TECHFEST 2021-22
Researcher Discovery Platform

Project Report
RD-216466

Name	Qualification
Anirudh CP	Second Year, VIT Chennai
Dr. Rajakumar Arul	Professor, VIT Chennai.

Overview:

The Researcher Discovery Platform is an aggregation tool to compile the details of various researchers. The application is capable of providing relevant search results based on the input of domain. The researches are listed in a results page along with key details and the ability to view more details if desired. Various universities across the country have been used as the sources. These can be used as filters. The results can be sorted according to H-Index, citations and name as per user preference.

Solution:

There are multiple indexing forums in the market and each list many details about a researcher. However, the most important feature of any complete data set is consistency and reliability. This is missing in many indexing forums as they fail to capture essential data.

Hence, the source for application must be a reliable source. The source selected here is the IRINS database. It lists researchers from across the country and currently has more than 65000 people in the dataset. A search in the IRINS website however fails to list researchers according to domain and is generally not up to expectations.

The solution is to hence, create our own database. The IRINS dataset creates a profile for every user. This can be scraped for vital information such as ORC-ID, H-Index, Citations and most important of all, the expertise of a researcher.

This can be used to find researchers in a given domain. A search based of the expertise of each researcher could lead to complete and proper results. These can be further filtered and sorted and then served to the user in a simple yet powerful UI. The dataset we have created can be updated and rebuilt quickly and efficiently as well. This can be used to query from easier and quicker than ever.

Approach:

The application has been written with scalability in mind. Hence, the technologies and frameworks used are reliable and robust.

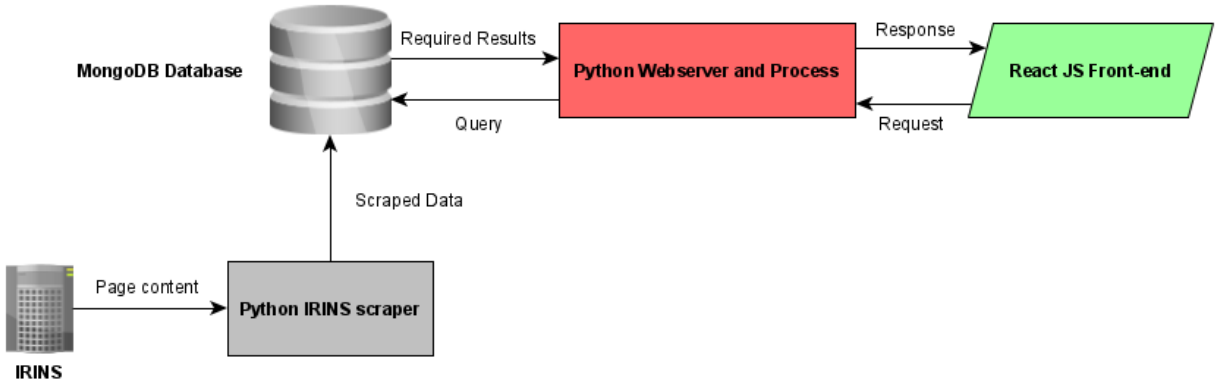
The front-end of the application has been done using React JS. The React JS framework allows for quick load times and ease of component reusability. The front-end has been written in a multi-page approach with redirects when required. The components are split in a reasonable manner to allow for simple data flows and ease of comprehension of code to aid in debugging, maintenance and scaling.

The back-end has been written in a python-based implementation. Since web servers are meant to be asynchronous, we have used a Tornado based web-server. This can be used to provide synchronous behavior when required and can be used in production environments as well.

The database where the data is stored is implemented in MongoDB. This is for fast load times and the ease of implementation due to a NoSQL implementation. This is vital to store the expertise which is required to search.

The database itself has the data provided to it via a scraper implemented in python. Scrapy has been used to boost speed and reliability. The scrapy scraper can be initiated at any time to insert the records into the database.

Application Architecture:



As explained above, the major components of the program interact with as shown in the diagram. A request is sent from the React JS front end. A request from the React JS front-end typically looks like JSON segment shown below:

```
{'query_name': 'Machine learning', 'page': 1, 'sort': "Sort by H-Index”,
'order': "Descending", 'filters': [], 'pageTotal': 5}
```

The above query is sent via a POST request by the API at the client-side to the server at the /results domain. The response is then waited for by the client.

The search query is then broken down in the client side and the major query details are extracted. The web-server then queries the database for the search query and applies any filters required. Once the appropriate filters are applied, it extracts the number of results needed and send the data to the client. A typical response from the server is as shown below:

```
{"query_name": 'Machine learning', "page": 1, "filters": [], "records":
<records>}
```

The <records> here indicated the multiple records sent to the client. Each record in the list is a JSON object and has multiple attributes as shown below for a sample result:

```
{'name': “ABC XYZ”, 'honorific': 'Dr', 'qual': ’QWERY Institute’, 'exp':
['machine', 'learning', 'data', 'analysis'], 'cite': 100, 'hindex': 12,
'orcid': '1234-5678-9012-3456', 'link': 'https://orcid/'1234-5678-9012-3456'}
```

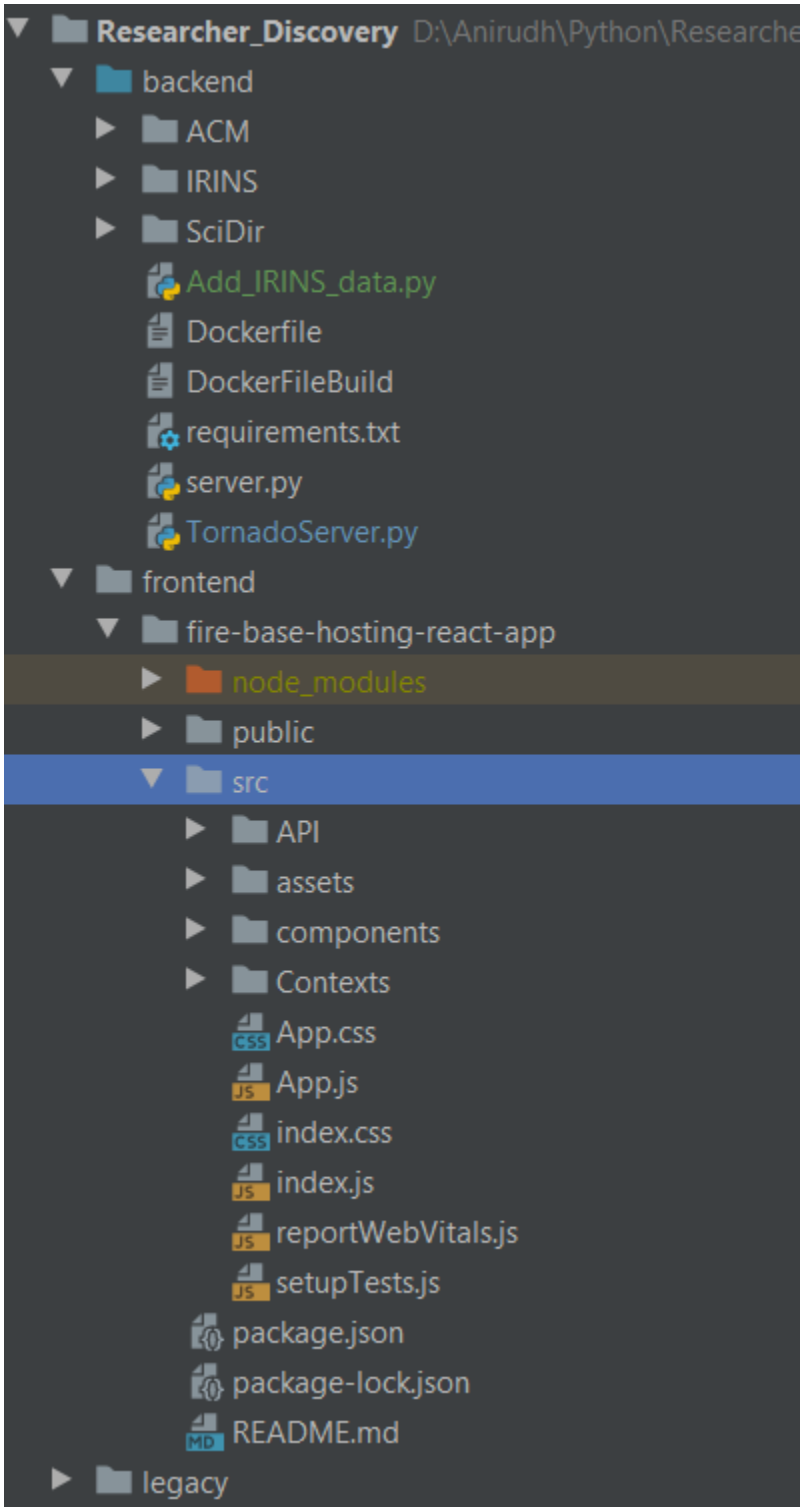
These results are then received by the client. They are then displayed in a simple and effective UI. Any changes to the query such as the application of a filter or changing of a page, leads to another call from the API and the process is then repeated.

In case of a field not existing, the application does account for this as well. For example, many researchers do not have an ORC-ID. This will lead to the link being the IRINS profile page. In case the client does receive such a record it will leave the field blank.

The application also has other scrapers that have been implemented in the past. The Science Direct and the ACM scrapers are regarded as the dynamic scrapers. They too can be configured to provide search results. However, since they have limitations in terms of speed and reliability, they are not set as the default. The python web-server however, does have the functionality to run these scrapers in runtime for results.

The dynamic scrapers are hence a future scalable feature in case the client desires more data from other sources as well. These can be upgraded to provide the same type of data as the IRINS dataset is capable of.

Implantation Approach and Documentation:



The application is divided into two major directories. The front-end and the back-end.

The back-end consists of multiple folders each for the scrapers which have been implemented in scrapy. There are also docker-files available. These can be used to deploy the application in a service such as Cloud Run or Kubernetes. The dockerfilebuild is for installing any requirements to the container. The dockerfile is used to actually dockerize the application backend. This can be run with a docker run command written in the dockerfile.

The other major files there are TornadoServer.py. This is the actual server in python which interacts with the other parts of the application. The Add_IRINS_data.py can be run at any time to add data from the IRINS webpages to our database.

The actual computation in the server is carried out in the post() method of the class ResultPageRequestHandler. Here is where the query is used to get the data and eventually send a response.

The front-end is a React JS framework. This has a source folder where the major functionalities are written. We can see the components in the ./components directory. The API used to interact with the back-end is in the API directory. The ./assets contains the images and the other data needed to run the program. The options.json file in the assets folder is used to load the options in the filtering menu in the front-end results page. A context has been used as well to send data between home and result page.

The components have been called one within another and lead to a tree. This can lead to data flow between these components as well. Each page has a component and each page is also split into multiple components.

The legacy folder is to hold the older scrapers code and the older flask server code as well.

Running the Application Locally:

- Open two terminals.
- Install the dependencies in the ‘backend/requirements.txt’.
- In the first terminal go the frontend/fire-base-hosting-react-app in the application. Use a cd command for this. Node modules must be installed here (IMPORTANT). Only the package-lock.json within the folder has been provided.
- Run ‘npm start’.
- Wait for the deployment server to be initialized.
- In the second terminal go the backend folder. Use a cd command.
- Run ‘python .\TornadoServer.py’
- Wait for message of 'Server has started listening...'

Technologies Used:

- Install **python** (latest version) from: https://www.python.org/downloads/
- Run the installation wizard
- Run python –version in command prompt and pip –version to check if both are installed properly.
- If not working check path, example: C:\Users\cpani\AppData\Local\Programs\Python\Python38-32\Scripts
- Run the following commands in the command prompt:
    - pip install scrapy
    - pip install selenium
    - pip install urllib3
    - pip install scrapydo
    - pip install tornado
    - pip install pymongo
- Install React JS for Front end. Reference (https://www.youtube.com/watch?v=0twjvW0c1r0)
- Install nodeJS from https://nodejs.org/en/
- LTS version is good, needs administrative permission for installing node JS.
- npm install -g create-react-app. Run this command to help create a simple React JS application. This is NOT required to run the RDP application.
- Open two terminals.
- Install the dependencies in the 'backend/requirements.txt'.
- In the first terminal go the frontend/fire-base-hosting-react-app in the application. Use a cd command for this. Node modules must be installed here (IMPORTANT). Only the package-lock.json within the folder has been provided.
- Run 'npm start'.
- Wait for the deployment server to be initialized.
- Go to http://localhost:3000/
- In the second terminal go the backend folder. Use cd backend.
- Run 'python .\TornadoServer.py'
- Wait for message of 'Server has started listening...'