Overview

The goal of this project is to provide a frontend to allow users to submit a type of calculation they want to perform on a molecular structure and display relevant data regarding the calculation. The frontend of this project is built on Angular 16 utilising NgRx for state-management and the open source java applet JSmol to display the chemical structures. The backend is built using FastAPI and PostgreSQL database.

Authentication

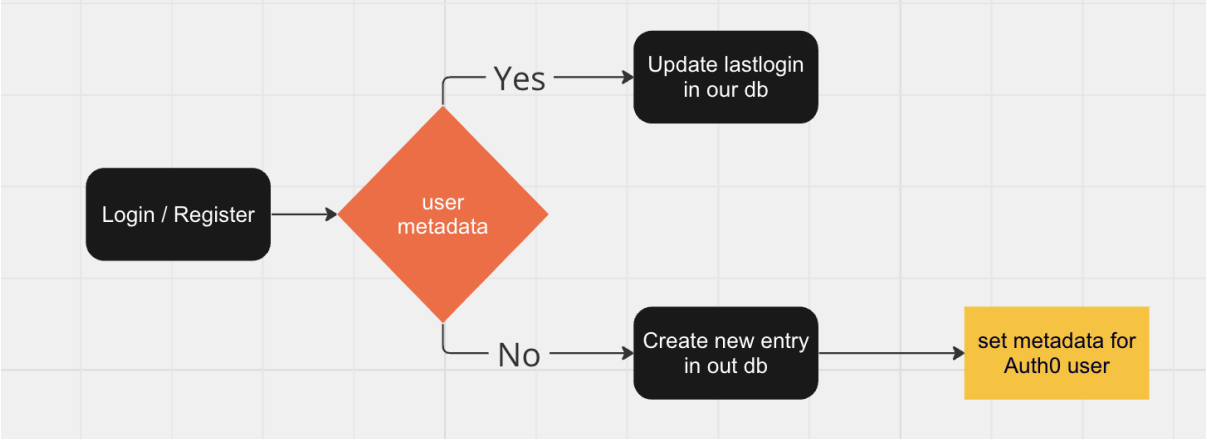
Our web application utilizes Auth0 for managing user authentication and login processes. The Auth0 account, designated as “ubchemica”, is overseen by Professor Thachuk.

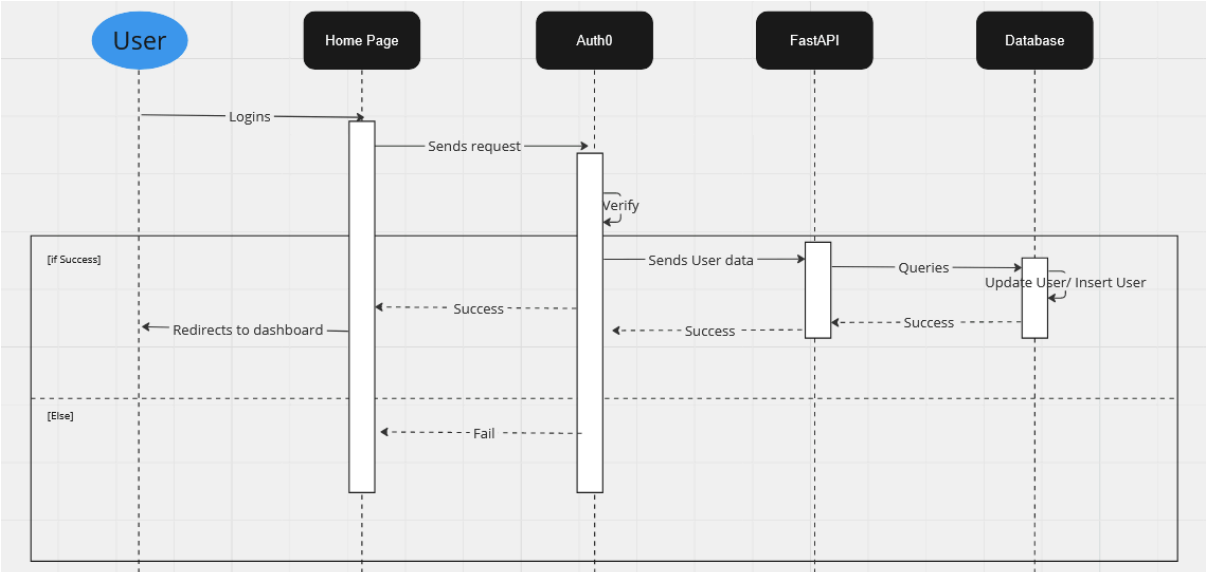
Configuration:

- Frontend: The connection to the Auth0 account is configured in the Angular project under the file UI/src/environments/environments.ts.
- Backend: The basic environment set up and token verification are handled in backend/app/util.py.
- All the sensitive credentials are stored in the .env file in each repository.

Synchronization between Auth0 database and our own:

- To manage user state upon successful authentication, we use an effect called handleUserAuthenticated$ located in UI/src/app/store/effects/user.effects.ts. This effect ensures that user data is current and properly synchronized with our backend services.





Frontend

Built with Angular16 using tailwindCSS for styling alongside daisyUI. Colour parameters are set inside tailwind.config file.

Repository link: [UBCC3/UI]

File Structure

The structure of the project is currently set up so that the directories in Features are the routes. Each directory in Features would be the container of that route and would consist of component folder which would contain all the components for that page

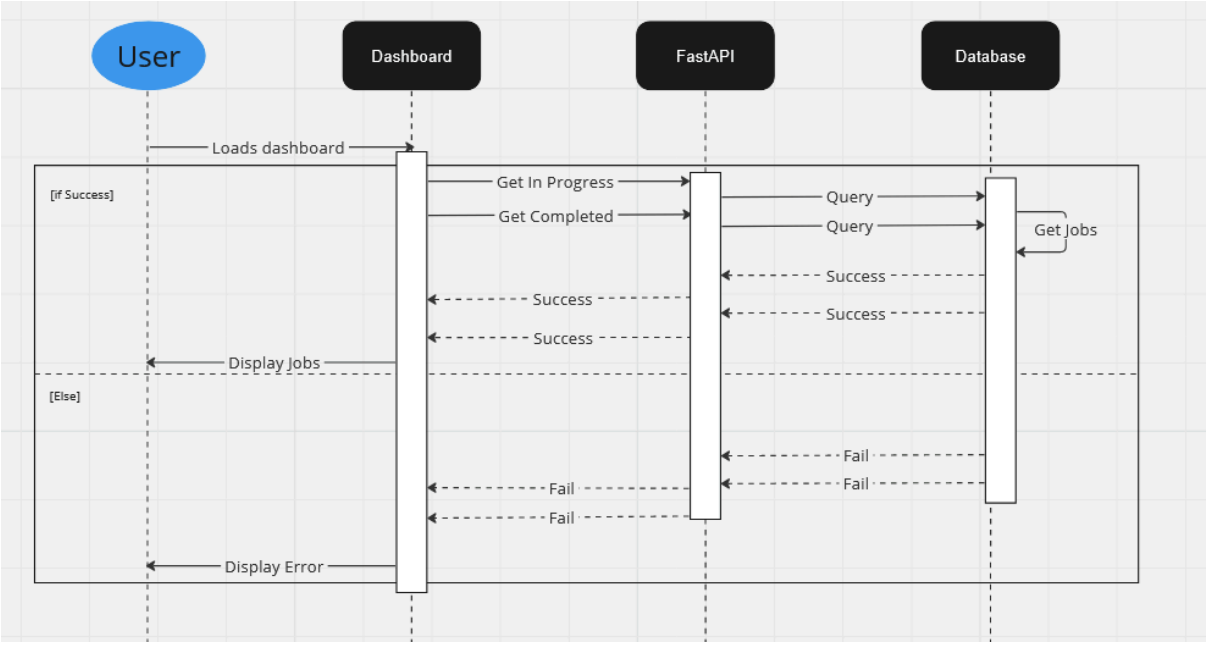
Shared directory contains code (components, models, services) shared across multiple part of the project

Store directory contains all NGRX related code

Dashboard

Upon successful authentication, the user will land on the dashboard which will display all the jobs that are currently in-progress or the jobs that have finished running.

Currently the fetch requests to pull the in-progress and completed jobs are only sent on initial load. The request to fetch the data to have the UI display updated state of the job will need to be implemented. Send fetch request every 60s might be a good interval



The search function is disabled on the frontend, backend functionality for the search bar has not been built yet.

New Calculation

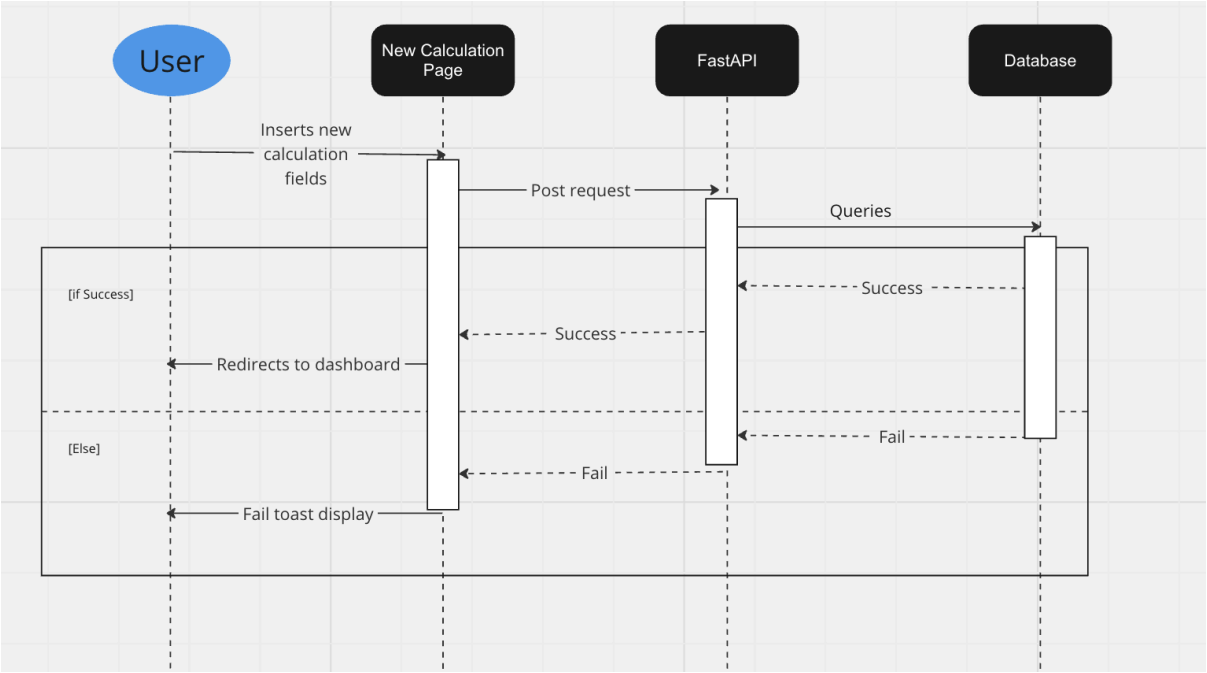
When a user creates a new calculation, they will first need to fill in a form with all the parameters for the calculation and then send a post request to the back end. The back end would then create a new entry in the Jobs table with the parameters submitted, if the source of the file that is submitted is uploaded from the user and not from an existing calculation, a new entry in the Structures table would also be created as well as a new entry in the s3 bucket would be uploaded.

Buttons to add a file by using 3rd party database or from previous calculation is currently commented out as the specification have not been fully flushed out.

The help icons beside each field on the form would also need to be discussed on what description should be put beside each one.

In the more settings drop down, the values for wave theory are just placeholders, and need to get specifications on what values to put as well as what other fields should be included in the more settings section.

Sample testing data: UI/src/assets/1aho.pdb



Navbar

Help icon and notification icon on the navbar is currently disabled, no specifications have been decided on what the behaviour of the icon should be (tooltip on hover or modal pop up on click) or what the content should include.

Notification specifications have not been discussed (what is considered a notification, what would be the CTA for the notifications and what kind of information would be included in a notification). Backend also needs to be updated to handle the notifications

Edit Structure

The CTA on the navbar are all placeholders other than the back CTA. Specifications on what to display and the functionality should discussed

Result

Upon the completion of a job, users can access detailed results by interacting with the job's entry. Clicking on the job name on the Dashboard page redirects the user to the results page, which displays comprehensive information related to the job. The results page presents general information about the job retrieved from our database, along with specific results generated from the PSI4 cluster.

Job Detail Component

- Location: UI/src/app/features/result/components/job-detail

- Functionality: This component is responsible for fetching and displaying detailed job information from the database using the job ID. It renders critical data for user review directly on the UI.

Download ZIP Component

- Location: UI/src/app/features/result/components/download-zip
- Functionality: This component provides a functionality for users to download a ZIP file containing all the result files generated by the PSI4 cluster for the specific job. It offers a button, enabling easy access to download these files.

Result Detail Component (lots of future work need to be done)

- Location: UI/src/app/features/result/components/result-detail
- Functionality: This component is intended to handle the downloading of the result.json file (more information about the file) from the S3 bucket, which contains structured results data for the job. Upon successful download, it aims to render all information dynamically based on the content of the result.json file, allowing detailed inspection of the job results.
- Note: Currently, the actually api route which used to download result.json file from s3 is replaced by a testLocalFilePath for testing UI purposes.

JSmol

To install JSmol, can follow instructions from the documentations
JSmol scripting documentations

To load JSmol into the component:
Calling the function in ngAfterViewInit insures everything from the component is rendered before trying to generate the code for jsmol

```
 ngAfterViewInit(): void {
       this.appletElement = this.appletContainer.nativeElement;

       this.loadJSMolWithPromise()
           .then((appletHtml) => {
               this.appletElement.innerHTML = appletHtml;

               if (this.file) {
                   setTimeout(() => {
                       this.loadFile();
                       this.setRightClickMenuAccess();
                   }, 100);
               }
           })

           .catch((err) => console.error('error loading jsmol: ',
err));
    }
```

```
loadJSMolWithPromise(): Promise<string> {
       return new Promise((resolve, reject) => {
           const script = document.createElement('script');

           script.src = '../../../../assets/jsmol/JSmol.min.js';

           script.onload = () => {
               // reference to jmol applet object
               this.appletObject = Jmol.getApplet('jsmolApplet',
info);

               // get html for jmol applet
               const appletHtml =
Jmol.getAppletHtml(this.appletObject);

               resolve(appletHtml);
           };

           script.onerror = (error) => {
               console.log('error', error);
               reject(error);
           };

           // add script to html
           this.renderer.appendChild(document.body, script);
       });
   }
```

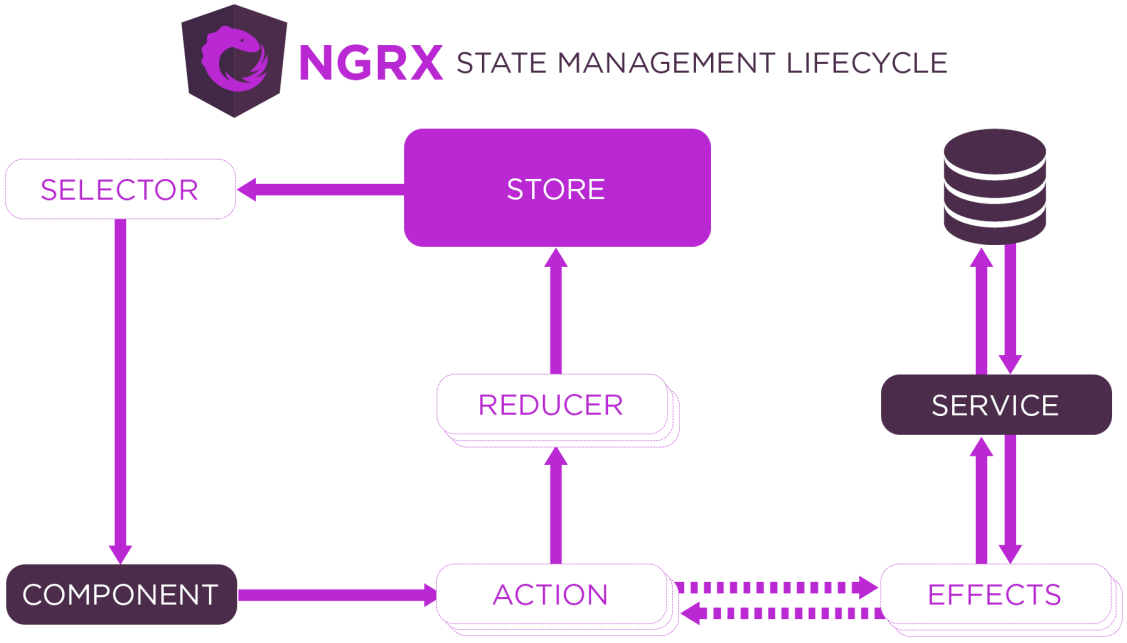
To load a file into JSmol:

```
loadFile() {
       if (this.file) {
           if (this.source == SourceEnum.CALCULATED) {
               const fileContentsAppended = `load data "model"
${this.file} end "model"`;
               Jmol.script(this.appletObject,
fileContentsAppended);
           } else {
               const reader = new FileReader();
               reader.onload = (event) => {
                   const fileContents = event.target?.result as
string;
```

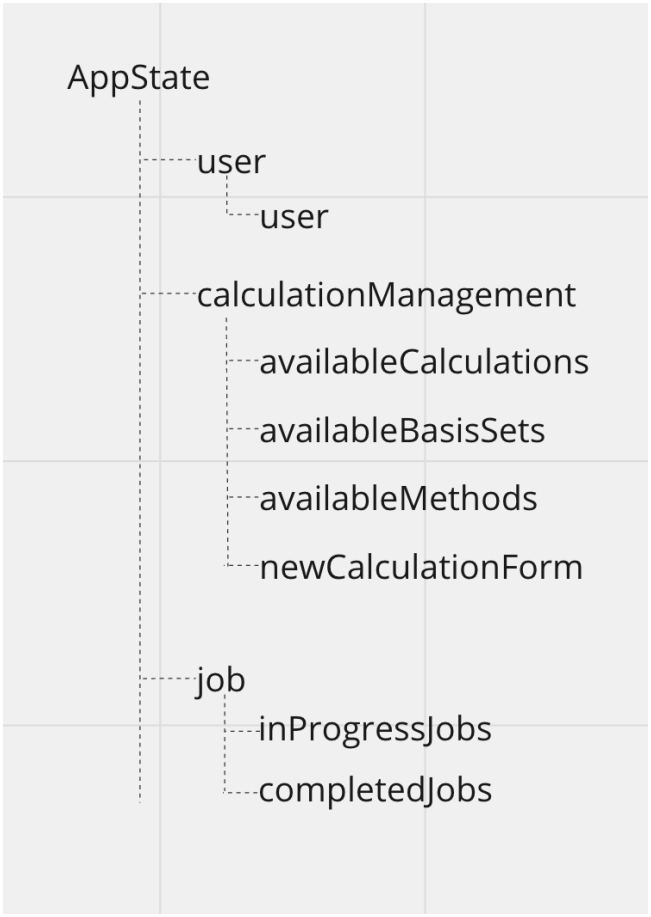
```
                    const fileContentsAppended = `load data "model"
${fileContents} end "model"`;
                    Jmol.script(this.appletObject,
fileContentsAppended);
                };

                reader.readAsText(this.file);
            }
        }
    }
```

NgRx

The overall flow of the state-management will follow:



The AppState of the current project looks like:



Notification should also be included in the AppState when specifications have been figured out. Can be part of the user state or a separate state.

Environmental variables (.env)

There are environmental variables that are read from a .env file in the root for the backend folder.
This is what the file should look like:

```
AUTH0_DOMAIN = auth0domain.ca.auth0.com
AUTH0_CLIENTID = auth0_clientid
AUTH0_SECRET = auth0_secret
AUTH0_ISSUER = https://auth0domain.ca.auth0.com/
AUTH0_ALGO = RS256
BASE_URL = http://localhost:8000
FE_URL = http://localhost:4200
```

Testing

Testing should be added to the project, currently there are only boilerplate test files made. NgRx testing should also be added to ensure the selectors are correct as well.

Backend

The current idea for the backend structure is to have FastApi handle all the crud operations and have the AWS built solution to run and select jobs from the DB and run them using modified PSI4.

Repository link: UBCC3/backend

FastAPI

Currently the FastAPI backend is used to handle CRUD requests for the application.

File Structure

Database

Each schema has its own management.py file which includes all the functions that relate to the schema query.

Routers

Each route would be in their own route file. All api endpoints related to that route would be handled in that file. Each endpoint is in the form of a python function like so:

```
@router.get("/", response_model=Union[list[JobModel], JwtErrorModel])
async def get_jobs(
   response: Response,
   token: str = Depends(token_auth)
   ):
   return job_dicts
```

Cluster

The cluster folder manages all functionalities related to interacting with the PSI4 cluster. This includes job submission, status checks, and result handling.

Cluster Communication
In app/util.py, we implemented a function called cluster_call(), which serves as a centralised function for all communications with the PSI4 cluster. It constructs command data, initiates an SSH connection, and handles the execution of Python scripts on the cluster, managing both input and output.

Key Functionalities in cluster.py:
Job Submission:

- Function: submit_job()
- Trigger: Initiated when a new calculation is submitted through the UI.
- Description: Handles the submission of jobs to the PSI4 cluster.

Job Status Checks:

- Function: check_jobs_status()
- Trigger: Periodically invoked by a BackgroundScheduler in app/main.py.
- Description: This function fetches the current status of running jobs from the database, sends them for status checking to the PSI4 cluster, and updates the job records in the database based on the feedback received.

Result Handling:

- Function: upload_results()
- Trigger: Initiated by check_jobs_status() function when a job status updated to completed, failed, or canceled.
- Description: Asynchronously manages the uploading of job results from the PSI4 cluster. It handles both archives and result.json result data, and ensures they are correctly uploaded to designated storage paths.

Job Cleanup:

- Function: clean_results()
- Trigger: Initiated when both archive and json result data are uploaded to s3 with a success response status.
- Description: Responsible for cleaning up result files on the cluster side once a job is definitively concluded

Job Cancelation:

- Function: cancel_job()
- Trigger: Initiated when user clicks on the cancel button on the UI.
- Description: Cancels the running job on the PSI4 cluster.

Relational Database

For a relational database, postgreSQL is used. The production uses Amazon's RDS with traffic only allowed through our EC2 instance.
For local development, setup a regular SQL server using postgre and run the table definition code inside "datamodel.sql" in the UI repository to set up the schemas.

Environmental variables (.env)

There are environmental variables that are read from a .env file in the root for the backend folder.
This is what the file should look like:

```
AUTH0_DOMAIN = auth0domain.ca.auth0.com
AUTH0_CLIENTID = auth0_clientid
AUTH0_SECRET = auth0_secret
AUTH0_AUDIENCE = https://auth0_audience
AUTH0_ISSUER = https://auth0domain.ca.auth0.com/
AUTH0_ALGO = RS256
RDS_PASSWORD = database password
RDS_HOST = database host (aka localhost)
RDS_USERNAME = database username
RDS_PORT = database port (5432 for postgre)
RDS_DBNAME = database name
```

```
AWS_ACCESS_KEY_ID = AWS_access-key
AWS_SECRET_ACCESS_KEY = aws-secret-key
AWS_SESSION_TOKEN = aws-session-token
AWS_REGION_NAME = ca-central-1
S3_BUCKET = ubchemica-bucket-1
BASE_URL = http://localhost:8000
FE_URL = http://localhost:4200
CLUSTER_LOC = ../cluster-api/main.py
```

Open Babel

Open Babel is used to convert different chemical file extensions. Need to add this into the backend so that the files are consistent. Using Open Babel all incoming structure files are converted to xyz before run on psi4 or stored in bucket storage (S3).

Installing Open Babel

There are two parts to installing Open Babel:

1. Installing the appropriate binaries on the executing machine.
2. Installing the python library using pip

Installing the Binaries

Depending on the operating system you are running (in deployment we currently use Ubuntu) the binaries may be available as a package. This is true of most mainstream linux distributions as well as Microsoft Windows.
If that is not the case, consult the link above, specifically the “Compiling Open Babel” section, however, this is not a process that is easy or necessary in most cases.

Installing the openbabel Library

The library has been added to the requirements.txt file in the backend repository and should be installed upon running the appropriate code, documented in the repository.

To install the library individually run:

```
pip install openbabel-wheel
```

PSI4

The PSI4 library needs to be edited so the output file generated from it can be formatted to contain only the data we need as well as including api calls to the FastAPI backend to store simple results (energy, etc.) directly into a column on the database.

Currently trying to make a local build from the forked source code running into issues trying to import the python library, no issue using the library directly from the terminal.

Planned workout for this is to try installing a Psi4 onto a fresh install of a Ubuntu machine and then trying to edit the source code from the installed path.

Building From Source Code

1) Create and activate conda env:
   *yaml files can be found in UBCC3/psi4/devtools/conda-envs

```
 conda env create -n p4dev -f /path/to/os_conda_env.yaml --solver
libmamba && conda activate p4dev
```

2) Configure psi4 against activated conda env:

```
cmake -S. -Bobjdir_p4dev -GNinja
```

3) Move to build dir and build

```
cd objdir_p4dev && cmake --build .
```

4) Prints the paths that should be executed

```
stage/bin/psi4 --psiapi
```

5) Run a test

```
psi4 ../tests/tu1-h2o-energy/input.dat
```

Nginx

For deployment, the Elastic Compute instance runs Nginx as a web server and reverse proxy. It routes all HTTPS traffic to the backend port while serving the static Angular files that are built.

Testing

The testing on the backend is done using unittest, each function should have test cases for it.
Currently the functions for jobs and structure do not have tests for them yet.

Cluster Scripts

Repository link: UBCC3/cluster-api

This repository contains Python scripts designed for deployment on the PSI4 cluster, facilitating interactions with the project backend. It includes essential scripts such as submitting a job, checking a job status, data processing, and results generation.

Communication with Cluster

The cluster used to run the calculations are Beluga computers provided by Alliance Canada for research facilities. Each of these clusters uses SLURM for job queuing. To run a job, a user has to enter their console and run a SBATCH bash script. This script will then run on the head node which assigns however many nodes are needed to the job, schedule it in queue, and eventually run and return its results with some data for the job (time elapsed, num of nodes, ...). To avoid misuse and possible security compromise, the user console is accessible through very strict means: either a physical key or limited one-time ssh commands run through specific IP addresses.

Our EC2 instance is on the whitelist for IP addresses allowed to run commands on the cluster. However, the list of commands allowed is very limited. As such, in order to convey the information needed with the cluster using the cluster_call function we pass certain information needed for the cluster via command line arguments as a string.

Sample Input

There is a sample_input.json in the cluster_api repository with an example of what the input sent to the cluster for a job submission will look like.

Each json dictionary will have two keys: ‘action’, and ‘parameters’. Action will be one of 5 words: submit, cancel, upload, check, clean up. Parameters will be details of the job pertaining to the action word.

Sample Output

Upon the completion of a job, various result files are generated by the computational processes. A dedicated function then compiles these files into a result.json file, which is specifically designed to facilitate the rendering of job results to the UI. The sample.json file serves as a template, demonstrating the expected structure and content of the result.json file. This template is crucial for ensuring consistency in how information is formatted and presented, guiding the integration of UI components that display job results.

Job information section
This section contains the data that is obtained from the cluster while not being stored in our database. This section is currently treated as a placeholder for potential future data fields, and is adaptable to include any additional fields as needed.

Structure information section

- “visualData”

    - “dataContent”: Intended to hold a long string (e.g. the content in the testing file 1aho.pdb), which is used to load data into the JSmol viewer on the UI.
- “genericData”
  Contains relevant information to the structure.
- “tableData”
  Represents structured data, detailing various properties of the structure.
    - “Is_graphable”: A boolean value indicates whether the data is graphable or not. Some future work that needs to be done is to determine a way to know which columns to graph and which are the 'x' and 'y' values.
    - “headers”: Defines the columns of the table, with “span ”indicating the number of columns a header label should span.
    - “footers”: An optional field that provides additional insights or analysis about the results presented in the table.
    - “Clickable”: An optional field that specifies which row “id” and “index” of the array “value” for row is clickable with the action “dataContent”, which would be a function name that already written in the result folder of the UI repository to perform adding a layer to the basic structure.

Example “tableData:

```
{
    "tableName": "Experimental Vibrational Frequencies",
    "is_graphable": "true",
    "headers": [
        {"label": "name", "span": 1},
        {"label": "predefined basis sets", "span": 1},
        {"label": "energies in hartrees", "span": 1}
    ],
    "rows": [
          {"id": 1, "value": ["Pentane, 2-methyl-", "G1", "-236.524913"]},
          {"id": 2, "value": ["Pentane, 2-methyl-", "G2MP2", "-236.529269"]},
          {"id": 3, "value": ["Pentane, 3-methyl-", "G1", "-236.523934"]}
    ],
    "footers": "e.g. analysis about the table result",
    "clickable": [
          {"id": 1, "index": 2, "dataContent": "sample string"},
          {"id": 2, "index": 2, "dataContent": "sample string"},
          {"id": 3, "index": 2, "dataContent": "sample string"}
    ]
},
```

Corresponding UI:

Experimental Vibrational Frequencies		
name	predefined basis sets	energies in hartrees
Pentane, 2-methyl-	G1	-236.524913(clickable)data
Pentane, 2-methyl-	G2MP2	-236.529269(clickable)

Pentane, 3-methyl-	G1	-236.523934(clickable)
Sample footer		

QCEngine

QCEngine is an interface which facilitates running psi4 inside Python code. The code that runs QCEngine is called run_qcengine on the cluster-api repository. Its package is available through PyPi and can be installed through pip.
QCEngine can return the results on its computation as a dictionary in Python which we then parse through to create the sample output json file. We also dump the entire dictionary in a ".out" file which is then zipped and archived in S3.

CI/CD Pipeline

Docker

Although there were attempts to containerize the backend server, the DockerFile is not yet functional and needs further testing and tuning.

Github

Github actions are running per Pull Request to ensure tests and lints pass before merging is allowed. Currently only the front-end actions are set up to run tests and lint on the changes to ensure the build succeeds.

Backend and cluster actions have yet to be implemented.

Terraform

Terraform has been discussed to handle the deployment of the project. Will be incorporated into the project at a later date.