

# **Final Project Report**

By: Anu Kumar

## **Project: Places to See Finder**

Using the “Places to See Finder” Flask application, a user can easily see a list of all the top ranked places to visit in a state and city of their choice. Users can further refine their results by entering a category criteria, such as “beaches”, “parks”, “hotels”, etc. to view a refined list of places to see in the category of their choice.

The homepage of the application shows:

- One dropdown menu to select a US state
- A search bar to enter a city name
- A search bar to enter a location category
- A table showing all the categories and subcategories of “places to visit”

Once the user submits their search criteria, the application redirects to the next page, which shows a list of all the top rated places to see and details about each place. For each “place to see” entry on the page, the user can view:

- An image of the place
- Average Rating (out of 5)
- Number of Ratings Given
- Phone Number
- Link to Yelp to see more details about the place

**Github Link to Project Code:**

<https://github.com/kumaram123/placestoseefinder>

## Data Sources

1. **Data source:** Yelp Fusion (<https://docs.developer.yelp.com/docs/resources-categories>)  
**Category:** Web API you haven't used before that requires API key ❖  
**Challenge Score:** 4  
**Number of Records:** ~1600 records (I retrieved and used all the records)  
**Documentation Format:** I cached the data and stored in it a JSON file  
**Summary:** Yelp fusion requires an API key. In order to display the "places to see" that fits the category entered by the user, I used the "/categories" endpoint to obtain all the categories which of the listings on Yelp. An API key is required to access in order to get all Yelp business categories across all locales.

Evidence of Caching:

Screenshot of cached data:

Important fields/attributes:

1. "alias" : the child category's alias. Gives a different name for the title.
2. "title" : the child category's title. Used to find the different categories of "places to visit".
3. "parents": Info on which parent category this belongs to. I'll use this attribute to find which category the "place to visit" belongs to.
4. "country\_whitelist" :List of available countries

```
{
  "https://api.yelp.com/v3/categories": {
    "categories": [
      {
        "alias": "3dprinting",
        "title": "3D Printing",
        "parent_aliases": [
          "localservices"
        ],
        "country_whitelist": [],
        "country_blacklist": []
      },
      {
        "alias": "abruzzese",
        "title": "Abruzzese",
        "parent_aliases": [
          "italian"
        ],
        "country_whitelist": [
          "IT"
        ],
        "country_blacklist": []
      },
      {
        "alias": "absinthebars",
        "title": "Absinthe Bars",
        "parent_aliases": [

```

Parts of code that implement caching/retrieve cached data:

```
if checkCacheExistence(API_CACHE_FILENAME) == False:
    api_results = make_request_with_cache(API_BUSINESS_PATH, PARAMETERS, API_CACHE_FILENAME, False)
else:
    api_results = checkCacheExistence(API_CACHE_FILENAME)
```

```
def make_request_with_cache(baseurl, params, CACHE_FILENAME, isScrape: bool):
    request_key = construct_unique_key(baseurl, params, isScrape)
    if isScrape == False:
        print("cache miss!", request_key)
        API_CACHE_DICT[request_key] = make_request(baseurl, params)
        save_cache(API_CACHE_DICT, CACHE_FILENAME)
        return API_CACHE_DICT[request_key]

    if isScrape == True:
        print("cache miss!", request_key)
        CACHE_DICT[request_key] = make_request(baseurl, params)
        save_cache(CACHE_DICT, CACHE_FILENAME)
        return CACHE_DICT[request_key]
```

```
def checkCacheExistence(CACHE_FILENAME):
    existingJsonFiles = []
    for x in os.listdir():
        if x.endswith(".json"):
            existingJsonFiles.append(x)

    if CACHE_FILENAME in existingJsonFiles:
        print("Cache hit!: ", CACHE_FILENAME)
        return open_cache(CACHE_FILENAME)

    else:
        return False
```

Cached file saved in project directory:

```
{ } yelp_API_cache_Novi.json
{ } yelp_scraping_cache_Novi.json
```

2. **Data Source:** Yelp ([https://www.yelp.com/search?find\\_desc=attractions&sortby=rating](https://www.yelp.com/search?find_desc=attractions&sortby=rating))

**Category:** Crawling and scraping multiple pages in a site you haven't used before ❖

**Challenge Score:** 8

**Number of Records:** ~100 records per location (I retrieved and used all the records)

**Documentation Format:** The program crawls & scrapes 10 pages of results (10 entries on each page) for each location searched, then caches the data, and stores in it a JSON file.

**Summary:** Through crawling and scraping multiple pages on Yelp, I collected data on all the "places to see". Yelp only displays 10 results per page, which is not enough results for my project, so I crawled and scraped 10 pages for each location in order to retrieve ~100 "places to see" for each location.

**Evidence of Caching:**

**Screenshot of cached data:**

**Important fields/attributes:**

1. "name": name of the place
2. "rating": average rating of the place
3. "reviewCount": number of people who have given a review for the place
4. "url": URL of the place on Yelp
5. "categories": list of categories the place belongs to
6. "thumbnail": URL of the image of the place to display an image to the user

```
{
  "bizId": "51EFwxbj2yJgE-4jNIPIA",
  "searchResultBusiness": {
    "ranking": 1,
    "isAd": false,
    "renderAdInfo": false,
    "name": "TreeRunner West Bloomfield Adventure Park",
    "alternateNames": [],
    "businessUrl": "/biz/treerunner-west-bloomfield-adventure-park-west-bloomfield-2?osq=attractions",
    "categories": [
      {
        "title": "Challenge Courses",
        "url": "/search?find_desc=Challenge+Courses&find_loc=Novi%2C+MI"
      },
      {
        "title": "Climbing",
        "url": "/search?find_desc=Climbing&find_loc=Novi%2C+MI"
      },
      {
        "title": "Ziplining",
        "url": "/search?find_desc=Ziplining&find_loc=Novi%2C+MI"
      }
    ],
    "priceRange": "",
    "rating": 4.5,
    "isDecimalRating": false,
    "reviewCount": 119,
    "formattedAddress": "",
    "neighborhoods": [],
    "phone": "(248) 419-1558",
    "serviceArea": null,
    "parentBusiness": null,
    "servicePricing": null,
    "serviceOfferings": [],
    "businessAttributes": {},
    "alias": "treerunner-west-bloomfield-adventure-park-west-bloomfield-2",
    "website": {
      "href": "http://www.treerunnerwestbloomfield.com",
      "rel": "noopener nofollow"
    }
  },
  "scrollablePhotos": {
    "isScrollable": true,
    "photoList": [
      {
        "src": "https://s3-media0.fl.yelpcdn.com/bphoto/ChZwh9I4GvR13beHdV8sQ/348s.jpg",
        "srcset": "https://s3-media0.fl.yelpcdn.com/bphoto/ChZwh9I4GvR13beHdV8sQ/1000s.jpg 2.87x"
      }
    ]
  }
}
```

**Parts of code that implement caching/retrieve cached data:**

```
if checkCacheExistence(CACHE_FILENAME) == False:
    all_results = []
    for b in BUSINESS_PATH:
        results = make_request_with_cache(b, PARAMETERS, CACHE_FILENAME, True)
        all_results.append(results)
    else:
        all_results = checkCacheExistence(CACHE_FILENAME)
```

```
def make_request_with_cache(baseurl, params, CACHE_FILENAME, isScrape: bool):
    request_key = construct_unique_key(baseurl, params, isScrape)
    if isScrape == False:
        print("cache miss!", request_key)
        API_CACHE_DICT[request_key] = make_request(baseurl, params)
        save_cache(API_CACHE_DICT, CACHE_FILENAME)
        return API_CACHE_DICT[request_key]

    if isScrape == True:
        print("cache miss!", request_key)
        CACHE_DICT[request_key] = make_request(baseurl, params)
        save_cache(CACHE_DICT, CACHE_FILENAME)
        return CACHE_DICT[request_key]
```

```
def checkCacheExistence(CACHE_FILENAME):
    existingJsonFiles = []
    for x in os.listdir():
        if x.endswith(".json"):
            existingJsonFiles.append(x)

    if CACHE_FILENAME in existingJsonFiles:
        print("Cache hit!", CACHE_FILENAME)
        return open_cache(CACHE_FILENAME)

    else:
        return False
```

**Cached file saved in project directory:**

```
{ } yelp_scraping_cache_Dallas.json
{ } yelp_scraping_cache_Las Vegas.js
{ } yelp_scraping_cache_Miami.json
{ } yelp_scraping_cache_Novi.json
```

3. **Data source:** US Cities Database (<https://simplemaps.com/data/us-cities>)

**Category:** CSV or JSON file you haven't used before with >1000 records

**Challenge Score:** 2

**Number of Records:** ~31000 records (I retrieved all the records and used data on all US states)

**Documentation Format:** I downloaded the CSV file directly, formatted the data to fit my requirements, then cached the data and stored in it a JSON file

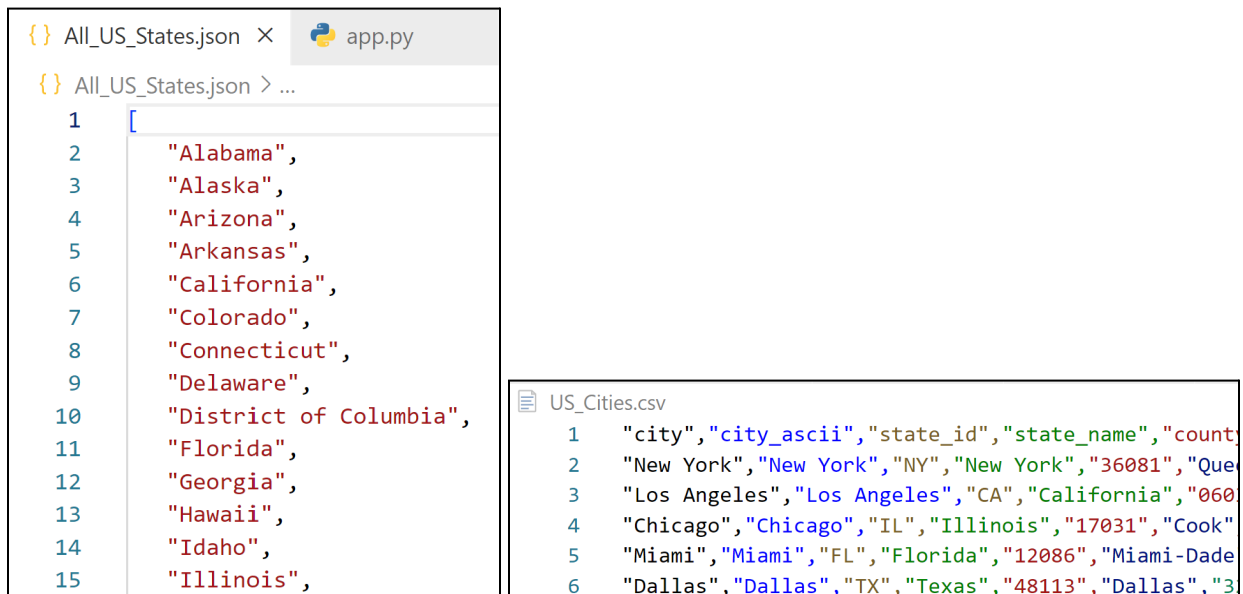
**Summary:** I will use the US states data to generate a dropdown search menu for the users to choose a state in which they would like to view "places to visit".

**Evidence of Caching:**

Important fields/attributes:

1. "city": The name of the city/town.
2. "state\_name" : The name of the state or territory that contains the city/town
3. "city\_ascii": city as an [ASCII](#) string.
4. "city\_alt": Alternative name of the city/town
5. "state\_id": The state or territory's USPS postal abbreviation.

Screenshot of cached data:



The screenshot shows a code editor with two files open. The left file, 'All\_US\_States.json', displays a JSON array of US state names: Alabama, Alaska, Arizona, Arkansas, California, Colorado, Connecticut, Delaware, District of Columbia, Florida, Georgia, Hawaii, Idaho, and Illinois. The right file, 'US\_Cities.csv', shows a snippet of CSV data with columns: city, city\_ascii, state\_id, state\_name, and county. The first row of data is: New York, New York, NY, New York, 36081, Queens.

Snippet of CSV File:

Parts of code that implement caching/retrieve cached data:

```
# load from cache or create cache
if cache.checkCacheExistence("All_US_States.json") == False:
    cache.save_cache(states, "All_US_States.json")
else:
    cities_list = cache.checkCacheExistence("All_US_States.json")
```

## Data Structure: Trees

I have used a tree data structure to store the categories and subcategories of all the “places to see” in a hierarchical manner. The “categories” information was accessed using an API key from the Yelp Fusion “categories” endpoint. The API data was cached and stored in a JSON file. The “categories” information contains various attributes regarding each category, but the two attributes I used to create the tree are: “parent\_alias” (main category) and “alias” (name of the category). Under the parent category “Active”, there are many subcategories such as “amusement parks”, “boating”, “flyboarding”, etc. In order to organize all the categories of “places to visit” in a tree data structure, I have created a “TreeNode” class with the following member functions: `__init__()`, `add_child()`: adds a child to a parent node, `get_level()`: gets the level of a specific node, and `print_tree()`: formats and prints the tree in a readable format. To begin, I set the root of the tree to be “categories”. The second level of the tree will be all the “parent aliases”, which are the parent categories. The third level of the tree is all the sub-categories within each parent alias. For example, the flow would be something like: Root: “Category” => Parent Alias: “Active” => Sub-Category: “Boating”.

Snippet of cached “categories” JSON file:

```
{
  "https://api.yelp.com/v3/categories": {
    "categories": [
      {
        "alias": "3dprinting",
        "title": "3D Printing",
        "parent_aliases": [
          "localservices"
        ],
        "country_whitelist": [],
        "country_blacklist": []
      },
      {
        "alias": "abruzzese",
        "title": "Abruzzese",
        "parent_aliases": [
          "italian"
        ],
        "country_whitelist": [
          "IT"
        ],
        "country_blacklist": []
      }
    ]
  }
}
```

## Screenshot of TreeNode class and creation of the tree:

```
class TreeNode:
    def __init__(self, data):
        self.data = data
        self.children = []
        self.parent = None

    def add_child(self, child):
        self.child = child
        child.parent = self
        self.children.append(child)

    def get_level(self):
        level = 0
        p = self.parent
        while p:
            p = p.parent
            level += 1
        return level

    def print_tree(self):
        print(' '*self.get_level() + '|--', end = '')
        print(self.data)
        if self.children:
            for each in self.children:
                each.print_tree()
```

```
def run():
    categories = scraper.checkCacheExistence("yelp_API_cache_Dallas.json")
    key = list(categories.keys())[0]

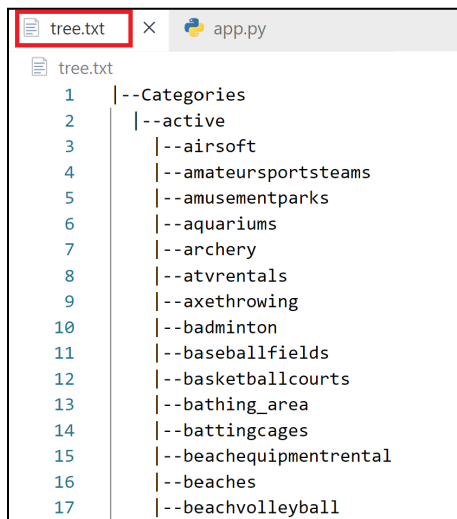
    parent_aliases = []
    for i in categories[key]["categories"]:
        for j in i["parent_aliases"]:
            if j not in parent_aliases:
                parent_aliases.append(j)

    parent_aliases = sorted(parent_aliases)

    root = TreeNode('Categories')
    for parents in parent_aliases:
        category = TreeNode(parents)
        root.add_child(category)
        for i in categories[key]["categories"]:
            for j in i["parent_aliases"]:
                if j == parents:
                    category.add_child(TreeNode(i["alias"]))

    root.print_tree()
```

## Screenshot of a portion of the saved tree as .txt file:



```
tree.txt
1 |--Categories
2   |--active
3       |--airsoft
4       |--amateursportsteams
5       |--amusementparks
6       |--aquariums
7       |--archery
8       |--atvrentals
9       |--axethrowing
10      |--badminton
11      |--baseballfields
12      |--basketballcourts
13      |--bathing_area
14      |--battingcages
15      |--beachequipmentrental
16      |--beaches
17      |--beachvolleyball
```

## Screenshot of a portion of the saved tree as .JSON file:



```
tree.json > ...
1 {
2   "categories": {
3     "active": [
4       "airsoft",
5       "amateursportsteams",
6       "amusementparks",
7       "aquariums",
8       "archery",
9       "atvrentals",
10      "axethrowing",
11      "badminton",
12      "baseballfields",
13      "basketballcourts",
14      "bathing_area",
15      "battingcages",
16      "beachequipmentrental",
17      "beaches",
18      "beachvolleyball",
19      "bicyclepaths",
```

## Screenshot of readTreeJSON() function in readTree.py:

```
def readTreeJSON(treeFilename):
    ''' opens and returns the tree json file if it exists
    if the tree file doesn't exist, creates a new cache dictionary
    Parameters
    -----
    None
    Returns
    -----
    The opened tree JSON file
    ...
    try:
        tree_file = open(treeFilename, 'r')
        tree_contents = tree_file.read()
        tree_data = json.loads(tree_contents)
        tree_file.close()
        return tree_data
    except:
        print("Tree file does not exist")
```

## Interaction and Presentation

The Flask application I have created: “Places to See Finder” lets the user choose a specific state, city and category and shows the list of top rated places to visit. On the homepage of the Flask App, users can enter their desired state, city, and category. Upon clicking the “submit” button, the app redirects to the next page, which displays a table containing the following information about each “place to visit”

1. Name of the place
2. An image of the place
3. The average rating out of 5
4. Number of ratings given
5. The URL to Yelp to view additional information
6. Contact number

### Different Graphs/Displays/Presentation:

1. HTML dropdown to select a US state
2. HTML form to enter a city
3. HTML form to enter a category
4. HTML table to display all the categories and subcategories of places to visit
5. HTML table to display the places to visit & their information
6. HTML Images which display a picture of each “place to visit”

### Technologies Used:

1. Flask: My entire program runs within a Flask App.
2. Command line: Used for logging & tracking the flask app activity and to show the tree data structure & tree levels.

## Demo Video

Link to demo video:

<https://youtu.be/vgbScBzDWps>