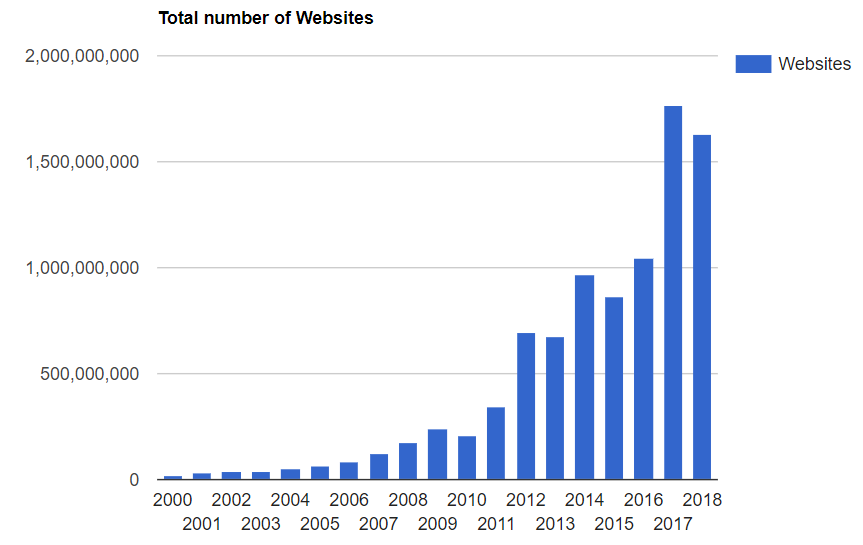
Introduction

Motivation

The internet has been, since it publicly launched in the 1990s, one of the biggest innovations and influencers towards building the technological world that we live in today. The World Wide Web has expanded over the course of the last 30 years to reach over 5 billion pages that are indexed through search engines, with over 1.6 billion web sites being active as of today.



But more importantly, over 50% of the world’s population is using the internet in 2019, either on a computer or mobile devices. Some of the most popular activities on the internet are researching, browsing social media or online shopping, as statistics show the more popular domains being *google.com, facebook.com, wikipedia.org, amazon.com.*

The high demand of internet usage makes it necessary to continue improving the browsing experience and create new technologies that can make it more efficient to accomplish certain tasks that are difficult to achieve otherwise. Using applications through voice commands has proven to be a necessary tool for the common user as most smartphones, TVs, speakers and car systems nowadays have such technologies implemented. Some of these include Google Assistant (enabled on 1 billion devices), Amazon’s Alexa (100 million devices), Apple’s Siri (over 500 million devices). It is also estimated that the usage of voice assistants will triple by 2023 as the home devices industry will keep improving their features and become more attractive to the public.

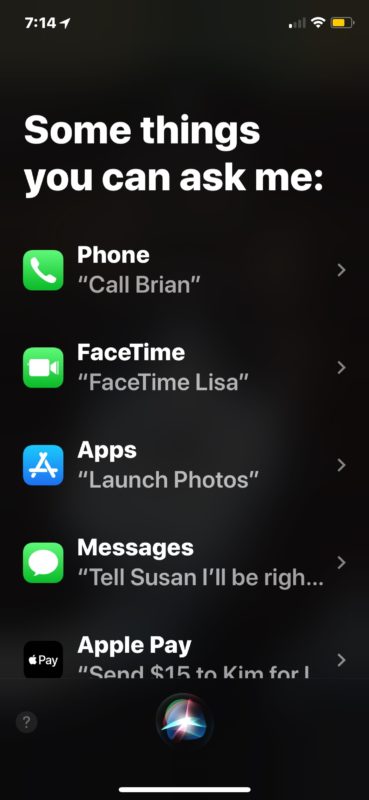
Regarding web accessing on computers, for over 3 years Google Chrome has been the most used browser because of its simplicity, speed, security and large catalog of user developed extensions. Just as mobile phones are getting more and more popular with using a voice assistant, the computer could also find it to be more convenient to be able to perform commands without having to use a mouse or keyboard. My application tackles the problem by dividing it in two parts, taking the vocal input from the user and initiating commands towards the browser based on that input.

Similar applications

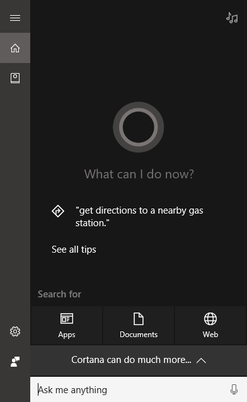
**Google Assistant** has launched in 2016 and it lets you interact with your Android device using voice commands. The way it works is starting up after the user says “Ok Google”, and wait for the user to specify their command. The assistant is able to find information on your device, such as contacts, messages, calendar notes or open applications, but also find information online such as the weather, bookings, points of interest, or random searching.



**Siri** is Apple’s voice assistant which launched in 2011, currently being compatible with every built-in application on Apple devices (interacting with them similarly to Google Assistant) and starts up with the command “Hey Siri”. It can only run on iOS devices while Google Assistant can run both on iOS and Android.



**Cortana** is a voice assistant launched by Microsoft in 2014 for Windows 8.1 and released also for Windows 10 and few other platforms. It is integrated in Microsoft’s Edge browser and uses Bing as a search engine, performing similar tasks to the other assistants, but on a computer.



Short description

While all these assistants are able to interact with your device to retrieve information and perform actions, on the browser level, their functionality is focused towards searching the web, finding topics on Wikipedia, telling the weather or finding certain locations or news, which only lead the user to the information that they need.

My application is focused towards handling the recurrent actions that a user performs on a specific web page, such a scrolling, clicking on buttons, selecting text, filling input fields or downloading content. To interact with the program the user needs to hold down the *RCtrl* key in order to input their command and then release it so the voice input can be recognized and transformed into text. Then, the text is processed by the application which will determine based on the similarity of it with the supported commands, what did the user want to happen, and then the function which launches the interaction with the browser on that specific command will be called and executed. If the command cannot be identified by the speech recognizer it will not be forwarded to the text analyzer and, similarly, if the text produced by the user’s input doesn’t resemble any of the supported commands, the user will be informed that they should try to repeat their request. Each request from the user is handled at a separate time and they will not be able to insert a new command until the previous one has finished its execution.

The complexity of the application depends on the function that the user requests at a certain time, many being executed in constant time such as pressing some button combinations on the keyboard or performing a javascript call, but others such as clicking on a button or scrolling the page require a linear time depending on the number of buttons that need to be searched on that page or the height that needs to be scrolled.

The purpose of my project is to create a program that can interact with the browser in a different way, that has not been popularized as of today and that, in my opinion, has the potential to be a more appealing way of using internet services to a certain group of users.

Chapters

1. Speech recognition and Browser functionality
2. Technologies used
3. Application architecture
4. ???
5. Conclusions

Chapter 1.

Speech recognition and Browser functionality

**Speech recognition** refers to the ability of a program to transform an audio input representing a speech, into a text representing the words of what was said. It is a popular technology used in many devices nowadays, such as mobile phones, household devices, car systems or robotics.

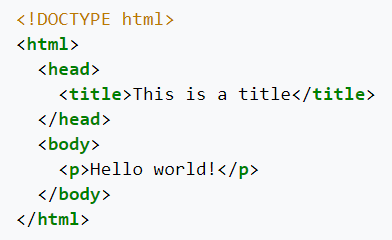
Some of the models that try to solve the problem of recognizing speech are:

* *Hidden Markov models* – “doubly stochastic process with an underlying stochastic process that is not observable (it is hidden), but can only be observed through another set of stochastic processes that produce the sequence of observed symbols” [1]
* *Dynamic time warping* – algorithm that analyzes words based on the speed of the speaker
* *Neural networks* – network trained to map vocal inputs to text outputs in order to be able to determine the output for new inputs

These technologies are used in implementing the speech recognizers available today, such as Google API, Microsoft API And CMU Sphinx.

Another important aspect in developing the application is to understand the **browser’s functionality**. The World Wide Web is a collection of web sites linked through URIs, that host resources which can be accessed through HTTP requests. In order to view the information listen on a web page, a user has to type an URL in the address bar of a browser and submit it, which will send a request to the host of the web site, returning the resources necessary to build the page in the browser.

Out of all the resources of a web site, the HTML file is the most important in determining the structure and the elements belonging to that specific page. The basic structure of an HTML file looks like this:



Finding the elements of a web page can be realized by searching through different tags or values inside the file, such as *<p>* or *<h1>* to determine paragraphs, *<button>* to find buttons, *<a>* for hyperlinks, or *<input>* for input fields. Then, the program would have to read or modify the information of that field.

Chapter 2.

Technologies used

**Speech Recognition Module**

Python’s *SpeechRecognition* module is a library that supports multiple speech recognition APIs, and has the ability to listen and record a microphone in the background, convert audio input to text and save audio data to files. Some of the APIs supported are: CMU Sphinx, Google Speech API, Wit.ai, Microsoft API, Houndify or IBM.

To use the **Microphone**, the user has first to install the additional *PyAudio* module, which is used to record audio input. On instantiation, the constructor of the *Microphone* can be given the following optional parameters:

**Microphone(device\_index: Union[int, None] = None, sample\_rate: int = 16000, chunk\_size: int = 1024) -> Microphone**

1. *device\_index*: an integer between 0 and the number of audio devices on the system, representing the index of the device that will be used to obtain audio input from; if unspecified, the default microphone will be selected
2. *sample\_rate*: the rate of samples per second at which audio chunks are recorded; a higher value can result in a better audio quality, but slower recognition
3. *chunk\_size*: the number of chunks in which the audio input will be separated

On instantiation, the *Microphone* object will check whether the user has assigned any optional parameters, and if so, whether they are the correct type (throws exception otherwise). Then it tries to import the *PyAudio* module and instantiate an object of it to use for the recording, raising *AttributeError* if the library hasn’t been installed. If no error occurs, all the components needed for the *Microphone* object will be saved as class variables.

The **Recognizer** is a class used for setting up different functionalities of speech recognition, starting and stopping the recording process and selecting the API that will perform the transcription. It has no additional parameters and on initialization, it will set some class variables to default values:

* *recognizer\_instance.energy\_threshold = 300 # type: float*

A variable that represents the perceived loudness of the sound, the value 300 being considered the threshold between silence and speech. But since each microphone has a different sensitivity that requires a different threshold, this value will be considered only as the initial (minimum) value and will need to be dynamically adjusted based on the hardware used and the loudness of the environment. This initial value can be modified to a lower value in the case that the microphone is not able to detect audio input, or to a higher value if the background noise makes it hard to understand and transcribe what the speaker is saying.

* *recognizer\_instance.dynamic\_energy\_threshold = True # type: bool*

In most cases, the energy threshold needs to be automatically adjusted based on the used microphone device and the background noise in the room. This value should be changed only if the ambient of the room remains constant and the initial energy threshold is set to the appropriate value.

* *recognizer\_instance.dynamic\_energy\_adjustment\_damping = 0.15 # type: float*

Approximates the speed at which the energy threshold is dynamically adjusted in one second (if the *dynamic\_energy\_threshold* property is enabled). A lower value means a faster adjustment, but it could miss certain parts of the audio input, while a higher value will decrease the rate of modification.

* *recognizer\_instance.dynamic\_energy\_adjustment\_ratio = 1.5 # type: float*

Represents the minimum ratio at which the speaker’s voice is louder than the background noise (if the *dynamic\_energy\_threshold* property is enabled). A smaller value would result in a harder differentiation between the two, while a larger value requires a quieter ambient noise.

* *recognizer\_instance.pause\_threshold = 0.8 # type: float*

The minimum number of seconds that represents silence between phrases. Lowering the value could generate a negative output for slower speakers, while increasing it could damage it for faster ones.

* *recognizer\_instance.operation\_timeout = None # type: Union[float, None]*

The number of seconds after which an internal operation (such as an API request) will timeout. The *None* default value means there is no timeout and the program will wait for the action to be performed.

In order to initiate the recording of the microphone, the following function will be used:

**recognizer\_instance.listen\_in\_background(source:AudioSource,callback: Callable[[Recognizer, AudioData], Any]) -> Callable[bool, None]**

Given an *AudioSource* instance as input (in this case, the *Microphone* object), it will transform it into an *AudioData* instance and send it as a parameter to the given callback function. It returns a function that, when called, will stop the background listening. The process will execute on a separate thread, so that the main thread will be able to perform other actions before it chooses to stop it.

When called, the function will check whether the given input is of the correct type (*AudioSource*), then it will initialize and start the thread that performs the listening. The thread is also marked as a daemon thread, which means that it will terminate automatically if the main thread completes (as long as there’s no other non-daemon thread present), and it can also be shut down at any time during its execution. The recording thread will continuously call a separate *listen* function that will handle the conversion of *AudioSource* to *AudioData* for each phrase, and, after it, the callback function which will do the recognizing. The thread will check every second if the stop function has been called.

The stop function is the function returned by the *listen\_in\_background* method, that, when called, will stop the recording thread from sending sound data to the converting function and thus finishing the thread. It has one optional parameter, *wait\_for\_stop*, that is set to a default *True* value, which will make the function wait for the thread to finish before returning. If set to *False*, the method will return right away, while the thread will still be on its way to end.

While *AudioSource* is an abstract class, implemented by the *Microphone* class, an *AudioData* object holds in the *frame\_data* variable a sequence of bytes that represent audio samples. This raw data can be then converted into different audio formats, if needed. The object also contains a *sample\_width*, representing the width of each sample in bytes, and a *sample\_rate* for the data.

**AudioData(frame\_data: bytes, sample\_rate: int, sample\_width: int) -> AudioData**

The *listen* function that performs the conversion from the *Microphone* to *AudioData* has the following structure:

**recognizer\_instance.listen(source: AudioSource, timeout: Union[float, None] = None, phrase\_time\_limit: Union[float, None] = None, snowboy\_configuration: Union[Tuple[str, Iterable[str]], None] = None) -> AudioData**

The recording takes place when the energy detected is above the *recognizer\_instance.energy\_threshold* value, and ends after few seconds of silence, given by the value of *recognizer\_instance.pause\_threshold*. It can stop also when there’s no more audio input sent to the method (when the stop function of *listen\_in\_background* is called). It has the following parameters:

1. *source*: the input *AudioSource* object
2. *timeout*: the number of seconds the function waits for an input phrase before exiting; if *None*, it will wait indefinitely
3. *phrase\_time\_limit*: the maximum number of seconds allowed for recording one phrase, before stopping and processing what was recorded in that timeframe; if *None*, a phrase will have no time limit
4. *snowboy\_configuration*: a configuration for a third party recognizer that can train a neural network based on the given inputs; if *None*, it will not be used

The way it works is verifying, first, that the source is a valid *AudioSource* object, then recording the audio input until there is a silence of *timeout* length (if not *None*), or until a phrase is long enough. If *dynamic\_energy\_threshold* is enabled, the *energy\_threshold* will be updated based on the current threshold, the *dynamic\_energy\_adjustment\_damping* and *dynamic\_energy\_adjustment\_ratio*:



The audio input of the microphone will be turned into a sequence of bytes that, concatenated, will represent the *frame\_data* element of an *AudioData* object. The transformation is done using *PyAudio*’s *read* function, that takes as a parameter the number of chunks to be read. After finishing, the function will return the newly created *AudioData* object.

**Google Speech API**

“Google has improved its speech recognition by using a new technology in many applications with the Google App such as Goog411, Voice Search on mobile, Voice Actions, Voice Input (spoken input to keypad), Android Developer APIs, Voice Search on desktop, YouTube transcription and Translate, Navigate, TTS. After Google, has used the new technology that is the deep learning neural networks, Google achieved an 8 percent error rate in 2015 that is reduction of more than 23 percent from year 2013. According to Pichai, senior vice president of Android, Chrome, and Apps at Google, “We have the best investments in machine learning over the past many years. Indeed, Google has acquired several deep learning companies over the years, including DeepMind, DNNresearch, and Jetpac”.”[2]

The way Google Speech API is integrated in the *SpeechRecognition* module is through the method *recognize\_google* of the *Recognizer* object.

**recognizer\_instance.recognize\_google(audio\_data: AudioData, key: Union[str, None] = None, language: str = "en-US", , pfilter: Union[0, 1], show\_all: bool = False) -> Union[str, Dict[str, Any]]**

The function would be called after obtaining an *AudioData* object from the *listen* function and has the purpose to obtain the transcription of audio data (represented as a sequence of bytes that build a specific format file) by sending a request to the API with the given input. It will return an error if the speech cannot be recognized or if the API cannot be accessed. As parameters, it takes:

1. *audio\_data*: the *AudioData* object that will be transcribed
2. *key*: a key required to access the API; if *None*, will use a default key
3. *language*: the language of the spoken text
4. *pfilter*: profanity filter
5. *show\_all*: if *False*, will return the response as a string, else, as a json dictionary

On implementation, the function will check that the given *audio\_data* is of the correct type, and then will convert the bit string of the *frame\_data* element in a byte representation of a FLAC file, using a third party executable. Then, it will send the request to the API and determine the result with the highest confidence rate from within the response dictionary:

****

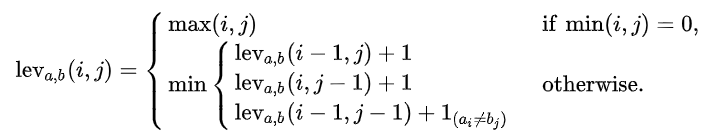
Possible outcome:

{    
   **"result"**:[    
      {    
         **"alternative"**:[    
            {    
               **"transcript"**:"hello world",  
               **"confidence"**:0.80891722  
            },  
            {    
               **"transcript"**:"hello-world"  
            },  
            {    
               **"transcript"**:"hello word"  
            },  
            {    
               **"transcript"**:"hello work"  
            },  
            {    
               **"transcript"**:"hallo world"  
            }  
         ],  
         **"final"**:true  
      }  
   ],  
   **"result\_index"**:0  
}

**Levenshtein Module**

Python’s *Levenshtein* module has implemented the ability to calculate the Levenshtein distance between two strings, and return a ratio between 0 and 1 of their similarity.

The Levenshtein distance between two strings represents a way to tell how similar the strings are, by calculating the minimum number of single characters edits it takes to transform one into the other. The possible edits to a character can be substitution, insertion or deletion.



The algorithm takes a recursive approach, by calculating the minimum number of edits between deleting a character from the first string, the second, or both, and checking whether that character from the same position matches in the two.

**Selenium’s Webdriver**

*Selenium* is a Python module that provides an API which can automate the interaction with the web browser [3]. It can be installed using the following command:

**pip install selenium**

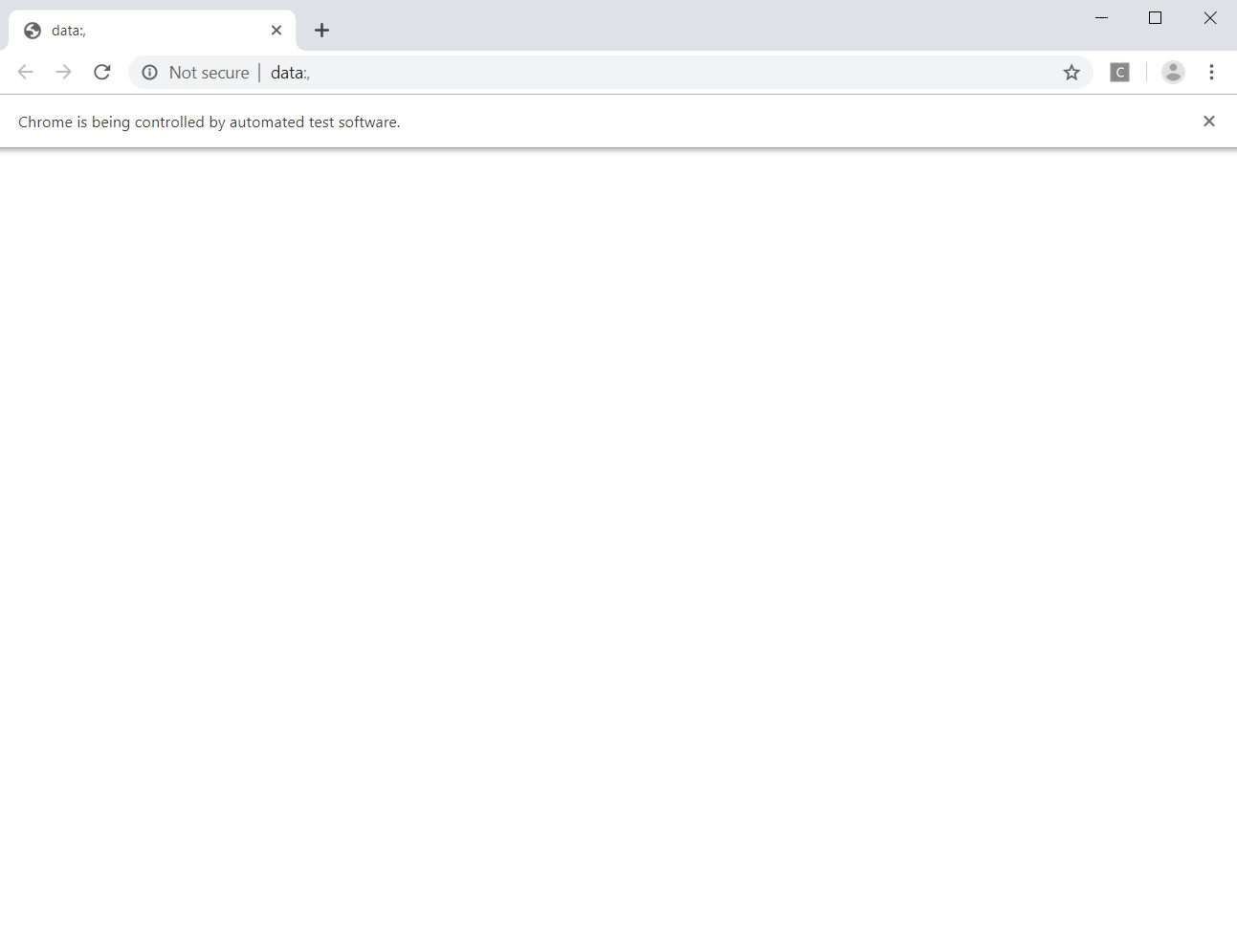
It also needs a web driver for a browser in order to be able to interact with it, which can be found on the developer’s web site:

|  |  |
| --- | --- |
| **Chrome:** | <https://sites.google.com/a/chromium.org/chromedriver/downloads> |
| **Edge:** | <https://developer.microsoft.com/en-us/microsoft-edge/tools/webdriver/> |
| **Firefox:** | <https://github.com/mozilla/geckodriver/releases> |
| **Safari:** | <https://webkit.org/blog/6900/webdriver-support-in-safari-10/> |

In order to launch the browser, the following lines need to be executed:



The first line imports the selenium.webdriver module and the second one instantiates a Chrome driver, which will launch the browser on a blank page. The module can also interact with other browsers, such as Firefox, Edge, Internet Explorer, Safari, Opera.



To navigate to a page on the browser, the *get* function is used:



Then, to find an element on the page, multiple functions can be used, that parse the HTML file and look for certain identifiers:

* *find\_element\_by\_id*: searches based on the id attribute of an element

*<form id="loginForm">*

*<input name="username" type="text" />*

*<input name="password" type="password" />*

*<input name="continue" type="submit" value="Login" />*

*</form>*



* *find\_element\_by\_name*: searches based on the name attribute of an element

*<input name="username" type="text" />*



* *find\_element\_by\_xpath*: performs a search similar to one in an XML file, being able to determine a relative path from one element to the other, or an absolute one from the root of the document (*html*); it is also able to find items of same rank based on their order and select only that with specific attributes

*<html>*

*<body>*

*<form id="loginForm"> … </form>*

*</body>*

*<html>*



* *find\_element\_by\_link\_text*: can locate a hyperlink (element with the tag *a*) that has a certain value given as a parameter
* *find\_element\_by\_partial\_link\_text*: locates the hyperlink whose value has a substring equal with the value given as parameter

*<a href="continue.html">Continue</a>*



* *find\_element\_by\_tag\_name*: determines the first element with a certain tag

*<h1>Welcome</h1>*

**

* *find\_element\_by\_class\_name*: determines the first element with a certain value for the class attribute
* *find\_element\_by\_css\_selector*: searches based on the syntax of the CSS selector given as parameter

*<p class="content">Site content goes here.</p>*

**

It is important to note that these functions will only return the first element found that respects the given characteristics (or nothing if there’s no such element). In order to find all of the elements, these functions will return a list containing them:

* *find\_elements\_by\_name*
* *find\_elements\_by\_xpath*
* *find\_elements\_by\_link\_text*
* *find\_elements\_by\_partial\_link\_text*
* *find\_elements\_by\_tag\_name*
* *find\_elements\_by\_class\_name*
* *find\_elements\_by\_css\_selector*

(The *id* attribute is unique for each element in the HTML document)

Another useful feature of the library is the ability to execute javascript commands, using the method *driver.execute\_script*. This enables the possibility of doing tasks that are not possible with the methods available in *Selenium*, such as scrolling, opening tabs or editing the HTML document.

All of the functionalities are applicable only to the current selected tab of the browser, that is indexed as an integer in the tab list, represented by *driver.window\_handles*. The values stored in the array are the names of the active tabs, which can be switched to by using the available function *driver.switch\_to\_window("windowName")*.

The implementation of these commands is based on sending a request to the browser, given the type of the request, the data added by the user and a javascript path identifier recognized by the server. Some of these identifiers are:



**Action Chains**

A useful library added to *Selenium* is the implementation of the *ActionChains* class. *ActionChains* allows for the automation of mouse movements, button presses and interaction with the context menus. It can be used for tasks such as hover on buttons to reveal context menus and interact with them, or drag and drop operations.

The implementation allows for storing multiple actions inside a queue in the *ActionChains* object, that can be performed together, in the order of storing. After adding the actions, the *perform* function will execute them in order using the same approach as the methods in the base *selenium.webdriver* module.



**Pynput Module’s Keyboard Listener & Controller**

The module allows for listening the keyboard and mouse for when an action takes place, and detect which key has been pressed, but also to control them and automate the pressing of buttons.

In order to control or listen to the keyboard, the following packages need to be imported:

* *pynput.keyboard.Key*
* *pynput.keyboard.Controller*
* *pynput.keyboard.Listener*

The *Key* package has the necessary mappings for all the keys on the keyboard.

The *Controller* can be used by simply instantiating it and using the *press*, *release* and *pressed* functions to handle pressing, releasing and holding down keys.



The *Listener* is a thread that will record actions until it is released, by either calling the *listener.stop* function, or returning *False* from one of its callbacks. The callbacks are set through the *on\_press* and *on\_release* parameters of the listener. Both of the callback functions take as parameter an object which represents the key that has been pressed or released by the user.



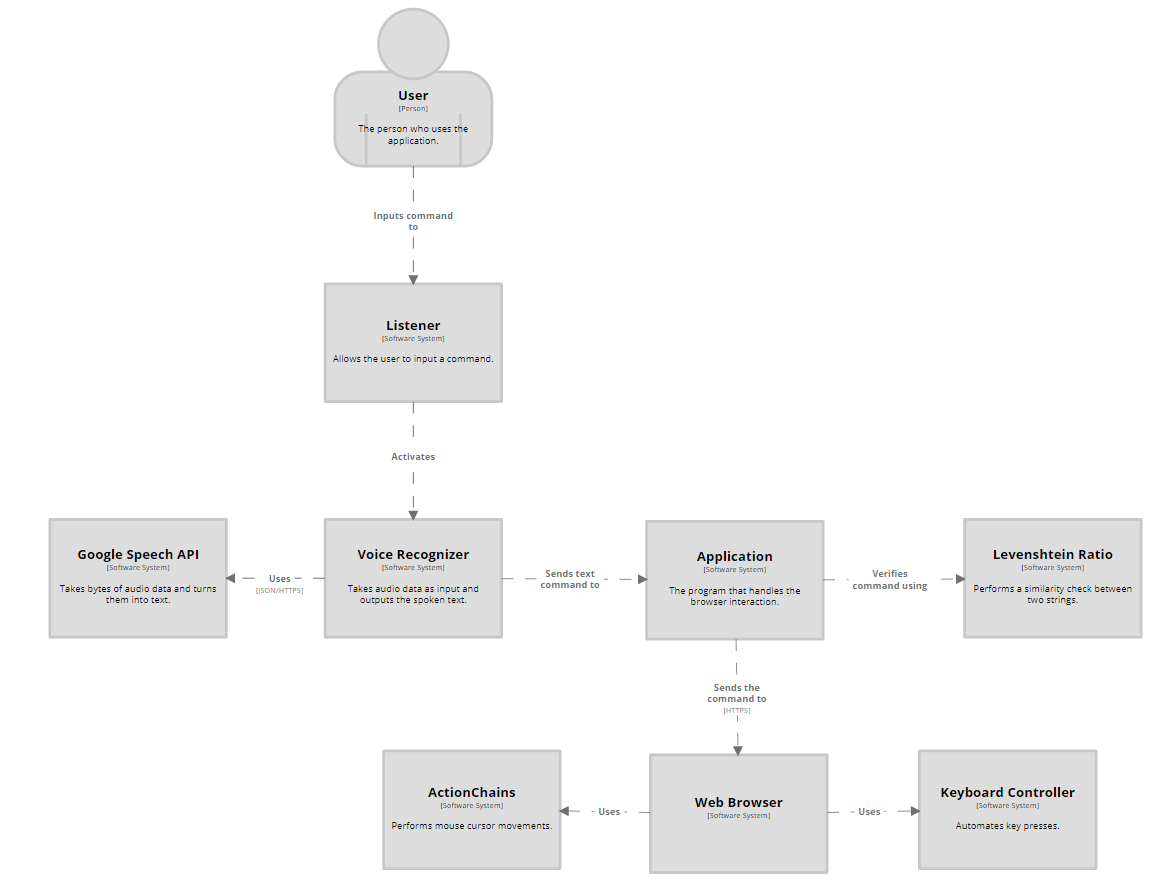
**// + daca mai adaug ceva**

Chapter 3.

Application architecture

The application’s functionality allows the user to input audio speech to the microphone, which is captured and converted to text using Google’s Speech API, then handled over to the browser interactor which will send a request to the web based on the command that should be performed.

The structure is divided in 2 principal classes, the *Voice* class, that handles the recording and recognition of the speech input, and the *Browser* class, that performs the interaction with the web driver.



**The *Voice* class**

The purpose of this class is to initiate the recording of the microphone, to stop the recording, and to implement the callback which calls the speech recognizer on the obtained audio data.

 On initialization, the object will contain a *command* field, that will store the text identified by the speech recognizer, an *executed* variable, that tells whether a command that has been identified has started execution or not, and a *stop\_listening* variable, that will store the stop function of the background listener.



In order to easily call for the program to start or to stop recording audio input, the commands that perform these actions have been added to a *listen* and *stop* functions inside the class.

The *listen* function will initialize a *Recognizer* and a *Listener* from the *SpeechRecognition* module, with no parameters which will prompt the library to select default ones, and call for the *listen\_in\_background* method of the recognizer that will continuously record audio input from the initialized microphone and obtain the sequence of bytes that makes it up. The *stop\_listening* variable will be set to the returned stop function that, when called, will stop recording the microphone and prompt the callback function to execute, in this case given as the *recognize* method to the parameters. The user is informed that the microphone is active with a message.

The *stop* function will call on the background listener’s stop function by using the variable in which the method was stored previously. The *wait\_for\_stop­* parameter is set to *False*, which means that the program will not wait for the listener thread to finish and execute the callback right away, in order to avoid unnecessary delays.



The *recognize* function is the callback set for when the background listener stops recording input from the microphone, and the information found so far should be transformed into text. The method takes as parameter the *Recognizer* object initialized before and the audio data computed by the *listen\_in\_background* function. It calls the *recognize\_google* function of the *Recognizer* with the audio input, which will attempt to output a JSON transcription with multiple possible outcomes and confidence levels and select the best possible option. The *command* variable will either be set to this “best guess” answer, or to an unidentified message if the API couldn’t understand what has been said. An intuitive message is also displayed in case the application cannot perform a connection with the API due to communication errors.

From the order of execution perspective, an object initialized from the *Voice* class will call on the *listen* function in order for the user to be able to input their command, and then on the *stop* method that will shut down the recording process and call on the recognizer. The program will then be able to read the identified text output from the *command* variable.

**The *Browser* class**

This class is used to initiate the web driver and to implement the execution of the commands supported by the application. Each command is implemented in a different method and can access the *driver* variable that sends requests to the browser.



On initialization, certain class variables will be defined that need to be accessed from multiple methods. These are:

* *driver*: the web driver that interacts with the browser
* *current\_tab*: the index of the current selected tab
* *scroll\_base\_speed*: the initial speed of scrolling
* *scroll\_update\_speed*: the value that the scrolling speed is updated with
* *scroll\_speed*: the current scrolling speed
* *button\_map*: a dictionary that stores the buttons found on a page
* *input\_map*: a dictionary that stores the input fields found on a page
* *selected\_field*: the currently selected input field
* *caps\_lock*: the value of the Caps Lock key

The main functionality of the class is to be able to easily perform browser related actions without rewriting large portions of code. Each command given by the user is executed in a separate function, that the program will wait for to finish before recording a new command. The methods that are implemented by the class are:

1. **Open(url)**



The function calls on the *get* method of the driver, in order to prompt the browser to go to the given *url*. If the action cannot take place, an error message is displayed.

1. **Quit()**



The function will call the *quit* driver method that will exit the browser.

1. **Search(text)**

****

It will find a search input field on the current page (usually having the attribute *name=”q”* in the HTML file), clear the text inside it (if there’s any), write the given *text* parameter by pressing each key in order of the characters and then press the *Enter* key to submit.

1. **Click(value)**

****

A function to click on the first button found that has its value similar with the given parameter. If no such button is found, the method will try to find a button that at least has the given parameter as a substring to its value. If no button is found, an error message is displayed.

The *buttons* variable will store all of the clickable items on the page (marked with the tag *button* or *a* for hyperlinks), using the *find\_elements\_by\_css\_selector* function of the driver. Then, the method checks for similarity between the value of each button and the given parameter. If none of the values are similar, it searches again in the hopes that the user wishes to find a button that only contains a specific string. An exception printing an error message is raised, if no button contains the parameter value, else the method will click on the button that has been identified using the *click* function of the library. The method also checks whether clicking on the button opens a new tab, by counting the number of tabs before performing the click operation and then recounting it after. If a new tab has been opened, the *switch\_tab* method of the *Browser* will be called in order to mark the newly opened tab as the current one.



The similarity between the value of a button and the given parameter is computed by the exterior method *similar*. The method takes as parameters two strings, removes all the nonalphabetic characters from them (using the regular expressions module) and makes all of their characters lowercase. They are compared based on the computed Levenshtein ratio between them, using the *Levenshtein* module, to determine if they are close to each other or not.

1. **Hover(value)**



This method will move the mouse cursor to the first button found that has the value similar or contains the given parameter. The implementation is identical to that of the *click* function, except instead of clicking on the found button, it will use the *ActionChains* library to initiate an action given by the *move\_to\_element* method, which will move the cursor to an HTML element. The action saved in the queue is executed using the *perform* method.

One possible use for this function is opening a submenu of a button, which is done by hovering on it.

1. **Page\_down(), Page\_up()**



These functions press the *PageDown* or *PageUp* keys on the keyboard. A keyboard object is used, which is globally initialized as a *Controller* object from the *pynput.keyboard* package.

1. **Scroll\_down()**

This method will perform a smooth scrolling of the current page, by using javascript’s *window.scrollTo(0, height)* function to scroll to a given height, calculated at each iteration and stored in a local variable.



Using javascript, the height of the page and the current height at which the display is on can be determined and stored in the *last\_height* and *sc\_height* variables.



Then, a loop will be created that will increase the current scrolled height until it reaches the maximum of the page. In order for the user to be able to stop the scrolling before it reaches the end, a *Listener* is used that will be activated if a new command is to be entered. If no command has been detected, the loop will continue its tasks.



If a *stop* command has been said, the scrolling will stop and the function will return. The speed of the scrolling will be reset to its initial value.



Else, if the user wishes for a faster or a slower scrolling, the *go faster* or *go slower* commands can be used. The *scroll\_update\_speed* will be either added or decreased from the current *scroll\_speed*, with the lower cap being the initial speed.



Javascript will be used to request the scrolling to be performed by the browser, then the new current height will be updated and also the total height.



When the current height reaches the height of the page, the program will use a new variable, *new\_height*, to recompute the total height of the page. If the *new\_height* is equal with the ­*last\_height*, it means that the scrolling is complete, the *scroll\_speed* should reset to the initial value and the method should return. Otherwise, it means that the page has dynamically generated new content as the user was scrolling it (or when the scrolling was finished), and the method should continue to scroll the page by setting the new *last\_height* value, until the user chooses to stop.

One possible use case for this feature is to automate scrolling for a social media web site, that generates new content continuously as the user reaches the end of the current content.

1. **Scroll\_up()**

This method is implemented similarly to the *scroll\_up* function. The user is still able to stop the scrolling or change its speed and there is no additional checking for whether new content can be generated since it’s not the case. The scrolling will stop when it reaches height 0 if the user doesn’t end it before.

1. **New\_tab()**



The function will open a new tab by using the javascript *window.open* method, will save the index to the newly opened tab (being the last index in the *driver.window\_handles* list) and will use the *switch\_to\_window* function to mark the tab as the current one.

1. **Switch\_tab()**



To move between the opened tabs, this function will switch to the next one in the *driver.window\_handles* list. If the last tab is the current one, the method will perform a modulo operation to return to the first in list.

1. **Close\_tab()**



Uses the *driver.close* function to close the current opened tab and switch to the next one. If only one tab is active, it will be closed and the browser will reopen.

1. **Refresh()**

****

Will refresh the current opened page using *driver.refresh* method.

1. **Back(), Forward()**

****

Will go either back or forward in the browser’s history, using javascript. If there’s no previous or next page, nothing will happen.

1. **Bookmark()**

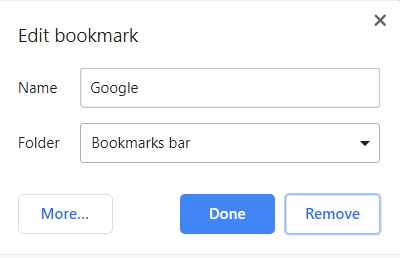


Will add the current page to bookmarks, with the keyboard shortcut *Ctrl+D*, using the keyboard controller. The *Escape* key is then needed to exit a the Bookmarks dialog.

1. **Remove\_bookmark()**

****

Will remove the current page from the bookmarks, if it has been added there previously. Using *Ctrl+D*, the Bookmarks dialog is open, then pressing *Tab* 4 times will select the remove button which is submitted.



1. **Save()**



Will perform the save operation, using the *Ctrl+S* shortcut.

1. **Source()**



Views the source code for the current page, using the *Ctrl+U* shortcut. Since the source is opened in a new tab, it must also be selected using the *switch\_tab* method.

1. **Submit(), Cancel()**

****

Simulate the pressing of the *Enter* or *Escape* keys.

1. **Find\_button(value)**

This function will map all the buttons found on a page (with the value containing the given parameter) to a number and store them in a dictionary. That way, the user will be able to choose which button he wishes to click by specifying the index at which that button is mapped.



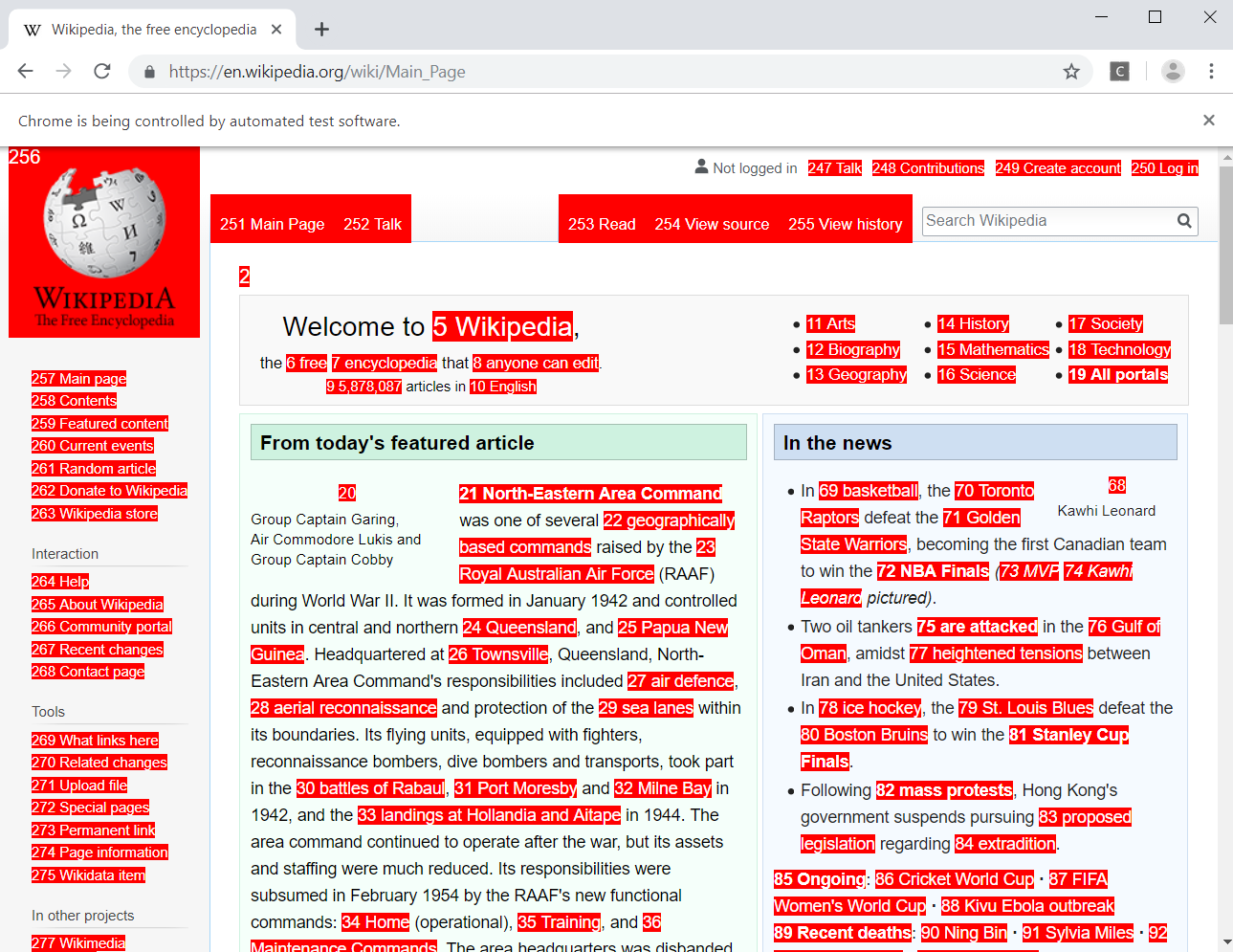
All the buttons on the page are identified by searching for the *button* and *a* tagged elements.



The *button\_map* dictionary contains as keys the indexes at which the buttons are stored, and as values the objects that represent the clickable elements on page.



The HTML file of the page is edited so that each of the selected buttons can be visible, by changing the background color and adding the index of the element to its value.



// de schimbat imaginea dupa ce repar afisarea la poze

1. **Choose\_button(value)**



Will choose the button from the list mapped previously (taking the index as a parameter) and click it. If a new tab opens, it will be selected. Also, if the index is not valid or the dictionary is empty, an error message will be shown. The dictionary is emptied if the operation is successful so that the program will not try to click on the same button from another page.

1. **Find\_input\_fields()**

The function will map all the input fields on a page in a dictionary, similarly to the buttons, in order for the user to be able to select a field which they wish to complete.



The input fields added to the map are only those that do not have a *type=”submit”* attribute, which have the purpose to submit the completed forms.



Apart from changing the background color, the text of the field is marked with the index at which it is mapped, and the type attribute is set to text so the user is able to see possible errors in passwords.

1. **Select\_input\_field(value)**



Selects the input field at the index given as a parameter by saving it to a class variable.

1. **Type\_text(text)**



Will type the given text parameter to the selected input field. It checks whether *Caps Lock* is enabled, then verifies that the current page is the same one where the input field was selected. If the given text is a key defined word, such as *space*, *backslash* or *comma*, it will write the mapped character. In order to write the words that make up the defined names, the user would have to spell them by characters.

1. **Clear\_text(positions=None)**



Will clear all the characters from a selected input field, or only the last *positions* characters. For clearing all characters, the library *clear* function is used, and for only removing some, the keyboard controlled will press the *Backspace* key multiple times. Nothing will happen if the selected field is not on the current page.

1. **Change\_caps(setter)**

Verifies the value of the setter, that should either be *on* or *off*, and activates or deactivates the *Caps Lock* key accordingly.

**Application flow**

On startup, the application will initialize a *Browser*, *Voice* and *Controller* objects, and start an infinite loop which will call for a *Listener* object at each iteration in order to wait for the user to input a command. After the user does so, a variable will be used in order to determine that the command just given by the user has been recognized, started its execution and should not be executed again until the user inputs another command. Then, the recognized command will be compared with the commands supported by the application using Levenshtein’s ratio, in order to bypass a certain degree of error between what the user said and what the speech recognizer has guessed, and to determine which command should be performed, if any. If a command has been identified, it will be separated in two parts, one that represents the identifiers for that command and another that is the user’s input to that command (if the command requires any). The *Browser* created object will then call the function which handles the command with the additional input, if necessary, and after the command has been performed, the application will continue looping and request a new command from the user. The program will stop when the user inputs the *exit* command, which will close the browser and exit the main loop.

The user can input a command by holding down the *RCtrl* key and releasing it when they have finished speaking. The list of commands supported by the application:

* *Go to <Url>*
* *Search <Text>*
* *Click on <ButtonValue>*
* *Hover on <ButtonValue>*
* *Page down*
* *Page up*
* *Scroll down*
* *Scroll up*
* *Open new tab*
* *Switch tab*
* *Close tab*
* *Refresh*
* *Go back*
* *Go forward*
* *Bookmark*
* *Remove bookmark*
* *Save*
* *View page source*
* *Submit*
* *Cancel*
* *Find the button <ButtonValue>*
* *Choose button number <Value>*
* *Find input fields*
* *Select input field <Value>*
* *Type text <Text>*
* *Clear (<Positions>)*
* *Caps <Setter>*
* *Exit*

Sources:

// de unde am preluat niste informatii statistice si legate de aplicatii similare pentru introducere

<https://www.internetlivestats.com/total-number-of-websites/>

<https://www.internetworldstats.com/stats.htm>

<https://www.worldwidewebsize.com/>

<https://www.juniperresearch.com/press/press-releases/digital-voice-assistants-in-use-to-triple>

<https://www.pocket-lint.com/apps/news/google/137722-what-is-google-assistant-how-does-it-work-and-which-devices-offer-it>

<https://www.pocket-lint.com/apps/news/apple/112346-what-is-siri-apple-s-personal-voice-assistant-explained>

<https://en.wikipedia.org/wiki/Cortana>

// informatii din capitolul 1

<https://en.wikipedia.org/wiki/Speech_recognition>

<https://en.wikipedia.org/wiki/World_Wide_Web>

<https://en.wikipedia.org/wiki/HTML>

// informatii din capitolul 2

<https://pypi.org/project/SpeechRecognition/>

<https://github.com/Uberi/speech_recognition/blob/master/reference/library-reference.rst>

<https://en.wikipedia.org/wiki/Levenshtein_distance>

<https://selenium-python.readthedocs.io/>

References:

// de pe scholar

[1] Rabiner, Lawrence R., and Biing-Hwang Juang. "An introduction to hidden Markov models." ieee assp magazine 3.1 (1986): 4-16.

[2] Këpuska, Veton, and Gamal Bohouta. "Comparing speech recognition systems (Microsoft API, Google API and CMU Sphinx)." Int. J. Eng. Res. Appl 7.03 (2017): 20-24.

[3] Lawson, Richard. Web scraping with Python. Packt Publishing Ltd, 2015.