



Investigation into the Omnichannel Experience that Post Pandemic Education can Present

LM118 – Bachelor of Engineering in
Electronic and Computer Engineering

Final Year Project
Final Report

Emmett Lawlor
18238831

John Nelson

<Submission Date>

Abstract

This project is a brief investigation into what the future could present for third level education in a post pandemic era. In 2020, all students of Ireland got to experience online learning, and then blended learning. By universities supporting omnichannel (blended) learning, universities can support more students, with more students comes more corrections, larger classes, more lecturers and supporting staff, I will look at some frameworks that would assist universities in this heavy change.

One idea would be a place for students to build tools for students using institution APIs (Automated Programming Interface). This would assist universities in having competitive resources available to students compared to universities who rely on a single IT department. Students with access to certain resources, can build better more integrated applications with their Universities framework, I encourage this by building a webapp of my own and detailing the framework for students to follow, in the event something like this was to be adopted.

The webapp is compiled from information freely available on the internet (but tricky to extract) of UL student timetables. Students visit <http://unical.ie> and input their course year, course code and enrolled modules. It was built using Python, HTML and JavaScript. I explain the fundamentals to build a webapp as easy as possible for students to use as a template.

I gathered all 800 module syllabuses present on timetables and went on to gather the 15k course directory of Qualifax, having so much text data I wanted to train a Machine Learning classifier but the data was too messy and unformatted. I decided to help machines read and understand these documents by cleaning them and creating a summarisation set of keywords or topics. With a keyword tagged data set, we could use it to train AI in the future for applications like correcting assignments or creating chatbots for students.

To back up my interest into this blended learning format, I investigated some trends in Irish education according to CAO, Qualifax and UL. I found UL lecturers are moving more towards tutorials, showing guided self-learning has become a more popular method of delivery. In CAO we see more applicants, offers, acceptances and higher results each year while according to Qualifax; the average asking points for courses stays static, showing a possibility for stress on campus resources and surrounding communities (e.g. a housing crisis) in coming years if learning is to be fully restored to on-campus learning, showing blended learning is something institutions should embrace.

In Chapter 2 we explain frameworks and useful functions in said frameworks for our applications (scrapers for information retrieval, web-application and text analysis), I then move onto Chapter 3; Architecture of the Webapp, how I gathered information from CAO, Qualifax and UL. Chapter 4; I explain how the entire project was executed in a perfect world. In Chapter 5; I mention issues which arose and how I overcame them. In Chapter 6; we go over some key points of information from all data sources in a small analytical report.

In Chapter 7; the project is wrapped together to imagine a university which utilised these tools and how they could be evolved to automate the certain aspects of university, providing students and lecturers with the tools needed to perform in a blended or omnichannel learning experience.

The exact format of the report is to be discussed and agreed with the project supervisor. In this template, the following is used:

1. 1.5 line spacing apart from the title page,
2. In the main body of the text, 12-point Calibri font is used,
3. The use of underlining text is avoided,
4. The report is primarily aimed at being read on a computer screen,
5. Text in main body is justified left and right,
6. The report presented so that if it was printed, it would be printed 2-sided. This means that to ensure a new section is presented on the right side of the paper, blank pages may be required,
7. The document is to be checked using the in-built Accessibility Checker in Microsoft WORD (<https://support.microsoft.com/en-us/office/improve-accessibility-with-the-accessibility-checker-a16f6de0-2f39-4a2b-8bd8-5ad801426c7f>),
8. Consider accessibility of the document:
 - a. <https://support.microsoft.com/en-us/office/video-improve-accessibility-with-heading-styles-68f1eeff-6113-410f-8313-b5d382cc3be1>
 - b. <https://support.microsoft.com/en-us/office/accessibility-video-training-71572a1d-5656-4e01-8fce-53e35c3caaf4>
9. The Table of Contents is generated with the “formal style” with Calibri 12-point font,
10. The IEEE citation style of referencing is to be used. Endnote may be used to aid the creation and use of references,
11. Note the use of page numbering and the numbering style,
12. Replace the chapter titles with appropriate wording according to the project,
13. Remove sections that are not required,
14. Add header 2, header 3, etc. styles for chapter sections and sub-sections, and suitably format,
15. Figure captions are to be placed below a figure (to the left of the page), and table captions are to be placed above the table (to the left of the page),
16. The project Gantt chart and final presentation slides are to be included as appendices at the end of the report.
17. If a weekly progress report was kept during the project, include an appendix that presents each weekly progress report (summary of actions in each week of the project).

Declaration

This report is presented in part fulfilment of the requirements for the LM118 Bachelor of Engineering in Electronic and Computer Engineering **Final Year Project**.

It is entirely my own work and has not been submitted to any other University or Higher Education Institution or for any other academic award within the University of Limerick.

Where there has been made use of work of other people it has been fully acknowledged and referenced.

Name

Signature

Date

Table of Contents

Replace this text with a table showing the section/sub-section numbers, titles, and page numbers (such as shown below).

<i>Abstract.....</i>	<i>i</i>
<i>Declaration.....</i>	<i>iii</i>
<i>Table of Contents</i>	<i>v</i>
<i>List of Figures</i>	<i>vi</i>
<i>List of Equations</i>	<i>viii</i>
<i>List of Tables</i>	<i>Error! Bookmark not defined.</i>
<i>Chapter 1 Introduction.....</i>	<i>1</i>
<i>Chapter 2 Frameworks and Methods of Interest</i>	<i>3</i>
<i>Chapter 3 Architecture (Tech Stack)</i>	<i>14</i>
<i>Chapter 4 Execution (If perfection perfected itself)</i>	<i>30</i>
<i>Chapter 5 Problems and Solutions (Where there's a will there's a way)</i>	<i>31</i>
<i>Chapter 6 Background Interests, Hypothesis' and Analytics</i>	<i>32</i>
<i>Chapter 7 Conclusions and future work.....</i>	<i>33</i>
<i>References.....</i>	<i>34</i>
<i>Appendices</i>	<i>- 1 -</i>
<i>Appendix A: Updated project Gantt chart</i>	<i>- 2 -</i>
<i>Appendix B: Final presentation slides.....</i>	<i>- 3 -</i>
<i>Appendix C: Project poster.....</i>	<i>- 4 -</i>
<i>Appendix D: Project progress reports.....</i>	<i>- 5 -</i>

List of Figures

Replace this text with a table showing figure numbers, captions and page numbers

List of Equations

Replace this text with a table showing equation numbers, captions and page numbers.

Notable Events

Chapter 1 Introduction

This Final Year Project was created to be an experimental way of looking at the various software's and what data they will collect in a world where online interaction and work environments are getting more and more digitalized.

For one, when I started my FYP, the various “verses”, like Meta’s metaverse and Nvidia’s Omniverse were not a thing, now the next frontier of the web. This introduction of a 3D universe in a digital space for work and social activities, filled with NFTs, smart-contracts and Artificial Intelligence plays into the introduction of Web3.0.

Web3.0 is the perfect example of what is expected of the world wide web and how it is changing. It is an all-encompassing prediction of how we will interact with digital properties in the future. We will explore the possibilities and some driving research behind Web3.0, investigating it’s Semantic and AI properties and it’s influences on this project.

I see software as a solution to many problems of two separate parties, academic staff, and students. In a perfect world people would learn for free, ai would correct assignments and be available as a q&a chat bot 24/7, lecturers and PHD students would spend more time lecturing and researching, being recommended current top topics in their areas of expertise which can be procedurally generated into learning-formatted material based on information they’ve deemed worthy topics or explored.

I wanted to explore how to create an API and graphical interface, in doing this we can see how easy it is to create a portal to which people can visit and learn from or use interactive tools. I created a full stack web application plus various supporting scripts which mine data from different sources, and transforms it into different formats for applications.

Through collecting this various data, I realised how much textual data I had, I decided not to let the opportunity go to waste in the chance to process this data to create a recommendation model in which students can describe the career they want, and get recommended courses. This recommendation model is only an application of the pre-processing of unsorted textual data which is the true bulk of this part of the project.

Pre-processing of textual data is the foundation of any application of Natural Language Processing application, once text data is categorised and a model is trained, you can create continuous pipelines of document categorisation, chatbots, topic generalisation and recommendation models.

Through text analysis, we can deduct topics and categorise textual data so it can be used to train any type of artificial intelligence model you like, with categorised textual data we can create a predictive text model, informational chatbot, and automate or complete textual documents in a similar sense to openai’s GPT-3.

This project wanted to see how the applications of these types of foundational software stacks of API GUI and AI can assist students in creating a one stop shop for the start of the semester, and in finding a course which suits their description of themselves. Along the way I will be

exploring topics such as Clustering, Data Analytics, alternatives to Frameworks or software, as well as performing investigations into various aspects of Third Level Education.

Personal motivation for this project is the fact many people cannot afford Third Level Education. I wish to make learning resources which can be customized and consumed by any person of any level. By using aggregated content from the internet based on peoples interests, we can create summarised learning material automatically through topic parsing, using inputs supplied by people, students and lecturers.

I hope a platform like this one day could assist someone in learning a subject that is not a clear cut path on how to evolve it's topics or foundation. It could also be used as a platform for people to test out subjects before committing to a 4 year course, curving dropouts and helping people make informed decisions on their career.

This led me to create Unical, the full stack application where students could come at the start of a semester. I wanted to make an everything tool that streamlined students entry to start of semesters. The first few weeks of every semester are always stressful and will set the tone for the rest of the semester, it's important to make this as easy for students as possible.

The current state of Unical will be explained throughout this project, briefly, it currently supplies downloadable calendars with integrated alarms based on the iCalendar (.ics) standard used by many devices. These downloadable calendars are filled with events from a course timetable.

Using the information supplied by the university, Unical can recommend other content like recommended books by module professors. I will talk about how I built Unical and what I would of liked to evolve it to, and how I would build a Webapp like this if I was to recreate it.

The second motivation for this project as mentioned above, is helping people make informed, guided decisions on their transition from any level into Third Level, either as a mature student, leaving certificate student, transfer or exchange students. I've seen a lot of people close to me being let down by expectations of a course, this can de-rail your enthusiasm for at least a year, until the next CAO application.

I figured the solution to this is to make deciding on a course more personal. I wanted to have a tool which would curate courses based on the persons interests, or based on previous courses they read about and liked. Nowadays the best way of finding courses is by sifting through cao.ie or qualifax.ie, searching courses by keywords of what you have heard about elsewhere.

I will explain how I processed the text data, some better algorithms and some findings using Natural Language Processing techniques, and the best method of categorising unsorted textual data.

Chapter 2 Frameworks and Methods of Interest

In this chapter we will define this projects current scope and the foundations behind it. Then we will explore the scope of where I hope this project could go and what would be needed for that. We will explore frameworks I used and how I could use them in this project.

Scope, what we need our frameworks to support:

1. Multiple web scrapers to gather large amounts of data.
2. An API with GUI for a portal to serve and host data/tools for students.
3. Methods of text processing, cleaning and analytics for a recommendation model as a tool to add to our portal in 2.

Python was chosen as the foundational language for each part of our scope. I have been writing Python for four years now, and have worked professionally in Python for a total of 2 years. I chose it due to my experience in it, specifically in web scraping, which will be the most important part of this project.

Multiple Python Libraries were used, which will be credited to throughout this project. Some honourable mentions are;

- Pandas; A data science package, used to create and manipulate Data Frames.
- Scrapy; A web scraping package, used to request and parse static HTML documents.
- Selenium; A browser automation tool which can programmatically navigate and parse web pages, specifically DOM (Document Object Model) enabled sites, which are HTML documents manipulated typically by JavaScript, of which cannot be navigated by Scrapy.
- Gensim; A Machine Learning document parsing package, to represent large corpus of text data as just a few summarised topics. Pre-filled with algorithms for parsing textual data, which will be used to find similarities among documents.

Scope 1: Multiple Web Scrapers to Gather Large Amounts of Data.

Web scrapers, also called miners and/or bots, are programs designed to extract information from the internet; The most common use for web scrapers is to collect data for marketing purposes. For example, a web scraper might be used to collect data about the prices of items on a particular website so that a company can create a price comparison tool. [source]

Bots/Scrapers are used to do repetitive actions, such as scrolling or navigating of sites to extract pieces of information. Google uses web scrapers to aggregate information and recommend information based on keyword and context similarity of a given input.

Scrapy:

The framework for Python, Scrapy was released in 2008 and was developed by a web-aggregation company. [<https://en.wikipedia.org/wiki/Scrapy>]

Scrapy offers tools which are composed of:

- Auto-throttle, which assists in reducing strain on servers you are sending requests to, which also helps in staying undetected to sites, and is just a form of respect to not congest servers.
- Rotating Proxies, we can rotate proxies to remain undetected and avoid IP banning by sending different requests and performing different routines from different IP addresses. Luckily we did not experience being banned from any sources, but it is nice to have this back-up in the instance we do get banned.
- User-Agents, we can set different user-agents to make it appear as though we are using any device or browser we want the target site to believe we are using.

Combining these tools, we can develop scrapers with varying levels of detectability, not congesting servers, making traffic look as though is multiple users from different parts of the globe, we can appear as normal traffic. We can even use custom user-agents and headers to identify ourselves, and even log-in as our profile by inputting a auth token which is granted by most sites upon successful log-in.

Scrapy also offers Scrapy Shell, which is a kernel based environment for testing the expected behaviour of the scrapers, this will assist in debugging during development. We can use the kernel to step by step navigate and inspect the website and figure out the best way to develop the scraper. We can call the shell from our terminal using;

```
> scrapy shell "https://ul.ie"
```

Scrapy works by allowing us to utilize “*selectors*”. Selectors use the HTML attributes href , id , class , or tag to select a specific element from a web page, and then returns the information we need from that element. The documentation of Scrapy gives us excerpts of how we are to implement Selectors:

```
>>> from scrapy.selector import Selector
>>> body = '<html><body><span>good</span></body></html>'
>>> Selector(text=body).xpath('//span/text()').get()
'good'
```

[<https://docs.scrapy.org/en/latest/topics/selectors.html>]

Where the body is the HTML document, and the Selector object is a class predefined by the Scrapy developers. The parameter text in line 3 “Selector(text=body)”, we see a parameter which will be passed to the Classes `__init__()` function, this is a compulsory function which will run on the initialisation of the class, performing set-ups and initialising class attributes.

From our Selector() object, we can call a function xpath() (Selector(text=body_of_html).xpath(target_html)) where we pass the string of the xpath to search within the HTML element.

The ability to specify *attributes* (denoted by ‘@’ in xpaths and ‘::’ in css) is useful in being able to extract specific attributes of HTML elements, such as;

- Extracting links: @href
- The class of the HTML element: @class
- The id of the HTML element: @id
- The txt of the element; @text

We can even use different selector functions to specify whereabouts on the HTML element to extract, xpath was mostly used in this project due to the simplicity, but in instances xpath does not work we can use css(). CSS is usually what the stylesheet.css files in HTML directories are composed of. CSS is what gives HTML documents their colour, shape and layout, and are usually inherited into HTML elements using the “class” and “id” attributes, and can even be written into the HTML using the “style” element.

Taking HTML document:

```
<HTML>
  <body>
    <a href=http://ul.ie >University of Limerick</a>
  </body>
</HTML>
```

Demonstrating

With target being the text (“University of Limerick”):

```
xpath:      /html/body/a/@text
css:        html body a ::text
```

With target being the link (“http://ul.ie”):

```
xpath:      /html/body/a/@href
css:        html body a ::href
```

Using Scrapy:

With target being the text (“University of Limerick”):

```
xpath:      Selector(text = html_document).xpath('/html/body/a/@text')
css:        Selector(text = html_document).css('html body a ::text')
```

With target being the link (“http://ul.ie”):

```
xpath:      Selector(text = html_document).xpath('/html/body/a/@href')
css:        Selector(text= html_document).css('html body a ::href')
```

Both the xpath and css functions of Scrapy rely on parseL, another open-source library for parsing HTML. parseL uses regex and the native Python lxml parsing package to search the HTML for the specified element or attribute.

Often we need to figure out the xpath to the link we wish to navigate to, in this case it is the timetable URL. Xpaths are a tricky thing to compose by yourself when a webpage has a lot of HTML tags and elements. Luckily we can use the inspection tool in any browser to find the html element and extract it's xpath, css and others. [https://www.scrapestorm.com/tutorial/how-to-find-xpath-in-chrome/] Before doing so, it is important to go into your browser settings and turn off Javascript, since scrapy doesn't support JavaScript, you need to look at the webpage like how scrapy does.

You can turn off JavaScript by changing settings in your browser, or use the Scrapy Shell and its native function *open()* passing in the response, which will show you what scrapy sees in browser. If you wish to do this from a script you need to import the *open_in_browser()* function and pass in the response to this function.

We use the selectors so often Scrapy comes with a shorter version to which we can use. Instead of importing the Selector object, we can directly call *xpath()* and *css()* from the *html response* object.

Using Selector Class;

```
[In [26]: from scrapy.selector import Selector

[In [27]: link_html_object = Selector(text=response.body).xpath("/html/body/a/@href")

[In [28]: print(link_html_object)
[<Selector xpath='/html/body/a/@href' data='http://ul.ie'>]

[In [29]: link = link_html_object.get()

[In [30]: print(link)
http://ul.ie
```

- Here we import the Selector class, and initialize the class using the *body* of our *response*.
- We then call its “xpath” function, passing in the path to the link we wish to extract.
- *link_html_object* is now an instance of this selector class, there’s the data containing any href attributes of the xpath we provided. We can treat this variable as another Selector() class, just localised to only be able to access elements within the provided path. This function will always return a list, as we can point to multiple elements using our xpaths.
- Using the function *get()*, we extract the first element of the selector list. In the instance we collect a list of urls, and want to extract all of them too, we can use *getall()*, which returns a list of the data present.

Using Response Class;

```
[In [15]: link_html_object = response.xpath("/html/body/a/@href")

[In [16]: url = link_html_object.get()

[In [17]: print(url)
http://ul.ie
```

- Here we simply call the response object (created by our script, from the initial request to the target site), and call the *xpath* function directly from it and display our data similarly to above.

We see using the response object is a lot more convenient for us. This was selectors and how we can use them to find different parts of a html document, and extract information like links and text.

Response Class:

The response class is the first thing returned from our scraper when it makes a *request*. The response class holds all the information about what the target site thinks of our request. It contains the html document to be returned among other data like;

- status codes (200 = Success, 404 = Page not Found, 401 = Unauthorized Access)
- headers (user-agent, cookies)
- body (body of the html document)
- meta (meta is a field we can store data in, if we needed to pass data between requests and responses, as the function that sends the request can input meta data that the function which interprets the response can use.)
- text (all the text of the html document, including the HTML and text attributes)

The response class also has a function called “follow()”, we can use follow by calling our response object and then calling follow; response.follow(url), url can be a relative url (/academic-registry/) or an absolute url (https://www.ul.ie/academic-registry/). *Follow* also comes with a parameter called “callback”, to which we can pass in a function, when the response from the *follow* is returned from our target url, the response is immediately passed to this function and is performed.

Here we see an excerpt from a UL scraper I built. This scraper was a prototype and was not used in production. We will talk more about why it wasn’t used in Chapter 5, which includes a call from UL cyber security.

```
def LoggedIn(self, response):
    self.timetable['misc']['pwd'] = '' # clear password from memory.
    element_for_student_timetable = '//*[@id="MainContent_StudentTile"]/a/@href'
    # here we implement a shortened xpath type using regex.
    student_timetable_link = response.xpath(element_for_student_timetable).get()
    return response.follow(student_timetable_link, callback=self.GetTimetable)
```

For context; by the time we get to this function we have just logged in and want to tell the scraper to go to the timetable URL. I could’ve hard coded the timetable URL having the scraper

go from log in redirected straight to the timetable URL, but something like that can be pinpointed by IT administrators if they ever wanted to lock down on bots, so I try to make the actions it took look as human as possible, such as following the correct user flow.

In the parameters for the function we pass in an instance of the scrapers class itself (*self*), similar to how we can call “response.xpath()”, we can call “self.function()”, which is the same as calling the scraper itself “UL_scraper.function()”. This is mandatory when creating a function within a class.

Firstly we wipe the password as I do store the data submitted by users, and of course we do not want to store passwords, so as soon as we are logged in we wipe this from our memory.

Secondly, we define our xpath to the link. Here you see a shortened version of xpath, it is a relative xpath, and utilises regex. The * in regex represents anything, it’s a wild card, we tell it to find “*anything*”, *, that has an attribute of id which is equal to “MainContent_StudentTile”, go into this elements “a” tag, and look at the href attribute.

We then pass this xpath into our *xpath()* selector function, and extract the link using *get()*. We pass this link into the response’s *follow* function, along with the function to call upon *callback*, which is the GetTimetable function where we parse the timetable page of timetable.ul.ie.

This is a brief overview of how Scrapy will play a big part in this project, from utilities to implementations. There is so much more which Scrapy can do from requests all the way down to storing the data, but a top-level view is sufficient enough for it’s current use in this project.

Scrapy is very useful with HTML documents not heavily reliant on JavaScript. But we often see the case in which when we turn off Javascript for web pages, and the page completely breaks, or our desired information is not displayed. Often we are met with a web page saying “Please turn on JavaScript and reload”.

For DOM (Document Object Model, pages heavily reliant on JavaScript) webpages, we cannot use Scrapy, and require an alternative. We wish to still use scrapy, as it can be run in a Linux server environment and requires no arbitrary software, so it runs quite happily by itself and

Left: timetable.ul.ie with JavaScript, Right: timetable.ul.ie without JavaScript.

As seen in the figures above, timetable.ul.ie without JavaScript cannot run the function to display the year field, which we were able to fill when JavaScript was enabled. We will explore more DOM in the GUI portal, and how I implemented JavaScript to change and display selections dynamically.

Selenium comes with a lot of the same benefits that scrapy comes with, such as custom headers, user-agents, proxies. Selenium also operates similarly to Scrapy in how we can handle xpaths and selectors. A benefit with Selenium is that we can even change our settings to appear we are using a mobile, and navigate the mobile version of sites to get features not usually available. I recently used this ability to automate posting to Instagram, the ‘create post’ option only being available on the mobile site or app.

Just like any other browser we can control multiple tabs, open multiple windows, scroll down, scroll to specific elements, run JavaScript code in the browser just like in the console of the developer tools of most browsers. We can also type and use keyboard shortcuts just like we normally would, so Selenium is ideal for being able to simulate any human flows.

Before writing any code we need to have a browser installed, we then need to check our browser version and download the corresponding driver which we can get online. Put the driver in the same location as your script, you can use the driver from a different location, but this complicates the code as we will need to point the script to the file directory of the driver executable. If you were to move the code to a different machine, you’ll have to fix the path, and more than likely by then you’ll be using a different browser version, and so you will need to download the corresponding driver.

List of browsers compatible with Selenium for Python:

- Chrome
- Chromium
- Microsoft Edge
- Microsoft Internet Explorer
- Opera
- Firefox

- Safari

I chose Chrome for my browser, and therefore had to use the chrome driver from chromium.org. There is no specific reason as to why I chose it, there are definitely more lightweight browsers out there like Chromium, a browser for Raspberry PI, which would probably have been easier on a CPU, but realistically for this task we can use any browser we like, it doesn't change any code in our script other than xpaths, unless you use similar browsers like Chrome and Chromium, which would have the same xpaths.

To automate navigation of a webpage, we first need to find out a user path on our target site which we can replicate within Selenium, take notes of every action we take, such as clicking on drop-down menus to expose other links not directly available.

My objective is to automate the selection of course, and year on timetable.ul.ie, displaying every timetable combination possible. [Timetable.ul.ie](http://timetable.ul.ie) is composed of two inputs, course and year. We can imagine the loop as being “for each year of each course, display timetable”.

The only package present in the Python Selenium Framework, is the webdriver package, an API for interacting with all the different drivers of different web browsers. We will only be using the Chrome driver, so that's all we need to import from Selenium.

We initialise the driver by setting a variable equal to the browser class (`Chrome()`, `Firefox()`) of the webdriver package;

```
from selenium import webdriver

# Chrome browser, if driver is in path/local directory.
driver = webdriver.Chrome()

# Firefox browser, if driver is not in path/local directory.
driver = webdriver.Firefox(r'c:/users/mario/dir/to/driver.exe')
```

This *driver* object is now a programmable version of our browser. Using this driver we can perform normal actions such as telling the browser to scroll down, scroll up, click, as well as typing to fill input fields, or shortcuts like page up, page down, we can even open multiple tabs. But most importantly we can request websites.

```
from time import sleep

url = 'timetable.ul.ie'

driver.get(url) # code will block here until a response is received.

sleep(5)
```

At the same time our script makes itself down the code, we can watch it perform the operation in the browser window which opens upon initialisation of the driver. We can control this window to be displayed or not (headless), I prefer to keep it open, because if the script fails, the window will remain until closed and you can inspect where it failed and why.

Upon landing on the timetable.ul.ie page, we want to go to the tile “Course Timetable”, we can simply tell the *driver* to find this element and click it.

```
course_timetable_tile_xpath = '//*[@id="ctl01"]/div[5]/div/div/div[3]/a'

tile_html_element = driver.find_element_by_xpath(course_timetable)

tile_html_element.click() # clicks element
```

For this simple operation, all we will be using from Selenium is the function `find_by_xpath()` of the `webdriver` class, and `click()` of the `html` element class. Although we could've used other methods of finding HTML elements, such as;

Taking HTML element:

```
<form id='course_selector_form'>

    <input name='course' id='course_input' class='course_class' />
    <input name='year' id='year_input' class='year_class' />
    <button value='submit' />

</form>
```

With *driver*. as the prefix to each of these functions;

- `find_element_by_name('course')` will return the first input html element.
- `find_element_by_id('course_input')`
- `find_element_by_class_name('course_class')`

- `find_element_by_tag_name('button')` will return the button HTML element.
- `find_element_by_xpath('html/body/form/input[1]')` will return the second input HTML element.

We can select multiple elements of the same attribute definition as above, by replacing *'element'*, with *'elements'*.

- `driver.find_elements_by_tag('input')` will return a list of the HTML elements with the tag *'input'*.

Selenium is capable of a lot of more human things than Scrapy. I believe the Scrapy method of scraping will become obsolete, and Selenium will reign supreme. As captchas get more and more advanced, with the newest version of Captcha, running processes in the background to verify human/natural movement[<https://www.theverge.com/2019/2/1/18205610/google-captcha-ai-robot-human-difficult-artificial-intelligence>]. We can add in margins of error for the Selenium script, to sort of “wander” its mouse, and take it’s time going through websites, we can fool these types of captchas.

This was the Selenium Framework for Python and how I plan on implementing it to extract information from timetable.ul.ie. We will also use Selenium to extract data from qualifax.ie and this will be discussed further in Chapter 4.

Scope 2: An API with GUI for a portal to serve and host data/tools for students.

In this scope the main package which I will be using is FastAPI. FastAPI is a python Framework in which we can create APIs and is composed of;

- WSGI (Web Server Gateway Interface, specification of a common interface between web servers and web applications)
- Werkzeug is a WSGI toolkit that implements requests, response objects, and utility functions.

[<https://pythonbasics.org/what-is-flask-python/>]

FastAPI also supports rendering of dynamic webpages, meaning we can use python variables within the HTML. This means Flask can serve as a full-stack solution, handling frontend (HTML, JavaScript) and backend (Python, Data). I did not rely my entire stack on FastAPI, I often find separating out projects makes it easier to work on the separated components, in the

event we wish to switch the frontend Framework for something such as Vue or Bootstrap, instead of the “*vanilla*” JavaScript, CSS and HTML I used.

For this to work we need three components, the API itself, a graphical interface/website, and a means of hosting the two. At the time I had just found out about Heroku, a free means of hosting an API and trying to get a net zero cost to this project, I went with it. I knew I couldn’t of hosted the entire web-app on Heroku, but I figured I could save on the CPU cost of the API, leaving the more costly web-server to just keep the frontend.

With my API hosting sorted, I needed to host the frontend, I had experience with Google Cloud and AWS cloud computing platforms, when I would need to run time consuming algorithms, I would host them in the cloud to relieve my laptop of the job.

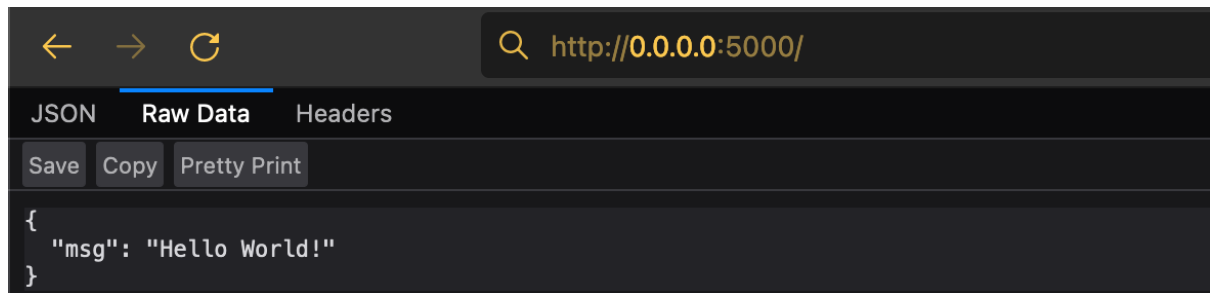
Google Cloud and AWS come with many cloud computing options, such as for ram, OS, number of CPUs, types of CPUs, etc. for this I am going to go with the bare minimums as we don’t require much, I opted for a Linux machine, with 1GB ram and 1 virtual CPU (t2.micro of AWS EC2 instance; <https://aws.amazon.com/ec2/instance-types/>). I chose Amazons AWS due to it’s free tier.

In our API we create endpoints which are basically functions called upon request to their URL path. We can have a working endpoint on our local machine in 7 lines of code, showing FastAPI really lives up to it’s name.

<pre># endpoints.py # contains functions API performs. from fastapi import FastAPI app = FastAPI() @app.get('/') # add url path async def hello(): return {"msg": "Hello World!"}</pre>	<pre># main.py # socket connections for API to run on. import uvicorn uvicorn.run("endpoints:app", host="0.0.0.0", port=5000, reload=True)</pre>
---	---

Left: file endpoints.py to put endpoints in, Right: file main.py to put socket settings.

On the left we are defining an asynchronous function hello(), which would be run upon a request done to our endpoint at “{host}:{port}{path}” which in the case above would be 0.0.0.0:5000/ which would then return our data. If we run *main.py* in our terminal using “python3 main.py”, we can visit the URL in our browser at <http://0.0.0.0:5000/>.

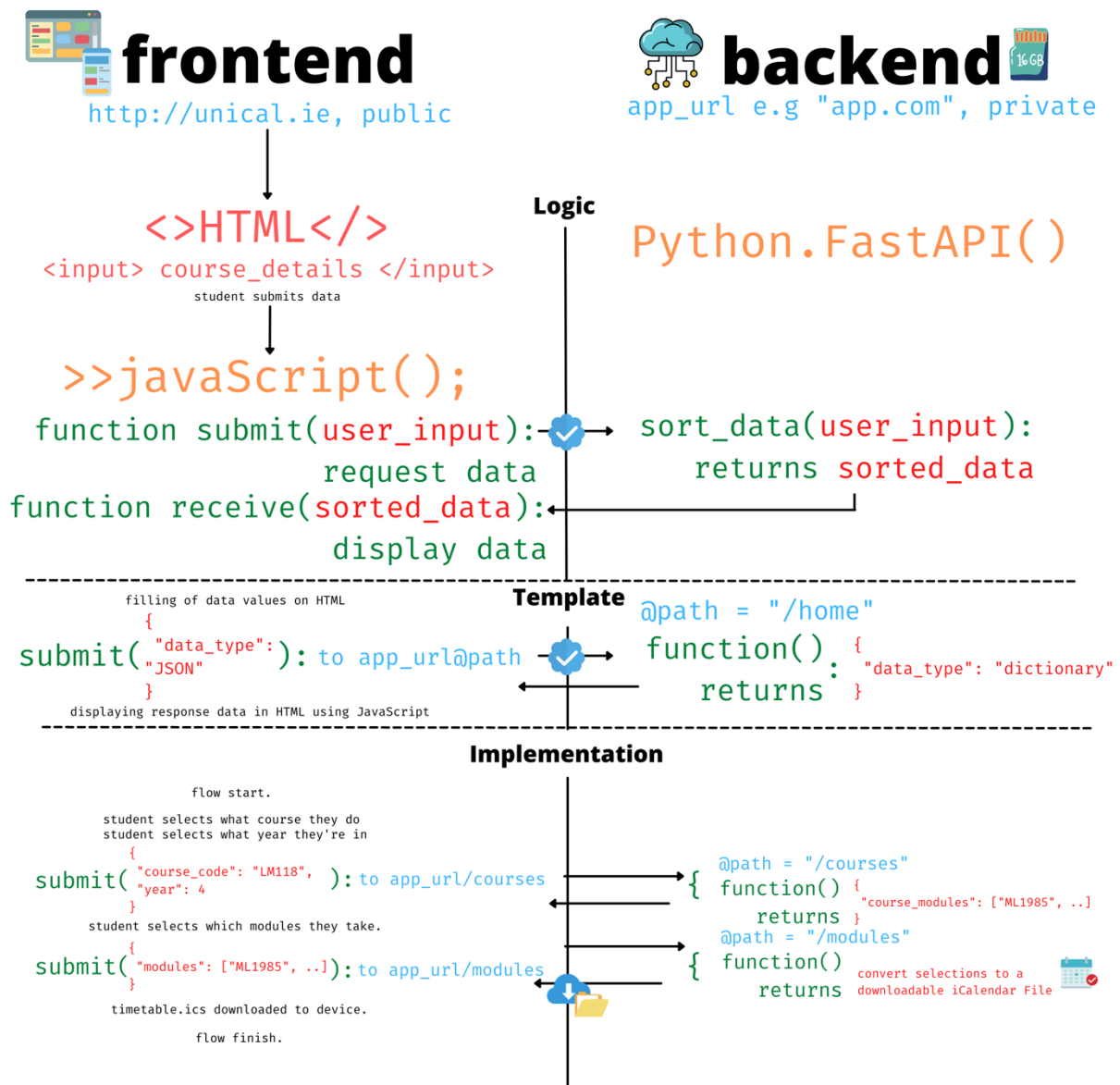


Local endpoint; <http://0.0.0.0:5000/> returning the defined data.

FastAPI offers a lot of Objects which we can utilize to return data to our users, different types of responses are;

- `StreamingResponse`; for a continuous buffer of data such as downloading a file or watching a video.
- `ORJSONResponse`; A fast JSON parser, for opening and reading JSON files, and sets the media type as json.
- `HTMLResponse`; Sets the media type of the response to a text/html response.
- `RedirectResponse`; Sends the user to the URL returned from our Python function.

This can be very useful in the event we want to return any types of data at all, for the current implementation the only data we need to be able to return is JSON data, and binary data (our iCalendar file, .ics file format). For the JSON data we can return the Python dictionary data type, which we can convert to JSON using the JavaScript of our front-end.



Planned implementation of web-app flow.

In the figure above, section “Implementation”, we see “flow start”, in more detail;

1. Student inputs their course year and course code into the HTML.
2. Upon submission, JavaScript will collect the values from the HTML make a call to the API url and corresponding endpoint ('path' in diagram).
3. API endpoint function filters dataset of all courses, returns modules found for the course year and course code combination.
4. JavaScript displays list of modules available.
5. Student selects modules they do, which is sent to API.
6. API generates iCalendar file by collecting all relevant events from modules, code year and course code selections, which is streamed back to the Student for immediate download to device.

Frontend:

For this we need to figure out how to prompt for inputs in HTML and how to request data using JavaScript. We want two types of inputs for the GUI, a “*select*” (list of “*options*”, e.g. 1, 2, 3, 4), and a simple *input* of type *text*.

<pre><!doctype html> <html> <body> <select> <option> 1 </option> <option> 2 </option> </select> </body> </html></pre>	<pre><!doctype html> <html> <body> <label>text input</label> <input type='text'> <label>numerical input</label> <input type='int'> </body> </html></pre>
---	--

Left; A *select* HTML element for limited options, Right; A *input* HTML element for custom inputs.

The code block on the right is for a keyboard input, I will use this to deter people not in UL from extracting any data or using the service at all, preventing unnecessary bandwidth usage and hopefully saving costs.

text input numerical input

Rendered HTML of *input* code block.

The code block on the right is a list of options to an input, I will use this as the year selector, by having options from 1 to 4, we can cater all years for all courses.



Rendered HTML of *select* code block of *options* 1 or 2.

Now that we have a way of inputting data, lets store it and get ready to send it to our API using JavaScript. Saving our *input* code block as *index.html* (index is usually the home page or landing page of websites.) file in our project folder, lets start on the JavaScript to extract a value from an input.


```

<!doctype html>
<html>
<body>
  <label>Course Code: </label>
  <input type=text id=course_code value="LM118">
  <button id=button>Submit!</button>
</body>
</html>

```

index.html

```

function getCourseCode() {
  var html_element = document.getElementById('course_code');
  var course_code = html_element.value;
  console.log(course_code) // prints to terminal.
  return course_code
}

```

function *getCourseCode()*; to get the value from the *input*.

```

var button = document.getElementById("button"); // find button.
button.addEventListener('click', getCourseCode); // bound the action 'click' on the
button element, to getCourseCode function. When button is clicked, getCourseCode will
run.

```

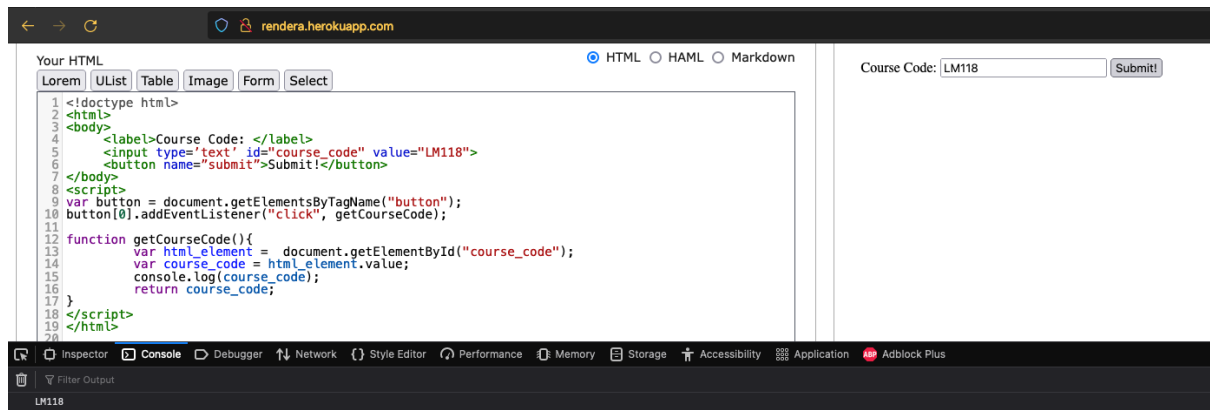
Code snippet to bound *getCourseCode()*; function to our button element, which will run on every click of the button element.

```

<!doctype html>
<html>
<body>
  <label>Course Code: </label>
  <input type=text id=course_code value="LM118">
  <button id=button>Submit!</button>
</body>
<script>
var button = document.getElementById("button"); // runs on load
button.addEventListener("click", getCourseCode); // bound the action 'click' on the
button element, to getCourseCode function. When button is clicked, getCourseCode will
run.

function getCourseCode(){
  var html_element = document.getElementById("course_code");
  var course_code = html_element.value;
  console.log(course_code);
  return course_code;
}

```



We see our HTML and JavaScript left, the rendered HTML on the right, and the console.log output at bottom (“LM118”), expose the console by right clicking on the webpage and inspect, then click console.

We point our variable *html_element* to the HTML element *input*, we use “*document*” object model as our representation of the entire HTML document. From here we can use a range of functions from *document* to find different elements, such as;

With *document.* as a prefix;

- `getElementById(element_id)` returns singular element where attribute id is equal to `element_id`.
- `getElementsByClassName(element_class_name)` returns list of elements with attribute Class equal to `element_class_name`.
- `getElementsByTagName(element_tag_name)` returns list of elements with a HTML tag equal to `element_tag_name`.
- `getElementsByName(element_name)` returns list of elements with attribute name equal to `element_name`.

Now that we have acquired the HTML element using the relevant means, we can access it’s attributes by simply calling an attribute which we see from the HTML tag;

Taking this HTML element as *html_element*;

`<input type=text id=course_code value="LM118">`

We can get it’s attributes by calling;

- `html_element.type` returns *text*
- `html_element.id` returns *course_code*
- `html_element.value` returns *LM118*

Once we have the data collected which we wish to query our database for, we can set up a “fetch request” to send this to our API endpoint. Fetch is a native JavaScript function, we can use it to request information from the server or API, all without reloading the webpage.

```
fetch(url, [options])
```

We pass in the URL of the address we want to request from, in this case it is the URL of the API, and a list of options to configure things like;

- Method; Methods are ‘GET’, ‘POST’, ‘DELETE’, ‘PUT’, and are the protocols in which we make requests over, GET is standard and simply gets data from the URL/API, POST is of more interest here as we can submit data to and receive data from our API.
- Headers; Set things like the user-agent, expected content-type (JSON, .ics file).
- Body; here we can store data in our request, on our API side, we will need to read this specific *body* field of the request to extract it.

There is many ways we can use *fetch*, we can add an *await* operator to have the code wait at the line of the fetch until a response is returned, or we can chain together multiple *then*’s, to synchronously run code on the response; I found this easier to debug, as the relevant code for each relevant request is chained together.

We can use the *then* function like so;

```
fetch(  
  url, [options]  
)  
.then(response => response.json());
```

```
var response = await fetch(url, [options]);  
  
response.json()
```

Left; using the *then* function, Right; equivalent piece of code using *await*

Using the *then* we create an alias for the object last returned from the previous function, so in a chain of 3 *then*’s, the third *then* has the result from the second *then*, so on.

```
fetch(  
  url, [options]  
)  
.then(  
  One => two  
)  
.then(  
  two => three  
)  
.then(  
  three => four);
```

So lets send data and receive it, taking the FastAPI endpoint;

```

# endpoints.py
# contains functions API performs.
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware

app = FastAPI()

app.add_middleware(
    CORSMiddleware,
    allow_origins=['*'],
    allow_methods=["post", 'get'],
    allow_credentials=True,
    allow_headers=["*"],
)

@app.get("/") # add url path
async def hello():
    return {"msg": "Hello World!"}

```

endpoints.py

```

# main.py
# socket connections for API to run on.
import uvicorn

if(__name__ == "__main__"):
    uvicorn.run('endpoints:app', host="0.0.0.0", port=8000, reload=True)

```

main.py

The only difference between this example and the previous example is the CORS middleware, which is basically some restricting security measures to ensure only listed origins (source of request), headers and methods are made to our endpoint, for demonstration purposes we don't need too much security, but we need the bare minimum before we can make a fetch request to the API from JavaScript, so a simple * denotes *anything goes*. Taking the HTML example and the JavaScript example;

```

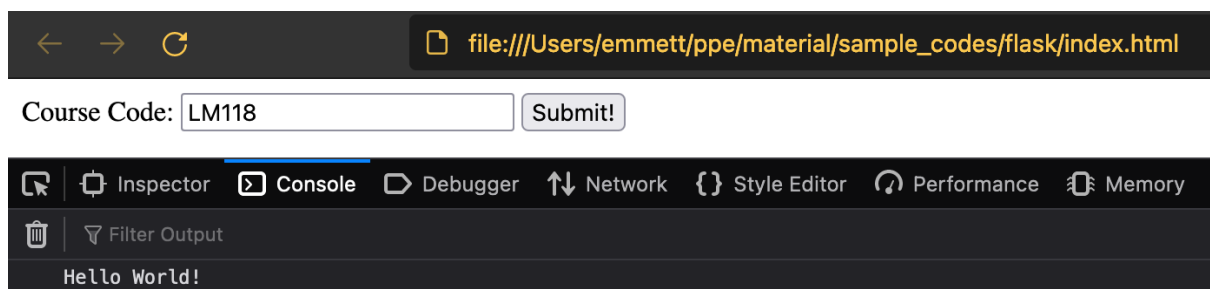
<!doctype html>
<html>
<body>
  <label>Course Code: </label>
  <input type=text id=course_code value="LM118">
  <button id=button>Submit!</button>
</body>
<script>
  var button = document.getElementById("button");
  button.addEventListener("click", submitCourseCode);

  function submitCourseCode(){
    fetch(
      'http://0.0.0.0:8000/'
    ).then(
      res => res.json()
    ).then(
      data => console.log(data['msg'])
    )
  }
</script>
</html>

```

Index.html, containing HTML and JavaScript.

Here we have done the same as before, except we bound any *click* on our *button* to run *submitCourseCode()*. I have configured the API endpoint, and the fetch request to be a simple *GET* request for just getting data, the API returns a dictionary with a key of “*msg*”, which points to the value “*Hello World!*”.



Upon click of “Submit”, ‘Hello World!’ is displayed in the console.

Now that we understand how to get data from our API, lets send data to it.

POST Request, sending data to API:

We can expand on the code above, and we are well prepared, being able to configure the *options* of the *fetch* we can change it to a *post* request, to send data to our API. We will store the *course_code* in the *body* of the *fetch* request.

In our API endpoint, it will need to read in the body of the request from the *fetch*, find the *course_code* within the *body* of the *request*, do something with it and send it back. This “do something with it” will be retrieval of course information from the UL timetable database when we have it all extracted.

So let’s define a new POST endpoint to add to *endpoints.py*;

```
from fastapi import Request
from json import loads
@app.post("/course-submit") # add url path
async def course_submit(incoming_request: Request):
    body_binary_string = await incoming_request.body()
    body_dict = loads(body_binary_string.decode())
    course_code = body_dict['course_code']
    msg = "You take: " + course_code
    return {"msg": msg}
```

In *endpoints.py*, new endpoint to which will accept post requests

Here we create a new path “/course-submit”, only accepting ‘post’ requests. We assign *incoming_request* to the *Request* object. *Incoming_request* contains a lot of information about the source of the request, such as the url of it’s origin, the method, and other things that we can define from our *fetch* in the JavaScript.

We *await* the full materialisation of the *Request*, and call the function *body()* from the class *Request*, this simply extracts the *body* of our request, although not in a very useful format, a binary string containing our data; `b'{"course_code": "LM118"}'`. We could use string manipulation techniques to extract the course name but we did not intend to do this, so decoding the binary string turns it to a plain-text text, and use the JSON parser package for python to parse a string using *json.loads(string)*.

The JSON parser will preserve the JSON data formation, and convert it into the JSON equivalent for Python, a dictionary; `{ 'course_code': 'LM118' }`, here we have the data in a key value format, so all we have to do is call the correct key (*course_code*) from the dictionary to get it’s value (*LM118*). Our function returns a dictionary consisting of a message of what course was submitted.

With logic in our backend implemented, let's prepare the *fetch*, we need to change the URL to match our new endpoints, as well as adding options to configure the fetch method to 'post'. We also need to reimplement the method of getting the value from the input, and send this value in the body of our fetch request.

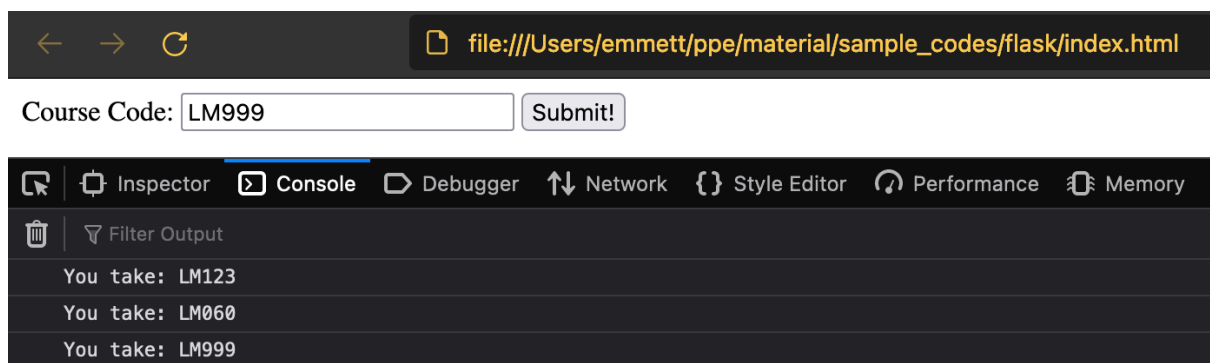
```
function submitCourseCode(){
    var course_code_input_value = document.getElementById('course_code').value;
    var data = {'course_code': course_code_input_value};
    var string_representation = JSON.stringify(data);

    var opts = { method: 'post', body: string_representation}
    fetch(
        'http://0.0.0.0:8000/course-submit',
        opts
    ).then(
        res => res.json()
    ).then(
        data => console.log(data['msg'])
    )
}
```

In index.html, redefine submitCourseCode() to get *input* value and *post* it to our new endpoint.

Using the same methods as before we extract the value from our *input* 'course_code', we assign it the same key we are expecting our data to be stored in in our API endpoint. We need to transfer the string representation of our JSON data, not the JSON data itself (due to compatibility issues across Python and JavaScript), we simply *stringify* our *data* using the *JSON* package native to JavaScript.

We configure our fetch request in the opts object, setting the method to 'post' and the body to the string representation of our JSON data. We simply make the request like we did before from there. Opening *index.html* and pressing submit, we get;



The functionality of index.html after adding the new post endpoint.

The JavaScript in our index.html now takes the value inputted at the moment of clicking “Submit!”, sends it to endpoints.py, which returns the msg, upon receiving the msg from our endpoints.py, the JavaScript in index.html logs the msg to our console.

Already I am seeing a template use for communication between frontend (index.html) and backend (endpoints.py, main.py). One thing this code does not do is display the message to the average user, right now we just log it to the console; *console.log(anything);*

Displaying Data:

We want to display messages to the user such as error messages, if their inputs aren’t valid, or a given course code doesn’t exist, and most importantly we will need to display a list of module codes to the student. We will need a way of creating HTML elements and add them to the html document in a specific position using JavaScript.

Thankfully we can create elements in a similar fashion as to how we find them, using a function of the *document* class, *createElement()*;

To build the element;

`<p>You take: LM118<p>`

Text HTML element to display a message, where LM118 is the course submitted.

We can create a function called *displayMsg()*; in our index.html to append the text element to the *document* object. We will call this function from our last *then* of the *fetch*.

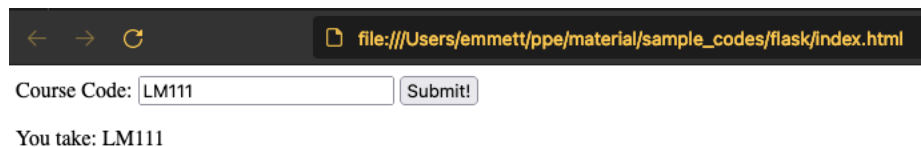
```
fetch(
  'http://0.0.0.0:8000/course-submit',
  opts
).then(
  res => res.json()
).then(
  data => displayMsg(data['msg'])
)
```

Calling *displayMsg()* and passing in our message received from API.


```
function displayMsg(msg) {
  // creation
  text_element = document.createElement('p');
  text_element.id = 'msg' // set the id
  text_element.innerText = msg; // update the text
  document.body.append(text_element); // append it to the body
  return true; // exit the function
}
```

Function *displayMsg()*, to create an element and display *msg* variable.

The function is straight forward, although to control the position of the element we need to note where we append the element to. Instead of just appending it to the documents <body>, we can append it to any element using any of the “document.getElementById...” functions.



A screenshot of a web browser window. The address bar shows the file path: file:///Users/emmett/ppe/material/sample_codes/flask/index.html. Below the address bar, there is a form with a label "Course Code:" followed by an input field containing the text "LM111". To the right of the input field is a button labeled "Submit!". Below the form, the text "You take: LM111" is displayed.

Results of the implementation of *displayMsg()*

Here we have now figured out a way to change our HTML, by utilising the JavaScript Document Object Model; *document*. We can now send and receive data between our backend and frontend, take inputs and display data to the students.

Using this framework and implementation I am confident we can create a portal to which we will can receive course data in the frontend, pass this to the backend where we filter our timetable data and return a list of modules, display them, then receive a selection of modules, to which we can generate an iCalendar file filled with the date and time of the corresponding Lecture, Labs and Tutorial events.

Demonstration project is available at https://github.com/xemmett/flask_app/

Scope 3: An information salient keyword extraction.

Basically I want to be able to extract sub-data from the data itself, I want to identify structure within a body of text, and I want to be able to find similar courses. I already am thinking to go institution to institution, as we would expect to find similar structure among the fields, and similar language, but we cannot use the same method of extracting things like LC requirements or course contents in NUIG as UL, so we need a single function that can identify these things when shown a sample of documents.

I also want to apply the function to the approach of going similar course to similar course, if I can identify similar titles and content within popular fields, weighing emphasis on the rarity of words like “Engineering”, “Sciences”, “Fourier Series”.

So our scope is:

- A single function to identify structure and important strings and terms, especially key pairs ("English: H2", "O3/H4 in Biology", "Graphic Design: Year 1: Students will....") when given a sample of similar terminology (course title approach) or similarly structured (course provider approach) documents.

I believe by being able to reduce the text data into hyper organised classified data, we can use this pipeline on any academic course document and have it ready to be applied in training machine learning models such as recommendation models, information system models, and even in text analysis, chatbots, and the list goes on.

The idea is to build a pre-cursor that reformats existing un-categorised data, into a common format with predictions of what the course document could be about, either stating a course name or a course discipline.

One method we will use we spoke about already in the scraper section as Regular Expression (“Regex”).

A recommendation system was the original goal, but I found the textual data was too “dirty”, causing the similarity analysis’ done with Gensim and clustering done with skikit

Recognition of Pattern Identified Entities: Features such as telephone numbers, e-mail addresses, quantities (with units) can be discerned via regular expression or other pattern matches.

Document clustering: identification of sets of similar text documents

Pre-processing usually involves tasks such as tokenization, filtering and stemming.

Chapter 3 Architecture (Tech Stack)

In this chapter I will go through my “tech stacks”. Technology stacks are the sandwich that holds the entire software project together.

Chapter 4 Execution (If perfection perfected itself)

Chapter to start on odd number page.

Chapter 5 Problems and Solutions (Where there's a will there's a way)

Java python tabular package

Information consistency issue

Cybersecurity with UL

Chapter 6 Background Interests, Hypothesis' and Analytics

Chapter to start on odd number page.

Chapter 7 Conclusions and future work

Information consistency systems, one source of truth.

Building image recognition scrapers, to avoid xpath and implement computer vision to identify and categorise data, using OCR and tesseract.

AI generated student and lecture content.

Better course search engine. Better recommendation to previously viewed courses.

References

Replace this text with the references. References to start on odd number page. Use the IEEE citation style for references.

Appendices

Replace this text with a list of the appendices. Appendices to start on odd number page.

Appendix A: Updated project Gantt chart

Replace this text with the project Gantt chart. This will be an update of the Gantt chart presented in the interim report, and reflect changes made during the project implementation. Appendix to start on odd number page.

Appendix B: Final presentation slides

Replace this text with the project final presentation slides (2 slides per page format). Appendix to start on odd number page.

Appendix C: Project poster

Replace this text with a copy of the project poster.

Appendix D: Project progress reports

Replace this text with the project progress reports, weekly or monthly, if created. If progress reporting was not part of the project, this appendix can be removed.