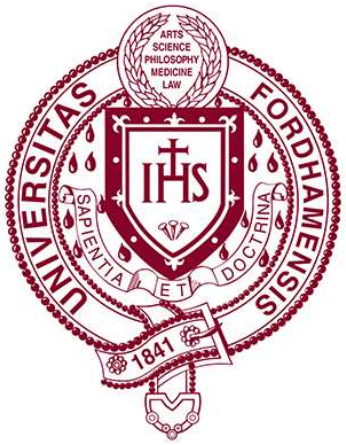


Fordham University

Graduate School of Art and Sciences

Summer 2023

Cloud Computing (CISC 5550)

Final Project

Rosa Sierra

The attached zip file contains:

- Project report (pdf)
- Dockerfile
- Start.sh
- Startup.sh
- Test.sh
- Todolist.db
- Todolist.py
- Todolist_api.py
- Templates/
    - Index.html
    - Login.html
    - Todo.png

What is cloud computing?

Cloud computing refers to the delivery of various computing services – including storage, processing power, networking, databases, software, and more – over the internet. Instead of relying on local servers or physical infrastructure, cloud computing utilizes a network of remote servers hosted in data centers to provide these services to users and businesses. The location where these technological solutions are made is called the "cloud", basically a set of servers available on the internet.

In my perspective, cloud computing has made a revolution in how approachable technology is nowadays. It has not only opened access to powerful computing resources but also redefined how we approach software development, service provisioning, and business management. The cloud breaks physical limitations, enabling even small-scale entities to leverage advanced tools and infrastructure that were once exclusively reserved for large corporations. Even individuals can use the tools provided by those cloud computing solutions enterprises, like google cloud computing or amazon web services.

The need to invest a large amount of money and time in hardware and deploying the required services has been eliminated with cloud computing solutions. And I think this is the greatest impact cloud computing can have in the tech market. As we can see in history, as technology and societies advance, it becomes easier for anyone to have access to them. Another big advantage of cloud computing is the implementation of the technique 'pay as you go', where I can see two main benefits: reduced cost for the users, and better performance for the cloud as not all the users will be demanding resources of the cloud indefinitely.

We have multiple cloud computing systems, like google cloud, amazon web services or azure each one of set with a large set of tools from complex virtual machines to already implemented apis we can simply run on our python or another programming language in just minutes.

Cloud Development and Deployment Project

I. Introduction

Objective

With a provided template of a web service of a to-do list, in this project we will explore different solutions to improve several areas of this to-do list, with some ideas but no liming to expanding the API, adding new functions (as google translate, or twitter API), and others...

Initial Version:

The latest improvement made to the to-do list was homework 4, in which I divided and created a frontend.py and a backend.py (API) files. Then, Deploy the app on the Google Cloud using Kubernetes and Docker Cloud. The web app used to look like the attached Figure 1, and the only functionalities of the to-do list was to create a new item, specifying what to do (which task) and when. Also, to mark as done or delete previous task.

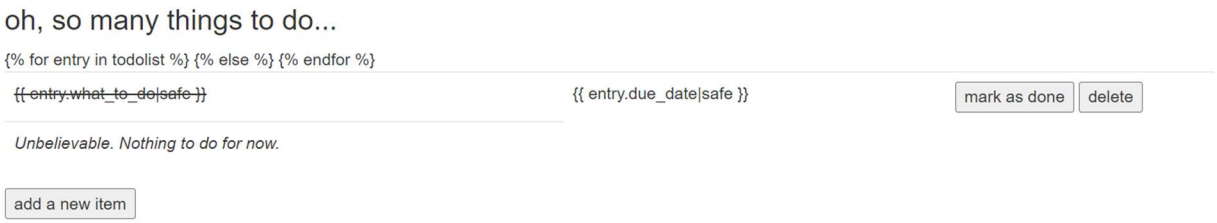


Figure 1. Original version of the to-do webpage

Final Version:

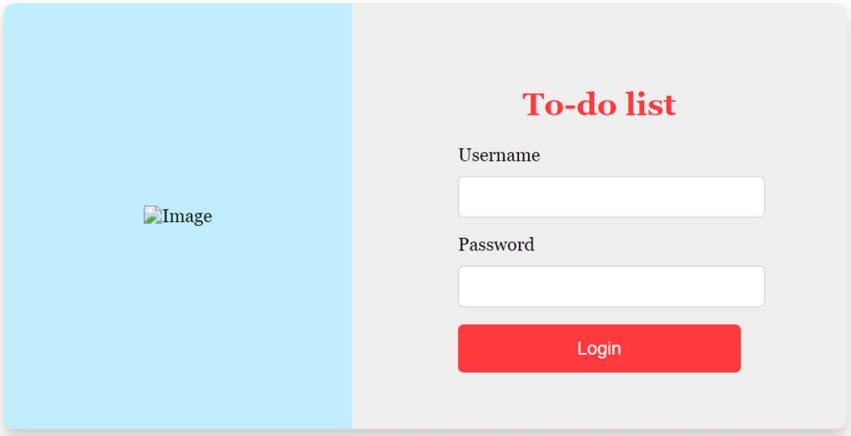
In general, the web app was improved in these main points, that we are going to go over in this report:

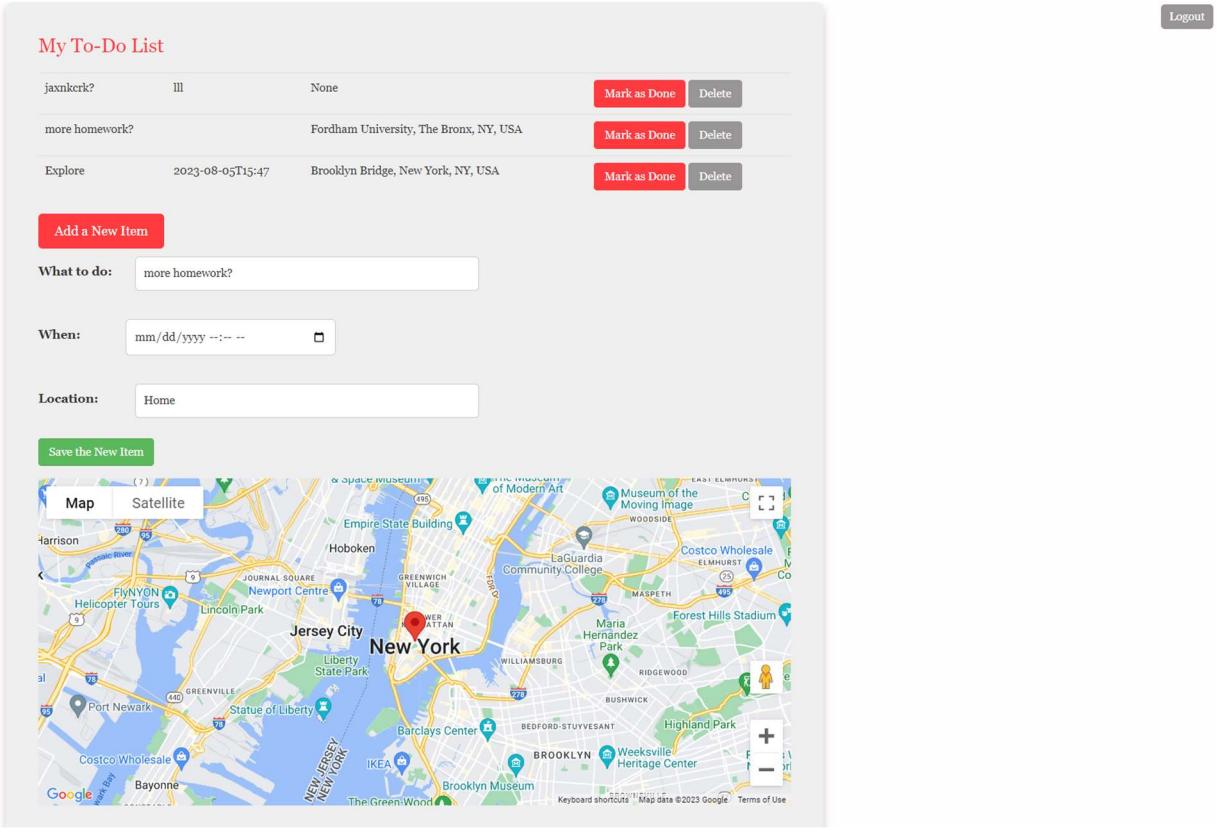
- Improve the visuals of the webpage.
- Robust security with log-in.
- Standardized 'when' column with a date-time selection.
- Create a variable 'location'.
- Added Autocomplete location Google API.
- Added a Map Visualization to support the location variable.

II. Implemented Solutions

a. Improve the visuals of the webpage.

Using html and CSS the final webpage files: index.html and login.html used the same palette color and font name to improve the appearance of the webpage to the user. See bellow how the webpage looks now compared to before [I have some troubles for making the image to appear in the login.html file]





b. Robust security with log-in/logout

As a security measure, I implemented a log-in/logout validation in the webpage to access the general to-do list. A way to extend this work is to implement cross validation and assign a list of to-dos to each user. But, for this project the log-in is treated as a user validation of the access to see and/or modify the to-dos. For implementing the log-in I made changes in the frontend, backend and even needed to create a new webpage login.html. But for the logout it was only necessary to make changes in the index.html and frontend file. The logout button can be found in the upper right corner of the to-do list webpage.

- Backend (todolist_api.py)

I created a dictionary of stored users/passwords valid to access the list on the backend(api).py, as an improvement we can make this more secure and make hash encryption for the passwords.

```
app.secret_key = 'your_secret_key'

users = {
    'user1': {'password': 'password1'},
    'user2': {'password': 'password2'}
}
```

Then, the validation of this user/password is also done from the username/password data provided from the html code (login.html)

```
@app.route('/login', methods=['POST'])
def login():
    data = request.get_json()
    username = data.get('username')
    password = data.get('password')

    if username in users and password == users[username]['password']:
        return jsonify({'message': 'Login successful'}), 200
    else:
        return jsonify({'message': 'Invalid login credentials'}), 401
```

- Index.html

We added the logout button to the index with connect to the frontend.py to logout the session:

```
<body>
    <div class="box-container">
        <div class="logout-container">
            <a class="logout-button" href="{{ url_for('logout') }}">Logout</a>
        </div>
```

- Frontend (todolist.py)

In the frontend we defined the request to the backend to the login

```
@app.route('/login', methods=['POST'])
def login():
    username = request.form.get('username')
    password = request.form.get('password')

    response = requests.post(f'{TODO_API_URL}/login', json={'username': username, 'password': password})

    if response.status_code == 200:
        session['username'] = username  # Start the session by storing the username
        #return redirect(url_for('index'))
        return redirect(url_for('show_list'))
    else:
        error_message = "Invalid login credentials"
        return render_template('login.html', error_message=error_message)
```

Also, for all the request (creating a new task, seeing the index.html (tasks), deleting) the systems first verify if the user has already login using session validation as bellow:

```
@app.route("/")
def show_list():
    if 'username' in session:
        resp = requests.get(TODO_API_URL+"/api/items")
        resp = resp.json()
        return render_template('index.html', todolist=resp)
    else:
        flash('Please log in to add tasks', 'error')
        return redirect(url_for('login_page'))
```

For the logout button on the upper right corner, the function was defined on the frontend file as bellow:

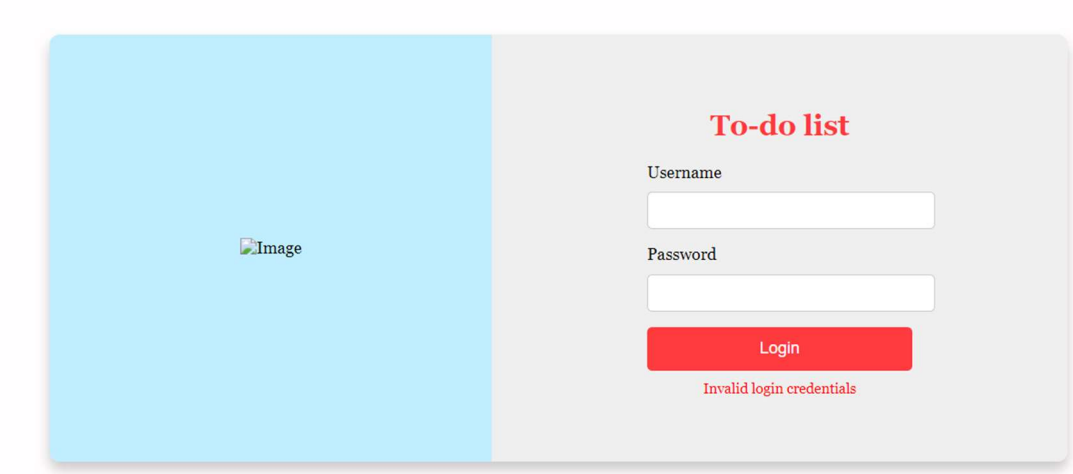
```
@app.route('/logout')
def logout():
    session.pop('username', None)  # Remove the 'username' from the session
    return redirect(url_for('login_page'))
```

- Login.html:

For implementing a log-in step in our webpage, I needed to create a front webpage to act like a barrier between the user and the list. In this login.html webpage the username and password are captured, and then validated by the backend.py file.

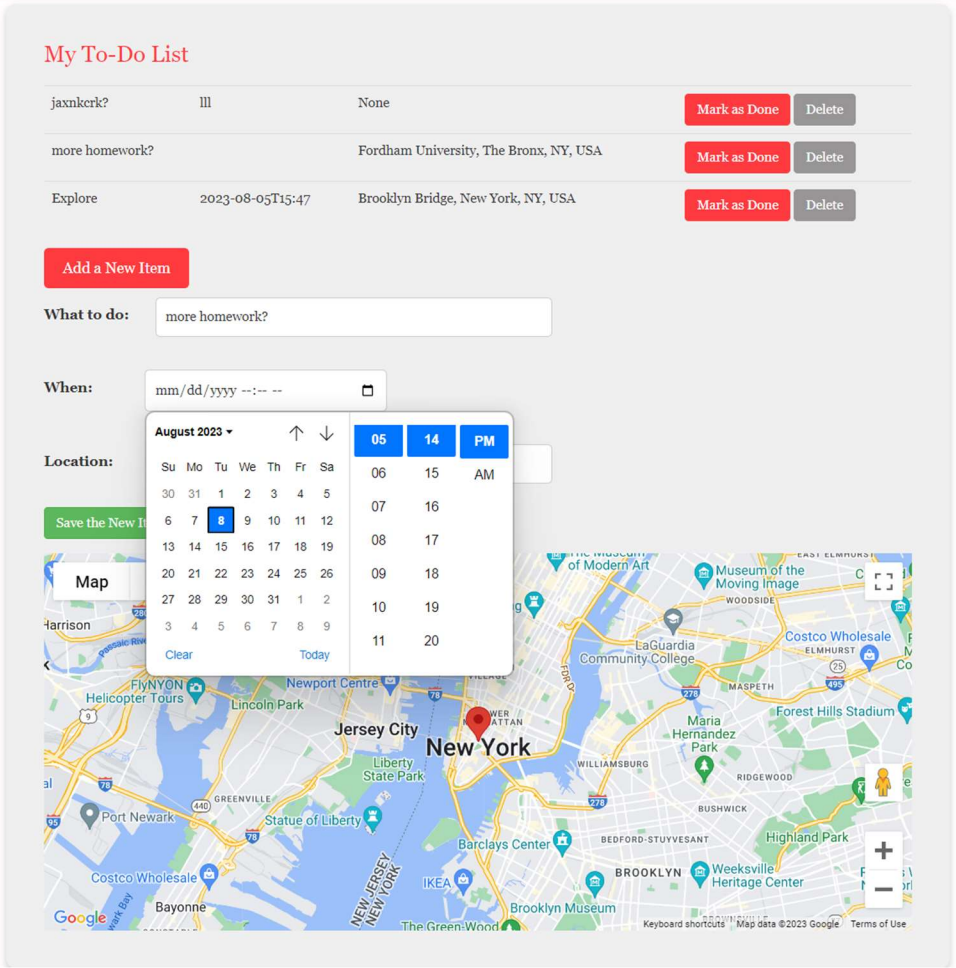
```
<body>
    <div class="container">
        <div class="left-container">
            <img class="image" src="todo.png" alt="Image">
        </div>
        <div class="right-container">
            <h2>To-do list</h2>
            <form action="/login" method="POST">
                <label for="username">Username</label>
                <input type="text" name="username">
                <label for="password">Password</label>
                <input type="password" name="password">
                <button type="submit">Login</button>
            </form>
            {% if error_message %}
                <p class="error">{{ error_message }}</p>
            {% endif %}
        </div>
    </div>
</body>
</html>
```

Also, if the log-in is not successfully, a message of 'invalid login credentials' defined on the backend code is printed on the screen as we can see below:



c. Standardized 'when' column with a date-time selection.

To add a new item to the to-do list, the column of date(when) now provides a standardized way of filling it selecting the due date and time of this task. For creating this standardized selection to the when column, I only needed to modify the index.html code as described below.



- **Index.html**

I only needed to modify the type of the input to 'datetime-local' which automatically enables the datetime selection.

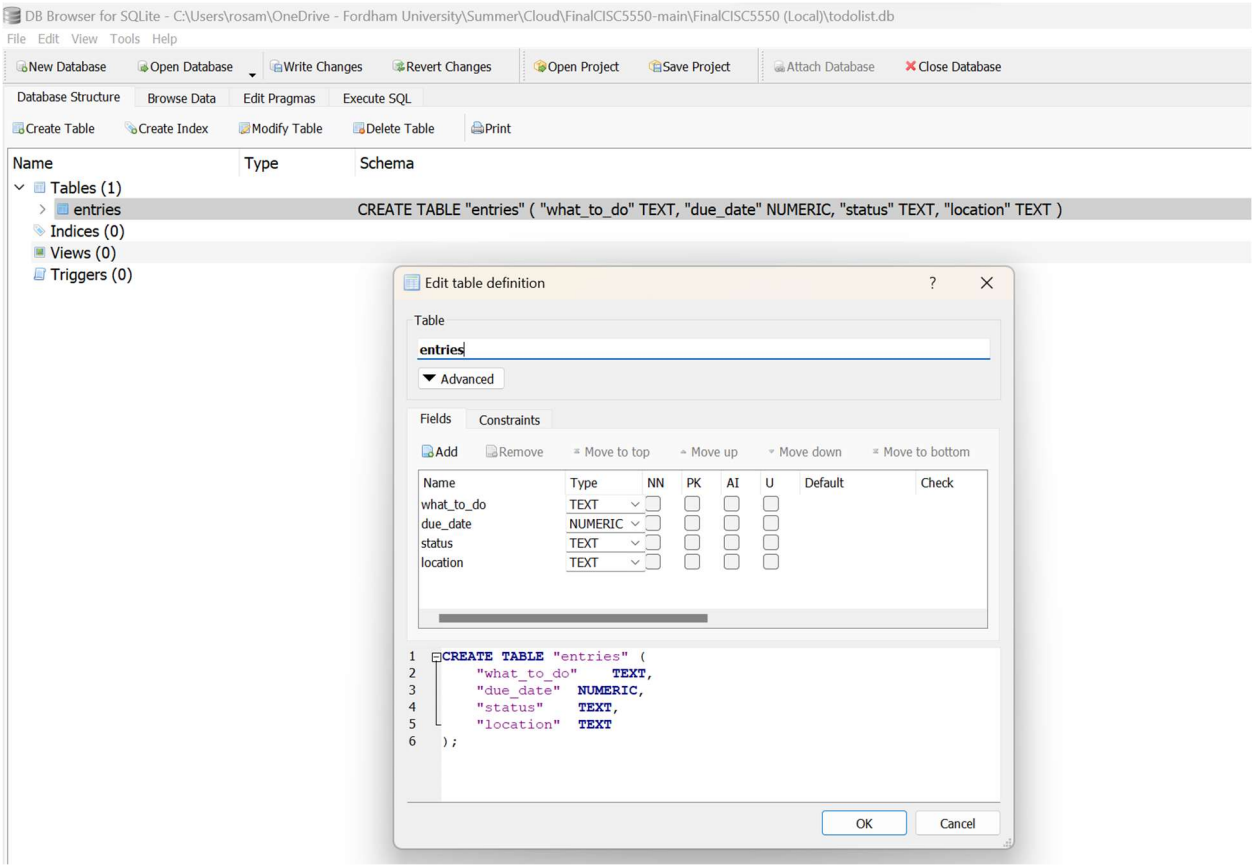
```
        </div>
        <div class="form-row">
            <div class="input-container">
                <label class= 'ex1' for="due_date">When:</label>
                <input type="datetime-local" id="due_date" name="due_date" value="">
            </div>
        </div>
```

d. Create a variable 'location'.

For extending the to-do list, I created a new variable called 'location' to be saved and shown in our to-do list. For implementing this I needed to modify the database, frontend, backend, and index.html files.

- **Database:**

Using SQL, I modified the query that creates the table entries, to add a new column of text called 'where', but then some difficulties were found with the code as 'WHERE' is also a command of SQL, so I renamed my column as 'location' doing the same steps on SQL.



- **Frontend, backend:**

The same logic was applied to both of the files, the idea was to add the column 'location' along with the others ('what to do','due_date' and 'status') in the lines that were necessary so our code would be aware that this new column would be on use. See bellow an example on the backend code:

```python
@app.route("/api/items")  # default method is GET
def get_items():
    db = get_db()
    cur = db.execute('SELECT what_to_do, due_date, location, status FROM entries')
    entries = cur.fetchall()
    tdlist = [dict(what_to_do=row[0], due_date=row[1], location=row[2], status=row[3])
              for row in entries]
    response = Response(json.dumps(tdlist),  mimetype='application/json')
    return response
```

- **Index.html**

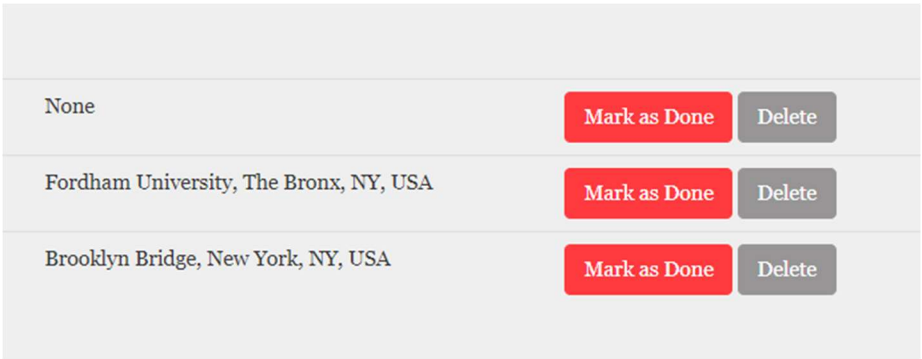
For the index.html file first the 'location' field was added so when the user clicks "add new item" location appears as a field to be completed.

```html
<div class="form-row">
    <div class="input-container">
        <label class= 'ex1' for="location">Location:</label>
        <input type="text" size="50" name="location" id="location" value="Home">
    </div>
</div>
```

Then, it was added along with the other variables, so after the user clicks "add task" the field location appears in the screen with the content.

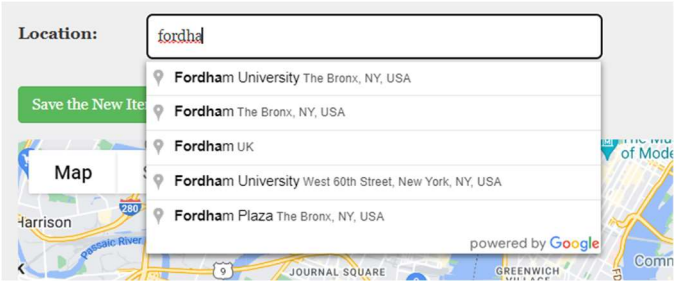
```html
<table class="table">
    {% for entry in todolist %}
    <tr>
        <td {% if entry.status=='done' %} class='done' {% endif %}>{{ entry.what_to_do|safe }}</td>
        <td>{{ entry.due_date|safe }}</td>
        <td>{{ entry.location|safe }}</td>
```

Webpage location in the index.tml:

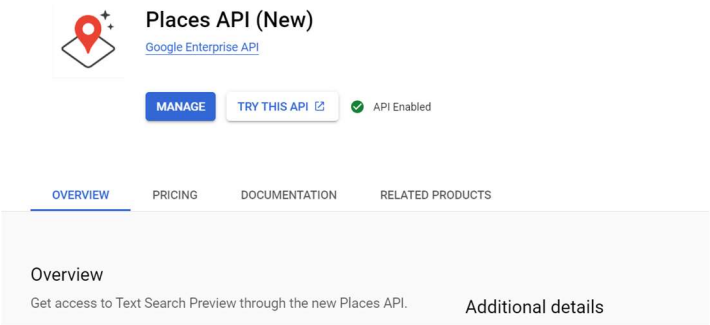


e. Added Autocomplete location Google API.

The idea of this functionality is that when the users start typing a location, the box with automatically suggest places based on what the user is writing. The idea of this function is similar to the date one, to basically standardized this field. For implementing this only the index.html was modified. See below the idea of the location suggestion



The first step to add autocomplete was to enable the google location API (or Places API). And then getting the API key associated with it.



- **Index.html**

The API key was introduced in the script source, and then the function Autocomplete was defined to work in based of the variable 'location'. [there are some lines of code on this screenshots that belongs to the map, please ignore it as we are going to talk about it next]

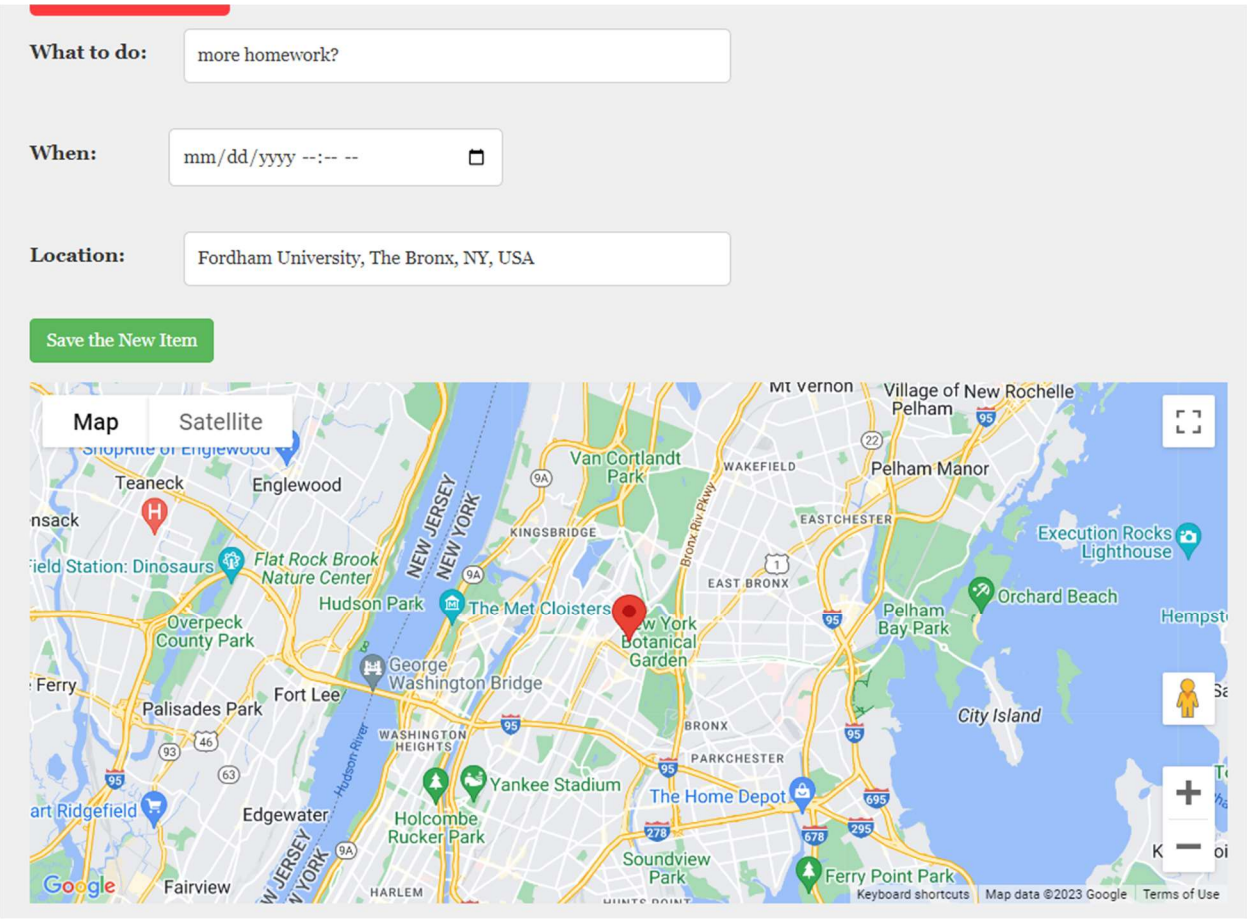
```
    <script src="https://maps.googleapis.com/maps/api/js?key=AIzaSyDdoiSj5ENlnSGyWtKEuRLrdw0v2L3qqFQ&libraries=places"></script>
<script>
    var show_entry_form = false;
    var markers = [];


    function toggle_entry_form() {
        if (show_entry_form) {
            $('#add-form').hide();
        } else {
            $('#add-form').show();
        }
        show_entry_form = !show_entry_form;
    }

    function initAutocomplete() {
        var input = document.getElementById('location');
        var autocomplete = new google.maps.places.Autocomplete(input);
```

f. Added a Map Visualization to support the location variable.

To continue expanding the location variable and the autocompleted, I expanded the google cloud API to show a map that confirms the location of the selected location. See below an example, if I select Fordham University, the Bronx from the suggested options, it will tell me where its exactly located on the map. A way to extend this work that I tried to do but didn't work is to find a way to maintain all the previously created tasks as bullets in the maps [that's the 'marker' you will see in the code]. For this implementation, I only needed to extend the previously done work in index.html.



- **Index.html**

```
function initMap() {
    var map = new google.maps.Map(document.getElementById('map'), {
        center: { lat: 40.71427, lng: -74.00597 }, // Set default center
        zoom: 12
    });

    // Create the fixed marker
    var fixedMarkerPosition = { lat: 40.71427, lng: -74.00597 }; // Set the position for the fixed marker

    fixedMarker = new google.maps.Marker({
        map: map,
        position: fixedMarkerPosition,
        title: "Fixed Marker"
    });

    markers.push(fixedMarker); // Add the fixed marker to the markers array

    // Optionally, you can add an info window to the fixed marker
    var fixedInfowindow = new google.maps.InfoWindow({
        content: "Fixed Marker Info"
    });

    // Open the info window when fixed marker is clicked
    fixedMarker.addListener('click', function () {
        fixedInfowindow.open(map, fixedMarker);
    });

    var input = document.getElementById('location');
    var autocomplete = new google.maps.places.Autocomplete(input);

    // Add event listener when a place is selected
    autocomplete.addListener('place_changed', function () {
        var place = autocomplete.getPlace();

        if (!place.geometry) {
            // User entered the name of a place that was not suggested
            return;
        }

        // Set map center to the selected place
        map.setCenter(place.geometry.location);

        // Create a task marker at the selected place
        createTaskMarker(map, place.geometry.location, place.name);
    });
}
```