Project

Deadlines:

25.11.2024 - 1 + 2 + 3

2.12.2024 - 4 + 5 + 6

9.12.2024 - 7 + 8

16.12.2024 - 9

Substructure search

Substructure search of chemical compounds is a crucial tool in cheminformatics, enabling researchers to identify and analyze chemical structures containing specific substructures. This method is widely applied in various fields of chemistry, including drug discovery, materials science, and environmental research. Substructure search helps scientists and engineers identify compounds with desired properties, predict reactivity, and understand the mechanisms of chemical reactions.

Modern chemical compound databases contain millions of entries, making traditional search methods inefficient and time-consuming. Substructure search utilizes algorithms that allow for quick and accurate identification of compounds with specified structural fragments. These algorithms are based on graph theory and the use of SMARTS (SMiles ARbitrary Target Specification) codes, ensuring high performance and precision in the search process.

SMILES

A key element in the representation of chemical structures is the Simplified Molecular Input Line Entry System (SMILES). SMILES is a notation that allows a user to represent a chemical structure in a way that can be easily processed by computers. It encodes molecular structures as a series of text strings, which can then be used for various computational analyses, including substructure searches. The simplicity and efficiency of SMILES make it a widely adopted standard in cheminformatics.

Here are some examples of SMILES notation:

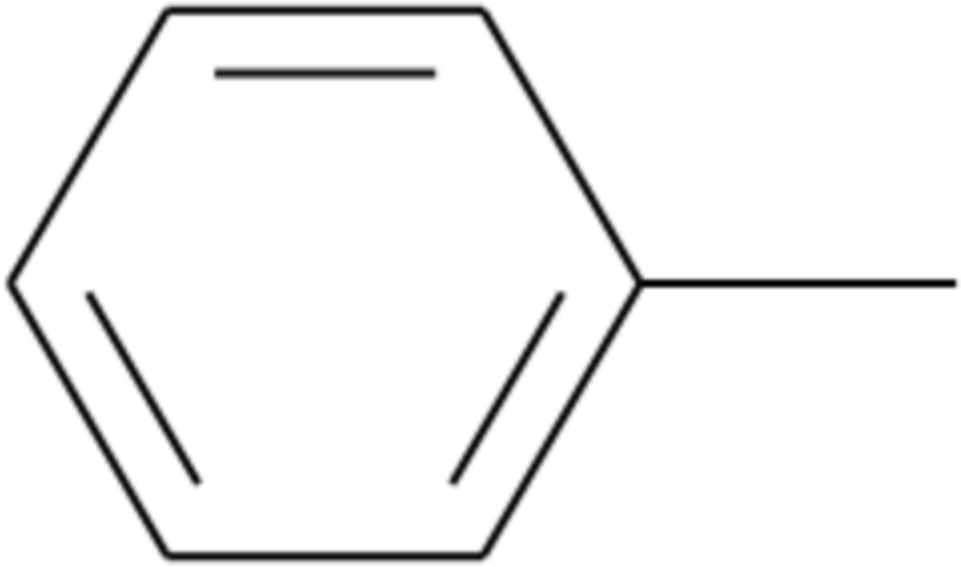
- Water (H_2O): O
- Methane (CH_4): C
- Ethanol (C_2H_5OH): CCO
- Benzene (C_6H_6): c1ccccc1
- Acetic acid (CH_3COOH): CC(=O)O
- Aspirin ($C_9H_8O_4$): CC(=O)Oc1ccccc1C(=O)O

Examples of Substructure Search

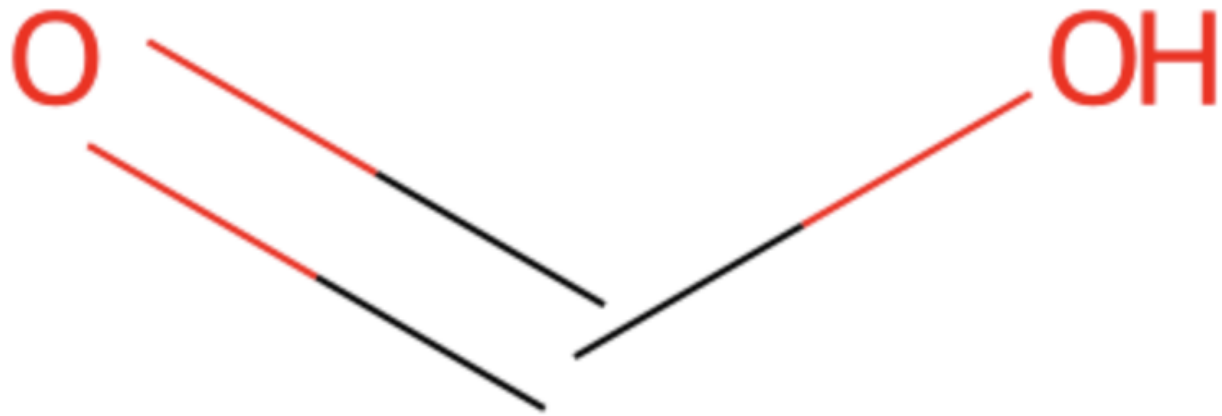
Below are some examples of substructure searches with visual representations:

1. Searching for the Benzene Ring:
    - Substructure (Benzene): c1ccccc1

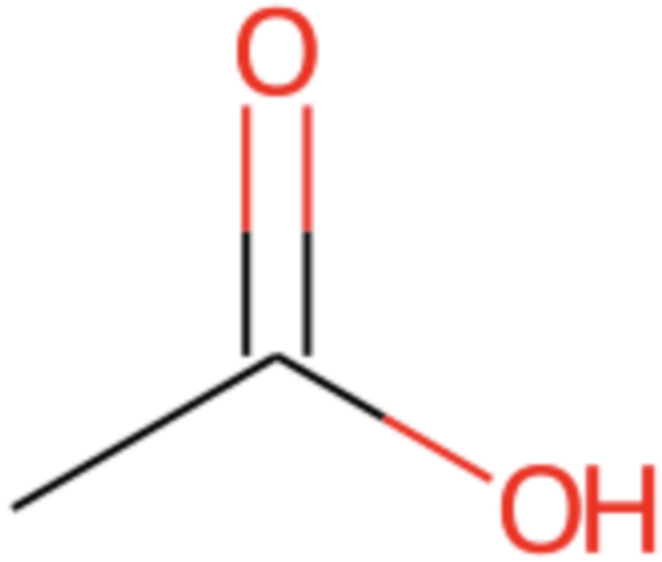
    - Example of Found Compound (Toluene): Cc1ccccc1



2. Searching for a Carboxylic Acid Group:
    - Substructure (Carboxylic Acid): C(=O)O



- Example of Found Compound (Acetic Acid): CC(=O)O



These examples illustrate how substructure searches can be used to find compounds containing specific functional groups or structural motifs. By using SMILES notation and cheminformatics tools, researchers can efficiently identify and study compounds of interest.

1. Introduction to RDKit

RDKit is an open-source cheminformatics software library that provides a wide range of tools for working with chemical informatics, particularly focusing on molecule representation, manipulation, and visualization. It is widely used in both academia and industry for tasks such as molecular modeling, chemical database searching, and molecular property prediction. One of RDKit's key strengths is its ability to seamlessly integrate with Python, making it accessible and flexible for a variety of applications.

In this introduction, we will explore the basic functionalities of RDKit, focusing on how to represent, manipulate, and visualize chemical structures using Python. We will cover the following topics:

1. **Installation of RDKit**
2. **Basic molecular representations**
3. **Substructure search**

4. **Molecular visualization**

1. Installation of RDKit

To install RDKit, you can use the following commands in your terminal.

```
pip install rdkit
```

2. Basic Molecular Representations

RDKit allows you to create and manipulate molecular structures easily. Here's how you can create a molecule from a SMILES string and visualize it:

```
from rdkit import Chem
from rdkit.Chem import Draw

# Create a molecule from a SMILES string
smiles = "CCO"  # Ethanol
molecule = Chem.MolFromSmiles(smiles)

# Draw the molecule
Draw.MolToImage(molecule)
```

3. Substructure Search

RDKit can be used to perform substructure searches, identifying specific functional groups or substructures within a molecule.

```
from rdkit import Chem

# Define the molecule and the substructure to search for
benzene = Chem.MolFromSmiles("c1ccccc1")
ethanol = Chem.MolFromSmiles("CCO")

# Perform the substructure search
match = ethanol.HasSubstructMatch(benzene)
print("Benzene ring found in ethanol:", match)
```

4. Molecular Visualization

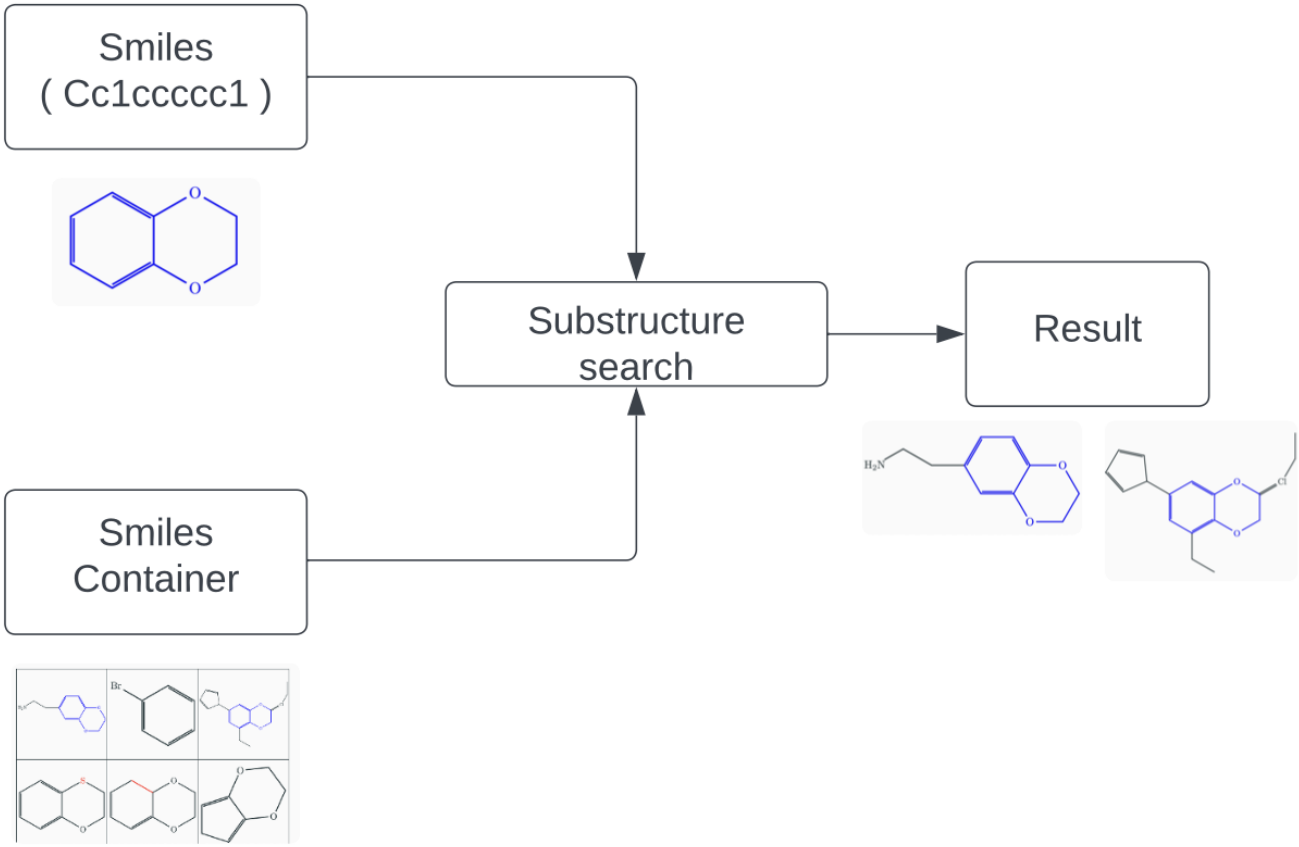
RDKit provides several options for visualizing molecules. You can visualize individual molecules or draw multiple molecules in a grid.

```
from rdkit.Chem import Draw
```

```
# Create a list of molecules
smiles_list = ["CCO", "c1ccccc1", "CC(=O)O", "CC(=O)Oc1ccccc1C(=O)O"]
molecules = [Chem.MolFromSmiles(smiles) for smiles in smiles_list]

# Draw the molecules in a grid
img = Draw.MolsToGridImage(molecules, molsPerRow=2, subImgSize=(200, 200),
returnPNG=False)
img.show()
```

Homework

Now you have everything to implement the substructure search function:



Implement a function in file that takes two arguments:

1. List of molecules as SMILES strings
2. Substructure as SMILES string Function should return a list of all molecules from argument 1 that contain substructure from argument 2

```
substructure_search(["CCO", "c1ccccc1", "CC(=O)O",
"CC(=O)Oc1ccccc1C(=O)O"], "c1ccccc1")
["c1ccccc1", "CC(=O)Oc1ccccc1C(=O)O"]
```

2. Introduction to FastAPI

FastAPI is a high-performance web framework for building APIs in Python. It combines ease of development, speed, and modern language features. Built on standards like OpenAPI and JSON Schema, it's a powerful tool for creating small to large-scale applications.

Installation

To get started with FastAPI, you'll need Python 3.7+ installed on your system. Follow these steps:

1. Install FastAPI:

Use pip to install FastAPI in your project environment:

```
pip install fastapi
```

2. Install a Server:

FastAPI itself does not include a web server. Use Uvicorn, a lightweight ASGI server:

```
pip install uvicorn[standard]
```

A Simple FastAPI Application

Here's an example of a basic FastAPI application:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root():
    return {"message": "Welcome to FastAPI!"}

@app.get("/items/{item_id}")
def read_item(item_id: int, q: str = None):
    return {"item_id": item_id, "q": q}
```

Running the Application

1. Save the above code in a file called main.py.
2. Run the application with Uvicorn:

```
uvicorn main:app --reload
```

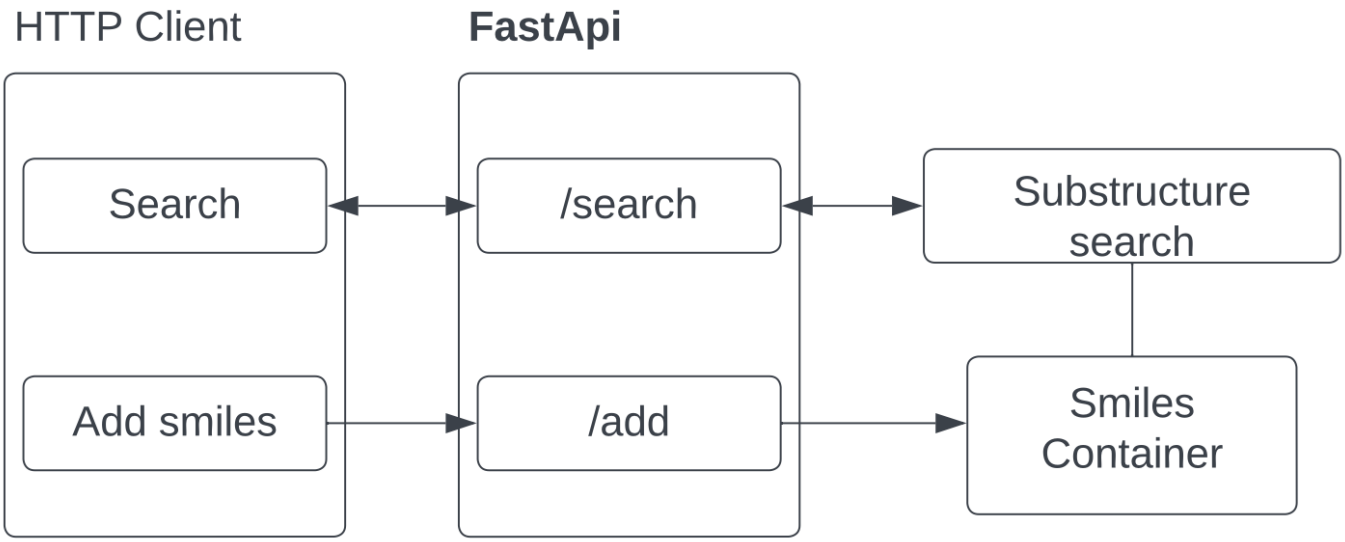
The --reload option enables auto-reloading when code changes are detected.

3. Once the server is running, you can access your API:

- Swagger UI Documentation: http://127.0.0.1:8000/docs
- ReDoc Documentation: http://127.0.0.1:8000/redoc

Homework

Now you need to make a server using FastAPI framework to use our algorithm.



It is necessary to implement the following API methods:

- Add molecule (smiles) and its identifier
- Get molecule by identifier
- Updating a molecule by identifier
- Delete a molecule by identifier
- List all molecules
- Substructure search for all added molecules
- **[Optional]** Upload file with molecules (the choice of format is yours).

We are not yet familiar with databases, so we can use standard data structures and store them in RAM.

3. Docker

Docker is a platform for building, sharing, and running containerized applications. Containers package an application along with all its dependencies (libraries, tools, configurations) into an isolated environment. This ensures reproducibility and portability, regardless of where the application is deployed.

Example:

```
FROM python:3.12
RUN pip install rdkit

WORKDIR /app/
```

```
COPY requirements.txt .
RUN pip install -r requirements.txt

COPY . .

EXPOSE 8000

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

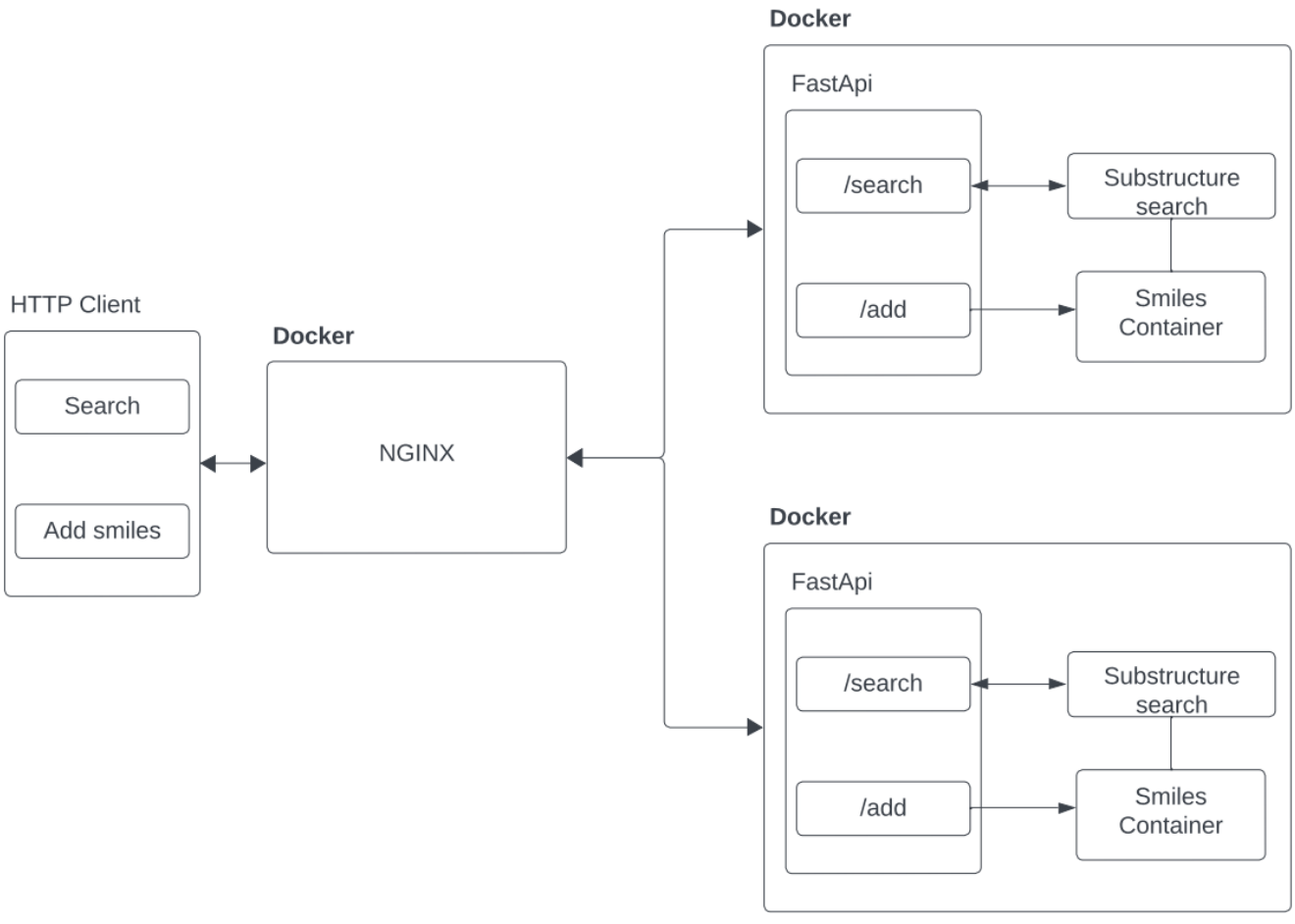
Homework

It is necessary to implement Dockerfile and docker compose for your Python FastAPI application with installed rdkit.

nginx

Nginx (pronounced “Engine-X”) is a high-performance, open-source web server and reverse proxy server used for serving web content, managing traffic, and improving application performance.

Originally developed to address the C10k problem (handling 10,000 simultaneous connections), Nginx is widely adopted for building reliable, scalable systems.



- Add method to check balancing:

```
@app.get("/")
def get_server():
    return {"server_id": getenv("SERVER_ID", "1")}
```

- Configuration file for nginx **nginx/default.conf**:

```
upstream webapp {
    server web1:8000;
    server web2:8000;
}

server {
    listen 80;

    location / {
        proxy_pass http://webapp;
    }
}
```

- Define services, networks, and volumes in `docker-compose.yml`.

```
version: '3.8'
services:
  web1:
    build: ./src
    volumes:
      - ./src:/src
    environment:
      SERVER_ID: SERVER-1

  web2:
    build: ./src
    volumes:
      - ./src:/src
    environment:
      SERVER_ID: SERVER-2

  nginx:
    image: nginx:latest
    ports:
      - "80:80"
    volumes:
      - ./nginx:/etc/nginx/conf.d
    depends_on:
      - web1
      - web2
```

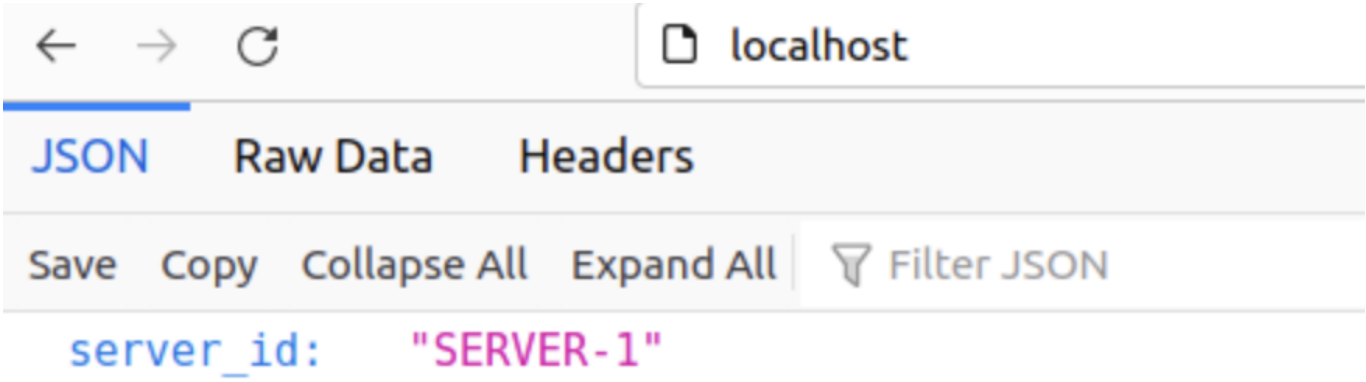
- Run the following to build and start the application:

```
docker-compose up --build -d
```

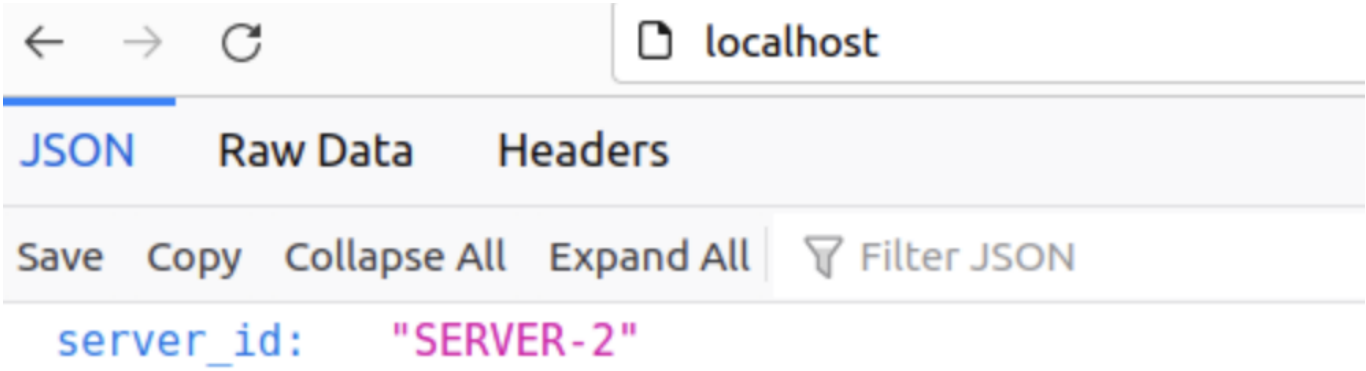
- View the running containers:

```
docker-compose ps
```

- You should be able to access the application on your browser.



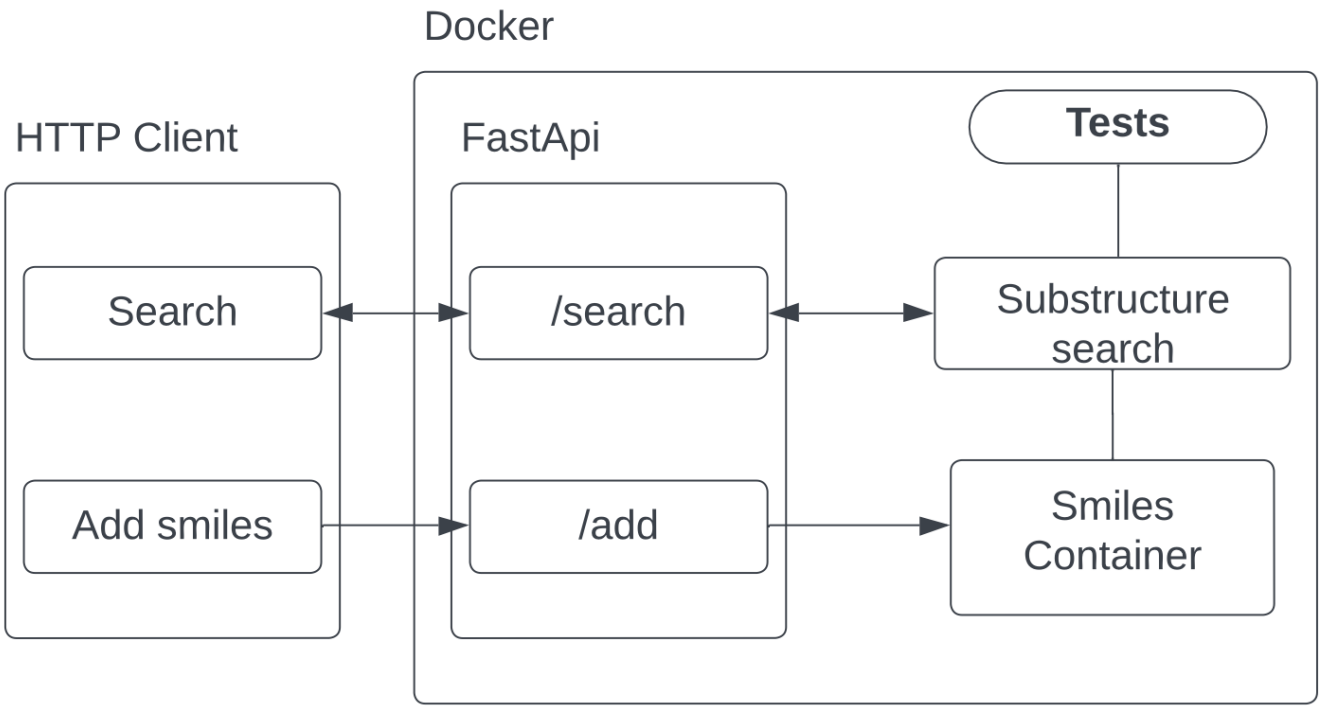
- Refresh to confirm that the load balancer distributes the request to both web containers.



4. Testing

Homework

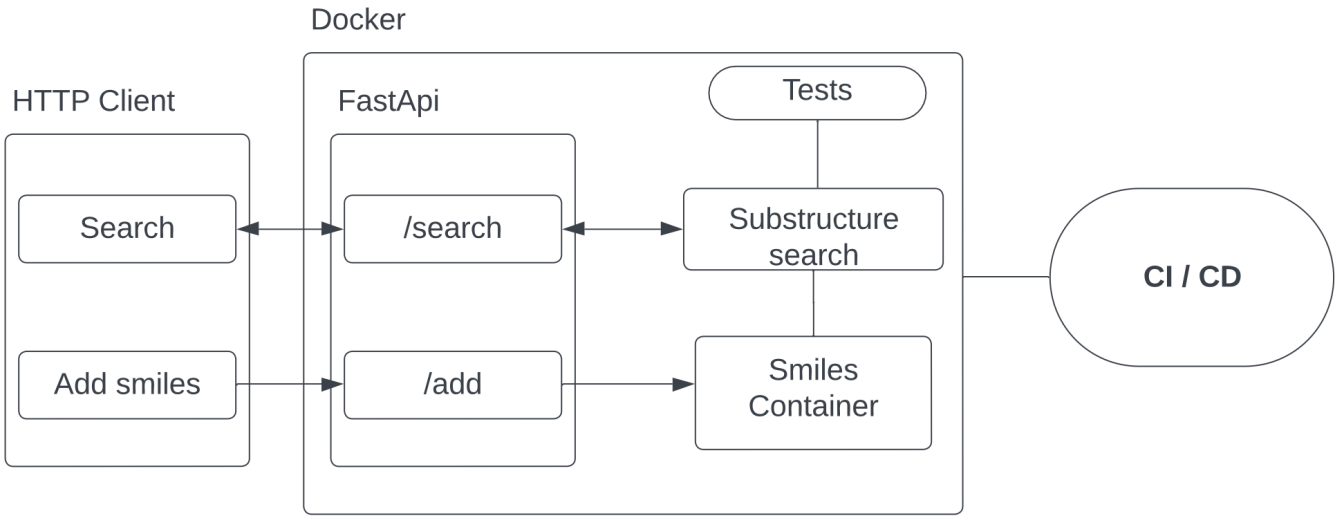
The most challenging part of our molecule data storage is the substructure search. The goal of this homework is to cover it with tests to make sure that this function works correctly.



5. Continious integration

Homework

Add CI using github actions to automate the process of testing your code after each push to the repository. Also add a static analyzer using flake8.



6. Database

Previously we stored our molecules in RAM using a dictionary data structure. This was quite inconvenient, as each time the application was restarted, our molecules were cleared.

Example

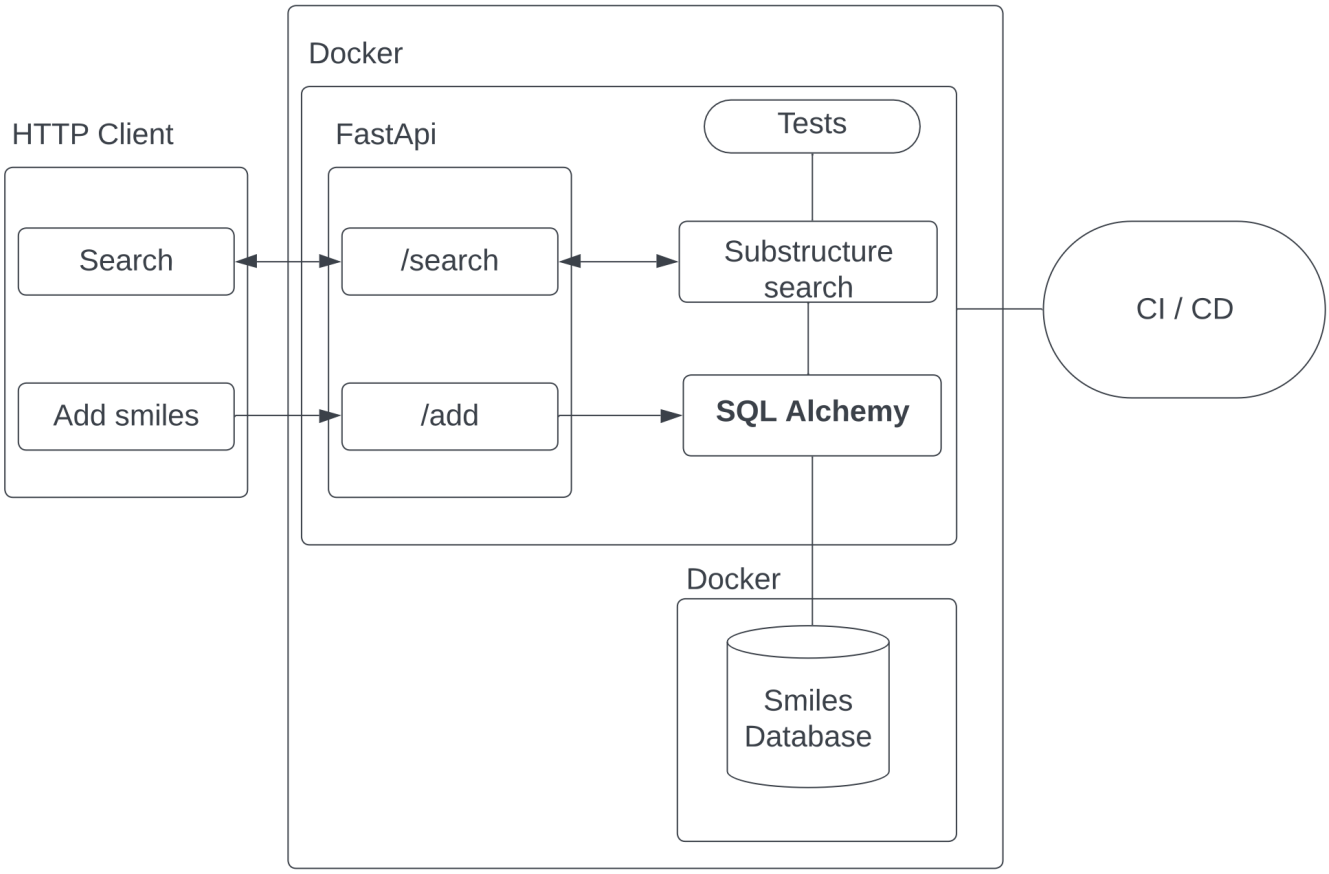
Modify `docker-compose.yml`. It will be better to store credentials to `.env` file

```yaml
version: '3.8'

services:
  db:
    image: postgres:15
    container_name: postgres
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: db
    ports:
      - "5432:5432"
    volumes:
      - db_data:/var/lib/postgresql/data

  web:
    build: .
    container_name: fastapi_app
    ports:
      - "8000:8000"
    depends_on:
      - db

volumes:
    db_data:
```

Use SQLAlchemy to setup database connection.

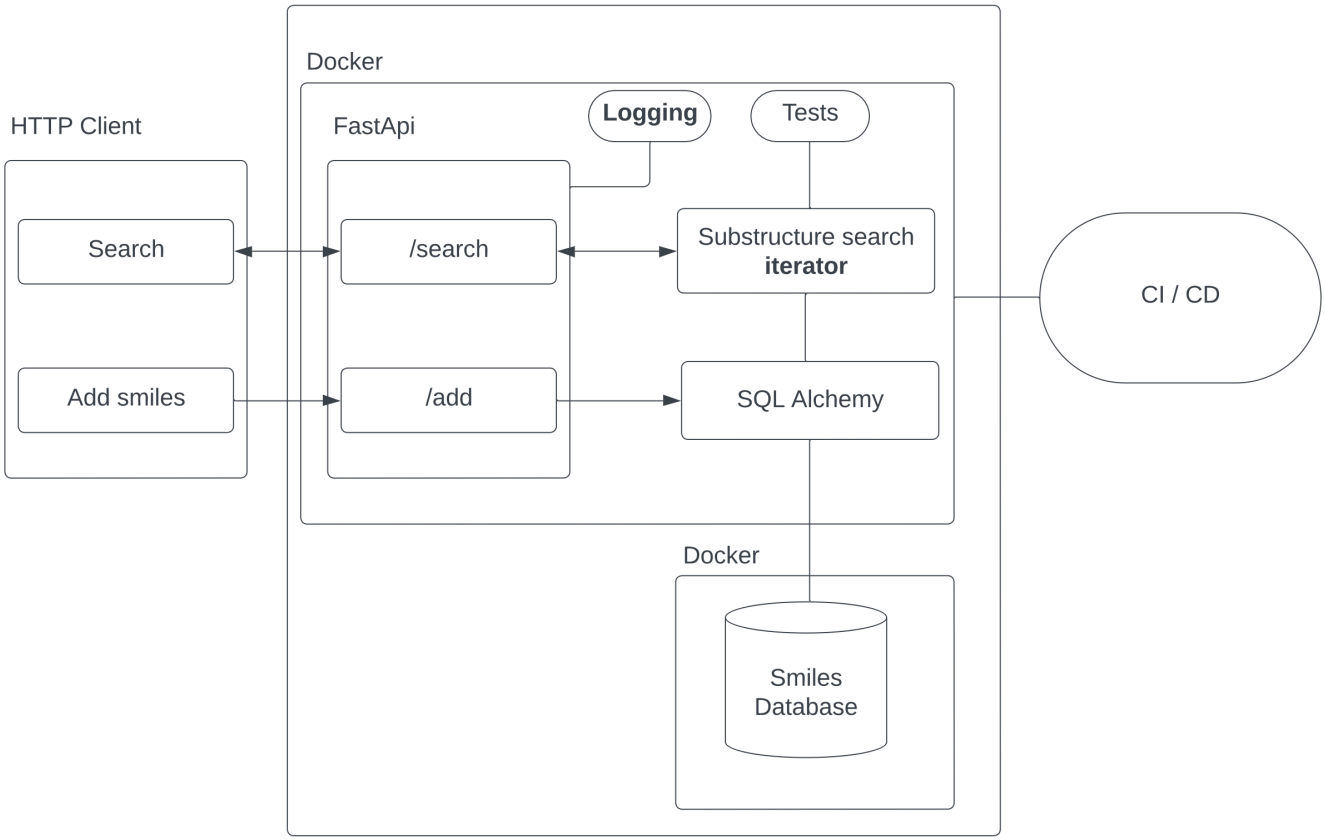
Homework

Add a database for storing molecules.



7. Logging/Iterator

Homework



Logging

An important part of the application's operation is logging. The `print` function is enough for small projects, but a good practice is to use the `logging` package. Add logging to the project so that you can understand from the logs what activity is happening in our application.

Iterator

We don't always need all the molecules at once when searching. Sometimes we need, for example, the first 100. Modify the API method `List all molecules` by adding the `limit` argument to limit the required number of molecules in the response. Don't forget to remake the function into an iterator using the knowledge from the lecture.

8. Redis

Redis (Remote Dictionary Server) is an open-source, in-memory data structure store that can be used as a database, cache, and message broker. Redis is well-known for its high performance, providing extremely fast access to data by keeping it in memory. It supports various data structures like strings, lists, sets, and hashes, making it versatile for different use cases. Commonly, Redis is used for caching, which helps reduce the load on the main database and speeds up the application by storing frequently accessed data in memory.

Why Use Redis?

Redis is particularly beneficial in scenarios where you need to store and retrieve data quickly. It can be used to cache the results of expensive database queries, improving the performance of your application by reducing latency and the number of requests hitting your primary database.

Key benefits of using Redis include:

- **Caching**: Store frequently accessed data in memory to reduce the load on your primary database and improve application performance.
- **High Performance**: Redis operates entirely in-memory, providing extremely fast data access.
- **Scalability**: Redis can handle large amounts of data with minimal latency, making it suitable for high-traffic applications.
- **Versatility**: Redis supports a variety of data structures, enabling a wide range of use cases beyond simple caching, such as session management, real-time analytics, and leaderboards.

Integrating Redis with FastAPI

Integrating Redis with a FastAPI project using Docker and Docker Compose allows you to easily manage and deploy your services in isolated containers, ensuring consistency across different environments.

1. Set Up Docker Compose

Next, create a `docker-compose.yml` file to define your FastAPI service and the Redis service:

```yaml
version: '3'

services:
  web:
    build: .
    command: uvicorn main:app --host 0.0.0.0 --port 8000
    volumes:
      - .:/app
    ports:
      - "8000:8000"
    depends_on:
      - redis

  redis:
    image: "redis:alpine"
    ports:
      - "6379:6379"
```

This configuration defines two services:

- **web**: The FastAPI application, which will be built from the `Dockerfile` and served using Uvicorn.
- **redis**: The Redis service, which uses the official Redis Docker image.

2. Use Redis in Your FastAPI Application

Now, integrate Redis into your FastAPI application. Below is an example of how you can use Redis for caching search results.

Create a `main.py` file for your FastAPI application:

```python
from fastapi import FastAPI
import redis
import json

app = FastAPI()

# Connect to Redis
redis_client = redis.Redis(host='redis', port=6379, db=0)

def get_cached_result(key: str):
    result = redis_client.get(key)
    if result:
        return json.loads(result)
    return None

def set_cache(key: str, value: dict, expiration: int = 60):
    redis_client.setex(key, expiration, json.dumps(value))

@app.get("/search/")
async def search(query: str):
```

```
    cache_key = f"search:{query}"
    cached_result = get_cached_result(cache_key)

    if cached_result:
        return {"source": "cache", "data": cached_result}

    # Simulate a search operation (e.g., querying a database)
    search_result = {"query": query, "result": "Some data"}  # Replace
with actual search logic

    set_cache(cache_key, search_result)

    return {"source": "database", "data": search_result}
```

In this example, the search endpoint first checks if the result is cached in Redis. If it is, the cached data is returned. If not, the search is performed, and the result is cached for future requests.

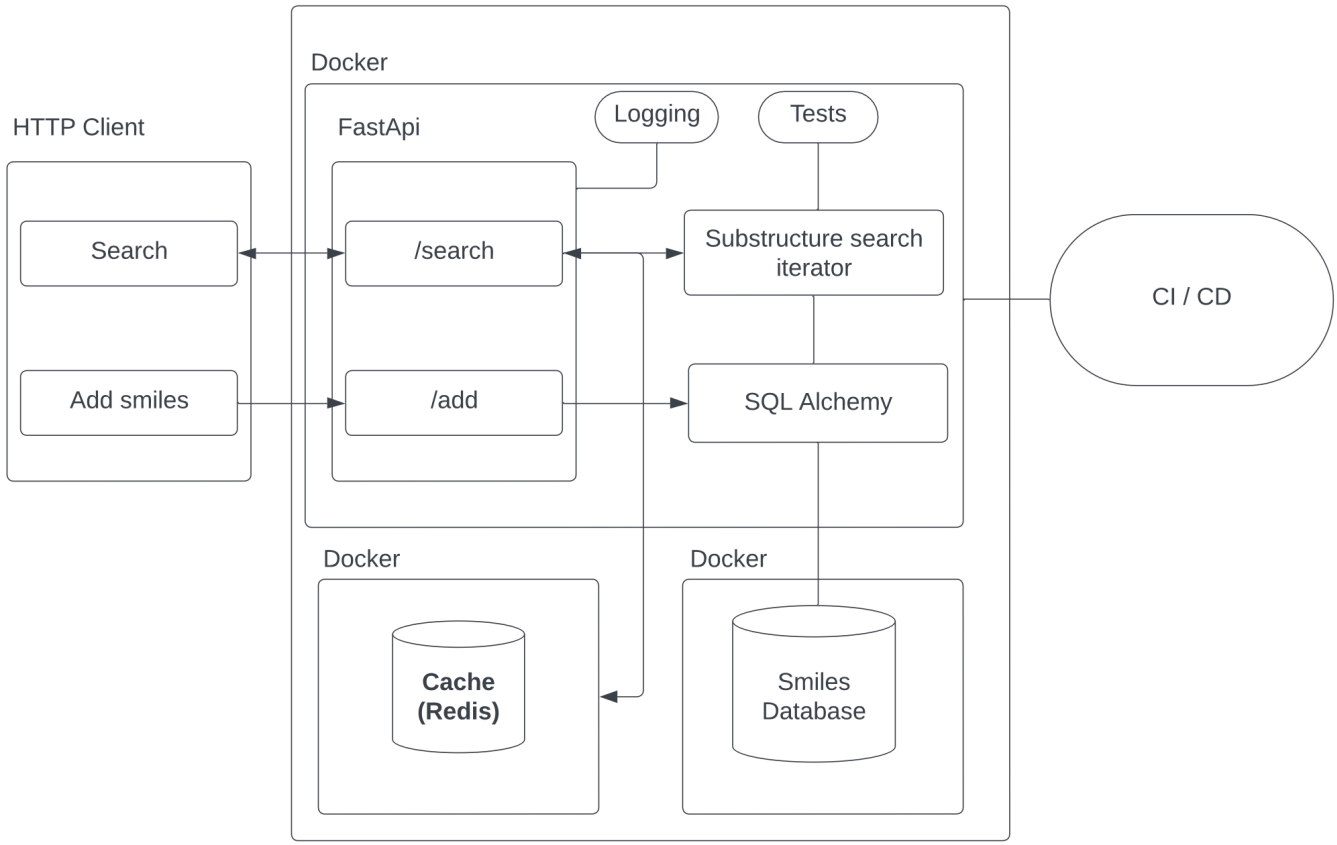
3. Running the Application

To start the application with Redis, simply run:

```
docker-compose up --build
```

This command will build your FastAPI service, start the Redis container, and make both services available. Your FastAPI application will be available at `http://localhost:8000`, and Redis will be running on port 6379.

Homework



Modify the search functionality in your FastAPI project to use Redis for caching search results.

Instructions:

1. **Set Up Docker and Redis**:
    - Ensure that Redis is integrated into your FastAPI project using Docker and Docker Compose as described above.
2. **Implement Caching**:
    - Modify the search logic in your FastAPI application to check for cached results in Redis before performing the search operation.
    - If the search result is found in the cache, return it immediately.
    - If the search result is not cached, perform the search, cache the result, and then return it.
3. **Set Cache Expiration**:
    - Implement a mechanism to set an expiration time on cached search results to ensure the cache is refreshed periodically.
    - Choose an appropriate expiration time based on the nature of the data being searched.
4. **Testing**:
    - Test the application to ensure that search results are correctly cached and retrieved from Redis.
    - Verify that repeated search requests use the cache to improve performance and reduce load on the main database.

9. Celery

Celery is a distributed task queue framework that allows you to execute long-running tasks asynchronously, outside of the main application flow. It's particularly useful for handling background jobs, such as sending emails, processing images, running machine learning models, or any computationally intensive work that you don't want to block the main application.

Key benefits of using Celery:

- **Asynchronous execution:** Run tasks in the background without waiting for them to finish.
- **Distributed task execution:** Celery can distribute work across multiple workers on different machines.
- **Retry mechanisms:** Celery provides automatic retry on failure.
- **Integration with various brokers:** Celery supports multiple message brokers such as Redis, RabbitMQ, etc.

Step 1: Setting Up the Environment

1.1 Create Project Structure

1. Start by creating the following project structure:

```
/celery-fastapi-redis/
│
├── app/
│   ├── __init__.py
│   ├── main.py       # FastAPI app
│   ├── celery_worker.py   # Celery configuration
│   └── tasks.py      # Celery tasks
│
├── docker-compose.yml
├── Dockerfile
└── requirements.txt
```

Step 2: Celery Setup with FastAPI and Redis

2.1 Define Dependencies

In `requirements.txt`, define the necessary dependencies:

```
fastapi
uvicorn
celery[redis]
redis
```

2.2 Create FastAPI Application (`main.py`)

Create a simple **FastAPI** application that will handle API requests and interact with **Celery**.

```
# app/main.py

from fastapi import FastAPI
from app.tasks import add_task
from celery.result import AsyncResult
from app.celery_worker import celery

app = FastAPI()

@app.post("/tasks/add")
async def create_task(x: int, y: int):
    task = add_task.delay(x, y)
    return {"task_id": task.id, "status": task.status}

@app.get("/tasks/{task_id}")
async def get_task_result(task_id: str):
    task_result = AsyncResult(task_id, app=celery)
    if task_result.state == 'PENDING':
        return {"task_id": task_id, "status": "Task is still processing"}
    elif task_result.state == 'SUCCESS':
        return {"task_id": task_id, "status": "Task completed", "result":
task_result.result}
    else:
        return {"task_id": task_id, "status": task_result.state}
```

2.3 Define Celery Worker (`celery_worker.py`)

Create a Celery configuration file to set up the Redis broker.

```
# app/celery_worker.py

from celery import Celery

celery = Celery(
    'tasks',
    broker='redis://redis:6379/0',
    backend='redis://redis:6379/0'
)

celery.conf.update(task_track_started=True)
```

2.4 Define Celery Task (`tasks.py`)

Now, define a simple Celery task, such as adding two numbers together.

```
# app/tasks.py

from app.celery_worker import celery

@celery.task
def add_task(x: int, y: int):
    return x + y
```

Step 3: Dockerizing the Application

We'll use **Docker** to containerize the FastAPI app, Celery worker, and Redis broker.

3.1 Create `Dockerfile`

Define a `Dockerfile` to create a Docker image for the FastAPI app:

```
# Dockerfile

FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY ./app /app/app

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

3.2 Create `docker-compose.yml`

Use **Docker Compose** to define services for FastAPI, Celery, and Redis:

```
# docker-compose.yml

version: '3.8'

services:
  fastapi:
    build: .
    ports:
      - "8000:8000"
    depends_on:
      - redis
    volumes:
      - .:/app
```

```
  celery_worker:
    build: .
    command: celery -A app.celery_worker worker --loglevel=info
    depends_on:
      - redis
    volumes:
      - .:/app

  redis:
    image: "redis:alpine"
    ports:
      - "6379:6379"
```

Step 4: Running the Application

1. **Build and start the containers:**

   ```
   docker-compose up --build
   ```

   This will start the FastAPI app, Celery worker, and Redis container.

2. **Testing the application:**

   - Open your browser or use `curl` to send a POST request to add a task:

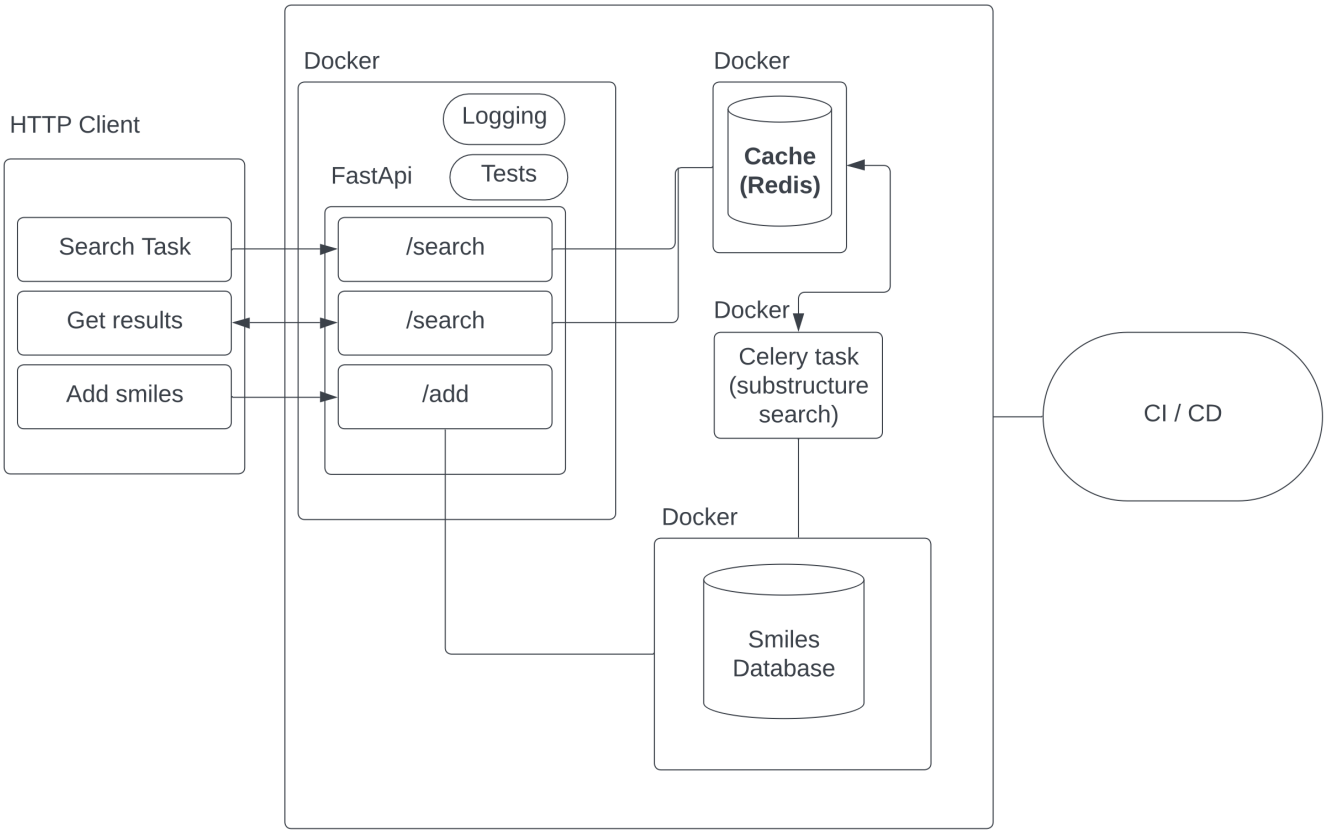
     ```
     curl -X POST "http://127.0.0.1:8000/tasks/add" -H "Content-Type: application/json" -d "{\"x\": 5, \"y\": 10}"
     ```

   - You will receive a response with the `task_id`. To check the status of the task, use:

     ```
     curl -X GET "http://127.0.0.1:8000/tasks/{task_id}"
     ```

   This will show whether the task is still processing, completed, or failed.

Homework



Modify the substructure search functionality in your FastAPI project to use Celery. It is necessary to modify the API. Our logic will be as follows: we will send a request to start the substructure search task and then we can send request to get results if it is ready as shown in the example.