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

21/3/2022

Abstract

The creation and demand of digital resources for third level institutions has steadily risen for years, from developing infrastructure to submit assignments online at the start of the century, to adjusting to complete digital delivery in 2020, due to the pandemic. Student centric applications will play a key role in digitization, built using pre-existing university frameworks and well populated data repositories. To grasp this better, I created a webapp, http://unical.ie which creates a downloadable calendar of a students timetabled course events. The result was a deep-understanding of data availability, extraction, cleaning, protection and privacy.

As digitization assists institutions in reaching people, proper course selection processes will be needed to further filter out course changes and deter drop-outs. Today comparison and selection is harder due to un-structured information. Thus, I hoped to remove the typical keyword search done today; I investigated how to create a tool in which applicants can input a body of text and receive course recommendations. The tool is developed using data from the course directory of Qualifax to derive a testing set for Machine Learning Applications such as a recommendation tool.

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	**Emmett Lawlor**
Signature	**Emmett Lawlor**
Date	

Table of Contents

1. Introduction

This project outlines my experience building an un-official university webapp through extraction and formatting of data; I also investigate a pipeline to categorise and format unstructured course descriptions.

The webapp is a wrap-around application of data from my university portal, timetable.ul.ie. I initially sought to replace the active process of having to check a static webpage for scheduled classes. Thus I created http://unical.ie, which converts the static timetable webpage into a downloadable file in Universal Calendar Format (UCF). Compatible with all devices, the UCF file places timetabled events into the device's integrated calendar, with a reminder 15 minutes prior to the event.

Thereby making preparation for a class more relaxed, students can continue to focus on their studies instead of the time, room number, among other unnecessary awkward steps (*Figure 1*). Instead, students wait for the event to be pushed to their device, a small byte of information for your next 1-2 hours (*Figure 1.1*).

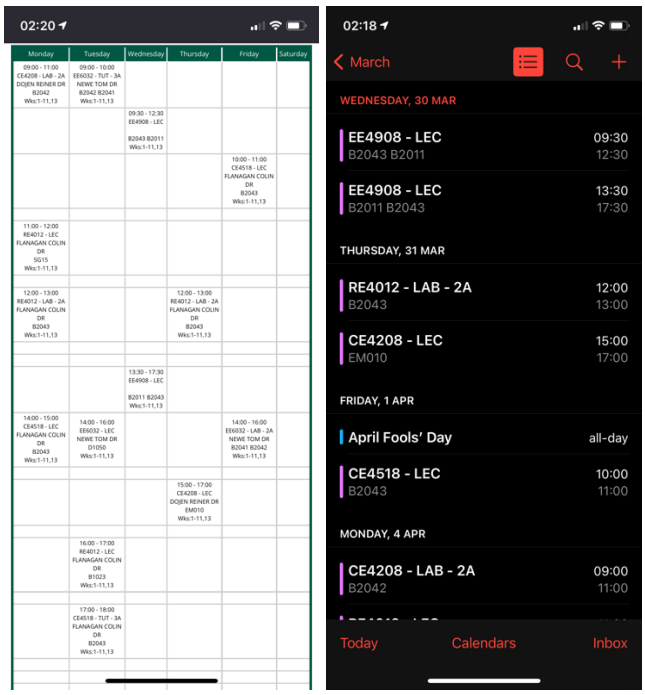


Figure 1.1: (Left) Standard university timetable. Retrieved upon log-in to timetable.ul.ie, days are left to right, time-slots are top to bottom, weeks which event are active are listed at bottom of each event, all relevant points of information for one event.

Figure 1.2: (Right) Calendarized version of timetable for Wednesday week 10 supplied from http://unical.ie, clearly displaying relevant information in a format which can be utilised to send reminders and gather a quick glance at your day.

This part of the project evolved from a personal application to public; being used almost 600 times. Along the way issues were encountered extracting data, and preventive measures from the university, one such was a UL cyber-security call, requesting that I don't release this webapp after I reached out to notify of the challenge I was facing extracting data for

individual timetables and asked for a work-around to my current Unical implementation of students giving me their student ID and password to SULIS, but was told such does not exist and I am not allowed to request passwords.

After the prototype being rejected by the university, I had to derive ways to extract the data needed from UL without actual access to the data needed. Therefor I built multiple data extractors, or “scrapers”, to gather all the data needed for the webapp to function, every course timetable from timetable.ul.ie and the required books per module from bookofmodules.ul.ie. This extracted data was then used as foundation for the webapp.

I see Unical as a one-use gateway at the beginning of each semester, for existing students to get their timetable and books and any other amenities that can assist in their organisation. This application could aggregate all the awkward pre-cursor steps to get prepared for the semester in and out of the classroom, just go to one site and get everything you need.

Unical will demonstrate the need for hyper-organisation between digital and physical realms and raises the conversation of data privacy and accessibility, as well as the usefulness of applications not developed directly by the institution, thus; if Universities should allow third-party applications from students.

In this project, we investigate textual data of course descriptions and names from Qualifax, to build a categorization pipeline, which would produce a meaningful tag or “keyword” for the inputted course description. To gather data I extracted 15,000 courses from Qualifax.

Using the Qualifax data; we gather an understanding of how academic institutions structure and word their course descriptions. With the core motivation behind this being to make finding a course easier, therefor we need to make the document easier for interpretation by a computer program; I decided to start with finding unique characteristics about the entire data set and using this to programmatically go through each entry in the data, and derive “keywords” and use this to group documents into categories.

By assigning word categories we could theoretically train a Machine Learning model on this corpus. I originally investigated building a classifier using unsupervised (non-categorised) data, but the classifier showed early signs that the accuracy is sub-par. This is due to the un-structured nature of the document; coming in many shapes and forms across various different institutions.

This puts the importance of consistent and well formatted up-to-date information under the microscope, for Universities to have their courses appear in recommendation engines, they will need to structure it in a plain way for the AI to understand it better.

Means of filtering and sorting this data can’t be done without understanding the data first, but once possible the idea would be to create a wraparound application of this data to make it easier for people to find the right course for them.

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 and how I could use them in each scope.

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

Checking the top ten most in-demand programming languages or most popular, Python JavaScript and Objective-C are always high in the list. Python is created from Objective-C, it is a wraparound which provides simpler, closer-to-English code compared to Objective-C or JavaScript.

Python is an Object-Orientated programming language, meaning our written code can be broken into Classes, which are workspaces that embody functions grouped together to perform specific repeated tasks. We give these classes meaningful jobs in our software's functionality.

We can download external "libraries" or "frameworks" for Python, which will contain a quantity of pre-defined classes. These frameworks are often written and maintained by a community of individual software developers, which is called "open-source" software.

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. (1)
- Scrapy; A web scraping package, used to request and parse static HTML documents. (2)
- 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. (3)
- icalendar; a framework which can format event data into Universal Calendar Format. (4)
- FastAPI; an API framework for Python. (5)

Choosing Python as the language is only the first step. Deciding on frameworks is the next step, for this we will need to figure out what it is we want the program to do, and find a framework or frameworks, which can help us achieve our scope.

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. Course Keyword Extraction.

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. (6)

This was actually my first introduction to web-scraping. I worked as an intern in a digital marketing company where I developed a tool which would compare products prices. The scraper/bot would aggregate the price of a product across multiple online outlets. Thus whichever outlet holds this data, can make informed business intelligent decisions on sales, discounts or pricing to have an advantage over competitors.

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 webpage documents based on keyword similarity of a given input. (7)

Bots are controversial topics. They are seen to be "bad" in the public eye, often being credited to worldwide shortages in consumer goods (8) and data privacy concerns. (9) There is a recognized importance of Text and Data Mining/Scraping, as data can provide valuable insights; with the right intent. I accordance with laws in the EU, I will state that all scraping activities were conducted for Research Purposes only.

Some popular Python frameworks used for scraping are:

- BeautifulSoup; A simple library for parsing only of HTML.
- Requests; A low-level library for making requests and receiving responses, can be used to request HTML documents or request data from APIs.
- Scrapy; Using the Requests library, Scrapy can make requests, receive responses. Parsing of HTML documents is also supported.
- Selenium; Web-browser automation which uses the DOM (Document Object Model) to find HTML elements and extract data from them.

Having used all these frameworks, Scrapy and Selenium are the most sophisticated high-level scraping tools. BeautifulSoup only being capable of just parsing HTML documents, and Requests only being able to get HTML documents, it was obvious to go with a more ready-to-go-out-of-the-box solution, like Scrapy; but as with all software, Scrapy had its limitations, meaning we needed to rely on Selenium for some aspects of scraping.

Scrapy:

The framework for Python, Scrapy was released in 2008 and was developed by a web-aggregation company. (11)
Scrapy offers tools such as:

- 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 and 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. (11) 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”
```

Figure 2.1: Launching scrapy shell

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: (12)

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

Figure 2.2: Using scrapy.selector.Selector class to extract text from HTML element.

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() where we pass the string of the xpath to search within the HTML element.

The ability to specify *attributes* is useful in being able to extract specific attributes of HTML elements, such as;

- Extracting href links.
- The class of the HTML element.
- The id of the HTML element.
- The text of the element.

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>
```

Figure 2.3: HTML document which displays a text, when clicked will bring the user to http://ul.ie

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’)

Often we need to figure out the xpath to the link we wish to navigate to. 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. (13) 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 would.

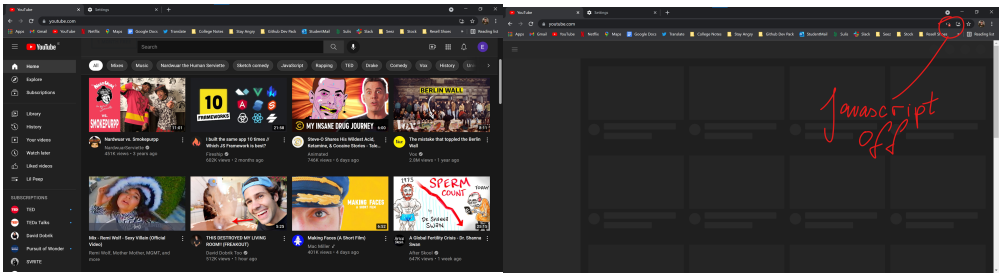


Figure 2.4: YouTube with and without JavaScript. Showing the importance of the DOM for most mainstream websites.

You can turn off JavaScript by changing settings in your browser, or use the Scrapy Shell and it’s native function *open(response)* passing in the response object, 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 can 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
```

Figure 2.5: Importing selector class and getting a href attribute using the xpath.

Here we import the Selector class, and initialize it using the *body* of the *response*.

- We then call it's "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
```

Figure 2.6: *Using response class to extract link from HTML.*

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

Here we see an excerpt from a UL scraper.

```
def logged_in(self, response):
        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.get_timetable)
```

Figure 2.7: *A function called logged_in, which accepts the Spiders instance itself and the response. This would be placed in the callback of the follow() function from a “log_in” function.*

For context in Figure 2.7; 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. We get the timetable “tile” element using the xpath, we extract the link and then follow it, passing the function *get_timetable()* into the callback parameter. This function will be called by it’s predecessor function, as *get_timetable()* is called here, *logged_in()* was passed as the callback.

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 define our xpath to the link. The * in regex represents anything or a “wild card”, that has an id attribute 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 *get_timetable()* 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 its current implementation 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 like in Figure 2.4, 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, HTML heavily reliant on JavaScript) webpages, we cannot use Scrapy, and require an alternative. We wish to still use scrapy, as it can be run on a low-level Operating System and requires no extra software, so it runs quite happily by itself and requires little resources; perfect to be hosted somewhere with little ram to run on demand.

Selenium:

Selenium is a web automation tool created in 2008 by an employee Jason Huggins at a software company called ThoughtWorks. Selenium was created and evolved as a software automation test tool with a focus on web browser automation for navigation and testing of web-apps and websites. (14)

Selenium, unlike Scrapy, requires companion software such as the browser itself you wish to automate, and the suitable driver for the browser. From experience, this limits our hardware capabilities as if we wish to use a single board computer like a Raspberry Pi, it may not perform as well when having multiple browsers open performing operations, compared to running an equal amount of instances in scrapy.

This isn’t due to Selenium being slow, it’s quite fast, it’s the webpages Selenium is usually required to scrape. Rendering of a DOM, which uses a lot of JavaScript for animations and different HTML or CSS aesthetics (such as ripples/focus around a button or input field you select.)

Selenium also limits you to one browser, as the html of a webpage can change browser to browser, so the xpaths or css selectors you select will usually only work with the browser you extracted them from.

So why do we use it?; Selenium, using the power of the browser, can render JavaScript, and therefore widens our grasp at what websites we can scrape, as most up-to-date sites will not work without JavaScript (as scrapy sees them).

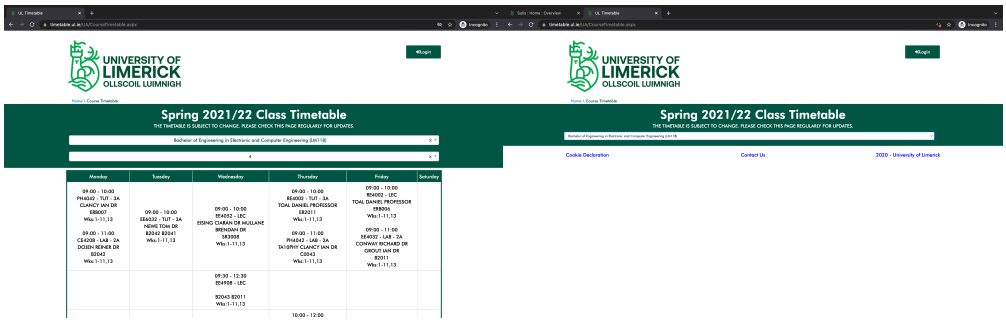


Figure 2.8: Left: timetable.ul.ie with JavaScript. Year field is presented after we enter our course title, which is needed to get timetable data. Right: timetable.ul.ie without JavaScript. Year field is not presented as the DOM cannot be updated due to JavaScript being disabled, visible by the red mark over the <> to the right of the url.

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. This is due to the Document Object Model not being able to be updated using the JavaScript. When we enter our course title, a request is made by the site to UL servers, requesting the course years, which then JavaScript will update the DOM with a select field with these year options.

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 on Desktop.

Just like any other browser we can control multiple tabs, open multiple windows, scroll down, scroll to specific elements and 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 therefor had to use the chrome driver from chromium.org.

To automate navigation of a webpage, we first need to map the 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, or needing to log in before requesting a page.

The only package present in the Python Selenium Framework, is the webdriver package, an API of an abundance of classes 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:/path/to/driver.exe')

# To download the relevant driver on execution of script
from webdriver_manager.chrome import ChromeDriverManager

driver = webdriver.Chrome(ChromeDriverManager().install())
```

Figure 2.9: Opening multiple browser windows using Selenium. Initialising the classes instantly opens and spawns a browser window, from here we can request a URL and begin navigation.

This *driver* object is now a programmable version of our browser. Using this driver we can perform normal actions such as requesting websites.

```
from time import sleep

url = 'timetable.ul.ie'

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

sleep(5)
```

Figure 2.10: Requesting timetable.ul.ie, then sleeping for 5 seconds. The sleep time can be varied, we use it to give the webpage a chance to fully load.

Upon landing on the timetable.ul.ie page, if 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
```

Figure 2.11: Clicking an element in Selenium, similar to Scrapy, we point our script to a specific element and extract.

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>
```

Figure 2.12: A template HTML element for a form, which has 2 inputs and a "submit" button. Each input element has a name, id and class attribute, we can use these to select specific data.

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.
- find_element_by_xpath('html/body/form/*') will all elements in form.

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'.

This was the Selenium Framework for Python and how I plan on implementing it to extract information from timetable.ul.ie and qualifax.ie. Ultimately Selenium can take advantage of the DOM to extract information, it is important to make the program simulate a human flow since JavaScript makes requests as you use any website, if you are making an unusual amount of requests and processing data too quickly, the server could prevent anymore data from loading.

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

Here we will need a virtual space which students can visit to utilise and interface with the tools we wish to deploy. We need to create HTML page with a DOM to curate retrieving and displaying of data based on student inputs. An API, or Automated Programming Interface will need to be created which will accept requests and return data in responses.

I was always interested in creating full-stack development and had never done so, I took this as an opportunity to learn. Full-stack development is the creation of the website/frontend/GUI and API/Backend/Databases, and a full-stack developer would be responsible for the entire development and deployment of the project at hand.

To complete full-stack development, I will write all of the HTML, CSS and JavaScript which makes up the frontend, and the Python which makes up the backend (API, databases). Here I will explore how HTML, JavaScript and Python can all interact and co-ordinate the structure and layout of the site. A method of data submission from frontend, data retrieval from backend and displaying on frontend again is needed.

Backend (Server Language, Python):

In the backend, 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.

FastAPI also supports HTML, this means Flask can serve as a full-stack solution, handling frontend (HTML, JavaScript) and backend (Python, Data), although 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.

In our API we create endpoints which are 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 its name.

```
# endpoints.py
# contains functions API performs.
from fastapi import FastAPI

app = FastAPI()

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

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

uvicorn.run("endpoints:app",
host="0.0.0.0",
port=5000,
reload=True)
```

Figure 2.13: 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 the URL template of "*{host}:{port}{path}*" which in the case above would be "{0.0.0.0}:{5000}{/}" which would then return our data, which is a dictionary in JSON format of key, value, where key is "msg" and the value is "Hello World!". 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/.

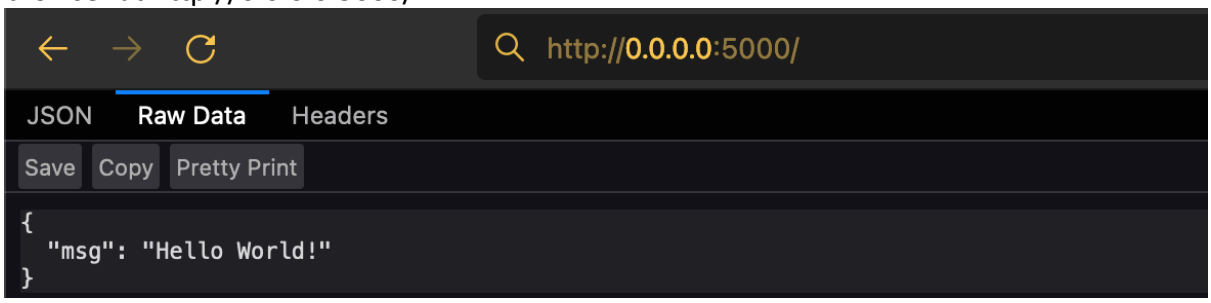


Figure 2.14: JSON data displayed of key = "msg" and value = "Hello World!", located at the local URL of 0.0.0.0, at port 5000, and path of "/". This is a fully functioning API endpoint.

FastAPI offers a lot of objects which we can utilize to change how the data will be sent in the response to our users, some 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 ({key: value}), which can be interpreted as JSON using the JavaScript of our front-end.

Frontend (Browser/Client Side):

For this we need to figure out how to prompt for inputs in HTML and how to request data using JavaScript. For this implementation we will investigate how we can mimic a course code and course year input and how to get the values of these inputs into JavaScript. Once we can get the values into JavaScript we can manipulate them and prepare them for sending to the API, but first let's just display the data to our console to ensure the JavaScript is working.

HTML structure:

We want two types of inputs for the GUI, a "*select*" with a list of "*options*", e.g. 1, 2, 3, 4, and a two *input* elements of type *text* and *integer*, we can begin prototyping the method flow of frontend and backend communication with a simple prototype.

```
<!doctype html>
<html>
<body>
    <select>
        <option> 1 </option>
        <option> 2 </option>
    </select>
</body>
</html>
```

```
<!doctype html>
<html>
<body>
    <label>text input</label>
    <input type='text'>
    <label>numerical input</label>
    <input type='int'>
</body>
</html>
```

Figure 2.15: Left; A select dropdown HTML element with two options, testing a course year drop-down. Right; A input HTML element for text and integer inputs for course code and course year.

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, to honour unjust data distribution and preventing unnecessary bandwidth usage; hopefully saving costs. We can save these to a HTML file, and open it in any browser to render the HTML layout.

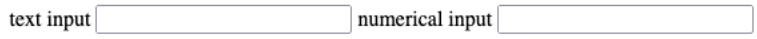


Figure 2.16: Rendered HTML of input code block. Text input accepts all characters whereas numerical input only accepts whole numbers.

The code block on the left is a list of options to an input, I will use this as a prototype year selector.

Figure 2.17: 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, let's 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>
```

Figure 2.18: HTML document of 3 elements nested in the body, a label, input and button. The input element has a type attribute of text, and id of course_code, and a value attribute which will pre-fill the box with a course code value.

Getting Data using DOM (Document Object Model):

JavaScript can be used to modify the DOM (Document Object Model) and get certain properties about HTML elements available through the DOM. (15) We will point a variable *html_element* to the *input* using the "*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 DOM, 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, or whatever value is inputted at the time.

Using the id attributes of input and button, we can find these elements in the DOM using JavaScript functions outlined above.

```
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
}
```

Figure 2.19: Function getCourseCode(): to get the value from the input element it calls the DOM object using document, we then get the input element using it's id. We then simply call the value attribute of the input element and log it to the developer console in the browser.

Next we want to bound this function to the click of our Submit button. So upon every click of the button, the function will fire; extracting the course code and logging it to the console. First we will need to find the button element using the document object model,

```
var button = document.getElementById("button"); // find button.
button.addEventListener('click', getCourseCode); // bound the event 'click' with the
button element, to getCourseCode function. When button is clicked, getCourseCode
will fire.
```

Figure 2.20: Code snippet to bound getCourseCode(); function to our button element, which will run on every click of the button element.

For the prototype demonstration, we can store the JavaScript in a <script> element of our HTML document under the <body> element. We can store the files in two separate files with our JavaScript in main.js and adding a script tag with a src attribute equal to the files path; *<script src='c:/path/to/main.js'>*.

```
<!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;
}
</script>
</html>
```

Figure 2.21: index.html with JavaScript code in a <script> element nested in the HTML document object.

We will then save this "*index.html*" file and open it in our browser.

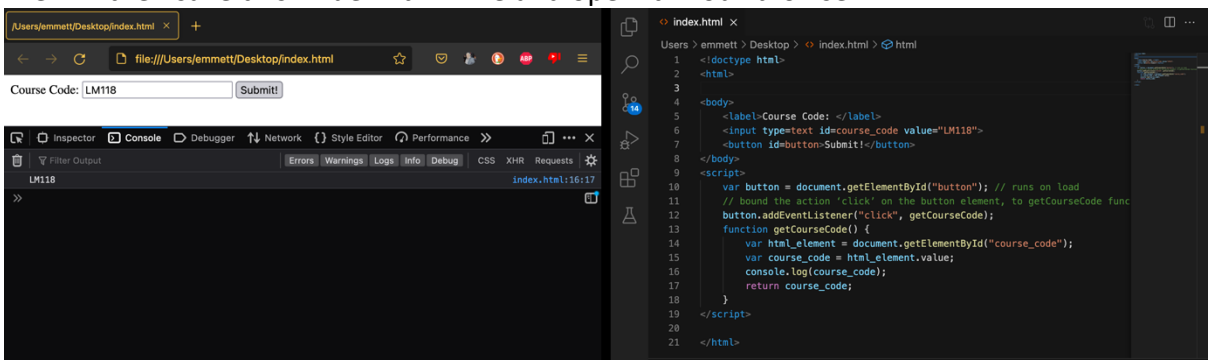


Figure 2.22: Open the developer console in your browser by right clicking and pressing "inspect", from here click "console", then the "Submit!" button and see the input value being displayed in the console.

In figure 2.22 see our Code on the right and the rendered HTML document on the left with the console.log output at bottom ("LM118"), expose the console by right clicking on the webpage selecting "*inspect*", then click "*console*".

fetch request, sending data to an API:

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 a server or API, all without reloading the webpage.

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

Figure 2.23: fetch request format. Where options would host a range of things like the type of request, the data we are sending, retries, timeout, etc.

We pass in the URL of the address we want to request data 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 *body* field to extract data from 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()
```

Figure 2.24: 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
   );
```

Figure 2.25: fetch template flow.

So lets send data and receive it, taking the FastAPI endpoint developed earlier, with an added middleware to handle Cross Origin Resource Sharing, or the transfer of data and communication between different websites of different domain names;

```
# 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 all urls.
   allow_methods=['get'],
   allow_credentials=True,
   allow_headers=["*"],
)

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

Figure 2.26: endpoints.py with added CORS Middleware to allow Cross Origin Requests, i.e. URLs not of the same domain as where our API is hosted ("0.0.0.0:5000" or "website.com").

The only difference between this example and the previous example is the CORS middleware, which comes with some restricting security measures to ensure only listed origins (source URLs of request), headers and methods can request from 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 *any* URL can make a request to the API.

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

if(__name__ == "__main__"):
   uvicorn.run('endpoints:app', host="0.0.0.0", port=5000, reload=True)
```

Figure 2.27: main.py example from earlier API prototype development.

Taking the HTML example and the JavaScript example from earlier, we can demonstrate the fetch request by replacing GetCourseCode with a function SubmitCourseCode, which we will evolve to get and submit data using a fetch request, first making sure the URL is correct by testing the current API implementation and display "Hello World!" from the API in the developer console;

```
<!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://0.0.0.0:5000/'
         ).then(
            res => res.json()
            ).then(
               data => console.log(data['msg'])
               )
   }
</script>
</html>
```

Figure 2.28: Updated index.html, with fetch request in function SubmitCourseCode which we will use to send our course code to the backend API.

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!*".

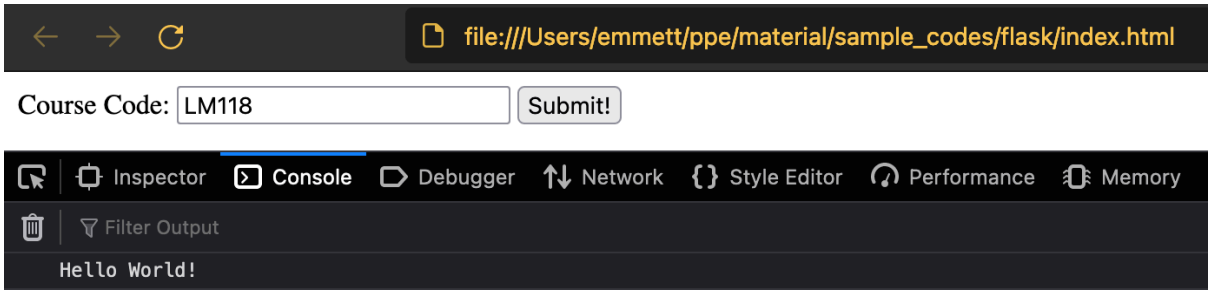


Figure 1.21: Upon click of "Submit", 'Hello World!' is displayed in the console.

Now that we understand how to get data from our API using fetch requests, 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 the Python script endpoints.py;

```
from fastapi import Request
from json import loads
@app.post("/course-submit") # add url path
async def course_submit(incoming_request: Request):
    # wait until request is fully loaded, data will be stored in the body, which we extract with body(). It will be a binary string in the form of b'{"key": value}'
    body_binary_string = await incoming_request.body()
    # use json.loads to convert a JSON string to an indexable JSON dictionary in Python. b'{"key": value}' of type binary, decode() -> '{"key": value}' of type string, loads() -> {"key": value } of type dictionary
    body_dict = loads(body_binary_string.decode())
    # index the Key 'course_code' to extract course code value from dictionary.
    course_code = body_dict['course_code']
    msg = "You take: " + course_code # Construct message
    return {"msg": msg}
```

Figure 2.29: In endpoints.py, new endpoint to which will accept post requests. Request is fully recieved, then we extract the body, decode it and convert it into a Python dictionary which we can index. The accepted data will be {"course_code": value }.

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, then using the JSON parser package for python to parse a string using *json.loads*(string), we can automatically convert a string in the format of a JSON data type to a Python Dictionary, the JSON equivalent in the Python language.

Thus presenting a dictionary such as; { '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 Python logic in our backend implemented, let's update the JavaScript function *submitCourseCode* and the nested *fetch*. We will re-implement the method to get a value from the input using the DOM, which will then be sent to the new endpoint 0.0.0.0:5000/course-submit. We will also add options to configure the fetch method to a

'*post*' request, telling the server we are sending data and append the data to the body of the 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://0.0.0.0:5000/course-submit',
          opts
        ).then(
          res => res.json()
          ).then(
            data => console.log(data['msg'])
            )
  }
```

Figure 2.30: Updated index.html. Created opts JSON object, set method param and body param. Added URL path 'course-submit' to point to our new Python function. The first then to interpolate the response as a JSON response, which we can then index to log the returned message to the console.

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, so 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;

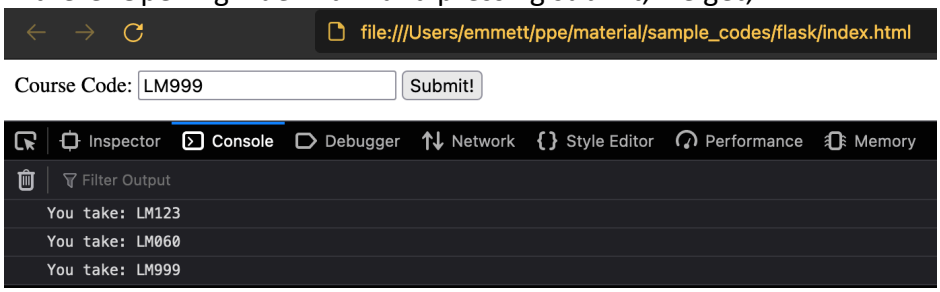


Figure 2.31: The functionality of index.html after adding the new post endpoint. Upon clicking of "Submit!", data is returned.

1. The JavaScript in our "*index.html*" now takes the value inputted at the moment of clicking "Submit!"
2. JavaScript sends the data to the Python file "*endpoints.py*", which interprets the request data, then constructs and returns the msg; "You take …".
3. Upon recieiving the msg from our endpoints.py, the JavaScript in index.html logs the msg to our console.

Already there is a template to use for communication between frontend (index.html); using HTML attributes and JavaScript DOM, event listeners and *fetch* requests, and backend (endpoints.py, main.py); using the Request class, string decoding and data type conversion. One thing this code does not do is display the message to the average user, right now we it is only logged to the console; using *console.log(anything);*

Displaying Data using DOM:

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* object model, *createElement();* (15)
To build the element;

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

Figure 2.32: 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.

```
function displayMsg(msg) {
    text_element = document.createElement('p'); // creates p element
    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
}
```

Figure 2.33: 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.getElement…" functions to localise the document object to a certain element.

We will then call this function from our last *.then* of the *fetch* request.

```
fetch(
    'http://0.0.0.0:5000/course-submit',
    opts
).then(
    res => res.json()
    ).then(
        data => displayMsg(data['msg'])
        )
```

Figure 2.34: Calling displayMsg() in the last .then after response being converted into JSON data type and passing in the message received from the request made to the API.

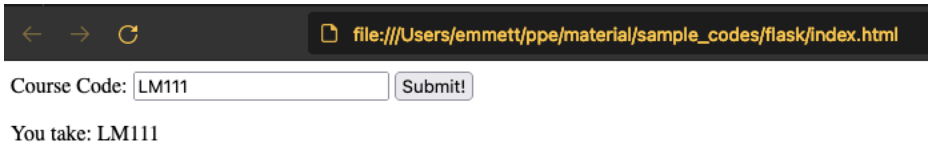


Figure 2.35: Results of the implementation of displayMsg()

Here we have figured out a method 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 the portal.

Using this framework and implementation I am confident we can create a portal to which we will 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 developed here is available at https://github.com/xemmett/flask_app/.

Scope 3: Course keyword extraction.

Here I will be weighing emphasis on the rarity of words like “Engineering” or “Sciences” in course descriptions.

I believe by being able to reduce the text data into categorised 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.

The Method I used was Regular Expression (“Regex”). I wanted to start low-level in order to really get an understanding of the data set and textual language used in Third-Level course documents. Regex is a method of forming templated sentences in order to find similar patterns or expressions within a document.

I stumbled upon this issue when attempting to build a recommendation model from the Qualifax course data. The model output was in-comprehensible, the data was too messy and my experience with Machine Learning was not enough to attempt complicating the model, so I figured I will attempt to understand and clean the data itself and see if the output improves. I will need to learn how to produce an indication as to what words are truly meaningful to the document.

Stop-words:

Stop words are a set of commonly used words in a language. (16) This is a set of words we wish to remove from the text data which carry no meaning and are not very useful for

analysis; words like “the”, “a” and “is”. There are stop lists available online, but I wished to create my own. Creating a curated set of stop words from the input data will be more useful for applying to this corpus and other corpora filled with similar documents.

Removing stop words we can improve a sentences comprehensibility “Engineering Students will need to meet the requirements of a H4 in Mathematics”, removing useless words like “will”, “need”, “to”, “meet”, “the”, “of”, “a”, “in”, we understand the topic is of “Engineering”, “Students”, “requirements”, “H4”, “Mathematics”, so we can search for documents similar to the more informational terms, perhaps returning a list of course documents which also mention the same thing.

To play devil’s advocate, stop words can still be useful; the pre-positional use of the word “in” indicates a link between nouns “H4” and “Mathematics”. This is tiny word would allow us to differentiate between courses that mention “H4 in Mathematics” rather than courses that just mention “H4” & “Mathematics”, as the “H4” might actually originate from a statement of “H4 in Chemistry and a H5 in Mathematics”, this is a simple thing to better refine search results and understanding between human and machine but might be difficult to implement as you would need to template a lot of sentences mention if there is a pre-positional noun then link the words either side.

Position of Speech, word tagging:

Position of Speech also known as *word classes*, is a grammatical method of categorizing words in sentences based on their function or position relative to words around them. (17) As mentioned in stop words, we can produce a better understanding of the context of the document by identifying important nouns.

“Bachelor of Engineering in Computer and Electronic Engineering”, is consisting mostly of nouns linked with prepositions “in”, “of” and the conjunction word “and”. If we were to remove all stop words and process the text document, we would lose statistical importance of the fact that this course description is describing a course interwoven with attributes from an Electronic Engineering course and a Computer Engineering course. So when curating the stop words set we must only reduce the set of certain words based on their function in the sentence.

3. Implementation

In this chapter we will explore the implementation of the webapp and the scrapers. For the implementation we will make some assumptions;

- The machine we are developing on has Python 3.8 or above installed.
- Pip (Package Installer for Python) is installed.
- Correct browser driver is installed and stored in the same directory of every Selenium script.
- We already know the desired flow of each website we are scraping.
- Basic knowledge of the packages outlined in this report so far and native packages to Python.
- We are familiar with the inspection tool common to all browsers.

Steps to complete:

1. Create spiders.
    a. Create and run UL spiders for timetable.ul.ie.
    b. Create and run Qualifax spider for qualifax.ie
2. Create API around UL data.
    a. Create methods of filtering json data.
    b. Open these methods up for access from the internet using FastAPI.

Creating Spiders:

Here we will extract data we need. First let's talk about the spider implemented in Scrapy; timetable.ul.ie and then the Selenium spider built for Qualifax. At one point I shifted my focus for collection on timetables, causing the timetable.ul.ie spider to need assistance from a separate Selenium spider, this was due to Privacy concerns from Cybersecurity in UL; we will discuss this in detail now.

Scrapy Getting Started:

Pre-requisites:

1. Install scrapy

    ```
    >>> python3 -m pip install scrapy==2.5.0
    ```

2. Create a template scrapy project.
    Using scrapy startproject <filename> we can create a boiler-plate project directory for working in scrapy. We then go into that directory and generate a spider by name and target URL.

    ```
    >>> python3 -m scrapy startproject UL_spiders
    >>> cd UL_spiders
    >> python3 -m scrapy genspider timetable timetable.ul.ie
    ```

We should now have a project file which looks like:

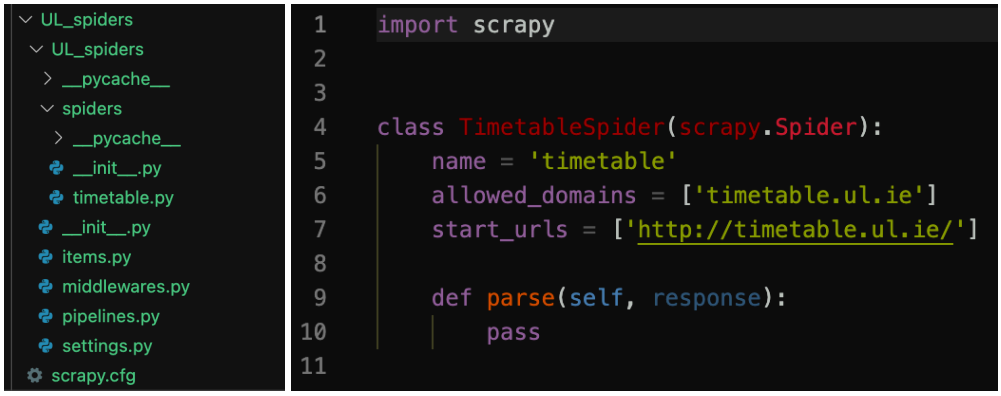


Figure 3.1: Left; Directory structure. All files generated by Scrapy. Right; Contents of timetable.py, our actual spider code, templated by Scrapy.

First we see a variable called *name*, this is a useful and necessary variable to call upon specific spiders to run. Secondly we see *allowed_domains* which basically tells our spider which domain to stay on the spider will not go on any webpages not under one of the allowed domains. The *start_urls* are urls we wish to scrape under the allowed domains. We can run this script from a terminal in the project directory using "scrapy crawl" and its *name*;

>>> scrapy crawl timetable

Here we expect to see our spiders log printed to the terminal as it starts and finishes as in figure 3.2. Let's examine these logs to differentiate what's useful and what is too much information.

Figure 3.2: Output from running Scrapy spider. Outputs from settings.py and middlewares.py are irrelevant unless we change anything in these files. Otherwise we will just look at the Scraper process, it's requests and their status'.

The more pages we begin to crawl the more request logs (status code, URL, method from Figure 3.2) get outputted to the terminal. We are given insights into how many pages were crawled, how many items (data structures, timetables in our case) were extracted, seen in the beginning of ***Scraper process*** "*INFO: Crawled 0 pages (at 0 pages/min), scraped 0 items*

(at 0 items/min)". This can assist us in debugging and regulation of useful requests made to the server, to which we can work on preserving bandwidth and minimalizing traffic.

Since the method is redundant to the current implementation, I will skip how I got them (using Selenium) and start on how to extract information from timetables now; to keep the topic on Scrapy.

Opening timetable HTML files in Scrapy:

In a nutshell, at this point I had used Selenium to save the HTML of each course-year timetable combination from https://timetable.ul.ie/UA/CourseTimetable.aspx, this will be talked about in the Selenium section. I will now explain how I parsed each file using Scrapy, I explain this first as the parser was written before Selenium was ever involved. This was due to the entire process first being for private student timetables (full Scrapy solution) but then eventually being adapted for public course timetables (Selenium and Scrapy solution).

We can view the difference between HTML retrieved using Scrapy and HTML retrieved using Selenium below in Figure 3.3. The Spider originally being developed for HTML files of the format on the **Left** and then having the xpaths changed to the HTML files of the same format as the **Right** from Figure 3.3.

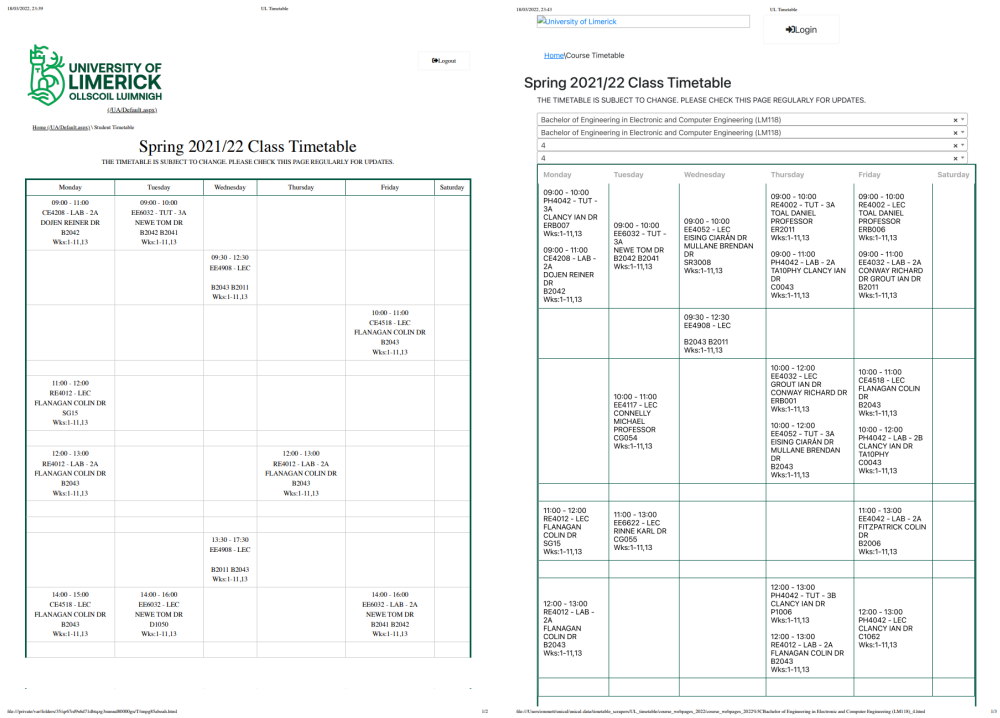


Figure 3.3: (Left) Private Personal Timetable HTML as retrieved by Scrapy. (Right) Public Course HTML when saved from a chrome browser using Selenium.

To extract any information from either of these tables, we need to open a HTML file to get a response within Scrapy. We've seen Scrapy's Spider function, *parse()*, will iterate over any URLs defined in *start_urls*. Instead of URLs, we will fill this function with the file paths of all HTML files saved to a folder in our projects directory, using the native python package; OS.

```
start_urls = []
for filename in os.listdir(directory):
    if filename.endswith(".html"):
        filepath = directory + filename
        start_urls.append('file://' + filepath)
```

Figure 3.4: Appending URL schemed filepath to start_urls. First we check if the file name ends with ".html", then we append it's entire filepath to "start_urls"; the list of URLs which will be scraped by the spider.

Here in figure 3.4, *directory* would be the folder in which we keep all the HTML files we wish to parse. From there we can use the os package to iterate over all the files in the *directory* if the filename ends in a dot html extension, we will append it, this just ensures we are only passing the accept HTML files into scrapy and not some random file. The resulting *start_url* element would be (on MacOS);

file:///directory/filename

or

file:///Users/emmett/fyp/timetable_scrapers/UL_timetable/course_webpages_2022/Bachelor%20of%20Engineering%20in%20Electronic%20and%20Computer%20Engineering%20(LM118)_4.html

Parsing data from HTML documents in Scrapy:

This function will need to read each cell/event, and extract all information needed to reconstruct it again for our calendar details. For each cell, we will create a data structure which contains;

- *day*: integer form (0=Monday, 5=Saturday)
- *title*: Module code – Delivery – Group?, str
    - *module:* Module code, string
    - *delivery:* Type of event, Lab, Lec, or Tut, string
    - *group:* Group if present, string
- *time:* start and finish, string
- *professor:* Professor listed, string
- *location:* Event location
- *active_weeks:* List of weeks which event occurs.

This function will need to use the HTML from our course timetable to extract the above information. First lets examine the structure of the timetable to plan our implementation. In figure 3.5 below, we see rows going down the way, and each row contains cells, and each cell contains a single or multiple events. We need our scraper to extract each event from each cell preserving it's day occurring and isolating it's data from any events it may be sharing the cell with.

We need to expect and handle cases where cells may have no data, a singular event, or multiple events. In a perfect case a cell will contain all the data as listed above^, but some edge cases will arise where events are missing data such as professor (e.g. figure 3.5, Wednesday 9:30am) or location. We will touch on inconsistencies and annoyances throughout the implementation.

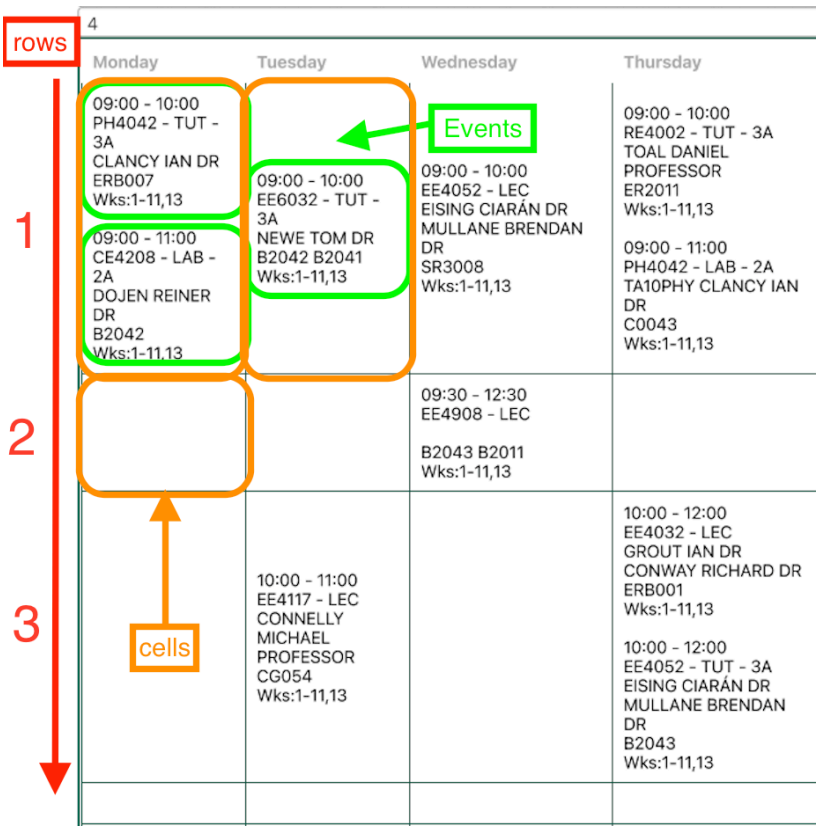


Figure 3.5: Layout of timetable. Rows going vertically, events within cells.

We can derive some pseudocode to out-line our implementation on the above timetable layout.

```
rows = scrapy select (rows_xpath_pattern)
days = [0, 1, 2, 3, 4, 5]
day_counter = 0

for each row of table:
        day = days[day_counter]
        for each cell of row:
                contents = scrapy select(cell text)
                …. *parse data
                day_counter = day_counter+1
```

*Figure 3.6: Pseudocode for getting text contents of each cell where *parse data has yet to be developed. We set the possible days and involve a day counter to keep track of which day we are on. We will increment the day for each cell in the row.*

So first we will find and select the table using it's xpath. We constructed our scraper to read the table down and across, extracting all cells per row. Each cell on each row corresponding to a specific day. We will simply take all the extracted text from a cell and pass it to be parsed.

First we can assign the xpath of our rows, this is quite simple, extracting 2 or 3 of the rows xapths reveals a similar pattern which we can use:

```
Row 1: /html/body/form/div[5]/div/div/table/tbody/tr[1]
Row 2: /html/body/form/div[5]/div/div/table/tbody/tr[2]
Row 3: /html/body/form/div[5]/div/div/table/tbody/tr[3]
```

Figure 3.7: Respective xpaths of the rows from Figure 3.5.

Each row as a similar xpath besides the numeration at the end, we can use a wild card expression to accept all rows of the table by replacing the table row or 'tr' tag with '*'. This will get every next available element leading from the 'tbody' tag, even non 'tr' rows which we would need to omit if any existed. The only row we will omit is the top row which outlines the days.

```
All rows: /html/body/form/div[5]/div/div/table/tbody/*
```

Figure 3.8: Common xpath expression for table rows.

```
rows_xpath = '/html/body/form/div[5]/div/div/table/tbody/*'
rows = response.xpath(rows_xpath)[1:] # Omit first row (column headers)
```

Figure 3.9: Getting all rows on timetable.

Now we can get the first cell to begin creating the format for an event. As mentioned earlier, we see that cell can contain 2 events, 1 event or even no events. Lets get and print the contents from some cells of the timetable. We will simply call the *'td'* (table data) tag xpath to get each cell element and then extract the *text* attribute of the cell and print it.

```
rows_xpath = '/html/body/form/div[5]/div/div/table/tbody/*'
rows = response.xpath(rows_xpath)[1:] # Omit first row (column headers)
first_row = rows[0]
first_cell = first_row.xpath('td/text()').getall() # get all text attribute values.
print(first_cell)
```

Figure 3.10: Printing a cells content or text.

Lets look at the output from some unique cells.

Cells with one event:

```
['09:00 - 10:00', 'CE4041 - TUT - 3B', ' FLANAGAN COLIN DR', ' A1051', 'Wks:1-12']
```

Cells with two events:

```
['09:00 - 10:00', 'EP4407 - TUT - 3A', ' TA3MMA  ', ' HSG022', 'Wks:3-12', ' ',
'09:00 - 11:00', 'EE6621 - LEC', ' RINNE KARL DR', ' B2011', 'Wks:1-12']
```

Cells with no event:

```
['\xa0']
```

Taking the first 5 elements of any list with data, we see *time, title (module code, delivery, group), lecturer, location* and *active weeks*. To be able to avoid incorrect storing of data from cells with no events and cells with two events, first we will check if the list of contents is longer than one, second we will use a compatible *time* object as the delimiter between two events in a single list.

```
def time_check(input):
    """
    purpose: to check if input is a time interval
          by attempting to convert the start and
          end time strings into time objects.
    """
    try:
        times = input.split('-') # start-end -> [start, end]
        start_time = times[0].strip()
        end_time = times[1].strip()
        # convert strings to time objects
        strptime(start_time, '%H:%M')
        strptime(end_time, '%H:%M')
        return True # input is time interval
    except ValueError: # input does not fit into time object
        return False
    except IndexError: # input does not have '-' in it.
        return False
```

Figure 3.11: time_check function. Simply converts compatible strings into time objects, if conversion is successful, then input is a time object meaning we are onto a new event.

Now that we have a means of delimiting, identifying and verifying events, lets map them onto the structure in Figure 3.12.

```
new_class = {
    'day': 0,
    'professor': 'Null',
    'module': 'Null',
    'group': 'Null',
    'delivery': 'Null',
    'location': 'Null',
    'active_weeks': [],
    'start_time': '00:00',
    'end_time': '00:00'
}
```

Figure 3.12: Templated desired format and structure.

With input; ['09:00 - 10:00', 'EP4407 - TUT - 3A', ' TA3MMA ', ' HSG022', 'Wks:3-12'] from the first cell of the table, we would want an output of;

```
new_class = {
   'day': 1,
   'professor': 'TA3MMA',
   'module': 'EP4407',
   'group': '3A',
   'delivery': 'Tutorial',
   'location': 'HSG022',
   'active_weeks': ['3-12'],
   'start_time': '09:00',
   'end_time': '10:00'
}
```

Figure 3.13: Resulting desired format and structure from input data.

Split is a function used to divide a string into a list using a passed delimiter. Some of the required data is hidden away in larger strings delimited by hyphens, we can simply *split()* the string at each hyphen in the title to extract the module code, delivery and group. Since not every event has a group, we will enclose it's assignment into a *try and except* meaning if there is an IndexError (meaning there is no index 2 in a list of [0, 1]), ignore it and just assign null.

```
# where class_data = [time, title, weeks_active]

# '9:00 - 10:00'
times = class_data[0].split('-')
start_time = times[0].strip()
end_time = times[1].strip()

# 'EP4407 - TUT - 3A'
title = class_data[1].split('-')
module = title[0].strip()
delivery = title[1].strip()
try:
   group = title[2].strip()
except IndexError as e:
   group = 'Null'

# '3-12'
active_weeks = class_data[-1].replace('Wks:', '').strip()
active_weeks = active_weeks.split(',')
```

Figure 3.14: Mapping of inputs time and title data to variables which will be assigned to our new data structure.

We are also using *strip()* on every string value, which ensures removal of whitespace at either side of the string common when scraping;

' TA3MMA '.strip() = 'TA3MMA'

As mentioned earlier; some timetabled events on other courses are incomplete, either missing professor or location or both, we also do not get any indication which is missing. If the length of the data is less than what we are expecting (5), we have no idea what the missing data element is, from inspecting I can see it's always either professor or location, so

I assign the unknown element to both keys, to help ensure trust in the information supplied to students by seeing familiar values rather than discarding the unknown data element.

We can of course handle this, by checking the length of the input event or class data, and if it's less than expected assign the last element from the input data to both professor and location, if there is no left over element (after time and title assigned), we just assign *professor* to unknown and *location* to nothing.

```
try:
    if(len(class_data) < 5):
        # data missing
        unknown_data = class_data[-2].strip() # second last element
        location = unknown_data
        professor = unknown_data
    else:
        # no data missing
        location = class_data[-2].strip()
        professor = class_data[2].strip()
except IndexError as e:
    professor = 'Unknown'
    location = ''
    pass
```

Figure 3.15: Mapping of professor and location.

From here we can just assign each value to the data structure defined in figure 3.12.

```
new_class['day'] = day
new_class['start_time'] = start_time
new_class['end_time'] = end_time
new_class['module'] = module
new_class['delivery'] = delivery
new_class['group'] = group
new_class['active_weeks'] = active_weeks
new_class['professor'] = professor
new_class['location'] = location

return new  class
```

Figure 3.16: Assigning values to data structure and returning

If we print the variable *new_class*, we can view our new structure;

```
{
 "day": 0,
 "professor": "TA3MMA",
 "module": "EP4407",
 "group": "3A",
 "delivery": "TUT",
 "location": "HSG022",
 "active_weeks": ["3-12"],
 "start_time": "09:00",
 "end_time": "10:00"
}
```


Figure 3.17: Event structure

This event will then be appended to a larger list of all events on the timetable, which belongs to a larger data structure called *timetable*, which has 4 values; course name, course code, course year and class; which is the field containing the list of events.

Before parsing our rows, we will define *timetable* as a variable in our *parse* function, and assign it the values of course code, year and name from the dropdown elements present on the webpage.

```
timetable = {}
course_name_code_el = response.xpath(course_name_code_xpath)
course_name_code = course_name_code_el.get()
# 'Bachelor of Engineering in Electronic and Computer Engineering (LM118)'
course_name_code_list = course_name_code.split(" ")
# ['Bachelor', 'of', 'Engineering', 'in' ...., '(LM118)']
course_code_w_brackets = course_name_code_list[-1]
# '(LM118)'
timetable['course_name'] = course_name_code.replace(course_code_w_brackets, '').strip()
# 'Bachelor of Engineering in Electronic and Computer Engineering'
timetable['course_code'] = course_code_w_brackets.replace('(', '').replace(')', '')
# 'LM118'
timetable['course_year'] = response.xpath(course_year_xpath).get()
# '4'
timetable['class'] = []
```

Figure 3.18: Assigning timetable object values

Now we will extract each cell and append the processed data to the list stored in the *class* field of *timetable*, thus building on the pseudo-code from earlier;

```
...in spider class
timetable = {}
assign course name, code, year to timetable

rows = scrapy select (rows_xpath)
days = [0, 1, 2, 3, 4, 5]
day_counter = 0
1. for each row of table:
        day = days[day_counter]

        2. for each cell of row:
                contents = scrapy select(cell text)
                contents = ['time', 'code-delivery-group', 'professor', 'location', 'active weeks']

                i = 1
                3. for each element in contents:
                        if element contains a time e.g 09:00-10:00: # time_check()
                                subdata = [this element]

                                4. for each element remaining:
                                        if element is a time:
                                                stop
                                        else:
                                                append to subdata

                                new_event = parsed subdata, day
                                append new_event to timetable

                day_counter = day_counter+1
```

Figure 3.19: Pseudocode for assigning timetable event values.

Here we have four *for* loops, each tackling a different level of our data, unwrapping it to the data we wish to parse and save. The first loop opens a row from the table, in the second we loop over each cell of the row and extract its contents. From here we want to iterate over the contents of the cell, we append subdata until another time object is hit or the iteration ends, and send this subdata for parsing.

In the case another time object is present (Figure 3.19: "*if element is a time"*) we stop or *break* the 4th *for*-loop. From here the 3rd *for* loop will continue until it reaches the very same element that broke the inner-loop, then the subdata parsing will begin for this new event. Nesting the loops seemed fastest at the time, and it's easy to keep track the data's condition at each level.

- 1st level is the whole row.
- 2nd level is an individual cell.
- 3rd level is a detection layer using the function time_check from figure 3.11; if a time object is present then go to 4th level.
- 4th level is a segment of the cell contents representing a whole event.

With our *parse()* function defined, we will create a function called *closed()*, it will be recognised by Scrapy and run upon the spider signalling that it is finishing its process, we will use this to save the data once the spider has extracted all timetables.

```
def closed(self, response):
    filename = r'ul_course_timetables.json'
    with open(filename, 'w+') as db:
        json.dump(self.UL_timetables, db)
```

Figure 3.20: Dumping json data in ULSpider.closed function. Function will run upon the Spider process completing and closes. Here we can perform a final shutdown function to save or "dump" the JSON data into ul_course_timetables.json

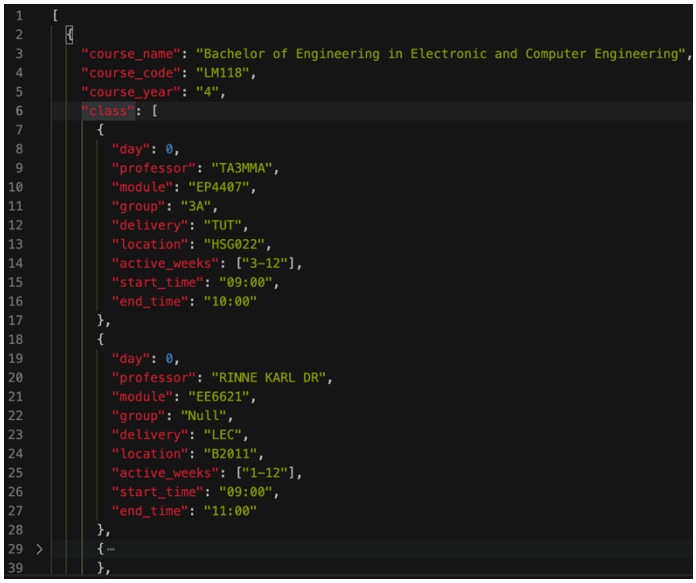


Figure 3.21: Contents of ul_course_timetables.json after the spider has completed and closed it's process. The data is now in a useable form, compatible with many available software and specifically for data transmission over the internet.

Here we see the method of saving our data and the file containing it in its final form; a JSON. We see Scrapy is relatively easy to use once you find a decent debugging method. I wrote this script multiple times in different ways and was one of the first steps to starting this project. The entire script takes 8 seconds to iterate over 458 timetables.

The decision to pursue familiarity as mentioned earlier when assigning the professor and location missing data; I wanted to collect and present as much information as I can. This stems from the anxiety that this project will be a third-party un-reputable source that supplies students with un-familiar files. I want to be able to present as much information to the students as possible without asking for them to specify too much about themselves to avoid appearing as if the site is phishing and improve transparency about the sites intents or purpose.

Third party applications built by students could appear a lot less suspicious if coming from a familiar source such as a university extension (i.e. student-app.ul.ie) also minimising the steps for students to acquire already public data such as course timetables could reduce the chance for error or data loss. We can make up for this by displaying more data than asked of by the student, showing we truly have the data to support our stated intents. For this we collect as much recognisable and familiar data as possible.

Create and Run UL Spiders (Selenium):

To extract the courses HTML and save it into a file to be parsed by the scrapy spider, we need to utilise Selenium in order to operate the course selection fields of name and year, then save the resulting HTML into a file, preserving it's structure for off-line parsing in Scrapy.

Selenium is a web browser automation tool, which is often used to automate a user flow to test your own website during development. Selenium can also be used in the exact same way but to collect data during the process. By automating websites we can passively extract data in a couple of hours that would usually take a human weeks to manually extract.

Due to Selenium running in the web browser it can utilise the DOM to interact with the webpage, as well extracting and manipulating data. Selenium's original purpose in this project was to only collect data from Qualifax, but ended up having to extract all the courses on https://timetable.ul.ie/UA/CourseTimetable.aspx too due to Scrapy being unable to iterate the directory without the DOM.

First objective is to collect all the HTML (source code which builds up structure and look of the website) for each course from timetable.ul.ie. We will save each courses HTML to a file in a directory for offline parsing in Scrapy.

Pre-requisites:

1. Install selenium

   >>> python3 -m pip install selenium==4.1.0

2. Install the relevant driver for the browser of your choice.
   a. For chrome visit https://chromedriver.chromium.org/.
   b. Place driver in same directory as project.
3. Create a .py file and import the webdriver class from selenium

```
from selenium import webdriver
from time import sleep
```

Figure 3.22: Import the webdriver library from Selenium.

4. Depending on your browser, start the relevant class of webdriver.

```
driver = webdriver.Chrome()
```

Figure 3.23: Initialise the browser class to the driver. This will immediately open the corresponding web browser window.

Saving the websites HTML from timetable.ul.ie in Selenium:

Once we have our browser opened from the pre-requisites, we will use Selenium to navigate to timetable.ul.ie, remove any pop ups and click the tile to go to the tool to select course timetables.

```
driver.get("https://timetable.ul.ie/UA/")

sleep(5)
cookies_accept = '//*[@id="CybotCookiebotDialogBodyLevelButtonLevelOptinDeclineAll"]'
driver.find_element_by_xpath(cookies_accept).click()

course_timetable = '//*[@id="ctl01"]/div[5]/div/div/div[3]/a'
driver.find_element_by_xpath(course_timetable).click()
```

Figure 2.36: Script to open a webpage; timetable.ul.ie and dismissing the cookies banner and clicking the course timetable tile.

We need to remove the pop-up otherwise Selenium will give us an error that when the driver attempts to click on the course timetable tile, another element will receive the click and that element blocking us is the cookies pop-up.

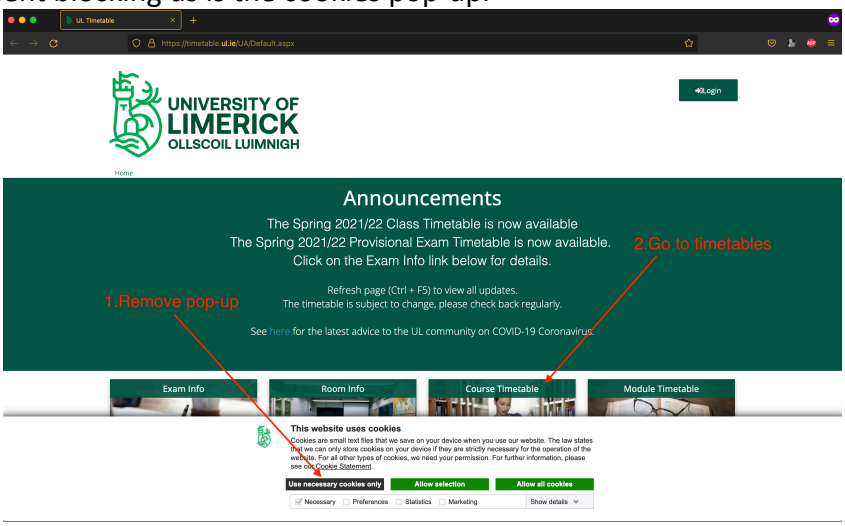


Figure 2.37: Selenium flow using our browser as a Graphical User Interface.

The script is now looking at a the webpage which has the course selection and the year selection, we can use pseudocode to outline our approach;

```
Click course selection to reveal options

for each course option:
        click course option
        course_title = course option text (name, code)

        click year selection to reveal options
        for each year option:
                click year option
                course_year = year option text

                file name = course_title + '_' + course_year + '.html'

                with file name open as file:
                        write HTML to file
```

Figure 3.24: Pseudocode code to iterate over timetable.ul.ie/UA/CourseTimetable.aspx.

Our first step is to click the course drop-down, from here we need to iterate this list clicking each course, upon clicking; the selections close and a year field is presented.

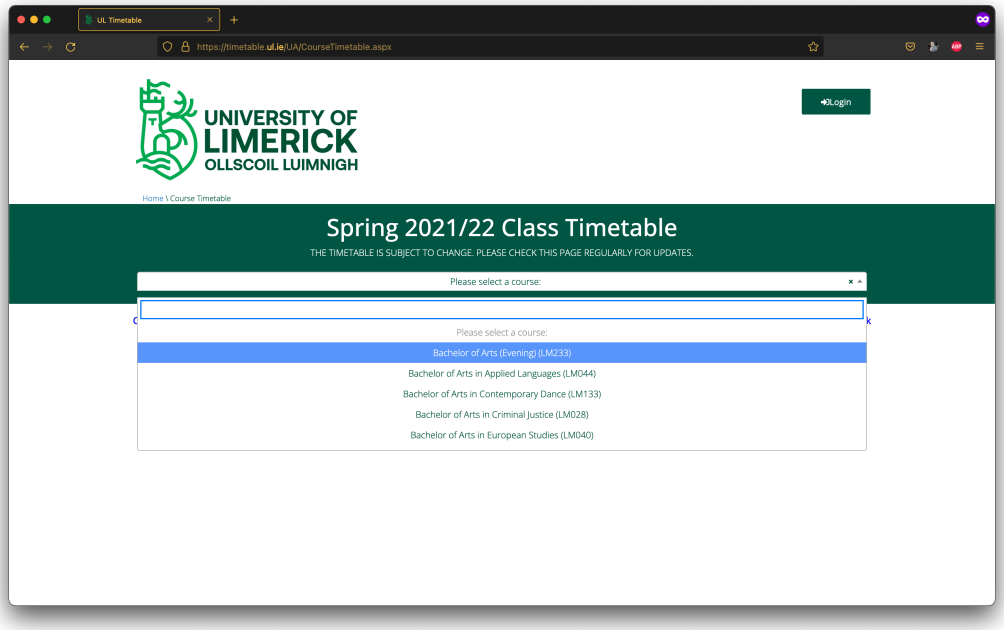


Figure 3.25: Clicking course selection to reveal course options.

Here a course has been chosen and the course drop-down closed, meaning the scraper will need to click this course selection drop-down again after iterating all the years for the current course. Thus we open the year drop-down and iterate.

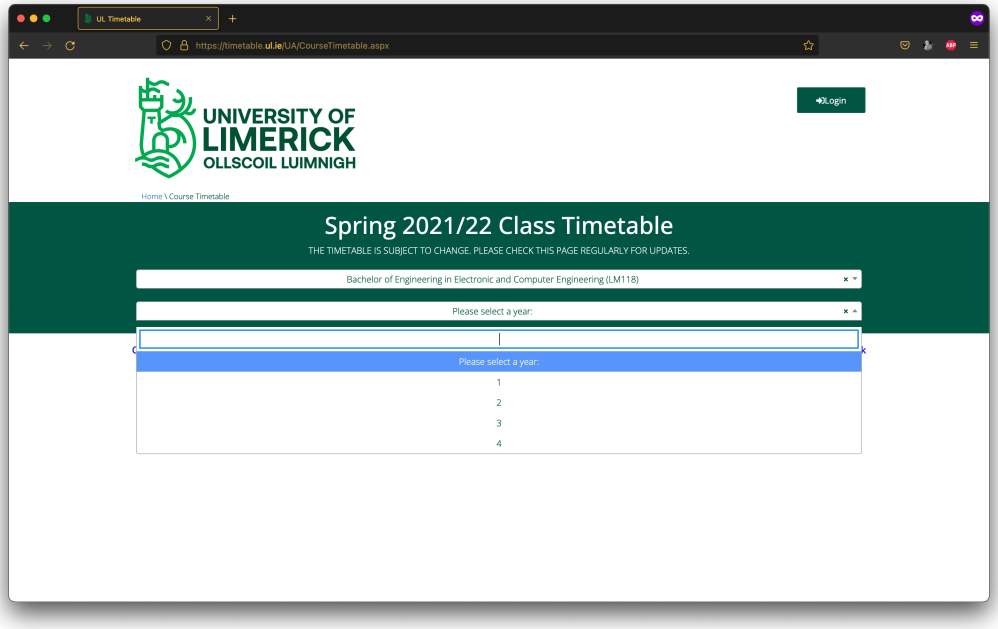


Figure 3.26: Clicking year option to reveal year selections.

Upon clicking the year option the drop-down closes and our timetable data is presented.

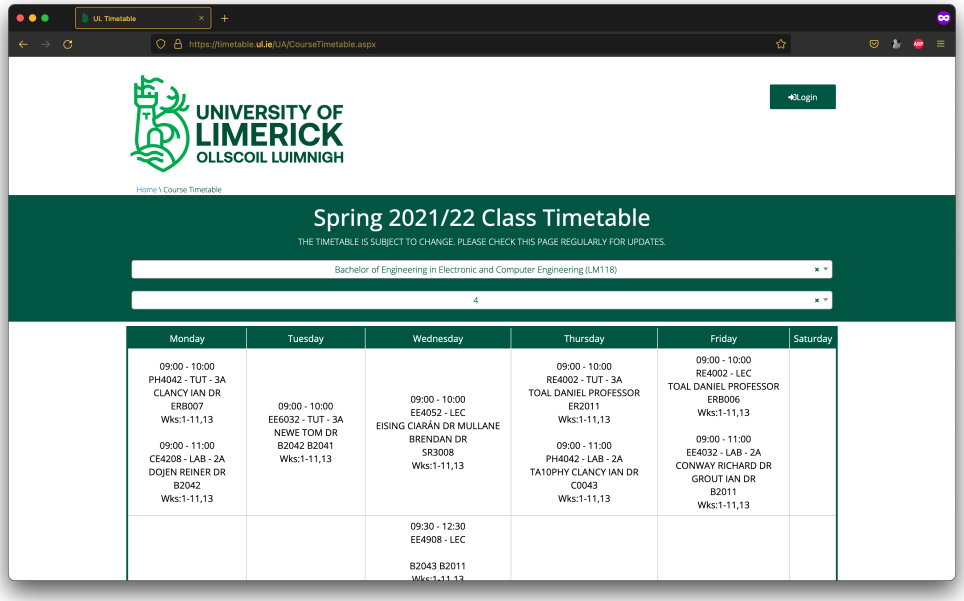


Figure 3.27: HTML we wish to preserve and parse within scrapy.

Once our timetable data is presented we need save it to a HTML file.

First lets define our xpaths:

```
course_selection_xpath = '//*[@id="select2-HeaderContent_CourseDropdown-container"]'
year_selection_xpath = '/html/body/form/div[4]/div/span[2]/span[1]/span/span[1]'
year_xpath = '/html/body/span/span/span[2]/ul/li[{}]'
course_xpath = '/html/body/span/span/span[2]/ul/li[{}]'
```

Figure 3.28: Xpaths for individual HTML elements that appear on the timetable selection tool.

We will be formatting the year and course xpaths with the index of which year or course we are on. Starting the index range from 1 to 500 (element 0 is the placeholder text 'Please select a course.'), we can run the spider until it fails to click a course selection above the indexable range from the course drop down selector using an Index Error, meaning we have gone outside the range of indexable courses present in the options drop-down.

```
for i in range(2, 500):
    driver.find_element_by_xpath(course_selection_xpath).click()
    sleep(2)
    try:
        course = options_xpath.format(i)
        course_name = driver.find_element_by_xpath(course).text
    except IndexError as e:
        break # spider stops
```

Figure 3.29: Looping over courses, with an exception for an out of range Index, upon having no courses left to iterate over.

If the spider successfully clicks a course then our loop is not broken, the webpage will load the year selection. We iterate over the year selection using a range from 1-8, when we know

our data better we can adjust indexes, if the course fails to select a year, the loop is broken and we return to the course loop above.

```
for year in range(2, 8):
    driver.find_element_by_xpath(year_selection_xpath).click()
    try:
        course_year_sel = options_xpath.format(year)
        course_year = driver.find_element_by_xpath(course_year_sel).text
        driver.find_element_by_xpath(course_year_sel).click()
        filename = course_name +'_' + course_year + '.html'
        html = driver.page_source
        save_file(filename, html)
    except:
        break
```

Figure 3.30: Looping over years. Extraction of the course year text, construction of the filename then final extraction of the page's source HTML, and saving it using our save_file function.

Here we gather relevant information to name the file and fill it with the HTML. The HTML is gotten by calling the drivers *page_source* attribute. This is the same as if we were to right click on the page in the browser and select "*View Page Source*".

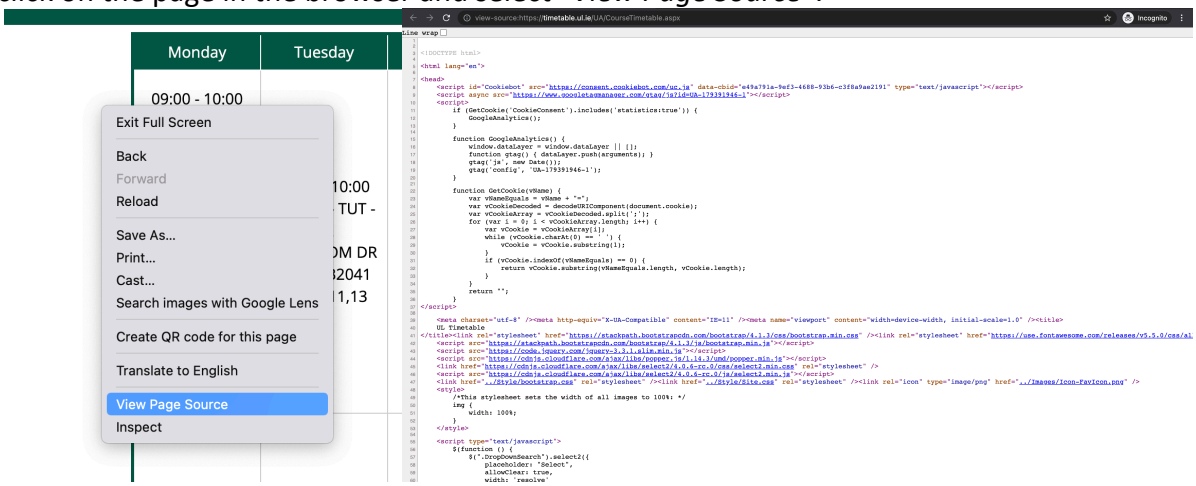


Figure 3.31: Viewing page source through browser. Code on the right will be the contents of the HTML file.

I could have begun parsing the timetable data here, creating JSON data and saving it to memory just as easy (definitely easier) as my current implementation. This comes back to my interaction with Cyber Security of UL. The Scrapy spider was already developed for parsing of my private personal timetable, so in the interest of time I decided to supply the Scrapy Spider with the HTML from the Selenium Spider, saving from needing to port the code from one framework to another.

Instead here we pass our HTML and filename into a save_file function. This function is similar to the *close()* function in Scrapy; we simply create and open a file of the passed *filename* and write the contents of our *page_source* into the file.

```
def save_file(fp, html):
    with open(fp, 'w+') as of:
        of.write(html)
```

Figure 3.32: Saving HTML to a file. A function with parameters for filepath and the html string.

From here, all we need to do is allow the process to run, and then point our Scrapy spider to extract all HTML files the directory of the Selenium spider saved them in.

We can see Selenium's usefulness in automating tasks on certain sites. It requires less set-up and files than Scrapy but it also a lot slower due to DOM enabled pages usually being larger in size, compared to the plain HTML document response Scrapy would get.

This was quite a low-level application of Selenium, but already it shows a lot more promise in future applications of web scraping. An interesting side note emphasising Selenium's importance is on sites like Facebook which tend to randomize their HTML patterns each day, meaning xpaths are not valid for very long. This will soon make xpaths, and HTML parsers like Scrapy and BeautifulSoup obsolete. An interesting workaround to this with Selenium is that we can take screenshots of the page and run these screenshots through Optical Character Recognition to extract data.

Create and Run Qualifax spider for qualifax.ie:

Qualifax is a site which visitors can search for courses using keywords or filters. Without inputting any keywords or filters and pressing search, we are presented with the entire directory of courses. Although the course directory tool is actually located in an iframe.

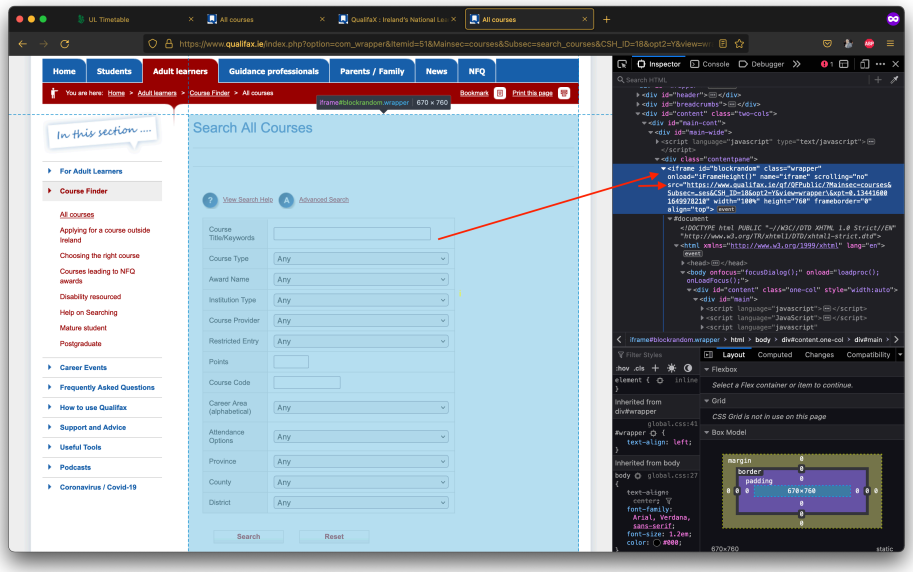


Figure 3.33: Qualifax course directory tool hosted in an iframe, with the actual source URL of the tool pointed out.

An iframe is a method of hosting another webpage within a frame like window or portal on the main site. The issue is that the DOM (Document Object Model) presented to us by Selenium on this page isn't the DOM which controls the directory tool, the DOM we wish to manipulate.

If we visit the URL of the source in the HTML element from figure 3.33 in Selenium, we can get the DOM for this respective page, therefor having manipulability over the directory tool's DOM.

We utilised Selenium to navigate the course directory browser located at; https://www.qualifax.ie/qf/QFPublic/?Mainsec=courses&Subsec=search_courses&CRAsort=&action=search&display=&CRT_ID=0%2C0&CSH_ID=18&PREV_CSH_ID=&AdvancedKeyword=&keywords_and_titles=title&all_or_any_words=all&full_or_part_words=full&FCT_ID=&FDM_ID=&keywords=&CRT_ID=0&QUA_ID=0&CTP_ID=0&COL_ID=0&RES_ID=0&points=&CRS_CODE=&CRA_ID=0&ATT_ID=0&PRV_ID=0&COU_ID=0&DST_ID=0.

Showing **1** to **50** of **14832** results

1 2 3 4 5 6 7 8 9 10 Next

New Search Refine Search

Compare	Tag All	Code	Course	Course Provider	NFQ Level	NFQ Classification
☐	Tag	TU282	Architecture - Bolton Street	TU Dublin - City Campus and TU Dublin - Technological University Dublin	Level 9 NFQ	Major
☐	Tag	TU231 TU232	Culinary Innovation & Food Product Development - Grangegorman	TU Dublin - City Campus and TU Dublin - Technological University Dublin	Level 9 NFQ	Major
☐	Tag	106332	Spirituality Awareness	Northern Ireland Hospice		
☐	Tag	111585	1921-1922 - Ireland: War and Peace	University College Dublin		
☐	Tag	101858	21st Century Teaching & Learning	Trinity College Dublin		
☐	Tag	97195	2D & 3D Digital Animation Production	Dundalk Institute of Technology	Level 7 NFQ	Minor
☐	Tag	CR_ECADM_6	3D CAD & Solid Modelling - Cork Campus	MTU - Cork Campuses and MTU - Munster Technological University	Level 6 NFQ	Special Purpose
☐	Tag	100229	3D Computer Aided Design Using Solidworks	Dundalk Institute of Technology		
☐	Tag	TU382 TU384	3D Design - Grangegorman	TU Dublin - City Campus	Level 9 NFQ	Major

Figure 3.34: Qualifax course directory interface. The DOM which we wish to manipulate.

Here the code will need block; clicking next, iterating over the 297 (1-50 courses per page, 14832 courses, 50/14832 ~= 297) pages extracting the links to every course, then the spider will visit each course link and extract all the information of each course page.

Qualifax lands us on the search tool in figure 3.35, from here we just need to click search to return all courses available on Qualifax. We will use our driver to request this search tool, and click search.

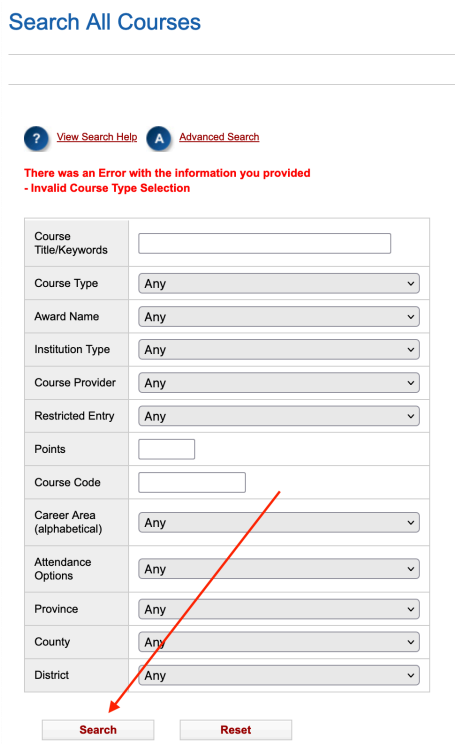


Figure 3.35: Landing page of the Qualifax course directory tool.

```
def get_results(driver):
    driver.get(search_tool_url)
    search_all_tag =
'//*[@id="main"]/table/tbody/tr/td[3]/table/tbody/tr[6]/td/button[1]'
    driver.find_element_by_xpath(search_all_tag).click()
```

Figure 3.36: Relevant code to direct the driver to the search tools url, then clicking the search button from Figure 3.35.

This lands us on the directory in Figure 2.37, in more detail below we show the links of interest, each course name contains a href attribute we will want the scraper to visit to extract it's full information.

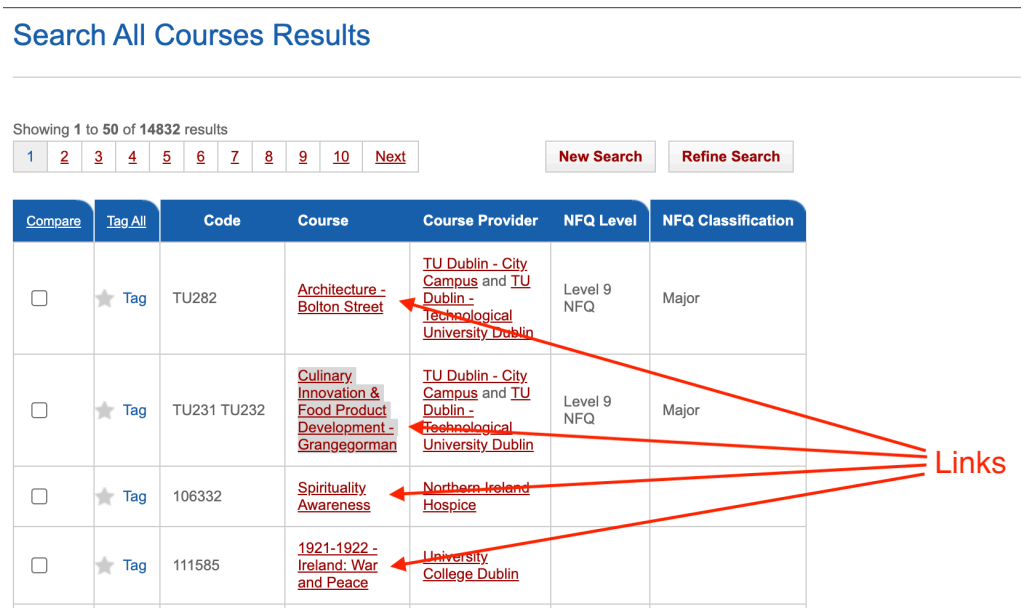


Figure 2.31: Qualifax course directory.

From here we extract all course links and save them to a JSON file. Let's write some pseudo code to extract our links, save them to a JSON object and click to the next page.

```
Links = []
for all pages in search results:
        get table element
        for row in table:
                get link
                append link to links
        click next page
```

Figure 3.37: Qualifax flow pseudocode. Creating a master list of all courses per page, to iterate and scrape later.

We can translate this into code as:

```
course_ links = [ ]
for i in range(2, 299):
    table_rows =  driver.find_elements_by_xpath(rows_xpath)
    for row in table_rows:
      cells = row.find_elements_by_xpath("td")
      try:
        course_cell = cells[3].find_element_by_xpath("a")
        course_link = course_cell.get_attribute("href").strip()
        course_links.append(course_link)
      except: # reached end of list.
        break # stop
```

Figure 3.38: Qualifax code implementation.

We iterate until there are no more rows with course links. The iteration range is 2-299, we will use the variable *i* to build a format the pagination query onto the URL, as we know there are 297 pages on Qualifax, we go a page above to break the query and ensure we hit all the pages.

The variable *rows_xpath* comes from the same method we used to find common rows on the UL timetable, by extracting two or three row xpaths, we can find a common xpath pattern and insert our wildcard.

```
Row 1: /html/body/div/div/form/table/tbody/tr/td/table/tbody/tr[4]/td/table/tbody/tr[2]
Row 2: /html/body/div/div/form/table/tbody/tr/td/table/tbody/tr[4]/td/table/tbody/tr[3]
```

Figure 3.39: Individual xpaths of each row of the table.a

```
/html/body/div/div/form/table/tbody/tr/td/table/tbody/tr[4]/td/table/tbody/*
```

*Figure 3.40: Common xpath for rows, using * to denote anything at the end of the tbody tag, either a th (table header, column headers) or a tr (table row), where th will be met by our exception.*

We then iterate each row, taking the 3rd index of the list of cells we extract from the row. This cell contains our link, which we get by getting the link element by it's '*a*' tag, then extracting the *href* attribute we get the link affiliated with this tag.

Thus with all links extracted for this page we need to navigate to the next page and repeat. Instead of repeatedly pressing next, inspecting the search URL, we can see where the page number is specified by navigating to different pages and seeing what changes.

```
Page 2: https://www.qualifax.ie/qf/QFPublic/?.....&Idx=2&searchaction=current
Page 3: https://www.qualifax.ie/qf/QFPublic/?.....&Idx=3&searchaction=current
Page 4: https://www.qualifax.ie/qf/QFPublic/?.....&Idx=4&searchaction=current
```

Figure 3.41: Qualifax pagination query idx. By setting this value the scraper can visit any page number available in the directory.

Using the URL query variable 'Idx', we can remove the number and format it using the *i* variable from the *for* loop which is incrementing from 1-297. We will add this to the bottom of our *for* loop to execute at the end of the course link extraction, and direct the spider to the formatted link.

```
query = base_link+pagination_query.format(i)
driver.get(query)
```

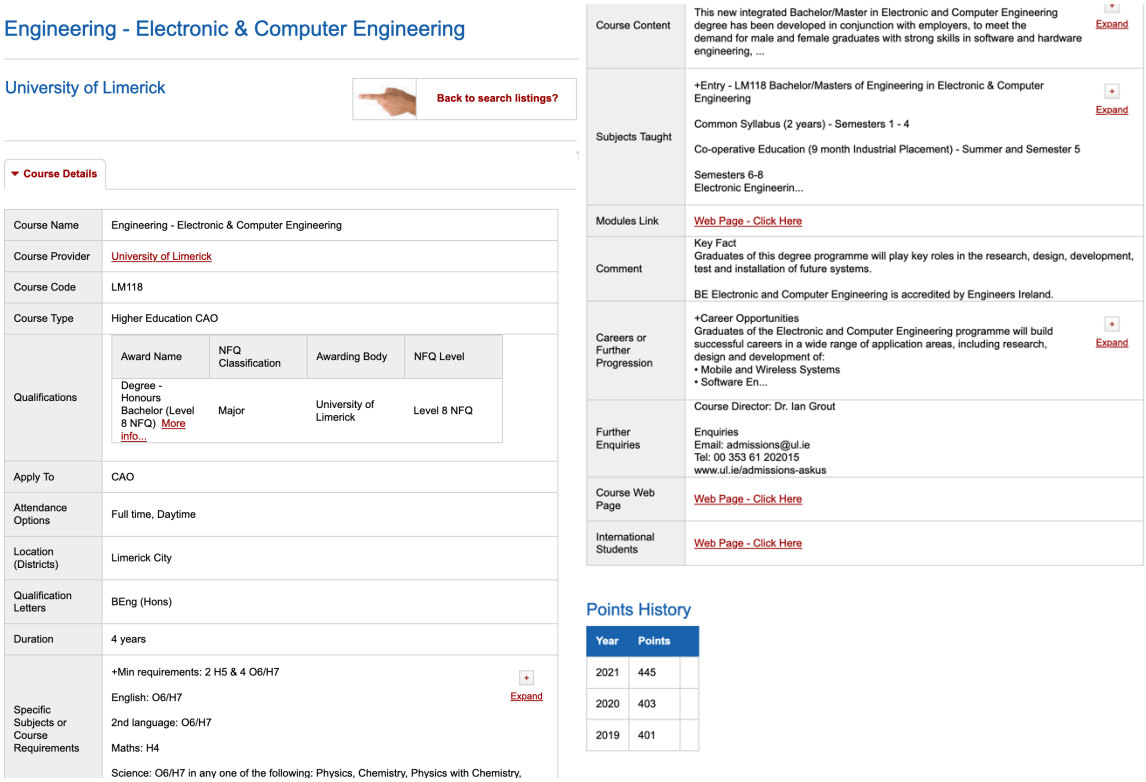
Figure 3.42: Formatting strings of links for Qualifax pagination, incrementing it's way through the directory.

Now that we have a means of extracting every course link on every page, we can simply use the same method as in the scrapy spider to dump our data into a file, since we are dumping a json file, we need to convert our list of links into a compatible object.

```
course_ links = {'links': course_ links}
filename = r'qualifax_course_links.json'
with open(filename, 'w+') as db:
   json.dump(course_links, db)
```

Figure 3.43: Converting data to dictionary and storing it as a json file.

We can now write another Selenium script to open this file, get all course pages and scrape the relevant information from them.



Engineering - Electronic & Computer Engineering

University of Limerick

Back to search listings?

Course Details

Course Name	Engineering - Electronic & Computer Engineering
Course Provider	University of Limerick
Course Code	LM118
Course Type	Higher Education CAO
Qualifications	Award Name: Degree - Honours Bachelor (Level 8 NFQ) More info... / NFQ Classification: Major / Awarding Body: University of Limerick / NFQ Level: Level 8 NFQ
Apply To	CAO
Attendance Options	Full time, Daytime
Location (Districts)	Limerick City
Qualification Letters	BEng (Hons)
Duration	4 years
Specific Subjects or Course Requirements	+Min requirements: 2 H5 & 4 O6/H7 English: O6/H7 2nd language: O6/H7 Maths: H4 Science: O6/H7 in any one of the following: Physics, Chemistry, Physics with Chemistry, Expand

Course Content	This new integrated Bachelor/Master in Electronic and Computer Engineering degree has been developed in conjunction with employers, to meet the demand for male and female graduates with strong skills in software and hardware engineering, ... Expand
Subjects Taught	+Entry - LM118 Bachelor/Masters of Engineering in Electronic & Computer Engineering Common Syllabus (2 years) - Semesters 1 - 4 Co-operative Education (9 month Industrial Placement) - Summer and Semester 5 Semesters 6-8 Electronic Engineerin... Expand
Modules Link	Web Page - Click Here
Comment	Key Fact Graduates of this degree programme will play key roles in the research, design, development, test and installation of future systems. BE Electronic and Computer Engineering is accredited by Engineers Ireland.
Careers or Further Progression	+Career Opportunities Graduates of the Electronic and Computer Engineering programme will build successful careers in a wide range of application areas, including research, design and development of: • Mobile and Wireless Systems • Software En... Expand
Further Enquiries	Course Director: Dr. Ian Grout Enquiries Email: admissions@ul.ie Tel: 00 353 61 202015 www.ul.ie/admissions-askus
Course Web Page	Web Page - Click Here
International Students	Web Page - Click Here

Points History

Year	Points
2021	445
2020	403
2019	401

Figure 3.44: LM118 Qualifax course page.

The course information and points history are two different tables, we can write a script that when passed a *table* HTML element we will parse it into a JSON object, going row to row, extracting the first cell and formatting it to be used as the key name, and the second cell as the value.

```
def parse_table(table):
    rows = table.find_elements_by_xpath('tr')
    data = {}
    i=0

    for row in rows:
        table_data = row.find_elements_by_xpath('td')
        values = [x.get_attribute('text') for x in table_data]
        column_name = values[0]
        column_value = values[1]

        if(column_name == 'qualifications'):
            subtable = parse_table(table_data[1])
            column_value = subtable
        else:
            if(column_value == 'Web Page - Click Here'):
                column_value = check_for_link(table_data[1])

        data[column_name] = column_value
    return data
```

Figure 3.45: parse_table(). When passed a table it will iterate the rows, extracting different data, reading sub tables and links.

parse_table uses a supporting function called *check_for_link* which checks text for links by attempting to extract the href attribute from the element to save the actual url instead of the call-to-action text of "Web Page – Click Here". This function is all we need to parse the course webpages, now to iterate over them.

Lets open the *search_results* our previous script provided us with;

```
def open_links():
    filename = r'qualifax_course_links.json'
    with open(filename, 'w+') as db:
        return json.load(db)['links']
```

Figure 3.46: open_links() loading in the JSON file into a Python script.

Now the bot will request every link in that list, parse the course table and a points table too if present (courses like apprenticeships would not have a points table).

```
courses = []
search_results = open_links()
for link in search_results:

    driver.get(link)
    try:
        table = driver.find_element_by_xpath(table_xpath)
        course = parse_table(table) # Parsing entire course data

        try:
            points_table = driver.find_element_by_xpath(point_table_xpath)
            points = parse_table(points_table) #
            course['points'] = points
        except: # no points
            pass
        finally:
            courses.append(course)

    except: # no course table present
        pass
```

Figure 3.47: Extracting tables using parse_tables().

Here we iterate each link, tell the driver to make a get request to the link, we try to parse the available table in the first *try* block. This *try* block will attempt to extract the data, from here then the next *try* block will attempt to extract courses, if no courses are found we do nothing by using '*pass*', then finally we append the course data to the master course list. If no table is found our scraper does nothing.

Now with all course information retrieved, and a list of our course JSON objects, we can save this to a JSON file for examining later by adding some code to when the scraper exits.

```
filename = r'qualifax.json'
with open(filename, 'w+') as db:
   json.dump(courses, db)
```

Figure 3.48: Saving data.

qualifax.json contents format;


```
[{
    "course_name": "Engineering – Electronic and Computer Engineering",
    "course_provider": "University of Limerick",
    "course_code": "LM118",
    "course_type": "Higher Education CAO",
    "qualifications": {
        "award_name": "Degree – Honours Bachelor (Level 8 NFQ)",
        "nfq_classification": "Major",
        "awarding_body": "University of Limerick",
        "nfq_level": "Level 8 NFQ"
    },
    "apply_to": "CAO",
    "attendance_options": "Full time, Daytime",
    "location_(districts)": "Limerick City",
    "qualification_letters": "BEng (Hons)",
    "duration": "4 years",
    "specific_subjects_or_course_requirements": "Min requirements: 2 H5 and 4
O6/H7\n\nEnglish: O6/H7 \n\n2nd language: O6/H7\n\nMaths: H4\n\nScience: O6/H7 in any one
```


Figure 3.49: LM118 Qualifax course page after parsing. The data is in an easily transmissible, key-value format.

We see Selenium implemented in a full extraction process here. Qualifax had hidden away the DOM for the directory tool possibly to deter scrapers, or maybe for development reasons keeping front-end and back-end separate for interchangeability.

Selenium was a successful candidate in the purpose of accessing data not accessible through usual means. This shows that technologies like Selenium will always preserve some sort of automation interaction among the web. Automation like this has proven very useful in doing repetitive tasks in the time a human could not.

Create API around UL data.

In this section we will utilise the Python framework FastAPI to build an API which will supply and return data according to inputs. It will need 3 pieces of functionality;

- Filter modules according to course year and course code.
- Filter timetable events according to course code, year and module code & group combinations.
- Construct iCalendar file using timetable events.

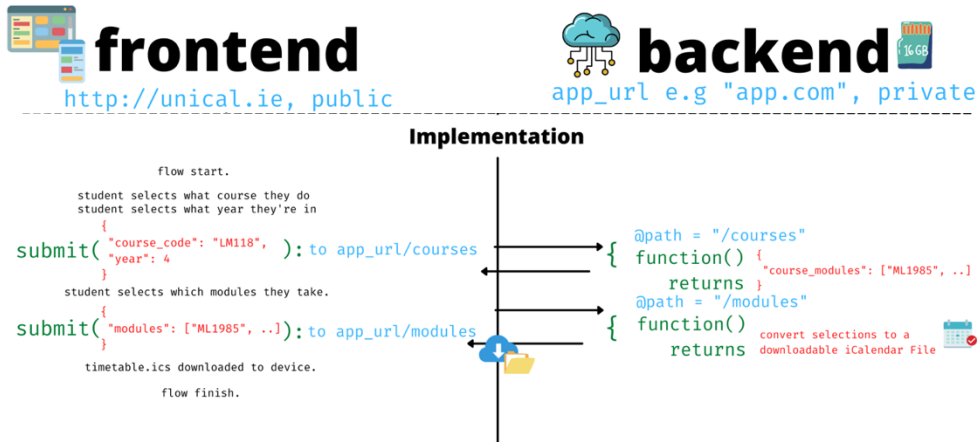


Figure 3.50: Planned implementation of web-app flow.

In the figure above, we see "flow start" of deployment API, which 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 using the DOM and makes 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.

Here we implement an API that is capable of creating .ics calendar files of events extracted from UL timetables. First we will build functions to filter the UL datasets, the original dataset scraped using Scrapy, and a derived dataset. The original data set will be used to construct calendars and the a derived dataset will be filtered to return a list of unique professor, module and group.

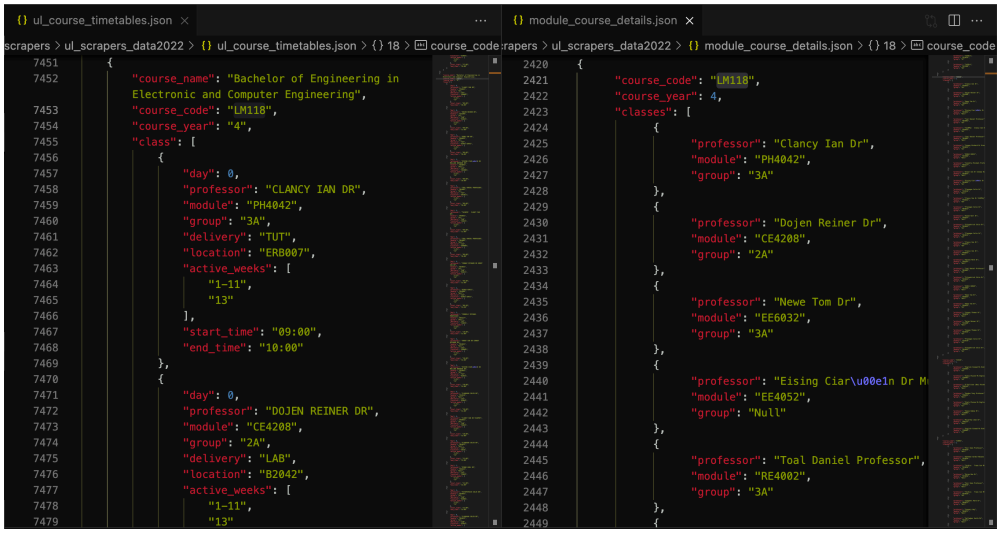


Figure 3.51: (Left) Timetables from Scrapy. (Right) Unique module combinations right, the list to be returned upon input of course year and code.

We will be using a data science package for Python called Pandas. Pandas is used to construct and manipulate data-frames; Data-frames can be easily created by reading in data using functions from pandas; *read_csv, read_excel*, or in our case *read_json*.

The DataFrame class in Pandas comes with a lot of functions, to manipulate entire columns of data. We will see some examples of data frames as we continue on.

Pre-requisites:

1. Install pandas

   >>> python3 -m pip install pandas==1.4.1

2. Create a .py script in a new directory

Filtering timetable dataset to specific Module events:

Let's load each file into the script using pandas:

```
databases = {
    "courses": "ul_course_timetables.json", # timetable.ul.ie
    "module_details": "ul_module_details.json", # bookofmodules.ul.ie
    "module_course_details": "module_course_details.json", # unique
module group-professor-code
  }

for name, datasource in databases.items():
    with open(datasource, mode="r", encoding = "utf8") as of:
```

Figure 3.52: Using pandas.read_json to construct a DataFrame from JSON files.

We will first use the file of unique module group-professor-code per course and year file; *module_course_details.json*, to build a function which will return this list of module combinations.

We will create a subset of the original data, using conditional logic we can select columns which match a certain value. We check for a entry which the column *course_code* matches a value *code* and the column *course_year* matches a variable *course_year*.

Using the conditional logic, all rows with a cell equal to the variable will be selected and all other rows will be discarded. First we filter the dataset by course code, taking a tiny slice of the data to be further filtered by course year.

```
def get_course(data, course_code='', course_year=0):
    data = data[data['course_code'] == course_code]
    data = data[data['course_year'] == course_year]

    if(len(data) == 0): # data is empty
        raise KeyError(f"No results")

    course = data.iloc[0]
    module_lists = clean_modules(course['classes'])

    return module_lists
```

Figure 3.53: get_course() which accepts course_code and course_year and filters the dataset down to a list of modules.

In *get_course*, we are passing in the data of *module_course_details.py* (unique module group-code-professor), and if the conditions are True the data should be filtered down to one course, if not we raise an error so we can alert our API that something has gone wrong and return an error message to the webpage.

With the data filtered to one course only (a single row structure), we select the specific row and cell of the resulting data and "clean" them in *clean_modules*, this is some code to reduce the number of options from 31 to 21 for the course of LM118 course year 4. It simply removes unnecessary module options based on an assumption that any module with an option for a non-Null group, does not also need an option for a 'Null' group; *if you are in tutorial group 3A, you are in the Lecture of group 'Null', as this is a global lecture group for all tutorial groups*.

```
course_code: LM118

course_year: 4

len of module_course_details.json: 407

results matching course_code of LM118: 4

..and matching course_year of 4: 1

data:
   course_code  course_year                                            classes
18       LM118            4  [{'professor': 'Clancy Ian Dr', 'module': 'PH4...

course:
course_code                                          LM118
course_year                                              4
classes        [{'professor': 'Clancy Ian Dr', 'module': 'PH4...
Name: 18, dtype: object

len of module_lists before cleaning: 31

len of module_lists after cleaning: 24
[{'professor': 'Dojen Reiner Dr', 'module': 'CE4208', 'group': '2A'}, {'professor': 'Flanagan Colin Dr', 'module': 'CE4518', 'group': '3A'}, {'professor':
'Flanagan Colin Dr', 'module': 'CE4518', 'group': 'Null'}, {'professor': 'Conway Richard Dr Grout Ian Dr', 'module': 'EE4032', 'group': '2A'}, {'professor'
: 'Conway Thomas Dr', 'module': 'EE4038', 'group': '3A'}, {'professor': 'Conway Thomas Dr', 'module': 'EE4038', 'group': 'Null'}, {'professor': 'Fitzpatric
k Colin Dr', 'module': 'EE4042', 'group': '2A'}, {'professor': 'Fitzpatrick Colin Dr', 'module': 'EE4042', 'group': '3A'}, {'professor': 'Eising Ciarán Dr
Mullane Brendan Dr', 'module': 'EE4052', 'group': '3A'}, {'professor': 'Connelly Michael Professor', 'module': 'EE4117', 'group': 'Null'}, {'professor': 'B
2043 B2011', 'module': 'EE4908', 'group': 'Null'}, {'professor': 'B2011 B2043', 'module': 'EE4908', 'group': 'Null'}, {'professor': 'Newe Tom Dr', 'module'
: 'EE6032', 'group': '2A'}, {'professor': 'Newe Tom Dr', 'module': 'EE6032', 'group': '3A'}, {'professor': 'Halton Mark Dr', 'module': 'EE6452', 'group': '
Null'}, {'professor': 'Rinne Karl Dr', 'module': 'EE6622', 'group': 'Null'}, {'professor': 'Ta10Phy   Clancy Ian Dr', 'module': 'PH4042', 'group': '2A'}, {
```

Figure 3.54: Output from get_course()

Now with a list of modules, lets filter our ul_course_timetables.json. We will want to filter by course code, course year, as well as a list of module code and group combinations. We will call the function *get_timetable()*, it will return the event data for the timetables scraped using Scrapy.

```
timetable = … conditional logic to choose course from ul_course_timetables.json
filtered_timetable = []
for each event in timetable:
        if event_module_code and group is in input_module_combinations:
                append module to filtered_timetable
return filtered_timetable
```

Figure 3.55: Pseudo code for filtering of timetables. Appending input module events to a list.

The conditional logic is the same as getting the list of modules; First we filter by course code then course year, if the dataframe is empty i.e. the length is 0, then that course code and year combination does not exist.

```
def get_timetable(data, course_code='', course_year=0, module_combinations=[]):
    data = data[data['course_code'] == course_code]
    data = data[data['course_year'] == course_year]
    if(len(data) == 0):
        raise KeyError(f"No results")

    timetable = data.iloc[0]['class']
```

Figure 3.56: Conditional logic for selecting course and getting classes

From here we filter out by course_code, year and module_combinations;

```
def get_timetable(data, course_code='', course_year=0, module_combinations=[]):
    ...
    specified_timetable = []

    for module in timetable:
        module_group = {'module': module['module'], 'group': module['group']}
        if(module_group in module_combinations):
            specified_timetable.append(module)

    return specified_timetable
```

Figure 3.57: Filtering timetable events to matching module_combinations

I reconstruct each module entry in the timetable to a JSON object like one in the module_combinations, we check if that JSON object is in the module_combinations and then append if so.

The only this missing are the module entries that have possible Nulls as their “group”, which are global events for anyone in that module, so we introduce them to our *module_combinations* if they are not already in it, in the beginning of function;

```
def get_timetable(data, course_code='', course_year=0, module_combinations=[]):
  # assumption that if you are in a tut or lab group you are in the Null (global)
group for lecs etc.
  module_combinations += [{'module':x['module'], 'group':'Null'} for x in
                module_combinations if({'module':x['module'], 'group':'Null'} not in
                     module_combinations)]

  ...

  ...
  return specified_timetable
```

Figure 3.58: Re-introducing the global events (group == Null) for the modules in module_combinations.

We can implement this using:

```
get_timetable(db, course_code='LM118', course_year=4,
                            module_combinations=[{'module':"CE4518", 'group': '3A'}])
```

Figure 3.59: Implementing get_timetable using some sample data.

Printing the changes our data undergoes;

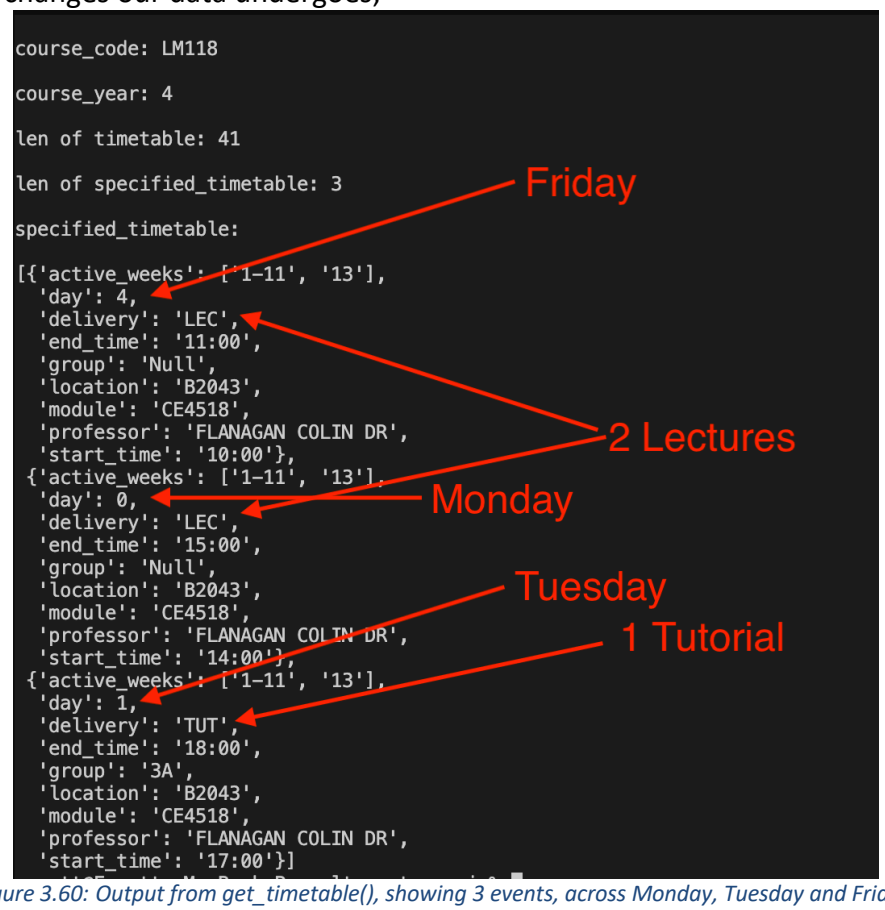


Figure 3.60: Output from get_timetable(), showing 3 events, across Monday, Tuesday and Friday.

We see 3 events in the *specified_timetable* for a person who would attend CE4518 in tutorial group 3A. We are looking for 2 lectures over Monday and Friday and a single tutorial on Tuesday. If we check the actual LM118 course year 4 timetable from timetable.ul.ie, we can check if our data is correct.

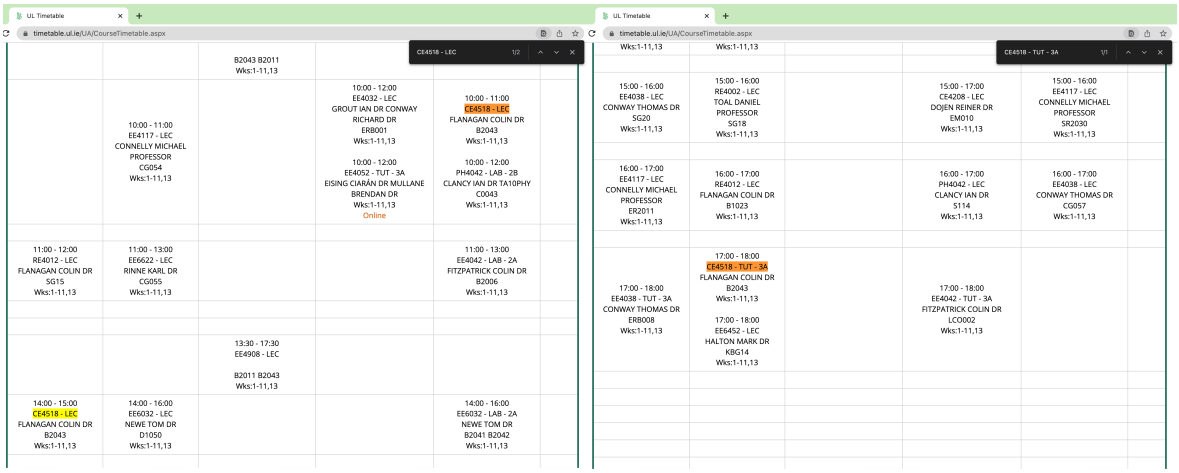


Figure 3.61: Two lectures and one tutorial, across Monday, Tuesday and Friday, validating our output.

Thus with a few more manual sanity checks we can validate our method and our data is valid. Once happy with the algorithm here and it's performance, we want to convert the event data into a .ics file format to build the calendar from this event data.

Creating a Universal Calendar Formatted file:

Pre-requisites:

1. Install icalendar

   >>> python3 -m pip install icalendar==4.0.9

2. Create a .py script in the same directory as above file.
3. Import packages into script:

   from icalendar import Calendar, Alert, Event

   from timedate import timedate

I learned the format of an .ics file by inspecting a few online, mostly google meets .ics files (attached to an email invite to a google meeting) they are binary text files with specific formatting and a .ics extension. The most important thing is to identify which arguments we want to set to what values.

```
1   BEGIN:VCALENDAR
2   VERSION:2.0
3   BEGIN:VEVENT
4   SUMMARY:Coffee with Luigi
5   DTSTART;VALUE=DATE-TIME:20220124T090000
6   DURATION:P00DT2H0M00S
7   RRULE:FREQ=WEEKLY;UNTIL=20220404T110000;INTERVAL=1;WKST=MO
8   LOCATION:Browser Castle
9   BEGIN:VALARM
10  ACTION:DISPLAY
11  DESCRIPTION:REMINDER
12  TRIGGER;RELATED=START:-PT15M
13  END:VALARM
14  END:VEVENT
15  END:VCALENDAR
```

Figure 3.62: Universal Calendar Format.

```
BEGIN:VCALENDAR
VERSION:2.0
BEGIN:VEVENT
SUMMARY:Coffee with Luigi
DTSTART;VALUE=DATE-TIME:24/1/2022 9am
DURATION:2 Hours
RRULE:FREQ=WEEKLY;UNTIL=4/4/2022 11am;INTERVAL=1;WKST=Monday
LOCATION:Browser Castle
BEGIN:VALARM
ACTION:DISPLAY
DESCRIPTION:REMINDER
TRIGGER;RELATED=START:15 minutes before
END:VALARM
END:VEVENT
END:VCALENDAR
```

Figure 3.63: Pseudo universal calendar format. Showing plain English for some values.

Within each BEGIN:VCALENDAR there can be any number of events, each event starts with the BEGIN:VEVENT keyword.
In each event we have;

- SUMMARY: title of event.
- DTSTART: first date and time of the event.
- DURATION: length of event
- DTSTART: first date and time of the event.
- RRULE: repetition rule.
    - FREQ: frequency of weekly, monthly, yearly.
    - UNTIL: end date and time
    - INTERVAL: every 1, 2, 3, 4…. FREQ periods
    - WKST: start of the week
- LOCATION; location

We can also give our events each an alert, which would prompt any calendar software to push a notification about the upcoming event. The reminder starts with keyword BEGIN:VALARM, in each alarm we have;

- ACTION: set to display to be a push notification
- DESCRIPTION: set to reminder.
- TRIGGER: what causes the event to go off.
    - RELATED: relative to start, minus 15 minutes to cause the alarm to trigger 15 minutes before.

We will create the function *create_ics*(), which will use the data from *get_timetable()* to translate the timetable into .ics format. First we will need to define the academic calendar in a python dictionary. All the keys will be week numbers from 1 to 13, and each value will be a list of dates in that week in mm/dd format. We will use the integer we assigned for event event's "*day*" value, to index a date in the corresponding week and use that for our start date.

```
academic_calendar = {
        "1": ["1-24", "1-25", "1-26", "1-27", "1-28", "1-29", "1-30"],
        "2": ["1-31", "2-1", "2-2", "2-3", "2-4", "2-5", "2-6"],
        "3": ["2-7", "2-8", "2-9", "2-10", "2-11", "2-12", "2-13"],
        "4": ["2-14", "2-15", "2-16", "2-17", "2-18", "2-19", "2-20"],
        "5": ["2-21", "2-22", "2-23", "2-24", "2-25", "2-26", "2-27"],
        "6": ["2-28", "3-1", "3-2", "3-3", "3-4", "3-5", "3-6"],
        "7": ["3-7", "3-8", "3-9", "3-10", "3-11", "3-12", "3-13"],
        "8": ["3-14", "3-15", "3-16", "3-17", "3-18", "3-19", "3-20"],
        "9": ["3-21", "3-22", "3-23", "3-24", "3-25", "3-26", "3-27"],
        "10": ["3-28", "3-29", "3-30", "3-31", "4-1", "4-2", "4-3"],
        "11": ["4-4", "4-5", "4-6", "4-7", "4-8", "4-9", "4-10"],
        "12": ["4-11", "4-12", "4-13", "4-14", "4-15", "4-16", "4-17"],
        "13": ["4-18", "4-19", "4-20", "4-21", "4-22", "4-23", "4-24"],
    }
```

Figure 3.64: Academic calendar mapped to a Python dictionary object.

Next we will initialise the icalendar *Calendar()* class, and add a version using the classes *add()* function, which takes the keyword first and value second as arguments. We can create an *Alert()* class similarly, and add the same attributes we seen in our template from the internet. We create a trigger relative to the start time of the event, scheduling the trigger for 15 minutes before.

```
def create_ics(events = []):

    calendar = Calendar()
    calendar.add('version', '2.0')

    alarm = Alarm()
    alarm['trigger;related=start'] = '-PT15M'
    alarm.add('ACTION', 'DISPLAY')
    alarm.add('DESCRIPTION', 'REMINDER')

    year = datetime.now().today().year
```

Figure 3.65: Initialising Calendar class and setting version.

From here we can create a for loop which iterates *events*. We can write pseudocode to explain the process:

```
for event in events:
        set title  = module_code – module_delivery – module_group?
        get start hour, minute
        get end hour, minute
        for week_range in active_weeks:
                get start week, end week from academic_calendar using weeks as index
                get start date, end date from start week, end week using event day as index
                create datetime object of start date & time
                create datetime object of end date & time
                event = create and format icalendar Event class
                add event to calendar
```

Figure 3.66: Pseudo code to format and fill a calendar with events in create_ics().

With an input in the same format as the *specified_timetable* from *get_timetable()*:

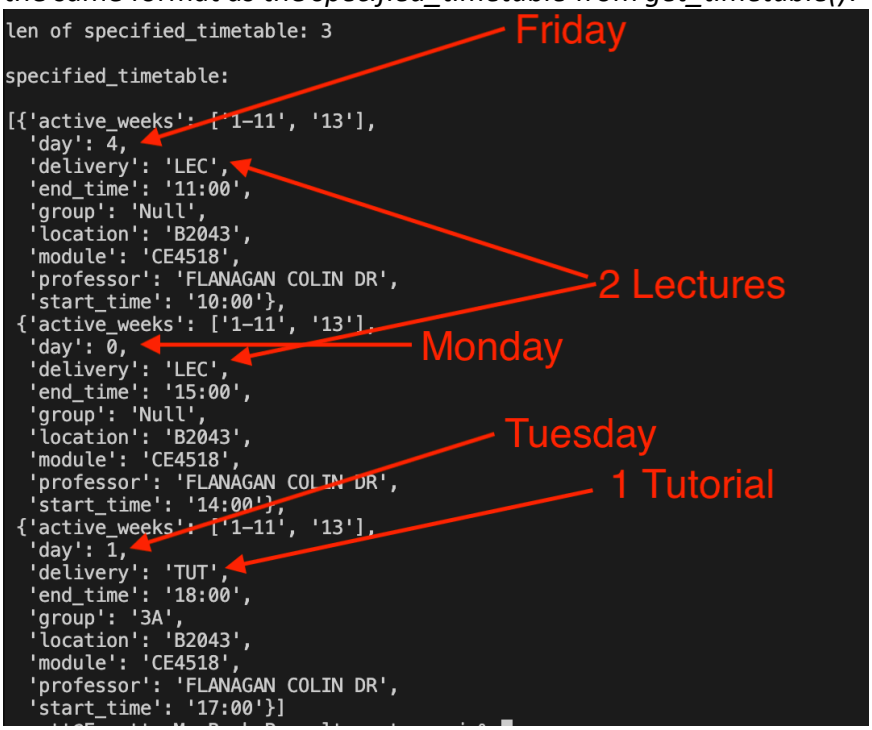


Figure 3.67: Output from get_timetable(), acquired from the method built in the previous section.

```
for event in events:
        active_weeks = event['active_weeks']
        # active_weeks = e.g. [‘1-3’, ‘9-11’, ‘13’] or [‘13’]
        # event[‘start_time’] = e.g. ’09:00’
        start_hour, start_minute = get_time(event['start_time'])
        end_hour, end_minute = get_time(event['end_time'])

        day = event['day']

        if(event['group'] != 'Null'):
            event_title = event['module']+' - '+event['delivery']+' - '+event['group']
        else:
            event_title = event['module']+' - '+event['delivery']

        for week_range in active_weeks:
            ...
            ..
```

Figure 3.68: In create_ics(), getting static module data.

First we compile some information, before creating events per *active week*. The function *get_time()*, simply converts a time string, e.g. ’09:00’, and returns just the 9 for hour and 00 for minutes by splitting the string at the colon. We also delimit and concatenate the module code, delivery and group as the *event_title*.

We then iterate over *active_weeks*; we need a start time stamp and a end time stamp for our weekly repeating event. To build a datime object we need to set *datetime(year, month, day, hour, minute, seconds)*, all date values in integer form. We can query the academic

table using the first week in a *week_range* which returns a list of dates, and an index of the day which the event occurs. I do this in a new function called *get_date_range()*.

```
def get_date_range(week_range='', day=0):
    active_weeks_list = week_range.split('-') # '3-12' then ['3', '12'] or '3' to ['3']
    start_week = active_weeks_list[0] # first element
    # get week number from academic_calendar
    # index the list of dates in week 3 for our events day, 0=Monday, 5=Saturday
    start_date = academic_calendar[start_week][day]
    # split the element into [month, date]
    start_month_date = (start_date).split('-')
    start_month = start_month_date[0]
    start_date = start_month_date[1]

    # if more than one week
    if(len(active_weeks_list) > 1):
        # calculate end dates
        end_week = active_weeks_list[1]
        end_month_date = academic_calendar[end_week][day].split('-')
        end_month = end_month_date[0]
        end_date = end_month_date[1]
    else:
        # event is one time occasion
        end_month = start_month
        end_date = start_date

    return int(start_month), int(start_date), int(end_month), int(end_date)
```

Figure 3.69: get_date_range(), Computing start date and end date. If there is no repetitive nature of the event, then it is a one time occasion.

We will build a function to pass the event data into, which initialises an *Event()* class, and set its variables; We denote the SUMMARY as the title of the event (module – group), the start as the start time, and location as the location. We set the repeat rule to be weekly until class_repeat_finish, which is gotten and created using the function in figure 3.69.

```
def set_event(event_title, class_time, location, class_repeat_finish):
    e = Event()
    e.add('SUMMARY', event_title)
    e.add('DTSTART', class_time)
    e.add('LOCATION', location)
    e.add('RRULE',{'freq':'weekly','interval':1,'wkst':'MO','until':class_repeat_finish})
    return e
```

Figure 3.70: set_event(), Creating and filling a ics format event.

Now we can implement these functions into our original *create_ics()* function and append a new *event* per *week_range* of each timetabled event.

```
def create_ics(events = []):
        ...
        ..
        for week_range in active_weeks:
           start_month, start_date, end_month, end_date = get_date_range(week_range,day)

           class_time =datetime(year,start_month,start_date,start_hour,start_minute,00)
           class_repeat_finish =datetime(year,end_month,end_date,end_hour,end_minute,00)

           location = event['location']
           formatted_event=set_event(event_title,class_time,location,class_repeat_finish)
           calendar.add_component(formatted_event)
    return calendar.to_ical()
```

Figure 3.71: Creating datetime objects from get_date_range() and using them in set_event().

Now with *event* defined, we add the component to the *calendar* class. The calendar is now fully formatted with all events fed from the timetable. If we inspect our output from *create_ics()*, using the input of just one event;

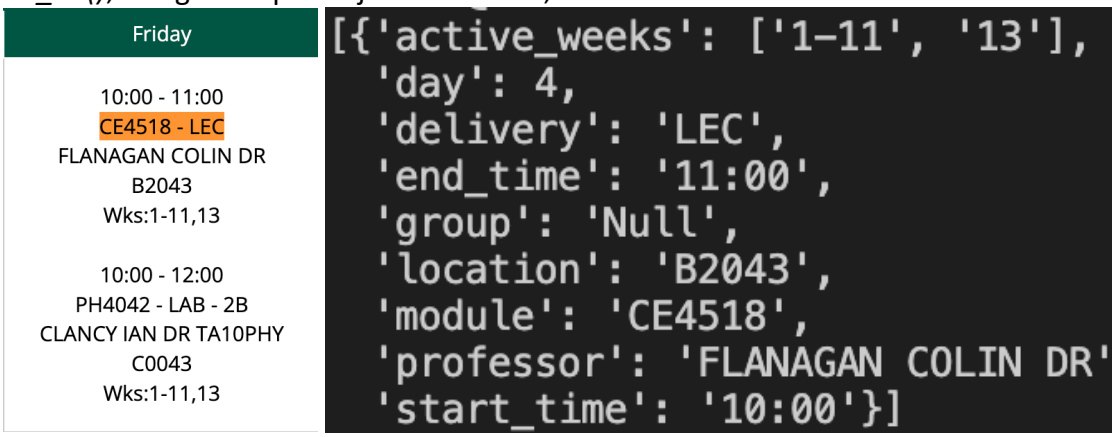


Figure 3.72: One timetabled event input, with two ranges of active weeks, as it appears on our timetable and in our data.

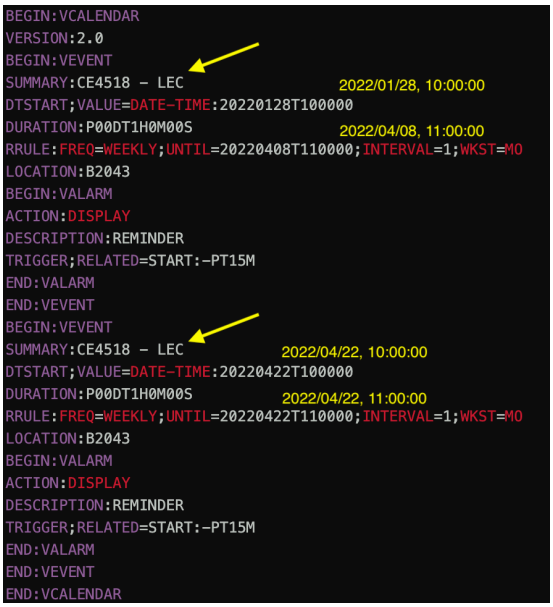


Figure 3.73: Output of create_ics() represented as a text file. The above timetabled event in Universal Calendar Format.

In figure 3.73, we see 2 lecture events, both have the same start time and hour, except one event repeats weekly from week 1 to week 11, and then the other event happens once in week 13, as timetabled.

We will import these files and their various functions to be used in the webapp script which will be in the same directory as these scripts.

Opening methods for access from the internet using FastAPI:

Pre-requisites:

1. Install fastapi

   >>> python3 -m pip install fastapi==0.75.0
2. Create a endpoints.py script in the same directory as above scripts.
3. Create a template webapp file as we did in Chapter 2, where we created a FastAPI app to demonstrate it's framework.
4. Import packages into same endpoints.py script:

   from fastapi import FastAPI, Request
   from fastapi.response import StreamingResponse
   from fastapi.middleware.cors import cors,
   from timedate import timedate
5. Create a webapp.py script in same directory.
6. Fill with contents described for hosting configuration:

   import uvicorn
   if __name__ == '__main__':
   uvicorn.run("endpoints:app",host='0.0.0.0',port=8080,reload=True)

I will have to omit a lot of parts of this code, as exposing the entire API infrastructure could be a security risk. But the format of the script is based on the template created in Chapter 2. We can outline the structure of the API using some basic code;

```
@app.get("/course/", tags=["GetModules"])
async def get_module_combinations():
    module_lists = get_course(module_course_details, "LM118", 4)
    return module_lists
```

Figure 3.74: get_course() implemented into a endpoint on our API.

Here we add an end point located at /course/, which you can only reach with a GET request. We simply return the result from our function which is the unique module combinations. We can then test this endpoint by running our webapp.py file and visiting localhost:8080/course/ in our browser.

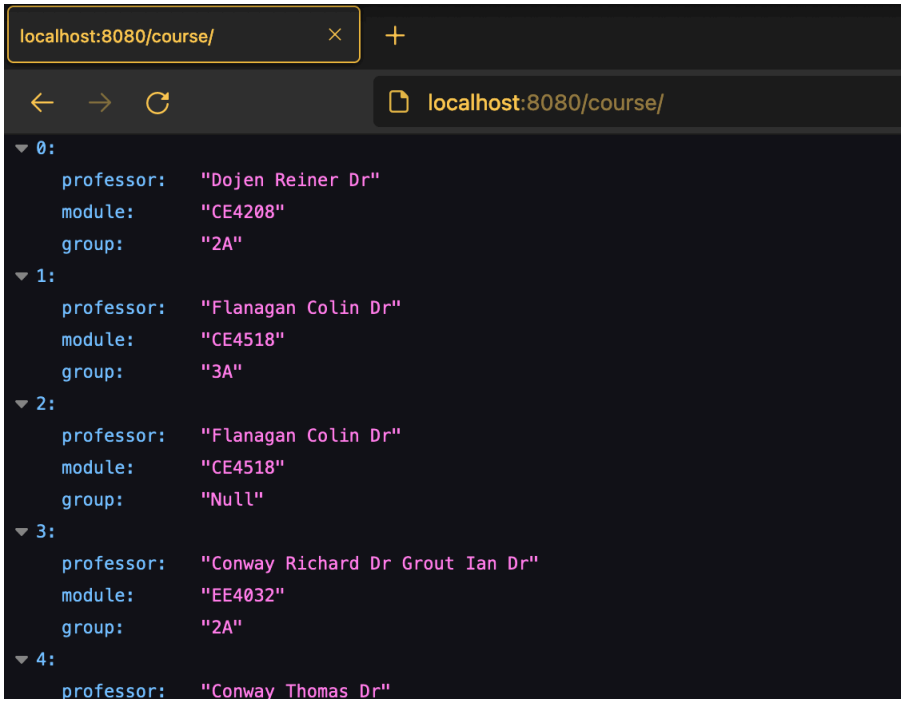


Figure 3.75: Response of localhost:8080/course/. Shows a successful creation and transmission of data over the API.

```python
@app.get("/timetables/", tags=["GetTimetable"])
async def GetTimetableDownload():

    timetable=get_timetable(timetables,"LM118",4,[{"module":"CE4518","group":"3A"}])

    calendarized_timetable = create_ics(timetable)
    media_type="text/calendar"
    response = StreamingResponse(iter([calendarized_timetable]), media_type)
    response.headers["Content-Disposition"] = "attachment; filename=timetable.ics"

    return response
```

Figure 3.76: Implementation of get_timetable() and create_ics() in a new endpoint 'timetables/'. We get the filtered and encode it as Universal Calendar Format using create_ics() from the above section. We set the response media-type so the client browser knows what data to expect.

Here we implement *get_timetable()*, and *create_ics()*, with our resulting binary calendar object *calendarized_timetable*, we can prepare our *response* as a *StreamingResponse()*, which takes an iterable object, in our case the *calendarized_timetable*, we will also set the *media_type* to a text file of format calendar. We can also name the file in the response headers using Content-Disposition.

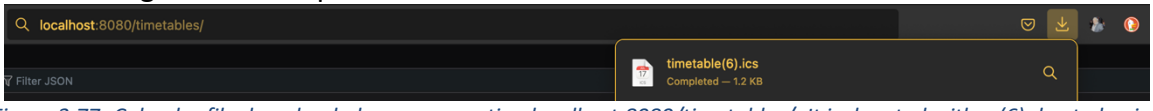


Figure 3.77: Calendar file downloaded upon requesting localhost:8080/timetables/. It is denoted with a (6) due to having downloaded multiple timetable.ics files already.

We now have a fully functioning API which implements the filtering of the data extracted from scrapers. With our API functional, we can construct the JavaScript and HTML which makes fetch requests and visitors to our site can interact and use our API.

The HTML was based on a template from; https://colorlib.com/wp/template/login-form-20/, the JavaScript works using *fetch* and *Document Object Modelling*, as described in Chapter 2. This can be adapted to any implementation we like.

FastAPI was a very good choice in the interest of time. I have worked with alternate frameworks like Django and Flask, but none are as straight-forward as FastAPI. This API now can be adapted to accept requests from any source, easily integrate-able with any other types of systems new or old, as long as the primary language has a way of making a request to this API, then data can be returned.

Writing the API script was the first step to being able to distribute the functionality for converting course timetables to Calendar format. Now students do not have to check a static webpage and can instead continue to study away and receive 15 minute alerts to go to class.

15 minutes seemed like the ideal timeframe to get from anywhere on campus to a classroom, always giving time to go the bathroom or complete your last task before leaving for class. The idea of this API was to reduce the amount of effort it takes to keep up with your day, we can remain passive and know that if we miss a class it is because either the phone is on silent or dead.

I hope to and can absolutely, continue adding functionality to this API, creating a suite of tools for students to start their semester; such as recommending books, YouTube videos, Wikipedia articles and informational articles based on their course material, something we can do by analysing the bodies of course and module descriptions, to figure out the key information as to which we would gather supporting material.

Conclusion:

Implementing the entire structure was a difficult process, but worth-it learning process. There were some points I was writing in 5 different languages trying to get the stack to work; HTML, CSS, JavaScript, Python and PHP. This was the first time I attempted a project of this size, the hardest parts were definitely making sure the data for the timetables was correct, testing a lot of timetables was time consuming and expensive of energy. It is very anxiety inducing to think if I mess up someone's timetable they are possibly stuck with incorrect notifications for an entire semester.

The second hardest part was developing the iCalendar format. There was very little documentation for the iCalendar library used in this project, it was very tough to figure out what worked and what didn't, it helped to eventually base my template off of a pre-existing Universal Calendar Formatted file from google.

Figuring out the StreamingResponse in the API was also very tiring, at the time I wasn't fully aware of what was required to build a response capable of returning a file. I was recommended the StreamingResponse from a friend in work who was familiar with FastAPI, and after changing the content type it worked perfectly.

This was all trial and error, taking several months to perfect and ensure data and method validity. Something like this could possibly take a very experienced programmer a significantly less amount of time, but it takes a student to understand the timetable layout and the need for an integrated calendar API for a static-excel-sheet timetable, and a motivated student to learn how to bring this to fruition.

4. Architecture (Tech Stack)

In this chapter I will go through the architecture or technology stack of the scrapers which gather data for the webapp, and the webapp itself. A technology stack consists of the frontend; how it looks, and the backend; how it works, including any middle-layers we use to help each end communicate, which all work as one unit to perform a function as efficiently as possible.

A side not is I regret not doing this earlier. I attempted development of this project without a clear architecture in my head, as I was learning as I went along, I did not know all the methods and technologies I would use before I stumbled across them during development.

I will be translating the code into Architecture to outline the pattern and flow of the webapp. I hope this will highlight some areas or complex integrations to fix or redact in the case I were to ever recreate this project.

We will first start with the architecture of the scrapers, from booting-up to storage of the data they are employed to collect.

We will then explain the architecture of the webapp, from where it is hosted, down to the functions which we created in Chapter 3.

Scraper Architecture:

Scrapers are the method of gathering data from websites, we can construct a flow which includes their target sites from starting of the scraper scripts, to storage of the data into the JSON files to be implemented in the Webapp or otherwise.

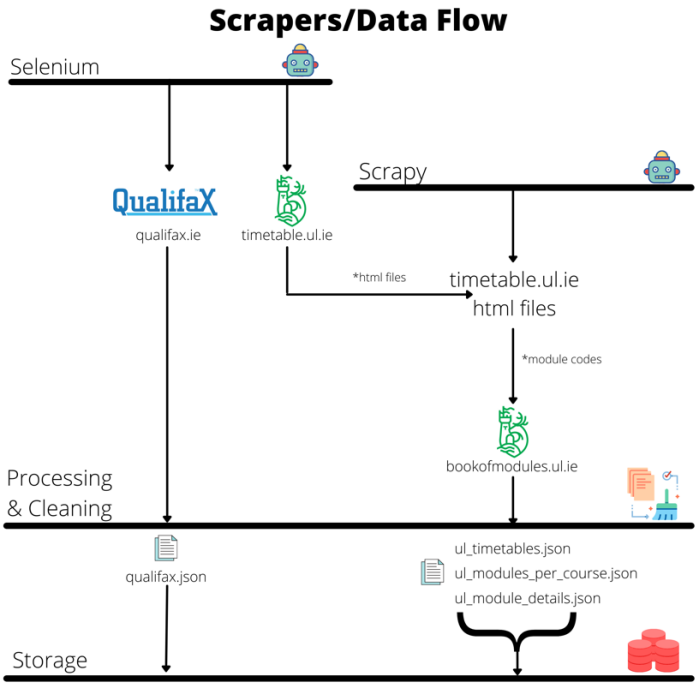


Figure 4.1: Scraper/Data Flow design and architecture.

From the initialising of the Selenium and Scrapy scrapers (depicted as “bots”), they scrape the data from their designated targets, they will then pass this data to be processed and cleaned into an appropriate format, which is then stored as their respective files.

This entire layer will utilise Python 3.8 and has dependencies on Scrapy 2.5 and Selenium 4.1 for scraping, and Pandas 1.4.1 for data cleaning and processing. The flow of data is based on the execution of the scrapers and thus is not optimal.

I would improve this layer by removing the pass-off between Selenium and Scrapy with the timetable.ul.ie HTML files. I would rather investigate ways to reverse engineer the official UL timetable API, used on the timetable.ul.ie site. With this API reverse engineered, we can then use it to get data as it requested by users on the Webapp side, instead of storing timetables in the way we do now.

Reverse Engineering the UL timetable API was not viable at the time due to the API requiring some data on what course timetable to return. One such field required the course title and code in the format “*course-title-(code)*” i.e. “Bachelors-in-Engineering-(LMXXX)”. I did not have course titles at the time, but after using Selenium, I could use this historical data to make requests to the UL API, improving speed and reducing reliance on Scrapers. Maybe in the future.

API (Automated Programming Interface):

Our API consists of the data we scraped and the logic needed to perform our webapps backend functionality. It is built entirely in Python and hosted on Heroku. The API has a private URL and will only accept request which originate from our frontend domain.

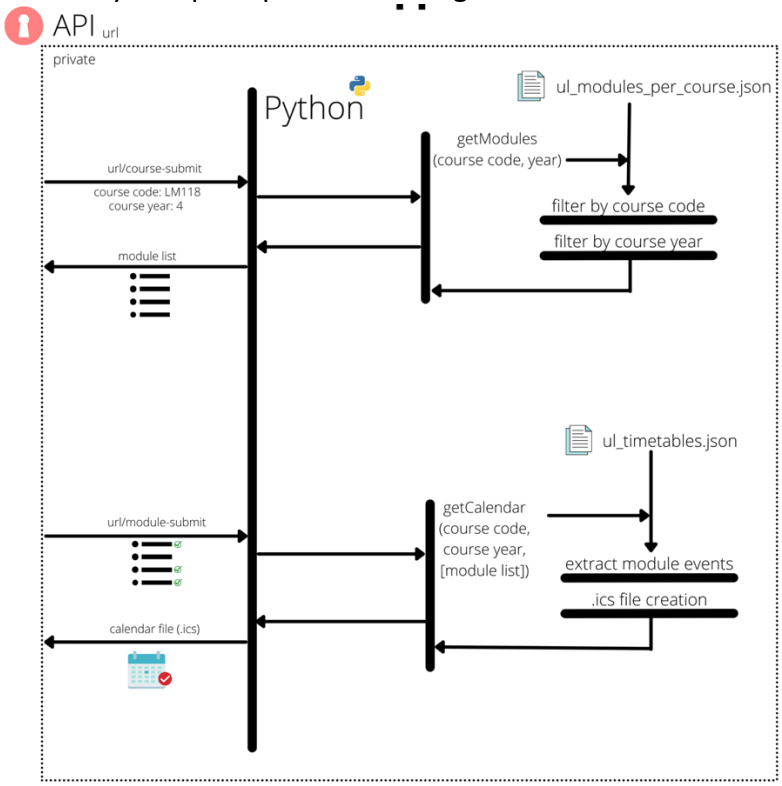


Figure 4.2: API flow and Architecture. Where student requests come in from Left and our backend architecture is on Right.

In figure 4.2, we see requests and responses coming in and out respectively from the left, carrying the data needed. The request is the starting point and the response is the end of each sub-process, reading top to bottom of the arrows on the left.

Upon a request made to the APIs private URL with extension "/course-submit" and bearing the correct parameters, we pass this data to the function; getModules(), where we filter the data in ul_modules_per_course.json, the json file with lists of module combinations to a given course code and year. We return the resulting module list.

A request is then made to our API url with extension; "/module-submit", with some selected modules from the module list, course year and course code. We run this through our function getCalendar, which filters ul_timetables.json for their timetabled events and creates a calendar file, which we return in the response.

Web-Application:

I based the concept of the webapps tech stack on a famous stack called the LAMP stack which consists of Linux, Apache, MySQL (A database manager) and PHP. The LAMP stack is quite old so we can swap out the MySQL for any sort of database manager, or in my case none at all, just our JSON files. We can also swap out our server language PHP for Python. We will however still use Linux and Apache for hosting the frontend.

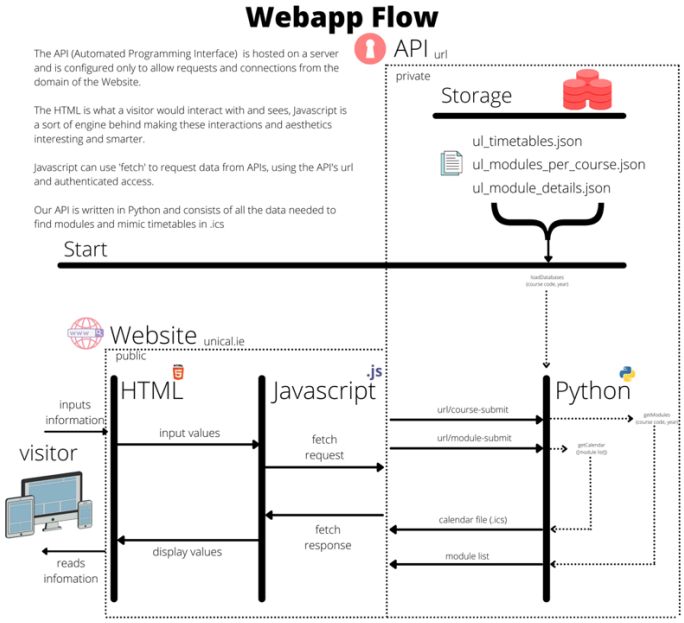


Figure 4.3: Web App flow and Architecture. You can see the aforementioned API and how the website integrates into this.

For the webapp we combine two separate components; the website and the API, or frontend and backend respectively. The entire stack contains 4 software languages;

- Website, Frontend;
  - HTML
  - CSS
  - JavaScript
- API, Backend;
  - Python

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.

The API keeps our data in active memory, and any action to filter the data is optimised within the code (i.e. filtering by course code first then course year, instead of course year then course code; resulting in a much cleaner filtration of the data). But still, users will be expected to make 2 CPU costly requests, filtration of courses down to modules, then modules into Universal Calendar Format, by using a free hosting service we can avoid CPU costs which can be the killer with hosting services like Amazon Web Services (AWS).

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. We can host webservers in the cloud through these virtual machines.

Google Cloud and AWS come with many Virtual Machine options 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, which did not end up being free and costing around $15 a month to run in times of high usage by students, going as low as $2 later in the semester.

The flow begins at the user who is greeted by the landing page of the website, they would then enter their relevant information and press "Get Modules" as shown in Figure 4.4.

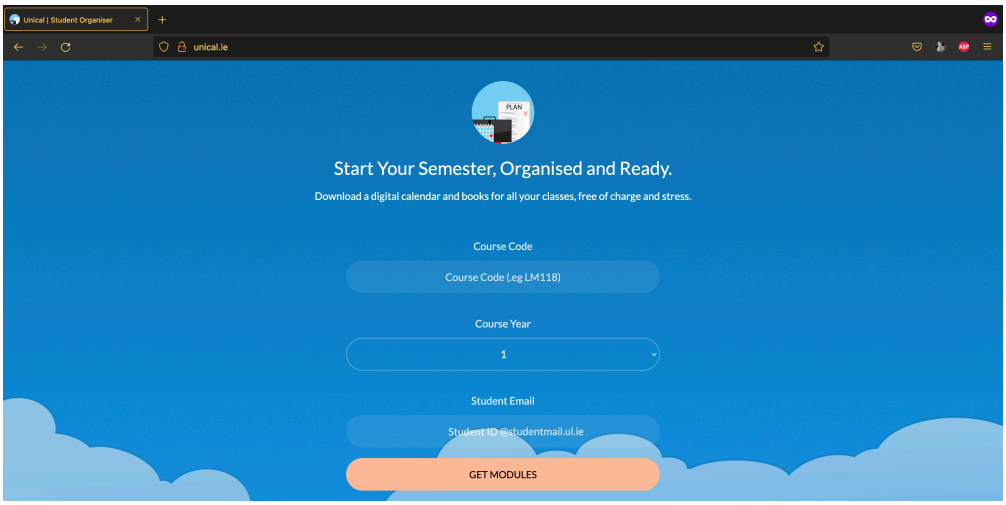


Figure 4.4: Landing page for visitors of the frontend, unical.ie.

Upon submission of this data, JavaScript gathers the information from the HTML and sends it to our API, specifically to the API endpoint; "/course-submit", the rest of the flow is described in the API architecture.

This is where the problem with hosting in separate places really shows, this round trip (request to response) takes around 12 seconds to complete on the users very first request, then taking milliseconds thereafter. This is due to the API and the GUI being stored on two different low-budget servers.

The list of modules the student receives in the response from the API of the above request is shown in figure 4.5. A list of all Module, Delivery, Group and Lecturer combinations according to their course and year is presented.

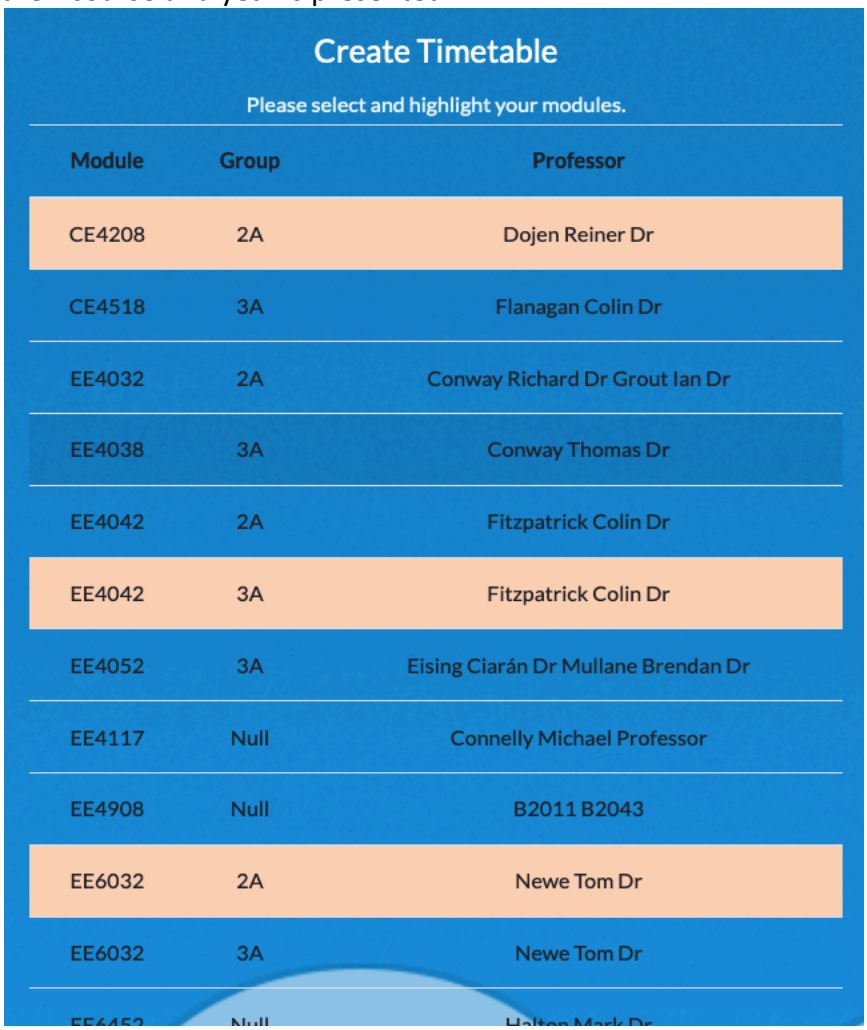


Figure 4.5: List of modules with some selected from LM118. Modules with tutorials and labs do not have a Null group presented as per our earlier assumption; if you are in a group then you attend the events of group Null too.

Visitors then need to select “Get Calendar” in figure 4.6 to submit their entries, then all accumulative data from the previous input (Couse Year, Course Code), plus this list of selected modules is sent to the API. The API accepts the request and streams the binary Universal Calendar Format in the response filled with events from selected modules.

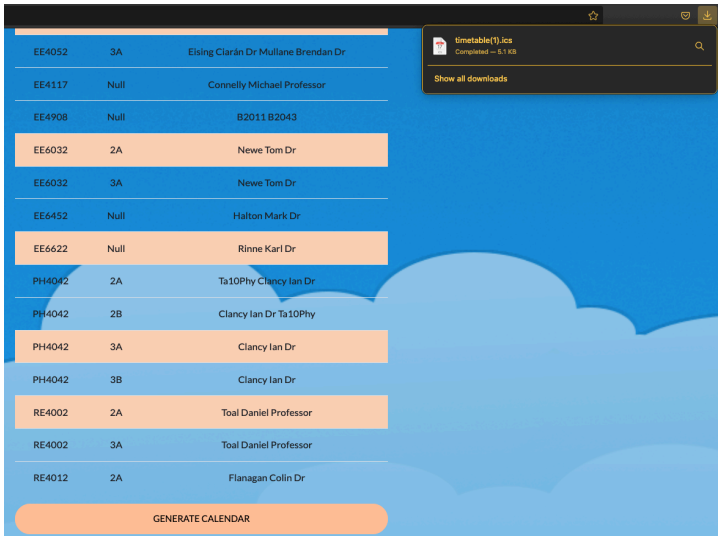


Figure 4.6: Timetable.ics file downloaded.

The next thing to happen is a calendar file is downloaded to the visitors browser, from here they can open this file in any calendar friendly app; Outlook, iCalendar, Android Calendar and Google Calendar, this will display all the events and is configured to notify students 15 minutes before events.

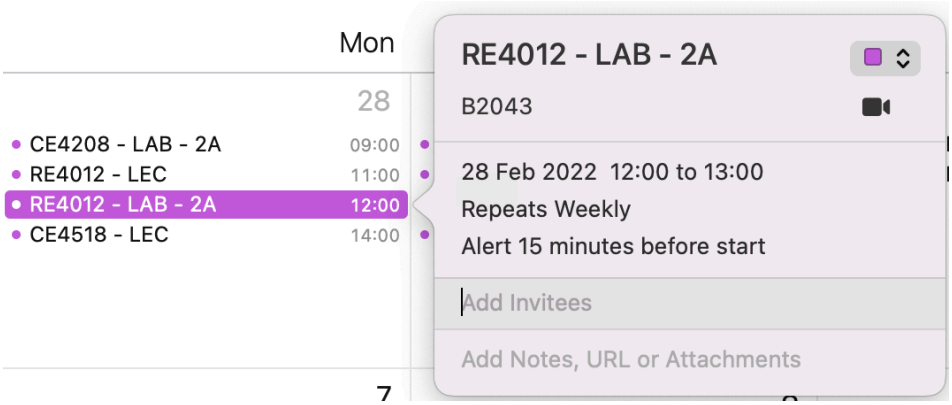


Figure 4.7: Timetable.ics opened on native calendar of MacOS.

Conclusion:

The architecture of the project as a whole is quite understandable, I feel I have gained a better understanding of the project by visualising it graphically. It is nice to take a look back and see the bigger picture as a whole instead of line by line of code.

I simply hosted and ran the scrapers on my laptop due to needing frequent debugging, although this would put strain on my computer as it would be running for 8-15 hours (Qualifax taking 15 hours, timetable.ul.ie and bookofmodules.ul.ie taking 8). Thanks to the scripts being fairly light weight we could move these scrapers onto a home server or even AWS cloud computing servers. In the interest of money though we cheap out on the environment for scrapers, as they do not require much. I would like to eventually add a database manager like SQL to store the data more securely, behind password and usernames and blocking any un-expected connections.

I would hope to improve the 12 second latency between AWS (Frontend) and Heroku (Backend) by eventually hosting the entire application in one place, ideally a server I have at home for complete cost effectiveness or entirely on AWS if the costs were feasible.

For the current implementation and the fact that the area in which this web-application will most likely be used is the University Campus, it would be fine to host it on a server in my campus accommodation in Limerick, as students requests wouldn't have to travel too far to reach my server.

If I were to deploy this application for multiple universities or institutions, hosting the application on a Content Distribution System like AWS or Akamai would make complete sense for optimal performance across a large area.

5. Problems and Solutions

During this project I faced a few problems. Due to this project being such a large learning curve for myself, having never used iCalendar, FastAPI, JavaScript, AWS or Heroku, I could never outline all the errors in all the code I faced during this project.

I will however outline some clever solutions to problems not introduced to me by my own written code. These are external factors I overcame by pivoting direction or writing supporting scripts.

Call from UL cybersecurity:

One of the first and most notable problems was when I was writing the first timetable.ul.ie spider, it would require UL login credentials such as password and student id. Before deploying; I emailed around in UL to figure out if there was a way in which my script wouldn't handle the login to keep passwords AWAY from me.

If there was a 'Log-in using UL' gateway existing, I could have visitors log in more securely, and use the verification returned by this gateway, to request their personal timetable page. I eventually received an email asking for a call to talk about it.

I was greeted by UL cybersecurity telling me that a log-in gateway does not exist, I will never get access to any data and I should not deploy the application, thus I changed direction to evolve the webapp to the password-free implementation deployed currently today.

The worst thing is that this was before the HSE attack. Strictness over public APIs or data in Ireland has only increased since, further contouring the line between student applications and university frameworks.

Tables in PDF form:

Due to how some older systems work, when we gather data from sources like CAO, it is in a PDF form. Some freedom of information requests quite often do not come in excel like files either, but PDF of table layout to actually deter computational analysis; this was made aware to me by Conor Ryan, an RTE Investigates presenter who hosted the "Universities

Unchallenged" episode wherein they get a lot of freedom of information requests to investigate third level expenditures.

Conor's warnings were laudable when I made some Freedom of Information requests to UL, all being returned in PDF format contrary to my request of being in a "Machine Readable Format such as CSV or Excel" therefor being incompatible with my Data Analysis software; Tableau and not directly malleable in Excel or Python.

But we can read tables from PDFs using a package called "tabula" for Python. Tabula is a wrapper of the original Java implementation, which will extract information from a table in a PDF file. Tabula proved very useful for extracting data from not so data friendly formats.

Information consistency:

After each extraction of data from a source using a Spider, I would investigate the extracted data and compare it with the original source, to make sure all timetables, courses or modules are extracted. Some data sources came naturally with incorrect totals or consistency probably from filtering on one of their own backends or databases.

In CAO points statistic files, the points step range would change year to year, starting at intervals of 5 in the early 2000s showing populations per range like; 205-210, 210-215, but throughout the years the interval changes, maybe from 5 to 15 to 10 to 50, making comparisons between different generations difficult. To overcome this I would take the largest range and aggregate any interim steps for this range across years. An interval of 200-250 in one year may encase 205-210…….245-250 intervals from another year.

Some other errors arose in the CAO data from its source, such as missing population percentages. In the CAO points file a percentage of the entire population is assigned to a points range (i.e. 4% of population scored between 200-250), sometimes the population would not total up to 100%, showing some unexplained missing data. I deem it unexplained as any edge cases such as people getting 0 or 625 were all taken account for, so some years there are 1-3% of the population for that year, missing.

For UL, I found inconsistencies across the timetables and book of modules, which is understandable, it is definitely a tedious task for staff to keep all information consistent for all modules. One inconsistency was the fact that allocated hours on the timetable per module would not equal the amount of hours per week allocated on bookofmodules.ul.ie. I utilised this inconsistency to see where the method of teaching is going for UL.

Another inconsistency was the "pre-requisites" field on book of modules. Some pre-requisite modules did not even exist on UL's bookofmodules, this was discovered when I attempted to perform an analysis to compare course timetables between semester 2 of 2020 and semester 1 of 2021, cautioning me to not rely on bookofmodules for much information.

Without much input from UL themselves, I had to conclude what would be meaningful data and what may be so outdated, it is not useful for comparisons to data supplied nowadays. The identification of this taxing process for staff to keep information consistent across

multiple sites inspired me to conceptualise a information consistency system, which would utilise bots and scrapers like I have here, to extrapolate information from a Ground-Truth source such as a timetable, then from here have a Selenium bot update information across BookofModules, Qualifax, CAO and other arbitrary places. This way staff is not expected to maintain the different sources but students can accurately gauge a courses structure and content.

6. Analytics

In this chapter we will analyse some data gotten from CAO, UL and Qualifax. We will go source to source, picking some Key Points of Information (KPIs) that caught my attention.

My intent is to communicate the rate at which traditional university delivery will be made redundant. One question on my mind, even without all this data and evidence pointing towards a state of blended delivery as the new normal, is what if blended learning isn't adopted and we continue learning as we are now.

In a couple of years, as people begin to get smarter, every course space is being filled and every house in the universities county and connecting counties are lived in, what happens? Students are already commuting exorbitant lengths to get to class, so more physical space on campus is not the smartest decision due to an already evolving traffic congestion problem on campus and the effects this would have on the environment as in a couple of years we will be in the ***heat*** of the Paris Climate Agreement.

So blended learning is inevitable, we will reinforce this point to warn universities in the hopes to avoid a catastrophic Housing, Climate and Third Level Availability crisis in Ireland.

Tableau is an analytical software which I had learned to use in my first job. We will load our CAO files gotten from CAOs press and marketing data tools (18), we will also load in the raw JSON scraper data files of UL and Qualifax courses.

Qualifax:

First lets get some KPIs from Qualifax about the average asking points per year;

Asking Points for CAO courses (qualifax.ie)

Avg. Points.2018	331.065
Avg. Points 2019	326.07
Avg. Points 2020	335.91

Figure 6.1: Average asking points per year for Cao Higher Level Entry courses as according to Qualifax.

First we see that there is no unusual fluctuation in asking points per year per course. This is good news for applicants, who have a verified reliable amount of wiggle room per year, no courses are drastically changing points. This is a good basis for argument that Universities are turning a blind eye to the improved results of candidates as reported by mainstream media in Ireland about predictive results being much better over previous Leaving Certificate years.

The only way for courses to counteract improved results in any year is by raising the points, to which can then be lowered later in the Second round of offers. Lets get a view on who

controls majority of the market per each segment. Institutions with the largest market share will set the standard for that segment of the market, and therefore will need to be the example setters for blended learning.

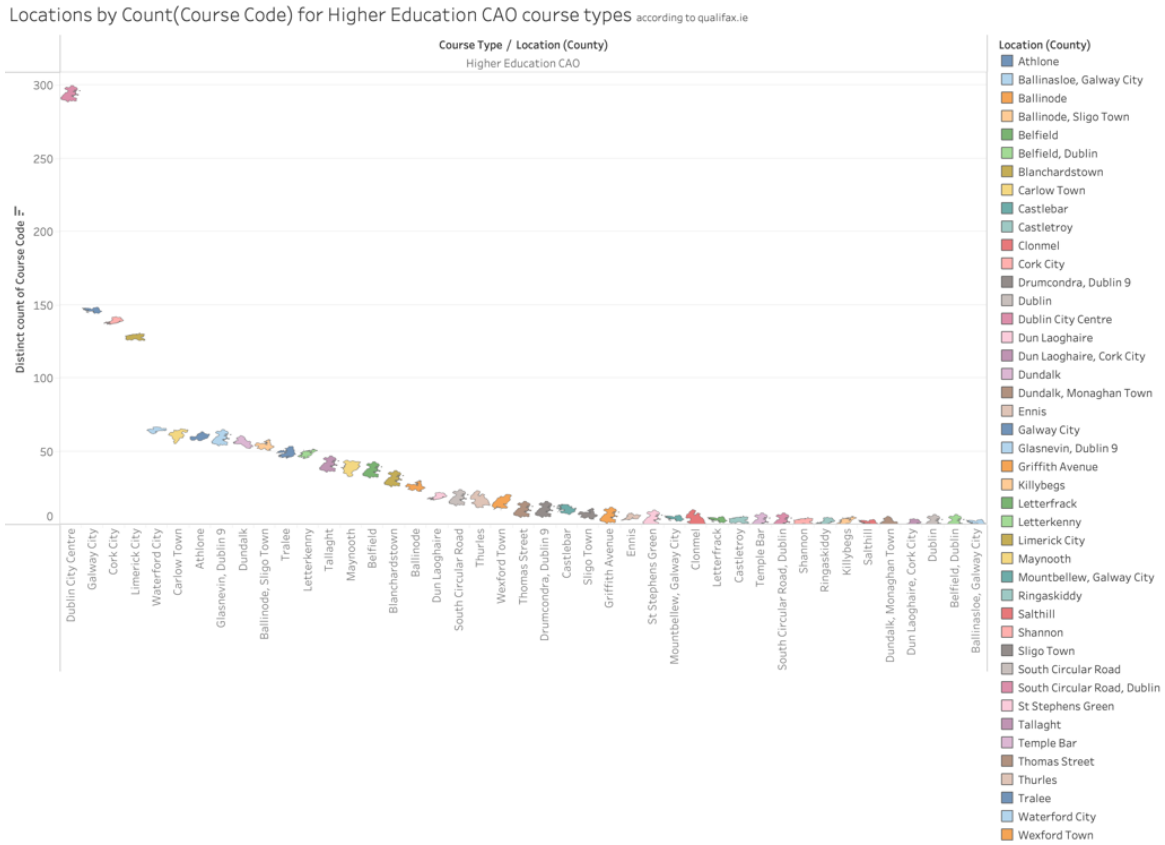


Figure 6.2: Most popular locations for CAO courses. X-axis being the courses location and the Y-axis is the count of courses occurring in that location.

We see Dublin City Centre is by far the most popular location, with almost 300, or 20% of all CAO applicable courses happening there. We can also see Galway City, Cork City and Limerick City, taking up about 10% of CAO courses between them.

This same ranking is held when taking into account all course types, with Dublin City Centre hosting about 14% of all courses on qualifax.ie. As institutions consider a fully blended approach to learning, they will compile some case studies to examine the feasibility. Dublin City Centre will stand for three things in this case study, over-crowding, rent-extortion and traffic congestion, if their County was to go the way of Dublin City Centre.

Dublins university overcrowding and housing crisis, while very profitable for people who may have money in universities and real estate, it is not very affordable for the average student.

Next, the distribution of course results over their providers, showing the top 10 Course Providers for CAO courses (figure 3), and all courses (figure 4).

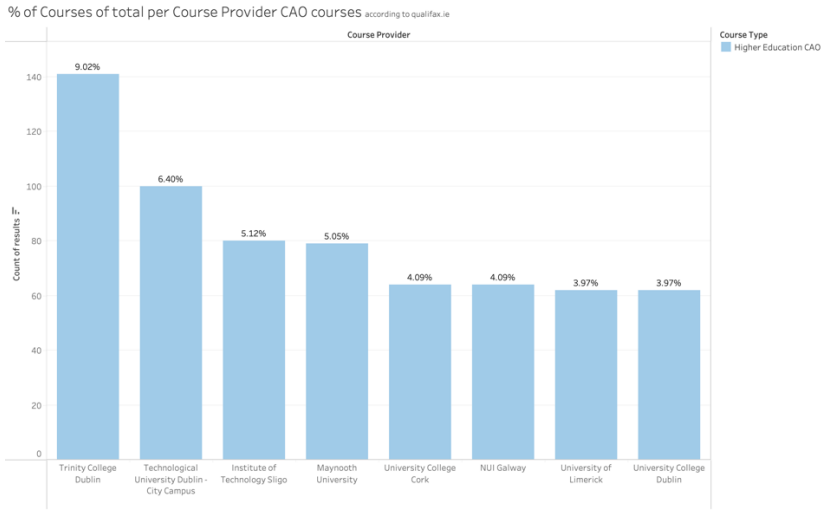


Figure 6.3: % distribution of CAO courses, showing top 10. With Trinity College Dublin hosting 9% of all courses.

Here we see majority of the CAO market is with Trinity College Dublin, offering almost 10% of all CAO courses on offer.

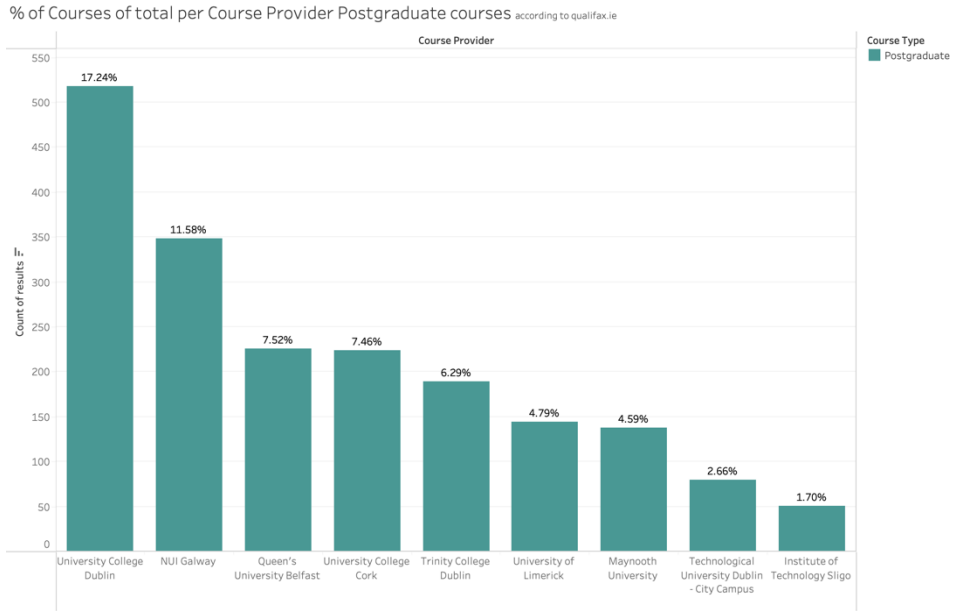


Figure 6.4: % distribution of Postgraduate courses, top 10, with University College Dublin hosting 17% of all available courses.

We see UCD (University College Dublin) holds majority of the market share by quite a lot, making up 17.24% of all postgraduate courses. Already ahead of the competition in terms of courses on offer, the universities at the tail end of figure 6.4 could use blended learning as a method of offering more courses, without actually offering more courses, by simply offering a physical and digital version of each course, they would be able to double their presence in terms of courses offered.

There are a surprising amount of courses from Queen's University Belfast on an Irish course directory, suggesting Northern Irish universities are looking for postgraduates from Ireland.

Blended learning approaches from these Northern Universities would benefit them in successfully enrolling students from Ireland.

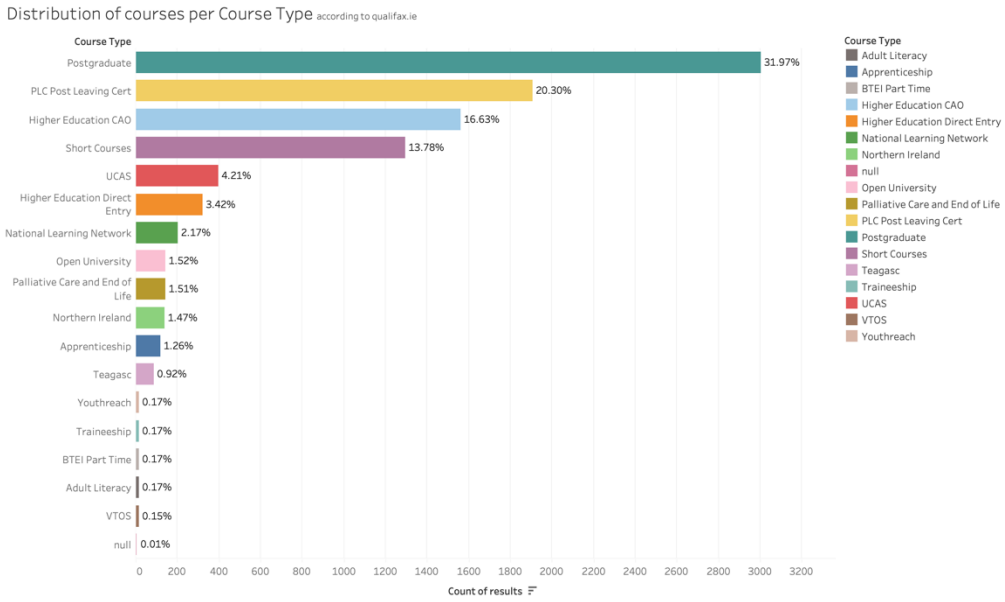


Figure 6.5: Courses by Course Type

Here we see Postgraduate courses dominate the market by its 32% share of 3004 courses. Currently UCAS, the course type for British and some Northern Irish Universities, has a 4.21% market share of Qualifax courses, it will be interesting to see how foreign Universities like these penetrate Qualifax, as they become more digitised and therefor more accessible, people from Ireland might opt for online learning from foreign universities, especially in hard-to-enter courses like Veterinary or Denistry.

Here we investigate the number of courses which mention placement. Placement is a competitivity contest between universities. By offering placements with renowned employers courses could attract more students.

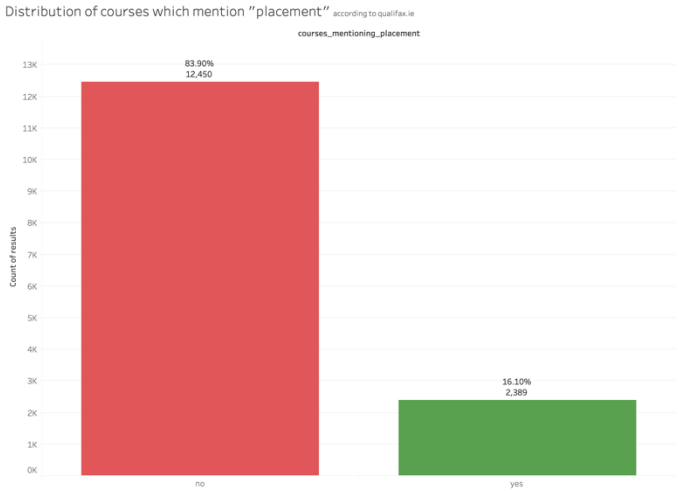


Figure 6.6: Distribution of ALL courses which mention "Placement".

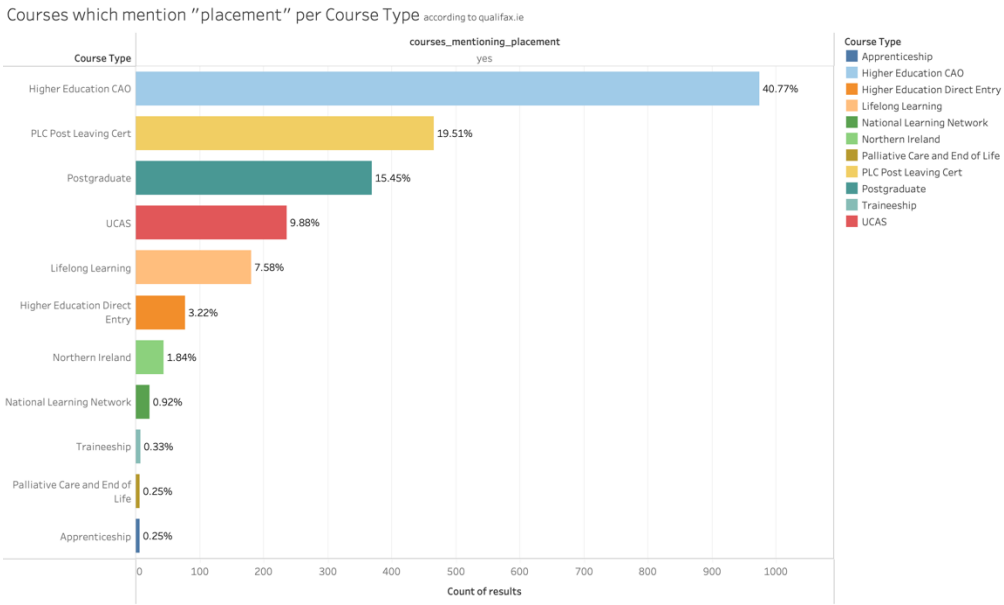


Figure 6.7: Courses that mention "placement" by Course Type. 40% of all CAO courses offer placement.

In figure 6.6, I created a Tableau equation to count the fields "Subjects Taught" or "Course Content" which mention the word "placement". Placement is something all CAO Higher Education courses strive to have, there are 1563 Higher Education CAO courses, and 62% of these mention "placement". It will be interesting to see how the others grow.

Considering PLCs are usually only one year, for people looking for an introduction to work in a field they like, PLCs are a good choice although only 12% of all PLCs mention "placement" of the 3004 present on Qualifax, meaning there is still room for more PLCs to offer placement.

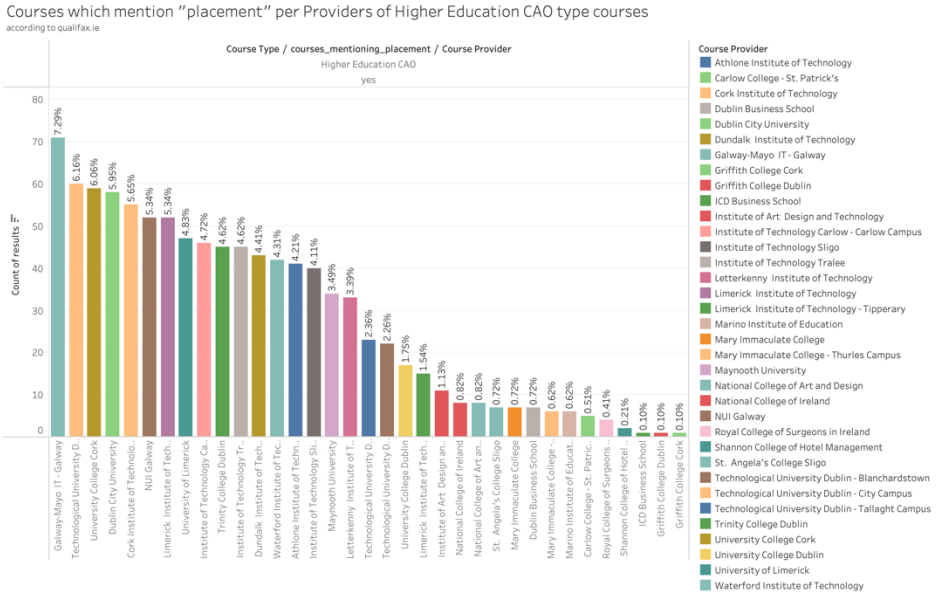


Figure 6.8: Courses that mention "placement" by Course Provider for CAO course types.

Here we see the provider who offers the most CAO courses mentioning "placement" is Galway-Mayo IT – Galway, who offers 7.29% of the 974 CAO courses which mention "placement".

Using Qualifax we can get some very reliable statistics about Third Level Education in Ireland, specifically the evolution of course points, and offerings. Performing an analysis like this each year can track courses and institutions evolutions. What new courses institutions are offering, or who now offers placement etc. are all catalyst factors to getting more students to go to college, further leading to inflation and over-allocation of campus and community resources.

CAO:

Next we will look at some statistics from cao.ie. CAO is the Central Applications Office, every applicant of a Higher Level CAO course sends their points and preferences to this source. This will give us an overview of the influx third level will see each year.

Qualifax will give us the market, what is on offer, who offers what and when they offer it. Using CAO we can get an idea of the demand, or available consumer base to Qualifax. We can use CAO applicant data to identify Key Points of Information to further confirm or contest the suspicions made that on-campus learning is not sustainable for much longer.

We will analyse CAO points, checking the trends of population who scored above the average asking points according to Qualifax in figure 6.1. A good gauge to over-population or over-enrolment of courses would be comparing Qualifax asking points with CAO applicant points; if quality of applicant points surpasses the quality of asking points, we can expect more people to enter courses around Ireland.

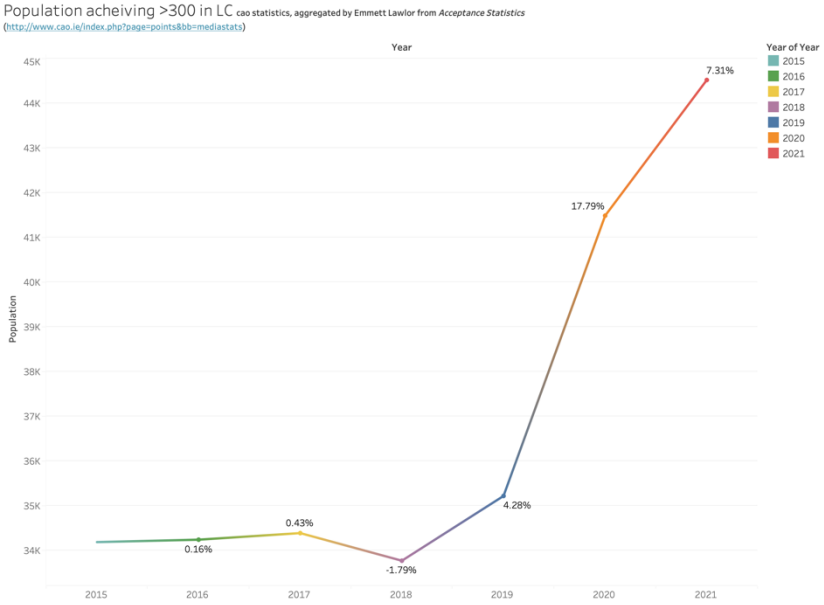


Figure 6.9: Population receiving >300 in LC. >300 was chosen due to the average in Qualifax being 315-335 in the last 3 years.

Here we see a graph which shows year to year the difference in population who achieved above 300 points. We see there has definitely been an influx of leaving certificate points above the Qualifax average, jumping a total of almost 25% in two years since 2019.

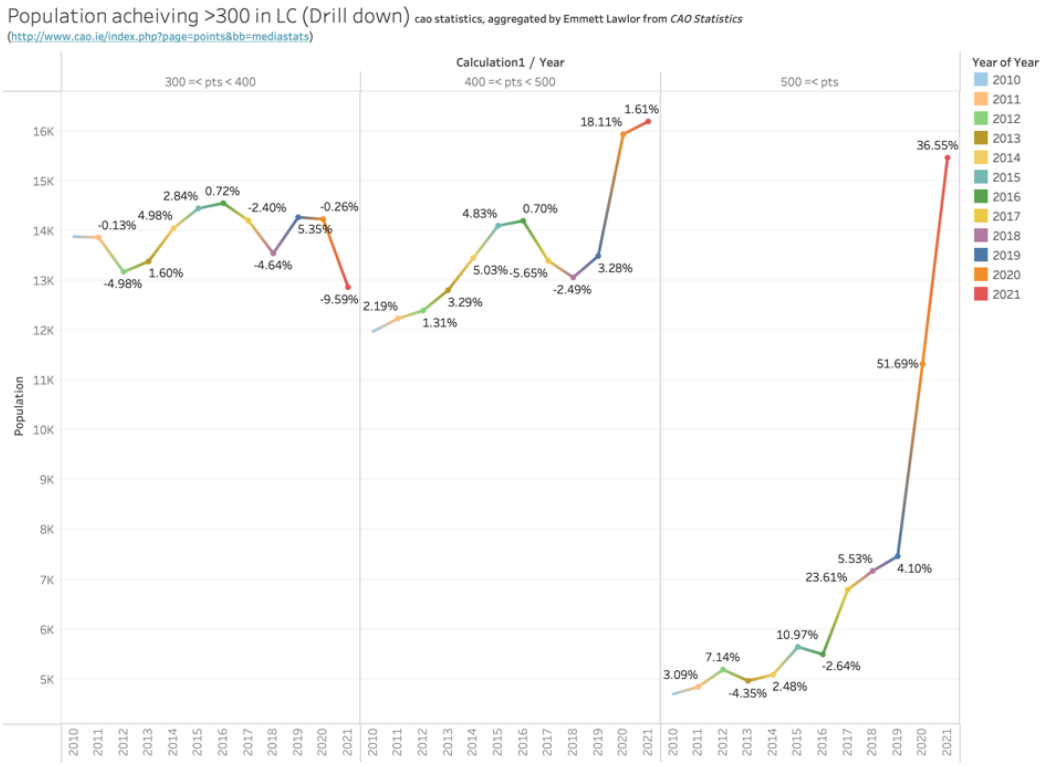


Figure 6.10: Population receiving >300 in LC intervals of 100.

With the number of people receiving specifically equal to or more-than 500 points jumping almost 88% since 2019 in figure 8, it is hard to determine definitely if these are repeats wanting to enter third level, overall better quality grades due to predictive grading, a combination of both or just natural growth. More repeats and natural growth being the scare factors towards over-crowding.

Temporary inflation due to predictive grading would be ideal due to it being temporary, but it is unpredictable for next year, do courses lower/increase their asking points, we are in a very un-predictable stage, too-low and you will have mass rejections to perfectly liable candidates, too-high and not enough students will enter third level, causing a huge round of second offers and even more overall un-predictability.

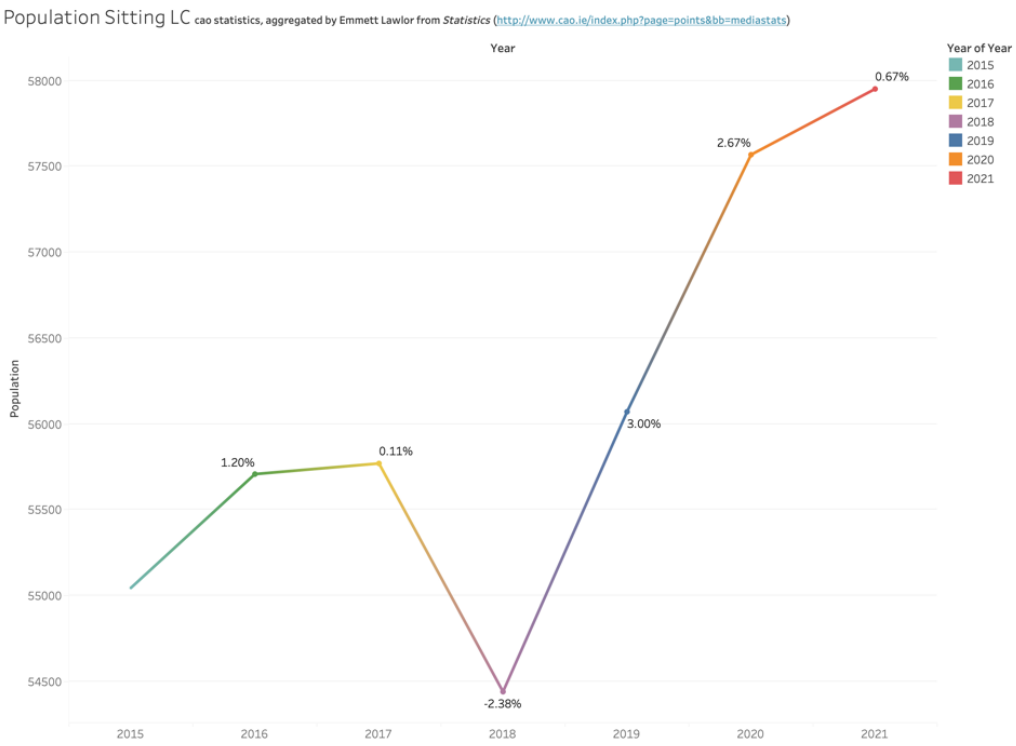


Figure 6.11: Population sitting LC last 6 years.

Overall the growth seems consistent for 2020 and 2021, showing something is definitely attracting more people to third level education.

Next we will investigate Acceptance data from CAO. This breaks down the number of Acceptances made in 2020 and 2019 (2021 was not available at the time of this report), for distinctly level 8 and level 6/7 courses. We are also given the number of First Preferences and Total Acceptances. Using this data we can assume courses are filling all possible places, we can check how many applicants accepted an offer to a "Course Classification" field, not individual courses.

First lets check the distribution of first preferences and total acceptances over 2020 and 2019 by level.

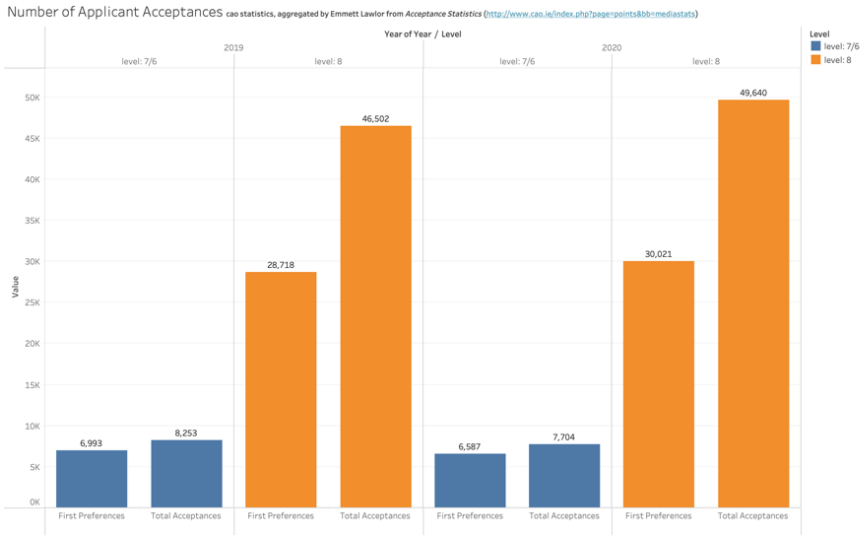


Figure 6.12: First Preference and Total Acceptances by Level by Year. Each year is split into Level 6/7 and Level 8, from here we display the population belonging to each field.

We see a small shift from Level 6/7s to Level 8 in 2020, with about 60% of total level 8, and 85% of level 6/7 applicants receiving their first preferences each year. Comparing these figures to figure 8, the number of people who received points surpasses the number of acceptances in 2020 by 225 and in 2019 by 1,316, showing a definite increase in hopeful applicants or an increase in University space.

Analysing CAO data, we can view what demand is being supplied to students. It's interesting to see more people receiving their first preferences and higher points, it seems as though students are not challenged with the scarcity of course-placement that was once present in Leaving Certificate.

University of Limerick:

In 2019 UL surpassed 3000 CAO offers for the first time ever (19). Likely taking advantage of the dip in people sitting the leaving cert according to figure 6.11 or new campus resources.

I made multiple freedom of information requests to UL, one regarding student contribution;

	AY 16/17	AY 17/18	AY 18/19	AY 19/20	AY 20/21
Student contribution	14,607,603	15,657,541	16,682,676	18,223,012	20,008,665
Alumni	4174	4474	4767	5207	5717
Growth		7.19%	6.55%	9.23%	9.79%

Figure 6.13: Student Contribution to UL.

Alumni and ***Growth*** are fields I calculated based on the Student Contribution row supplied by UL freedom of information. UL was able to increase their growth an additional 2% in AY 19/20 when they gave out CAO offers before the pandemic and keep it consistent into 20/21, showing blended learning has not really slowed people down in going to third level.

Comparing 2020 timetable data to book of modules, shows how much lecturers have changed their teaching style, using a Python notebook I was able to derive that of the 864 modules timetabled;

- 8.5% have longer timetabled Lectures.
- 7.5% have longer timetabled Laboratories.
- 28.1% have longer timetabled Tutorials.

Being book of modules it probably has not been updated for years, showing this has been a more natural change and not the result of the pandemic. It shows lecturers and students adapting to a more self-guided, hands-off approach to learning and teaching, something that works nicely with remote or blended learning, showing the way courses are going, not much will be needed to take them fully online anyways.

7. Conclusions and future work

In a world where people are getting smarter with unlimited access to the internet for information, it shows strain on the current second to third level framework, as more and more people apply and meet requirements to third level, there is only so much we can raise our points to. Once the points to every course is so high and places in the course continuously fill and all the houses in the county are lived in, institutions will have no other choice but to expand into online and blended learning.

For institutions to expand into online and blended learning, they will need to think of creative ways to meet the resources needs.
Such as;

- Having perfect synchronicity between online and physical spaces.
- Be able to process and handle the excess amount of paperwork like administration or assignments.
- Providing mass amounts of students with relevant tools or resources to complete school work, get answers to questions.
- Providing lecturers with relevant tools or resources to assist with the increase in number of alumni.

Due to university IT departments having an abundance of responsibilities, they are often slow to develop student-centric applications; by allowing students to develop and deploy applications, more niche apps can be built and given more attention by students solving a specific problem, rather than an IT department identifying, developing, deploying and maintaining them. This can assist in filtering resources and applications wanted by students, IT departments can then acknowledge and officially endorse popular or trending applications.

With support from universities, students can build more integrated apps, and are not limited by data available online. Something like an authentication gateway provided by the university could be integrated to unical.ie to give direct access to the students personal timetable, we could also go further to invoke assignment tracking and alerts.

I would like to expand my project by a lot; scrapers to handle information detecting out of date information and updating it, so applicants have up-to-date information based from one source of truth, e.g. a timetable, update all the timetabled hours in other sources, base course description on the syllabuses from the timetabled modules for the course, and update across qualifax, cao, etc.

With correct information stored and course descriptions accurate, Natural Language Processing of these course documents would be much easier and way more useful in application as the model will be trained on up-to-date correct data. The recommendation model I wanted to build would've been quite useless for the purpose of deterring drop-outs if all information on Qualifax was out of date.

Once information is consistent and up to date, I would like to begin implementing Ai generated student and lecture content. Using descriptions from the course description

fields, I could scrape the internet for related content, and use Natural Language Processing and Ai to convert things like articles and videos into a learning format like a PowerPoint presentation.

We could also train an AI on the information which it scraped from the internet based on relevant topics from the course description, to create a chatbot which could relieve lecturers of the emails they receive of questions from students. Since the bot knows the course syllabus and content, it can be more than just a glorified google search, but you can question it like “how much is my FYP worth?” and it would know.

The final thing I would like to complete is the recommendation model, which could convert the keyworded search process on Qualifax, to a “similar courses” or “based on your ideal description of a course here are …”, something I believe could help a lot of people in choosing the correct course, deterring drop-outs and providing a better experience in third level.

Overall I thoroughly enjoyed developing and believe this project was beneficial to my skill progression and I hope to have uncovered and brought attention to the inflation and impending destabilisation of Second to Third Level processes using data from Qualifax and CAO.

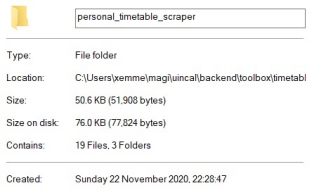
I also hope I have demonstrated any Students capability of building an application for the sole purpose of serving the University alumni. By universities promoting and endorsing these kind of applications, it could benefit the students career opportunities and also provide the university alumni and staff with a massive amount of free resources and tools without the IT department having to be involved at all, or even aware that an application like this was actually in demand by students.

The two key take away from this report can help universities prepare for the coming years and help influence a decision in offering fully online or blended courses. Physical learning cannot survive, and any University which believes otherwise may face stagnation in enrolments due to simply not having the space to house more students. Whereas digital universities can expand across the globe and back again, but the digital infrastructure to support this will be needed, and can be developed hand-in-hand with students for extra merit.

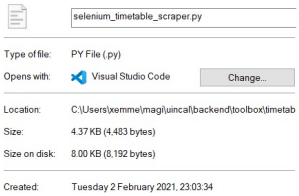
8. References

1. **Pandas Documentation. [Online] https://pandas.pydata.org/.**
2. Scrapy Documentation. [Online] https://scrapy.org/.
3. Selenium Documentation. [Online] https://www.selenium.dev/.
4. iCalendar PyPI Project Page. [Online] https://pypi.org/project/icalendar/.
5. FastAPI. [Online] https://fastapi.tiangolo.com/.
6. Price Comparison with Web Scraping. *datamam.com.* [Online] https://datamam.com/web-scraping-for-price-comparison.
7. What is google spider? *Look to the Right.* [Online] https://www.looktotheright.com/blog/what-is-the-google-spider/.
8. Consumers warned of bots hoarding PS5s and other must-have items. *The Guardian.* [Online] https://www.theguardian.com/money/2021/jan/22/scalper-bots-uk-xbox-series-x-playstation-5.
9. Is web scraping legal? Short guide on scraping under EU jurisdiction. *Discover Digital Law.* [Online] https://discoverdigitallaw.com/is-web-scraping-legal-short-guide-on-scraping-under-the-eu-jurisdiction/#2_Check_if_you_fall_under_the_Text_and_Data_Mining_TDM_exception.
10. Scrapy. *Wikipedia.* [Online] https://en.wikipedia.org/wiki/Scrapy.
11. Launch the Shell. *Scrapy Documentation.* [Online] https://docs.scrapy.org/en/latest/topics/shell.html#launch-the-shell.
12. Scrapy.Selector Documentation. [Online] https://docs.scrapy.org/en/latest/topics/selectors.html.
13. Xpaths in Chrome. [Online] https://www.scrapestorm.com/tutorial/how-to-find-xpath-in-chrome/.
14. Selenium Wikipedia. [Online] https://en.wikipedia.org/wiki/Selenium_(software).
15. JS HTML | DOM Documentation. *W3 Schools.* [Online] https://www.w3schools.com/js/js_htmldom_document.asp.
16. Stop-words Explained. *Opinosis Analytics.* [Online] https://www.opinosis-analytics.com/knowledge-base/stop-words-explained/.
17. Part of Speech English Grammar. *Thought Co.* [Online] https://www.thoughtco.com/part-of-speech-english-grammar-1691590.
18. Media and Statistics. *Central Applications Office.* [Online] https://www.cao.ie/index.php?page=mediastats.
19. UL CAO offers exceed 3000; first time in history. *University of Limerick.* [Online] https://www.ul.ie/news-centre/news/university-limerick-cao-offers-exceed-3000-first-time-its-history.
20. The rising conern around data and privacy. *Forbes.* [Online] https://www.forbes.com/sites/forbestechcouncil/2020/12/14/the-rising-concern-around-consumer-data-and-privacy/.

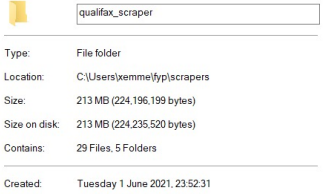
9. Appendix A: Key Work Dates



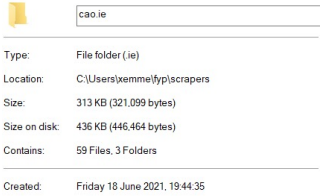
Started Personal Timetable Scraper



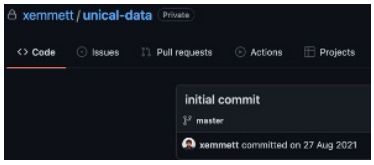
Started Course Timetable Scraper



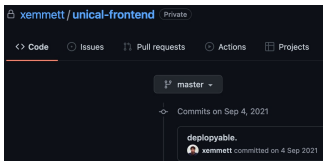
Started Qualifax Scraper



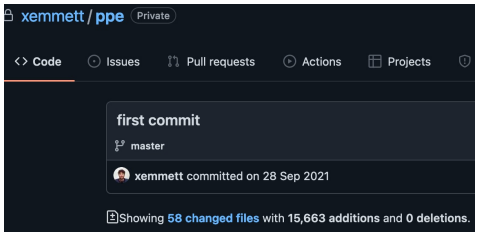
Started CAO investigation



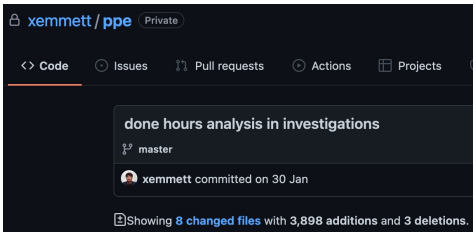
Deployed Webapp API



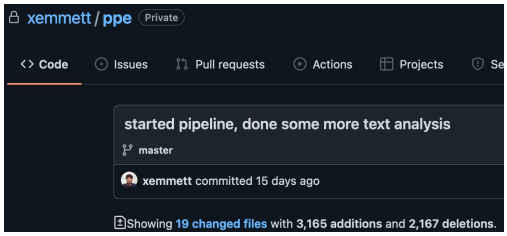
Deployed Webapp Front-end



ppe ~ fyp
Post Pandemic Education /
Personal Protective Equipment

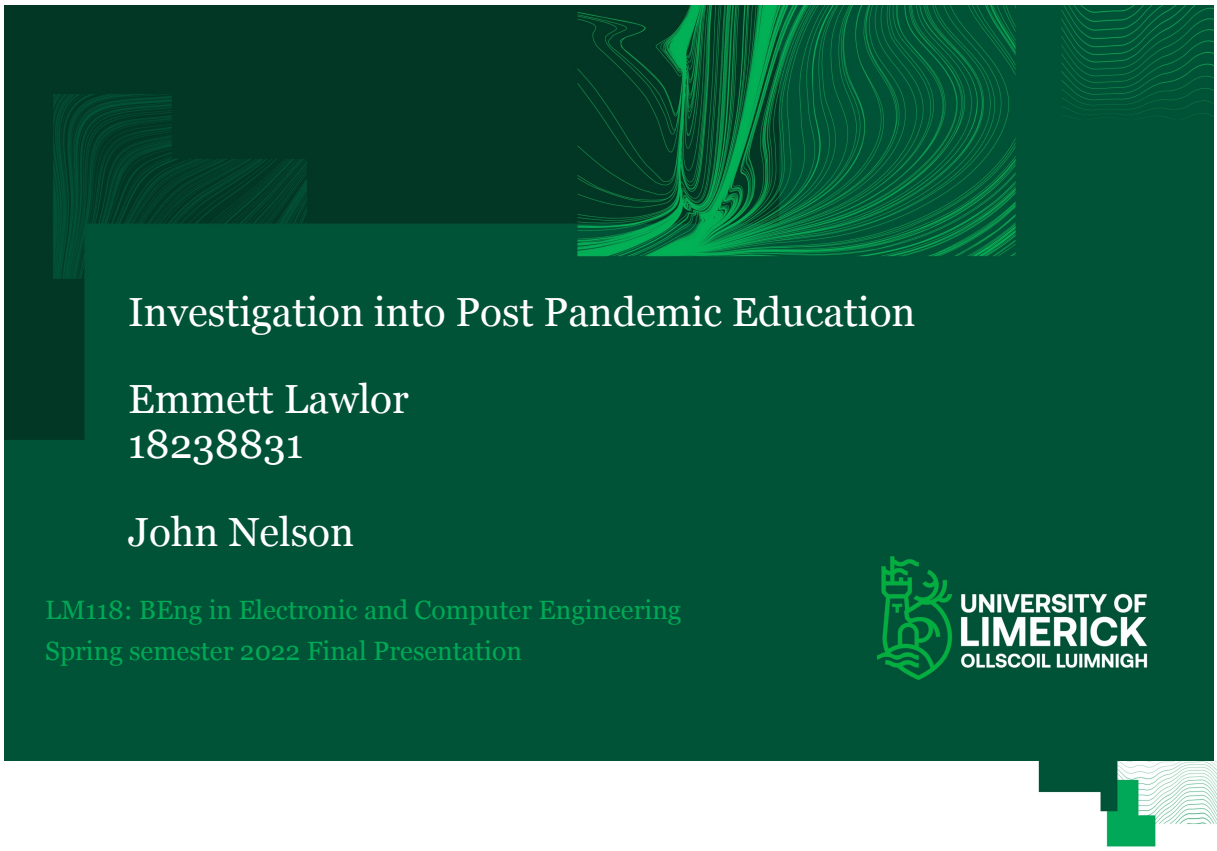


UL Timetable vs Bookofmodules
Hours analysis



Axed keyword analysis

10. Appendix B: Final presentation slides



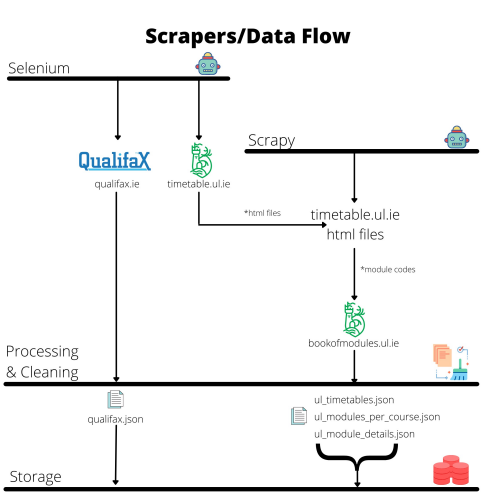
Presentation overview

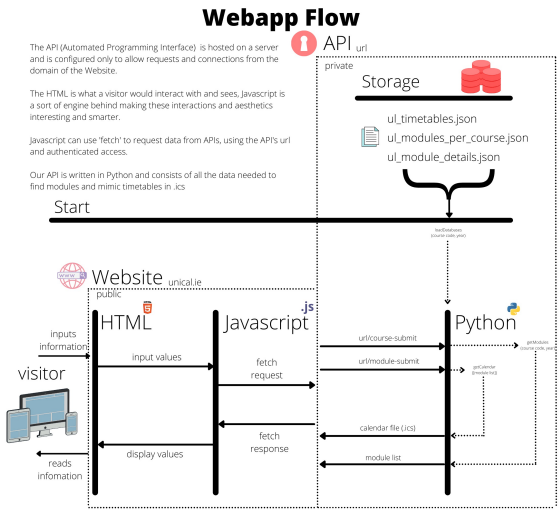
- Overview
- Key Work Dates
- Key results
- Problems and Solutions
- Potential uses of the system and future work
- Demonstration
- Conclusions

Project overview (1)

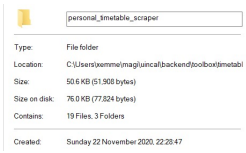
- Fully realised method of extracting course data from Qualifax and UL.
  - Using frameworks for Python, I created 2 types of scrapers, which read HTML from webpages and save it in a JSON format for easy indexing and use across internet.
- Fully realised Webapp used almost 600 times over 2 semesters.
  - Using timetable data I could recreate events into a Universal Calendar Format.
- Fully realised an analytical report investigating points data nationally and module data in UL.
  - Using Qualifax, UL and CAO data, I investigated some trends we saw during the pandemic.

Project overview (2)

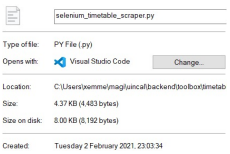




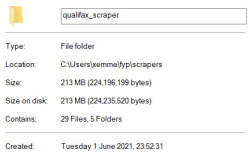
Key work dates



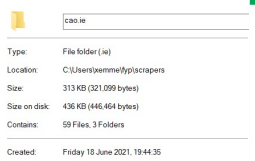
Started Personal Timetable Scraper



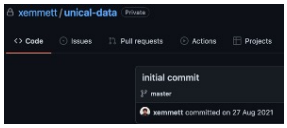
Started Course Timetable Scraper



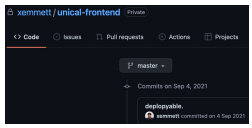
Started Qualifax Scraper



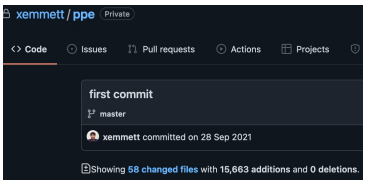
Started CAO investigation



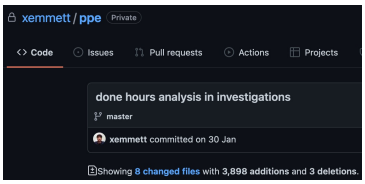
Deployed Webapp API



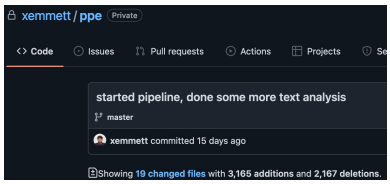
Deployed Webapp Front-end



ppe ~ fyp
Post Pandemic Education /
Personal Protective Equipment



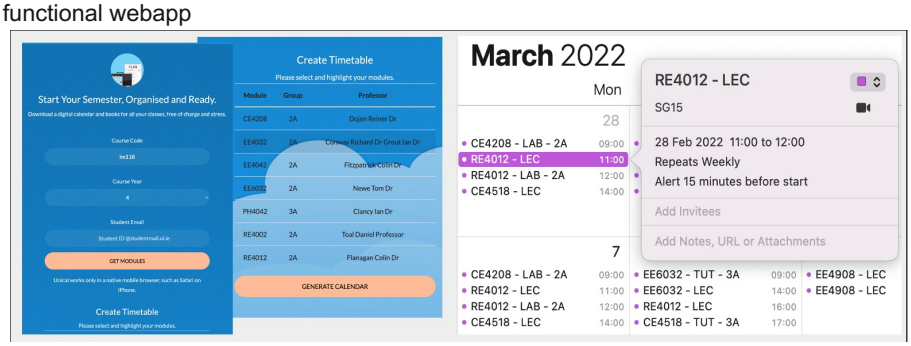
UL Timetable vs Bookofmodules
Hours analysis

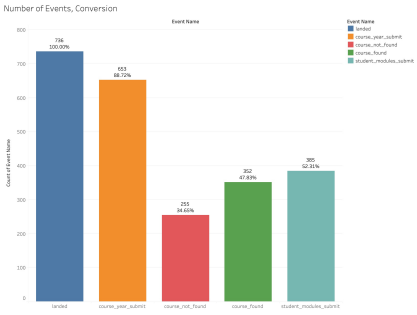


Axed keyword analysis

Key results

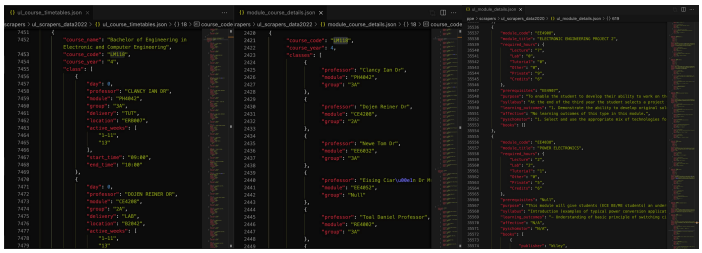
Fully functional webapp





Key results

Extraction of UL data



Extraction of Qualifax data

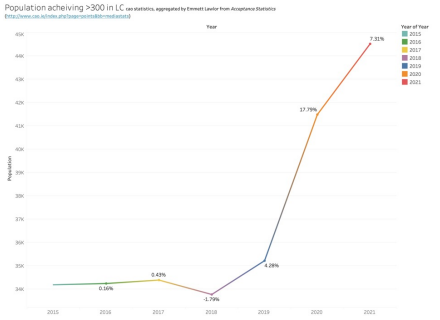

```
{
    "course_name": "Engineering - Electronic and Computer Engineering",
    "course_provider": "University of Limerick",
    "course_code": "LM118",
    "course_type": "Higher Education CAO",
    "qualifications": {
        "award_name": "Degree - Honours Bachelor (Level 8 NFQ)\n More info...",
        "nfq_classification": "Major",
        "awarding_body": "University of Limerick",
        "nfq_level": "Level 8 NFQ"
    },
    "apply_to": "CAO",
    "attendance_options": "Full time, Daytime",
    "location_(districts)": "Limerick City",
    "qualification_letters": "BEng (Hons)",
    "duration": "4 years",
    "specific_subjects_or_course_requirements": "Min requirements: 2 H5 and 4 O6/H7\n\nEnglish:
    "leaving_certificate_general_entry_requirements": "Irish Leaving Certificate Applicants \n\n
    "leaving_certificate_vocational_programme__lcvp": "For the purpose of satisfying minimum ent
    "qqi_fet_applicants_general_information": "QQI Pathways",
    "qqi_fet_entry_requirements": "http://www.cao.ie/index.php?page=fetac_search",
    "mature_applicants": "Applications are particularly welcome from mature candidates (at least
```

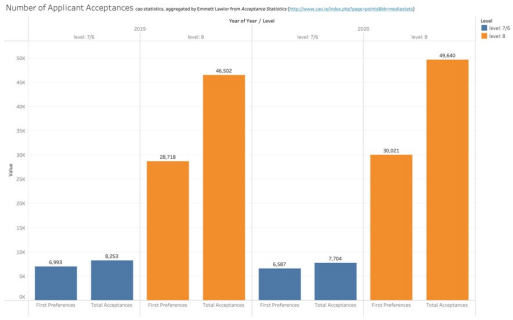

Key results

Possibility for inflation in Third Level

Asking Points for CAO courses (qualifax.ie)

Avg. Points.2018	331.065
Avg. Points 2019	326.07
Avg. Points 2020	335.91





Problems and solutions

- Sources from CAO being in PDF tabular form.
  - Tabula framework for Python.
- Inconsistent Data across sources.
  - UL, CAO both had conflicting data.
- Textual Analysis.
  - More questions than answers.
  - Information format and consistency.
- UL CyberSecurity.
  - Removed password entry to webapp

Potential uses of the system and future work

- Course competitivity.
  - Utilising scrapers and analysing external sources like Qualifax, institutions can find a competitive edge over other courses or districts, what to offer etc.
- A new hub for integrated tools.
  - Allowing students to create web applications such as the one in this project, under a institution framework with special accesses such as log-in using Institution portal. Institutions could have a flood of resources available, built by students for students or lecturers.
- Deter drop-outs, increase quality of student life.
  - By creating custom tools wrapped around data like Qualifax (course description data), we could build tools to help hopeful students find the right course.
  - By streamlining student tool creation, students can have newer ways of keeping their digital and physical academic life in check, in ways unknown to corporate software creators.
  - Helping students find the ideal course and structure around it, we can make a very stressful period in life easier.

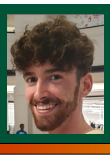
Demonstration

- Demonstration of Web Application.
  - Will demonstrate flow
  - How various aspects were completed.
  - Evolution.

Conclusions

- Quality of code progression.
- Third levels beware of strain on institution and community resources.
- A need for organisation and formatting of documents.

11. Appendix C: Project poster



Investigation into omnichannel experience of Third Level education in a Post Pandemic Era

E&CE | Department of Electronic and Computer Engineering

Emmett Lawlor

General Electronic and Computer Eng.

Introduction

As blended learning allows institutions to accommodate more people and give more offers, a need to digitize and automate the education sector becomes more necessary for alumni and staff.

This project focuses on the information students need before and during the attendance of their third level. It also provides a foundation for databases and webapps that could potentially assist both, students and professors, throughout their academic lives.

To do this, we used Spiders to extract and gather large amounts of data from websites and existing databases. The textual part of the data was then parsed for information retrieval and categorisation for potential ML/AI modeling.

Aim

1. Collect data relevant to students applying to third level (A) and students already in third level (B).
2. Collect and parse Qualifax data to help machines understand it and to build a cleaner directory and a better experience for finding courses. (A)
3. Collect and parse UL data, for creating web applications for UL students.(B)
4. Create a web-application for UL students to help in streamlining the beginning of the semester. (B)
5. Investigate the data retrieved and detail points of interest and trends that currently exist in third level education in Ireland.

Method

Using frameworks available for Python (a programming language), we extracted data from sources like timetable.ul.ie, bookofmodules.ul.ie, and qualifax.ie. We then deployed crawlers onto websites to extract more information and save it in a usable format.

With 300 timetables and 800 modules extracted from UL sources, we facilitated the webapp, which uses information submitted by students to create a calendar that displays the timetable of the modules they're enrolled in. It can then be downloaded to their devices.

Figure 1: Web app flow on MacOs

The entire stack was built from HTML, JavaScript and Python for free and is explained in this report so students can create APIs of their own.

With textual data on 15k courses extracted from qualifax.ie (course content, subjects, etc.), we created a method to analyse the text and derive important words and meaningful phrases.

By calculating the importance of each word and its rarity among the data itself and similar documents, we can make educational documents easier for computers to interpret, by preparing a training set for its application in fields like artificial intelligence and semantics analysis.

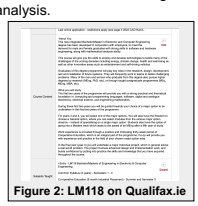


Figure 2: LM118 on Qualifax.ie

Bachelor/Master in Electronic and Computer Engineering degree has been developed in conjunction with employers, to meet the demand...

The analytical report is based on Data from CAO, UL and Qualifax. We use it to investigate how the Pandemic affected education, such as through UL module delivery changes, number of courses listed on Qualifax, and trends of CAO points, offers and acceptances.

Results

1. Two kinds of Spiders were created to overcome challenges arising from how the spiders interact with webpages. One automated a Chrome browser to crawl dynamic websites and the other simply requested and read HTML documents. 15k Qualifax courses and 800 UL modules were extracted.

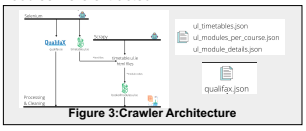


Figure 3:Crawler Architecture

2. In SEM1/21, 43% of the 600 visitors to the webapp submitted their course details, showing that students are open to utilising apps built by other students. This research can encourage students to create tools and institutions to provide resources/data to support creators.

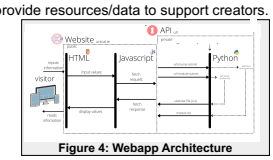


Figure 4: Webapp Architecture

Students creating apps for fellow students would relieve institutions of the competitive development of useful tools. Institutions should embrace this and allow students to utilise their APIs to create better, more integrated applications.

3. Textual analysis reveals the most popular keywords, their frequencies, rarity and importance in documents.

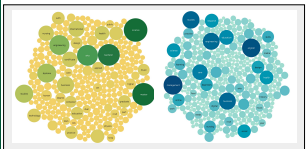


Figure 5: Most prominent words among UL (left) and Qualifax (right) course names.

4. In 2020, 28.1% of modules in UL increased tutorial hours. In CAO, there was a 6.7% increase in acceptances, but only a 3% increase in population. There was no real change in required points for courses on Qualifax. As more people qualify for third level, more institutions will need to rely on self & blended learning, especially with housing and campus space becoming more limited.

Figure 6: Asking requirements remain static while increases in CAO Applicants and Quality of Results.

Conclusion and personal reflection

I explored applications which could assist institutions to adapt in the future;

1. Spiders to find and aggregate relevant information for current and potential students.
2. A template for people to build APIs for students providing useful resources for their institutions alumni and staff.
3. Creating textual training sets for AI and Machine Learning modelling.

I found I have improved my technical skills and have overcome my first ever personal project to be used publicly. I demonstrated creativity, by overcoming problems and seizing opportunities

I hope to continue building a suite of useful tools and resources for current and aspiring students and staff.

Acknowledgements

FYP Supervisor: John Nelson

Frameworks: Selenium, Scrapy, FastAPI, Pandas

Code sites; geeksforgeeks, stackoverflow

Sources; UL, CAO, Qualifax

UNIVERSITY OF LIMERICK OLLSCOIL LUIMNIGH