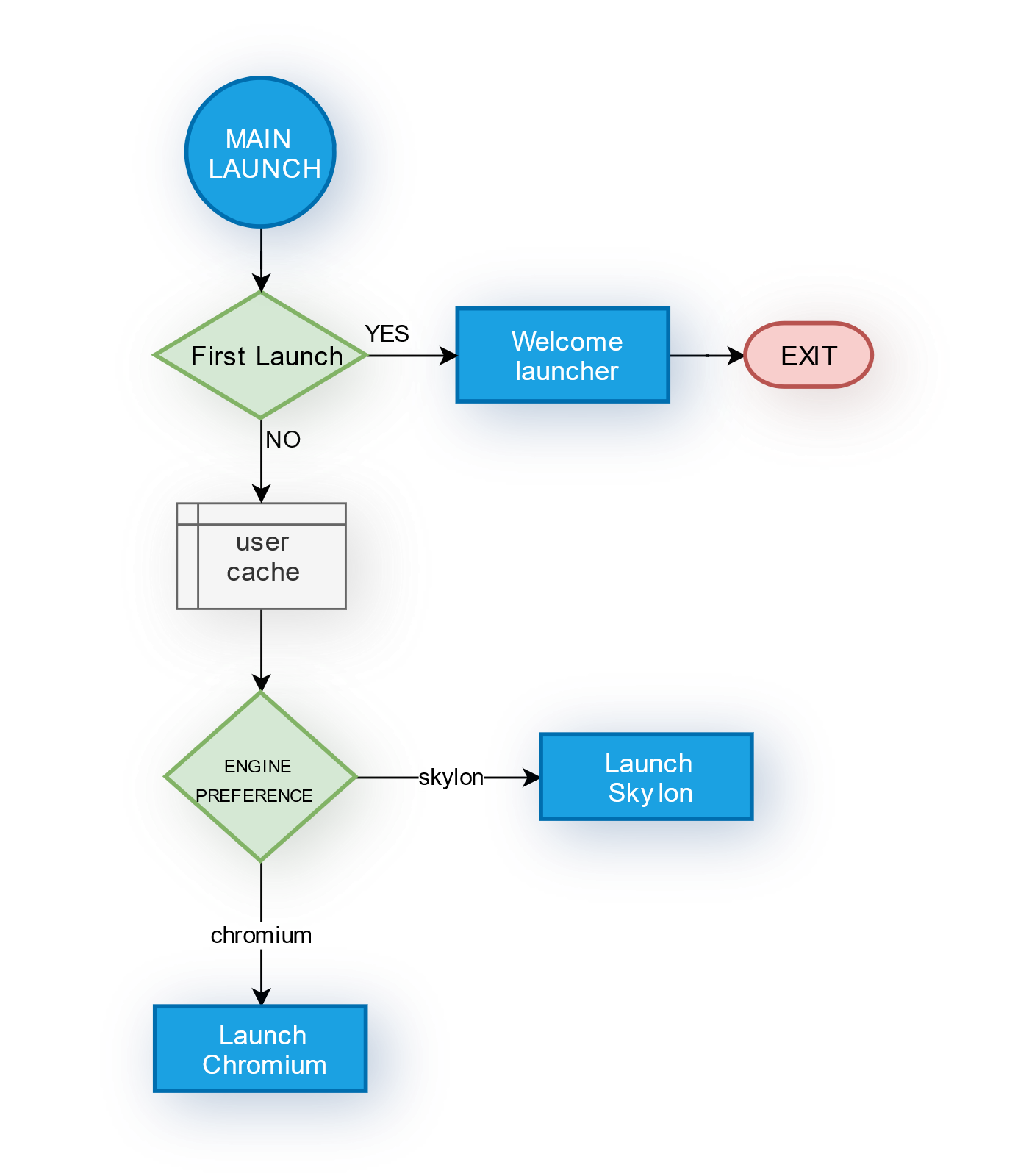


Figure 2: Main Launcher (entry-point)

The welcome launcher is a window created by CEF Python; it displays the webpage <https://sujalsinghx86.github.io/school-project-web/> as the GUI. The launcher also exposes some python functions to the website using CEF Python to get data entered into the input boxes. Then it connects to a real MySQL server hosted on AWS [here](skylon-users.ctq2ehobzwiu.ap-south-1.rds.amazonaws.com), storing the user’s browser preference, and if it already exists does not ask for the preference. It also caches the data locally using a binary file for quicker access.

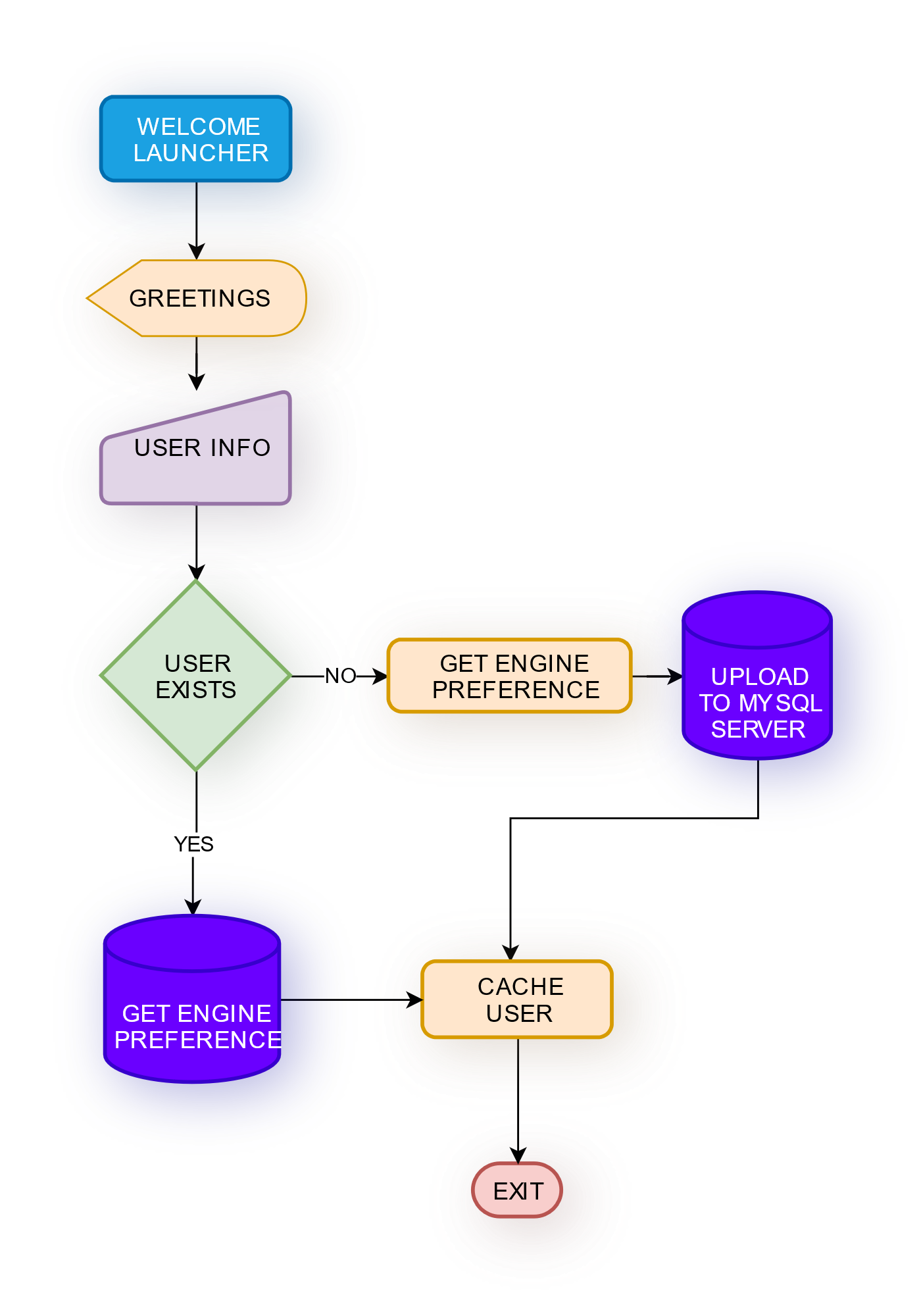


Figure 3: Welcome Launcher

Launching Skylon and Chromium are simple enough procedures, so no flow chart is required for them.

**How Skylon works?**

As mentioned earlier, browsers are complex and so here we explain only the features we have implemented in our browser.

Before one can understand how browsers work, one should be aware of the existence of **web standards**. Web Standards are in a way the **laws of the web**. All major browsers conform to these standards to avoid ambiguity in the web. The organization called **W3C (World Wide Web Consortium)**, founded by the creator of the web Tim Berners Lee. Although, HTML standards are being maintained by a separated body called **WHATWG**. Our browser also tries to conform to these standards to the best of our abilities.

Having the knowledge from these standards one can now figure out how to implement a rendering engine for a browser, our browser’s working can be demonstrated with some flowcharts:

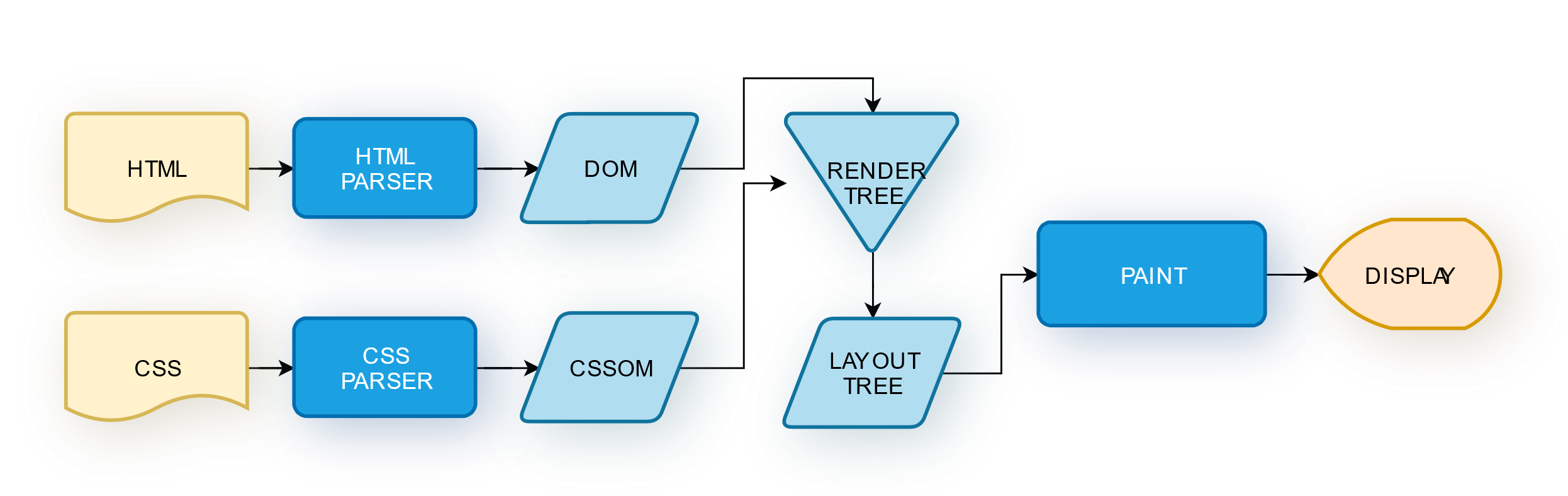


Figure 4: Rendering Engines

HTML documents are parsed into DOM objects and similar process happens for CSS; the outputs of these stages are combined to create a render tree. The render tree consists of only those elements that are to be displayed, for example an element with the display property set to none will not appear in the render tree. The render tree is then converted into a layout tree in which all elements know where they are to be positioned, how they look (color, dimensions, etc.). The painting process utilizes this tree to finally tell the GUI framework (PyQT5 in our project) to display items according to the tree.

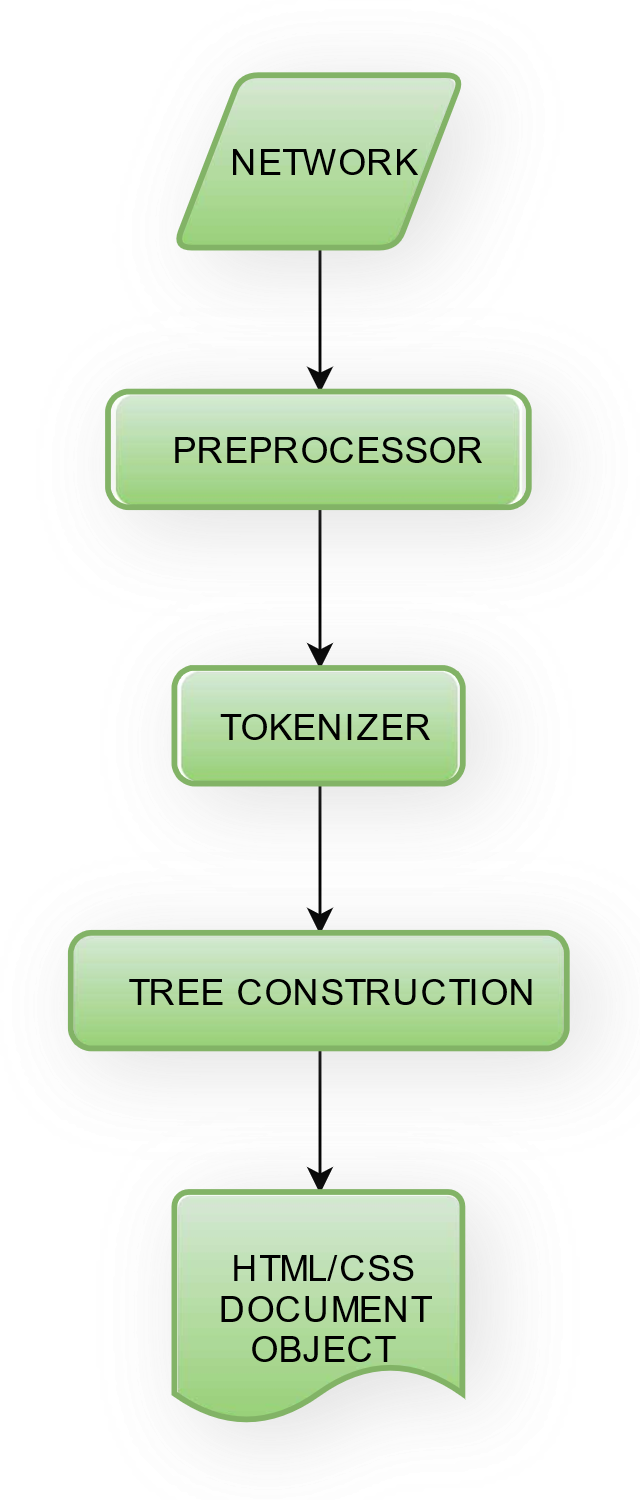
**Parsing**

Figure 5: Parsing Flow

Requested document is fetched through the internet.

Preprocessor removes unnecessary characters from fetched document.

The tokenizer breaks the input stream into its lexical components. For example, <html> into a tag element.

The tree construction stage takes in input from the tokenizer and makes a tree of parent and child nodes.

The above tree is contained in a Document object, that is further rendered into a web page.

**Tokenizing**

The standards model the tokenizer as a state machine. A state machine processes inputs according to the rules of its current state and inputs can change its state. Here is a tinier version of the tokenizer’s state machine diagram:

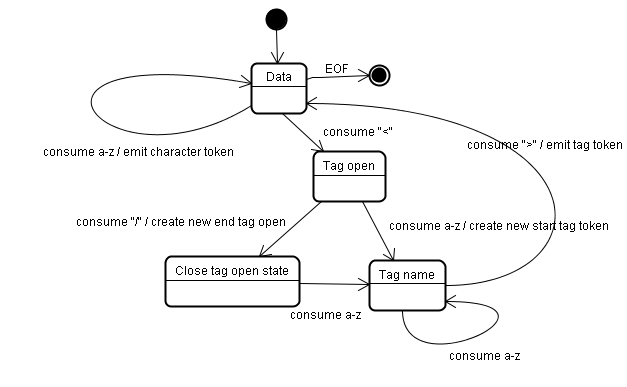


Figure 6: Tokenizer State Machine Diagram

If its not obvious what tokenizing does, here’s an example:

If someone was to develop a tokenizer for the English language the sentence “the cat sits” would be converted into tokens like {“the”: “article”, “cat”: “noun”, “sits”: “verb”}.

The next stage would further take these tokens to divide it into subject and predicate.

**SYSTEM REQUIREMENTS**

**Hardware:**

|  |  |  |
| --- | --- | --- |
| Component | Minimum | Recommended |
| CPU | *Dual core x86 (4 threads)* | *Quad core x86 (8 threads)* |
| GPU | *Intel UHD / AMD Vega* | *GTX 1650 / Radeon RX 570* |
| RAM | *4 GB DDR4* | *8 GB DDR4* |
| Storage | *~ 200 MB* | *~ 300 MB* |

**Software:**

* **Python** 3 - 3.8   
  (Note: Because CEF Python currently lacks support for python 3.8+, higher versions will not work)
* **GNU/Linux**, all major distributions are supported -  
  (Fedora, Debian and its derivatives, RedHat, CentOS, Gentoo, Arch, OpenSUSE, Manjaro, ...)  
   or  
  **Windows** (minor changes are required to run on windows)   
   or  
  **MacOS** (minor changes are required to run on MacOS)

*Tested on AMD Ryzen 5 3600 running Fedora Linux with GTX 1650 graphics card, 8 GB DDR4 3200MHz RAM, 256 GB SSD*

**PYTHON**

**(AS FRONTEND)**

Python offers a significant advantage over other languages for our project. As explained above, our project has to parse HTML and CSS and so a lot of string functionality is required, which happens to be built right into python’s easy syntax and also a large set of in-built functions. Python is object oriented and supports object inheritance allowing us to easily create nodes for the DOM tree in the parser.

Python’s clear, easy and feature rich syntax attracts more developers to the language, consequently python’s community is quite large and so if one ever runs into any problems it’s likely someone else has faced a similar issue and a solution exists for it. Additionally, huge projects like CEF Python (which is being used in our project) are ported to python too.

Some other reasons to use python: it’s open source! And so free of cost. Anyone can modify the language to one’s needs. As a matter of fact, many have done these modifications, implementations like PyPy allow one to eliminate one of python’s major drawbacks that is speed, giving our project scope for further speed enhancements. Also, Python is cross-platform, so you write the same code (in most cases) for all platforms. Python also has large availability for libraries, so one doesn’t need to reinvent the wheel every time. This allows for quicker development compared to other languages. Even ignoring the vast library availability, development is quicker in python as one doesn’t have to declare variable types.

In conclusion, the choice to use python as the front-end was because it has a simple syntax, large community, it’s open source and cross-platform, it has endless applications, large availability of libraries and offers small development times.

**MySQL**

**(AS BACKEND)**

# TODO

**Why binary files were also used?**

Instead of making a request to the MySQL server every time user data is required, the browser caches user data with a binary file to allow for quicker access.

**DEPENDECIES**

* **PyQT5**This is a cross platform GUI framework. To create a window, complicated calls have to be made to the operating system and the specific calls vary with each operating system. To avoid the hassle of unnecessarily dealing with these already solved problems, a GUI framework has been used.
* **CEF Python**Chromium Embedded Framework allows embedding chromium browser inside a window. CEF Python is the python version for CEF. As mentioned before this is being used to maintain backwards compatibility for websites written with JS. Also, since the Skylon engine doesn’t yet support variety of node types and layouts, welcome launcher of our browser is utilizing CEF Python to create a beautiful GUI. (CEF Python can create a standalone window without being embedded into another window)

**USER DEFINED FUNCTIONS**

**(UDF)**

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Parameters | Return Type | Description |
| *is\_first\_launch* | *None* | *Boolean* | *Returns True if binary file ‘user.cache’ does not exist else returns False* |
| *get\_user\_data* | *None* | *List* | *Reads ‘user.cache’ and returns data, defaults to [“John”, “Doe”, “johndoe@example.com”, “chromium”]* |
| *launch* | *None* | *None* | *Launches WelcomeLauncher if first launch else launches the browser of choice.* |
| *WelcomeLauncher* | *None* | *None* | *Launches Chromium with page as* [*https://sujalsinghx86.github.io/school-project-web/*](https://sujalsinghx86.github.io/school-project-web/) *and exposes python functions to the web page.* |
| *launch\_chromium* | *None* | *None* | *Launches browser with chromium as engine choice* |
| *launch\_skylon* | *None* | *None* | *Launches browser with Skylon as browser choice.* |
| *create\_user* | *first\_name, last\_name, email, engine\_choice* | *None* | *Creates a new user in the MySQL Database* |
| *user\_exists* | *first\_name, last\_name, email* | *Boolean* | *Checks if a user with given parameters exists or not by querying MySQL database.* |
| *update\_preference* | *email, new\_preference* | *None* | *Updates engine preference for given email parameter on MySQL database with new\_preference parameter* |
| *delete\_user* | *email* | *None* | *Deletes user entry from MySQL database for given email parameter* |
| *cache\_user* | *email* | *None* | *Fetches user data from MySQL database for given email parameter and creates binary file ‘user.cache’ for quicker access.* |

**GLOSSARY**

**Tokenizing:** The process of breaking down a string into a list of lexical components as defined by a grammar for a specific language (here HTML and CSS) through a series of steps. These steps depend upon type of grammar of the language.

**Tree construction:** The process of converting a list of tokens into a tree structure through a series of steps, in our project these steps conform to the HTML and CSS specification.

**DOM:** Document Object Model is an HTML Tree consisting of HTML elements as nodes which can be used as an API to manipulate the tree after a page is loaded.

**Chrome:** All the functionality except the browser’s engine is termed as the chrome of the browser.

**FUTURE ASPECTS**

This project opens a lot of doors for future development.

One primary aspect for development will be the Skylon engine itself which can be developed further to support a variety of features supported by the modern web such as SVG & raster image formats, XML, etc. The engine will also need a lot of development in the rendering module. The engine currently only supports text elements, further development would allow other nodes and much more layout options. Another thing which should be implemented in the future is a DOM API in python to be accessible by web pages rendered by the Skylon engine, just like it’s current accessible in JS for pages rendered by modern browser engines.

Another thing which allows further development is shifting the entire code base to the PyPy implementation of python, this would allow significant speed gains.

Additionally, the chrome of our browser could support multiple tabs, page history, bookmarks, etc in the future.

All in all, this project has taught us a lot and we plan to further develop this during our post-secondary education. One day we might stand a chance at competing against all the major browsers in existence today.

**BIBLIOGRAPHY**

**HTML SPECIFICATIONS**

[**https://html.spec.whatwg.org/multipage/parsing.html**](https://html.spec.whatwg.org/multipage/parsing.html)

[**https://html.spec.whatwg.org/#rendering**](https://html.spec.whatwg.org/#rendering)

**CSS SPECIFICATIONS**

[**https://www.w3.org/TR/css/**](https://www.w3.org/TR/css/)

[**https://www.w3.org/TR/css-syntax-3/**](https://www.w3.org/TR/css-syntax-3/)

[**https://www.w3.org/TR/css-box-3/**](https://www.w3.org/TR/css-box-3/)

**CEFPYTHON**

[**https://github.com/cztomczak/cefpython**](https://github.com/cztomczak/cefpython)

**EBOOKS**

[**https://browser.engineering/**](https://browser.engineering/)

[**https://htmlparser.info/**](https://htmlparser.info/)