Bachelor of Software Engineering

Diploma in Software Development

CS202A/B

Python & Databases

This report belongs to:

Chris (Hui Hui) Chong
210758235@yoobeestudent.ac.nz

Background

We have learnt that Python is a popular language used by statisticians, engineers, and data scientists to perform data analytics. It has a vast collection of libraries capabilities for numerical computation and data manipulation, and for graphics and data visualization.

Prior to this class, I have always wanted to learn about "web scraping", and some research showed that Python is the web scraper language of choice. It has BeautifulSoup, a Python framework that helps to handle the web scraping process smoothly, and combines with html5lib and lxml to parse scraped data.

I personally also have an interest in property/housing. Given the current volatile housing market, I thought this might be an interesting time to see if I could identify any property trends taking place.

Objectives

Based on the above, I consulted with my tutor and chose to create an application that has the potential to showcase the following skills:

1. Leveraging on Python's web scraping libraries and frameworks to obtain raw housing property data from a New Zealand property website,
2. Connecting the data from a database,
3. Be able to sort and analyse the data, with some visual elements.

Functional Requirements

The functional requirements have been kept to a minimum in order to keep with the required timelines:

FR#	**Functional Requirements**	**Want/Must**	**Achieved?**
FR001	Users can create their own account (with basic validation checks)	Must	Yes
FR002	Users can log into their account	Must	Yes
FR003	Users can edit their account information	Want	Yes
FR004	Users can scrape data from a specific region	Must	Yes
FR005	Users can perform basic analysis of the data scraped and view it in a simple, visual, manner	Must	Yes

Product

In the previous iteration of this project, I needed to use an external application, ScrapingBee, to retrieve data. This time, I managed to create a pure-Selenium application, with slightly more advanced/stable filtering to do the scraping.

```
current_dir = os.getcwd()
print("the current_dir is " + str(current_dir))
# # Define a relative path and filename of my HTML files (used to store temp data to reduce need to scrape)
beescraps = os.path.join(current_dir, 'proptrends\scraper\output', 'rawbeescraps.html')
print("Beescraps are: " + str(beescraps))

# Configure Chrome options
service = Service(executable_path="C:\Program Files (x86)\Google\Chrome\Application")
options = webdriver.ChromeOptions()
options.add_argument('--ignore-certificate-errors')
driver = webdriver.Chrome(service=service, options=options)

# Load the page
driver.get(source)
```

Because I am now using Selenium, I can now retrieve all the data from the webpage, including the advertised properties. This means that the number of listings in a page was no longer a fair assessment to determine when to move to the next page, or if I should stay on that page. This meant an overhaul of the scraping logic.

The scraping process is quite slow to run and quite compute intensive. In order not to flood the website and escape RealEstate.co.nz's bot filters, I decided to introduce long (human-like) wait times, stretching from 13 seconds to 130 seconds.

Logic Flow

The general scraping process is as follows:

1. There are 3 flags:
    a. scraped_listings_on_page
    b. new_scrapes
    c. total_page_listings
2. The scraping components have been broken down into suburbs. This makes it easier for the scraper to be re-run since it would not have to start from scratch.
3. The scraper would scrape the **main page** of a defined search (for example, Berhampore suburb in the Wellington City region).
4. It uses an initial randomised 5 – 10 second check to see if the page has listings.
   If it has no listings, it aborts:

```
# Quick first check for sanity
delay = uniform(5.03, 10.26)
sleep(delay)
check_page = BeautifulSoup(driver.page_source, "html.parser")
check_valid = check_page.find('h3', class_='text-xl.25 font-semibold')
if check_valid:
    check_valid = check_valid.text.strip()
    if check_valid == "Nothing to see here":
        print(
            "Check_valid is returning: " + check_valid + ". This means the search result is empty. Abort and move to next suburb.")
        return "next_suburb"
```

5. If it is valid, then wait for a random time between 13 to 130 seconds to give the site's elements ample time to load and simulate human wait times.
6. The raw data that I retrieve is massive. To mitigate this, I ran the data through a few python-enabled methods to filter out the unnecessary content. This reduced data is saved into html files.

This is useful for testing and troubleshooting, reducing the need to re-scrape the site

```
# Same for <noscript> (Facebook) tag and its contents
noscripts = page.find_all("noscript")
for script_tag in noscripts:
    script_tag.decompose()

# Finally remove <iframes>
iframes = page.find_all("iframe")
for frame_tag in iframes:
    frame_tag.decompose()

print(" Rubbish stuff should now be extracted. Let's save this into rawbeescraps...")

# writing over beescraps with the reduced page (no rubbish)
soup_string = page.prettify()
with open(beescraps, 'w', encoding="utf-8") as file:
    file.write(str(soup_string))
file.close()
# Close the browser
driver.quit()

# Now open the temp html file created
with open(beescraps, "r", encoding="utf-8") as f:
    page = BeautifulSoup(f, "html.parser")
```

7. Since the number of advertisements and listings vary, the scraper first does a quick run-through of all the listings to count how many there are on the page and stores it in the total_page_listings parameter

8. The scraper then checks the listings' URL (unique) with the database. If it exists, it moves to the next listing while increasing the scraped_listings_on_page flag.

9. If it does not exist, it increases the scraped_listings_on_page and new_scrapes count and starts extracting the relevant raw data correlating to the parameters I want to capture. Regex is then used to clean it.
   Note: The heavy lifting of the extraction and filtering work is done thanks to BeautifulSoup!

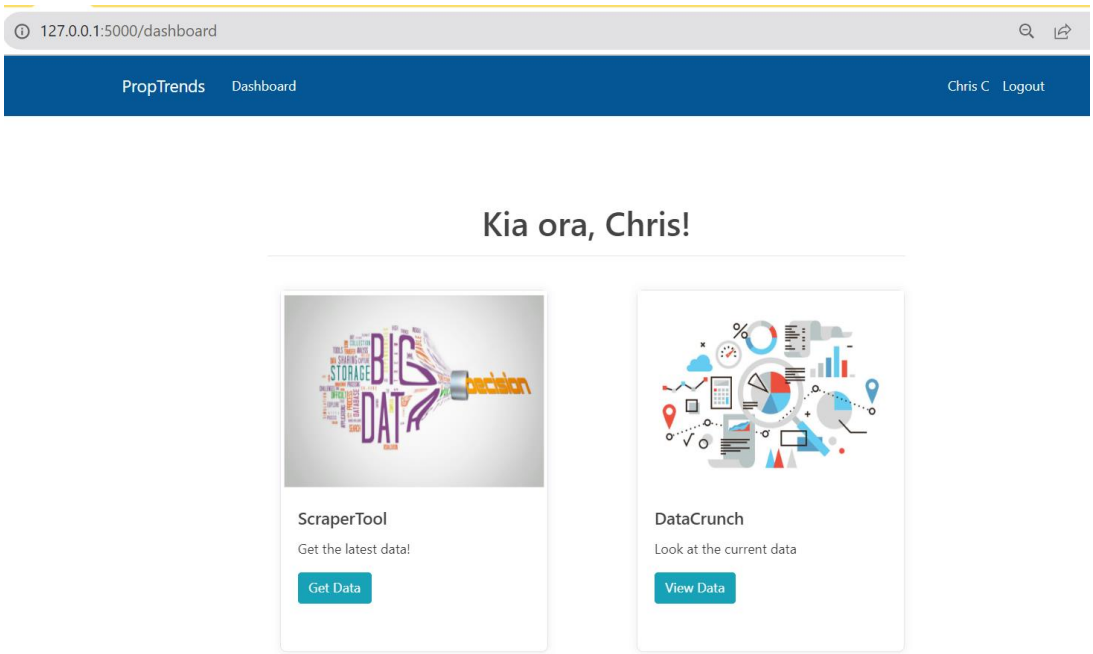
```
# First increase the new_scrape count
new_scrapes += 1
scraped_listings_on_page += 1
# COLLECT AND CREATE RECORD WITH ALL THE CONDENSED INFO FROM THE MAIN PAGE'S LISTING
# Extract address using regex:
address = one_listing_object.find('h3').text.strip()
suburb_pattern = r'[^,]*$'
remove_leadspace_pattern = r'^\s+'
if match := re.search(suburb_pattern, address):
    suburb = match.group(0)
    # Then remove the leading space
    suburb = re.sub(remove_leadspace_pattern,'', suburb)

    print("the address is: " + address + " with suburb: " + suburb)

    # Convert the region and district_city strings to ids for cheaper/more efficient parsing
    suburb_id = convert_to_id(Suburb_key, 'suburb_name', suburb)
```

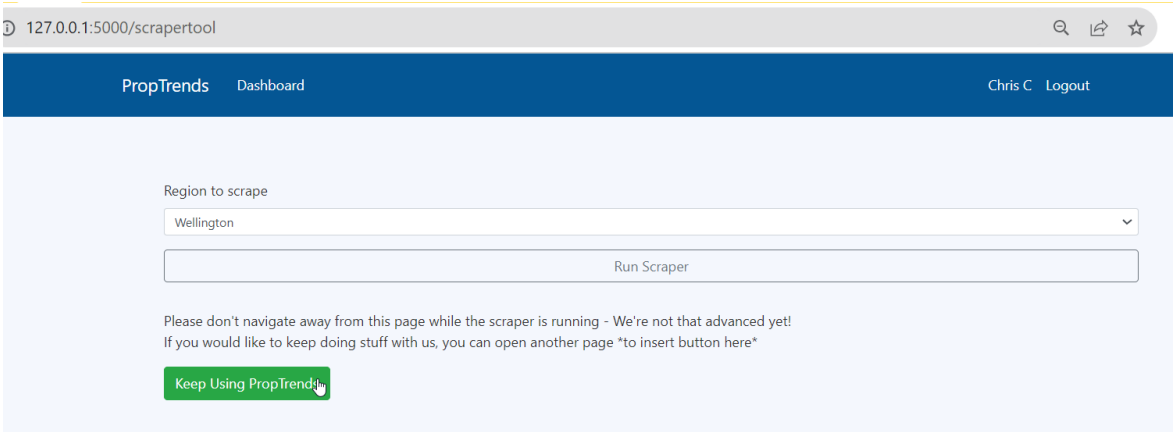
10. It then creates a record and commits the summary values to the database

```
# ADDING ALL EXTRACTED INFO TO LISTING DB TABLE
new_list = Listing(
    region=region_id,
    city=city_id,
    prop_url=prop_href,
    address=address,
    suburb=suburb_id,
    prop_type=proptype_id,
    list_price=price,
    beds=bedrooms,
    baths=bathrooms
)
db.session.add(new_list)
db.session.commit()
db.session.refresh(new_list)
listing_id = new_list.id
print("The listing id is :" + str(listing_id))
delay = uniform(7.03, 111.87)
sleep(delay)   # Adding a sleep time to make sure this doesn't flood the site with too many requests
# Now go into the actual listing (using the prop_href) and scrape details of the actual listing
values = scrape_indiv(mainurl, prop_href, listing_id)
print(values)
```

11. This triggers a scrape_indiv method that scrapes the individual listing for more detailed data, using similar logic as above.

This entire process can be triggered using the ScraperTool button. The ScraperTool currently only allows for the scraping of the Wellington region, but intends to support scraping other regions in the future.





Database architecture

The key area that I've implemented focuses on optimising the datatypes. This may sound strange, but an application like this would easily have tens of thousands of records in the first scrape, with increasing numbers over time. Because of this, the database has been designed to be as efficient as possible. Since integers are far quicker to search than strings, I decided to avoid repeated storing of the same strings by converting the Region, City, Suburb, and Property Type to one-to-many tables.

```python
class City_key(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    city_name = db.Column(db.String)
    citykeys = db.relationship('Listing', backref='cities', lazy=True, uselist=False)

class Region_key(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    region_name = db.Column(db.String)
    regionkeys = db.relationship('Listing', backref='regions', lazy=True, uselist=False)


class Listing(db.Model):
    # DECLARING VARIABLES TO COLLECT
    id = db.Column(db.Integer, primary_key=True)
    prop_url = db.Column(db.String, unique=True)
    list_date = db.Column(db.DateTime)
    address = db.Column(db.String)
    region = db.Column(db.Integer, db.ForeignKey('region_key.id'))
    city = db.Column(db.Integer, db.ForeignKey('city_key.id'))
    suburb = db.Column(db.Integer, db.ForeignKey('suburb_key.id'))      # This will be the suburb.id
    prop_type = db.Column(db.Integer, db.ForeignKey('proptype_key.id'))      # This will be the prop_type.id
    title_type = db.Column(db.String)  # Potentially change to id if there's a lot of listings with titletypes
    list_price = db.Column(db.BigInteger)
    beds = db.Column(db.Integer)
    baths = db.Column(db.Integer)
    size_m2 = db.Column(db.Integer)
```

```python
#==========================================================================================
# This method looks up a name in a key-table, retrieves or creates a new record,
# then returns the name's id
#------------------------------------------------------------------------------------------
def convert_to_id(Table_key, name_in_table, name_to_convert):
    # The table associates the name that we want to convert with int ids so that it's cheaper, and more efficient to store/call
    name_exists = db.session.query(Table_key.query.filter(getattr(Table_key, name_in_table) == name_to_convert).exists()).scalar()
    if name_exists:
        # Get Name id
        name_id = Table_key.query.filter_by(**{name_in_table: name_to_convert}).first().id
        print("existing name's id is " + str(name_id))
    else:
        # Create a new record using the name_to_convert and grab the new record's id
        new_name = Table_key(
            **{name_in_table: name_to_convert}
        )
        db.session.add(new_name)
        db.session.commit()
        db.session.refresh(new_name)
        name_id = new_name.id  # getting new name_id id
        print("new name's id is " + str(name_id))
    return name_id
```

	id [PK] integer	suburb_name character varying
1	1	Island Bay
2	2	Berhampore
3	3	Hataitai
4	4	Mount Cook
5	5	Mount Victoria
6	6	Kilbirnie
7	7	Newtown
8	8	Karori
9	9	Brooklyn
10	10	Owhiro Bay
11	11	Kingston
12	12	Aro Valley
13	13	Vogeltown

Query Query History

```
1 select * from suburb_key
```

id [PK] integer	prop_url character varying	list_date timestamp wi	address character varying	region integer	city integer	suburb integer	prop_type integer	title_type character va
225	/42405071/residential/sale/...	2023-08-...	10 Burrows Avenue, K...	1	1	8	2	[null]
232	/42400535/residential/sale/...	2023-07-...	8/5 Court Road, Tawa	1	1	20	2	[null]
2	/42394243/residential/sale/...	2023-07-...	126 Clyde Street, Islan...	1	1	1	2	[null]
26	/42388677/residential/sale/...	2023-07-...	31 Devon Street, Aro V...	1	1	12	4	Freehold
3	/42389987/residential/sale/...	2023-07-...	16/52 High Street, Isla...	1	1	1	1	[null]
227	/42399322/residential/sale/...	2023-07-...	72 Pitt Street, Wadest...	1	1	24	2	[null]
228	/42399441/residential/sale/...	[null]	3 Newman Terrace, Th...	1	1	31	2	[null]
233	/42406577/residential/sale/...	2023-08-...	45 Catherine Crescent...	1	1	21	2	Freehold
49	/42128665/residential/sale/...	2023-03-...	1/260 Wakefield Stree...	1	1	14	3	[null]
229	/42369145/residential/sale/...	[null]	47 Sefton Street, Wad...	1	1	24	2	[null]
234	/42407224/residential/sale/...	[null]	5/38 Haining Street, T...	1	1	14	3	[null]
92	/42391350/residential/sale/...	2023-07-...	3 Birkhall Grove, Strat...	1	1	30	2	[null]

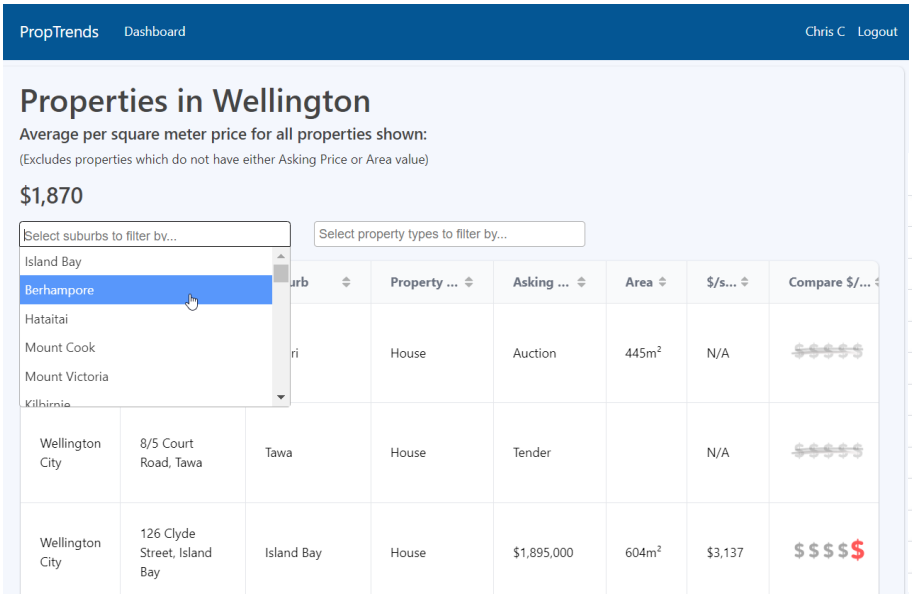
Query History

```
select * from listing
```

Data processing & Visualisation

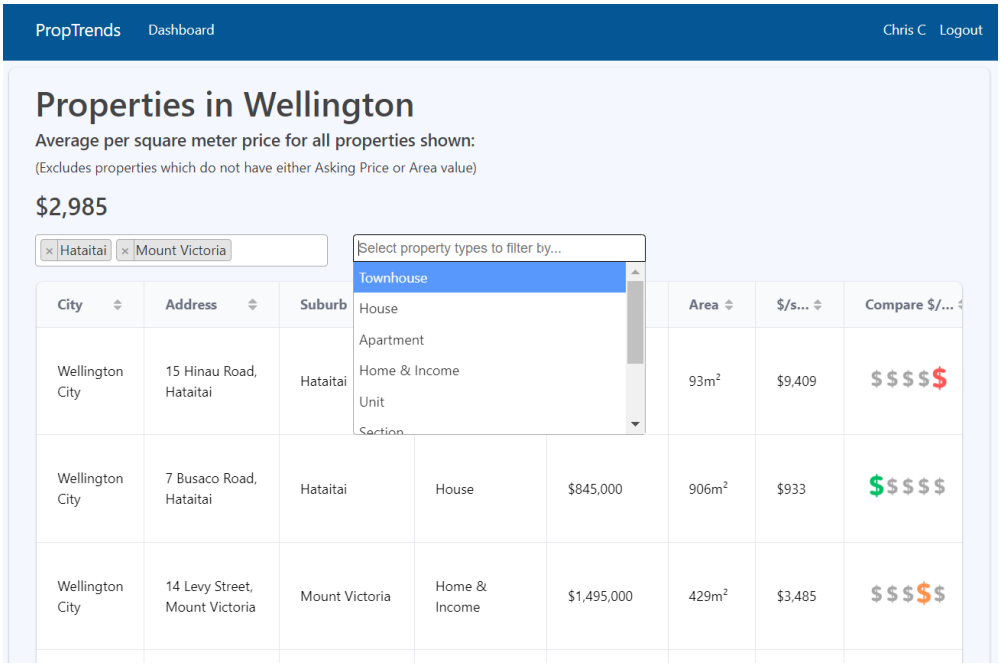
Due to the lack of time and scraping data, I decided to perform a very simple method of processing the data that would explore ways to query a database:

a. The user can filter by “Suburb(s)”. The suburb list is pulled from the Suburb_key.name and increases as different regions are scraped.



	id [PK] integer	suburb_name character varying
1	1	Island Bay
2	2	Berhampore
3	3	Hataitai
4	4	Mount Cook
5	5	Mount Victoria
6	6	Kilbirnie
7	7	Newtown
8	8	Karori
9	9	Brooklyn
10	10	Owhiro Bay
11	11	Kingston
12	12	Aro Valley
13	13	Vogeltown

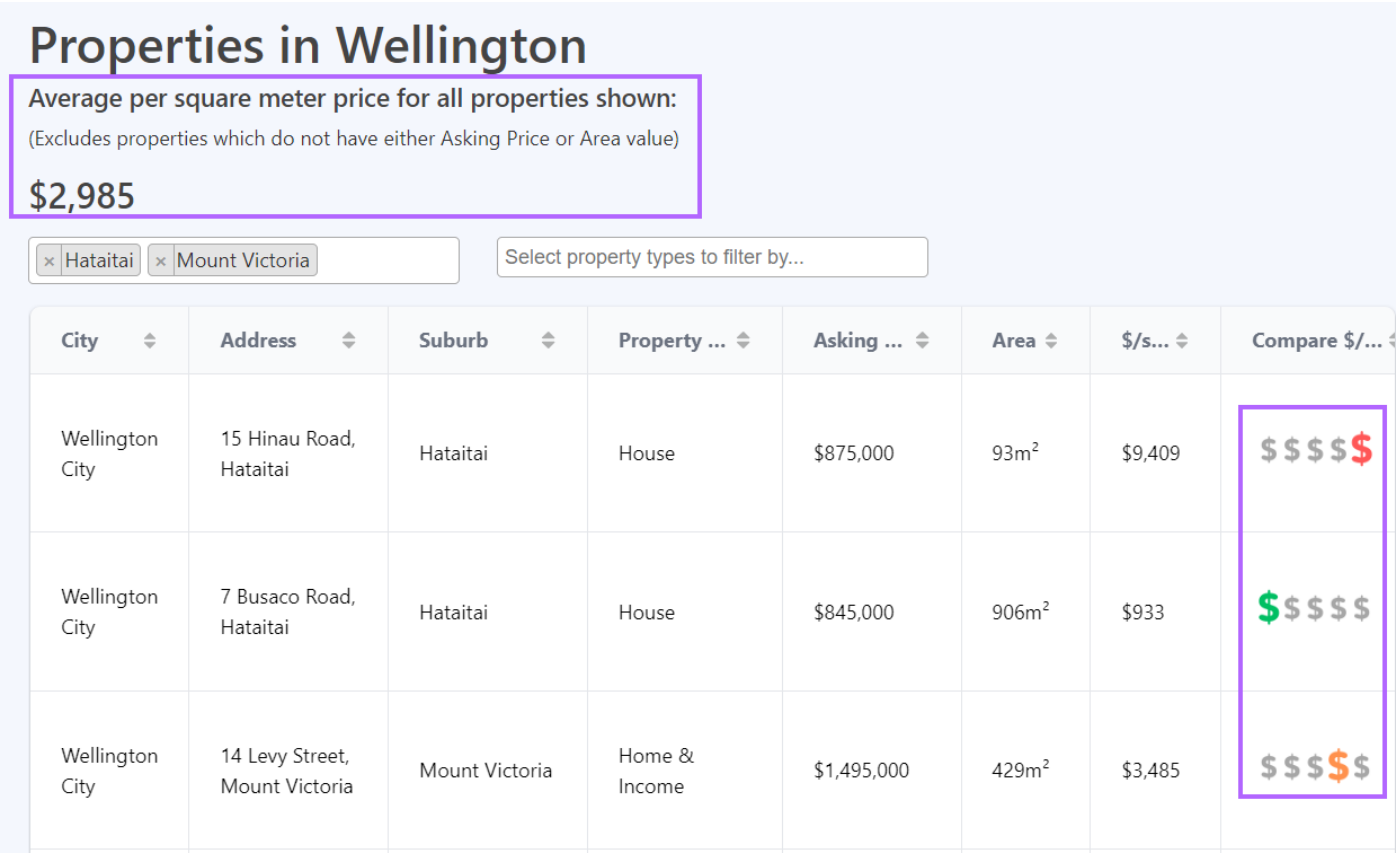
b. The user can add a secondary filter to narrow down the listings by property type:



id [PK] integer	type character varying
1	Townhouse
2	House
3	Apartment
4	Home & Income
5	Unit
6	Section
7	Multiple Properties
8	Lifestyle Section

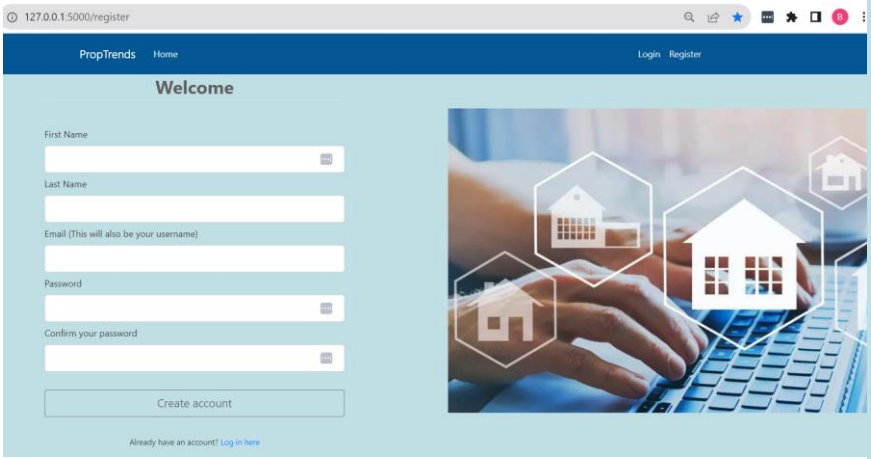
c. On the first page load and every time a filter is added/removed, PropTrends dynamically:
    1. Searches and discard all listings that have not listed either the Asking Price or Area,
    2. Calculates the average per square meter value for all the properties shown on that page,
    3. Calculates the per square meter value for that property,
    4. Compares each listing against the average per square meter. If a listing's price percentage difference is less than 0, the listing's asking price is more than the average value and is a bad deal. Conversely, if the average per square meter is more than 0, it is a good deal.

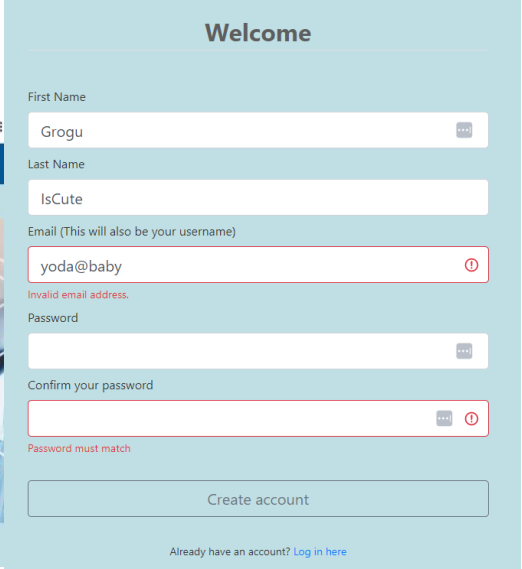
d. Understanding the percentage difference takes time and can be misunderstood. I have therefore translated this to a graph that instantly gives the user an indication of whether or not this is good value:

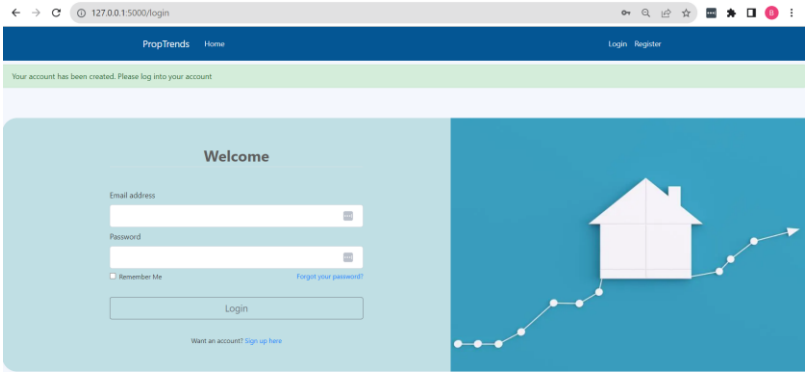


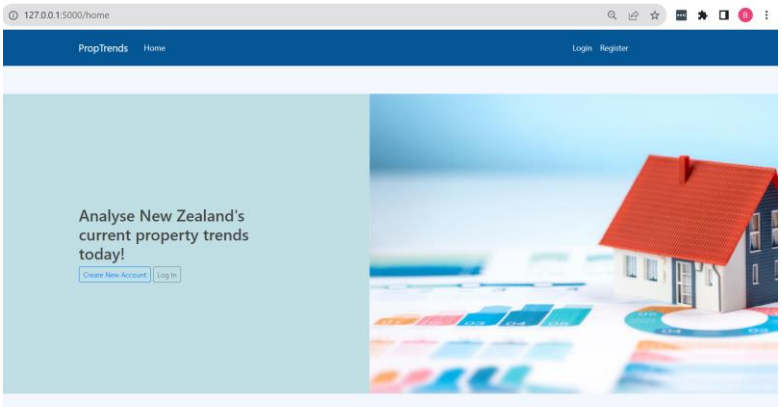
Basic User Capabilities

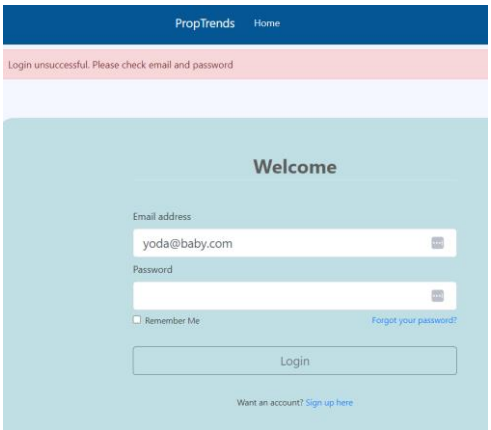
Of course, this application included the very basics: registration, login, and editing of user accounts, while including methods to handle passwords and image uploads. The screenshots taken below are meant to showcase snippets of this process.

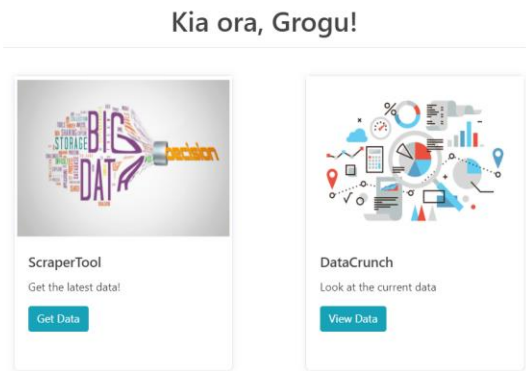

127.0.0.1:5000/register
PropTrends
Home
Login
Register
Welcome
First Name
Last Name
Email (This will also be your username)
Password
Confirm your password
Create account
Already have an account? Log in here

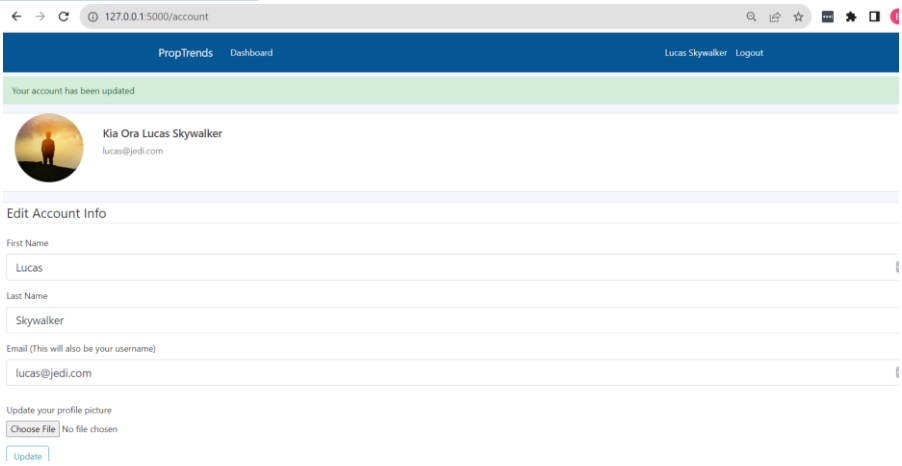

Welcome
First Name
Grogu
Last Name
IsCute
Email (This will also be your username)
yoda@baby
Invalid email address.
Password
Confirm your password
Password must match
Create account
Already have an account? Log in here


127.0.0.1:5000/login
PropTrends
Home
Login
Register
Your account has been created. Please log into your account
Welcome
Email address
Password
Remember Me
Forgot your password?
Login
Want an account? Sign up here


127.0.0.1:5000/home
PropTrends
Home
Login
Register
Analyse New Zealand's current property trends today!
Create New Account
Log In







Future Development

There is always a lot more work that can be done, however this was the best that I could accomplish during the three weeks. Given the chance to progress development on this application, my next steps would be:

1. Collect more data from different regions (via the same website),
2. Have more methods to filter and analyse data, especially historical data and trends over time,
3. Host the application on cloud servers,
4. Turn the scraping process into a serverless function (AWS Lambda or Google Apps Script) that is triggered at random times of the day Automate regular (daily) scraping at random times of the day,
5. Find a way to collect data from sold property listings (i.e. https://www.realestate.co.nz/residential/sold). This won't be fully accurate since not every sold property is listed, but would at least provide a better picture of the market,
6. Look into more feature implementation.

---- End ----