IfcOpenShell-Based BIM Application

Overview

This project uses the IfcOpenShell library to process Building Information Model (BIM) files in the IFC format. The application reads an IFC file, extracts relevant data about building elements, their properties, relationships and performs simple analysis, and saves the processed data to two separate JSON files. This project is containerized using Docker to ensure a consistent runtime environment.

Approach

1. IfcOpenShell Library: Utilized to read and process IFC files.
2. Data Extraction: Extracted building elements, their properties and relationships from the IFC file using python scripts.
3. Analysis: Performed a simple analysis.
4. Data Output: Saved the processed data and analysis results to a JSON file.
5. Application directory structuring: Divided the whole application into a group of independent microservices, namely elements_properties, relationships and automated tests.
6. Docker: Created a Dockerfile for each service to set up the environment, install dependencies, and run the service application in the container.
7. Docker-compose: Used Docker-Compose YAML file to orchestrate multiple services.
8. Included automated tests and scalability within the docker-compose file.

Pre-requisites

1. Basic domain knowledge of Building Information Modelling or BIM.
2. The IFC file provided is well-formed and follows standard IFC specifications.
3. The Docker environment has network access to fetch and install these dependencies during the build process.
4. The Docker runtime environment of minimum 16 GB memory and 8 cores CPU to handle the processing of the IFC file, especially if the file is large.

Docker -compose

Docker-compose gives the user an ability to orchestrate i.e., define and run multi-container Docker applications. Each container represents each micro-services of the application.

Instructions to build and run the application using Docker-Compose

1. Install Docker

1.1. Update your System:

- Update your existing list of packages.

  sudo yum update -y

1.2. Install Docker

- Install Docker using the yum package manager.

  sudo yum install docker -y

1.3. Verify Docker Installation:

  docker --version

2. Install Docker-Compose

2.1. Download the current version of Docker Compose:

  sudo curl -L "https://github.com/docker/compose/releases/download/$(curl -s https://api.github.com/repos/docker/compose/releases/latest | grep -Po '"tag_name": "\K.*?(?=")')/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

2.2. Apply executable permissions:

  sudo chmod +x /usr/local/bin/docker-compose

2.3. Create a symbolic link to /usr/bin for ease of use:

  sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose

2.4. Verify the installation:

docker-compose –version

3. Structure the application directory as per the requirement for efficient handling.

An example of the project directory structure according to the project case is as follows:

Ex.:

/path/to/your/project/directory/

| ---- docker-compose.yml

| ---- elements_properties/

| |-- Dockerfile

| |-- elements_properties.py

| |-- sherier_place_arch.ifc

| ---- relationships/

| |-- Dockerfile

| |-- relationships.py

| |-- sherier_place_arch.ifc

| ---- automated_test/

| |-- Dockerfile

| |-- tests_.py

| |-- sherier_place_arch.ifc

Create a Dockerfile for each services within the respective directory.

4. Create a Dockerfile
   A Dockerfile is a text based file with a series of commands to create an image.

4.1.Create a Dockerfile:

touch Dockerfile

4.2.Edit the Dockerfile

- Open the Dockerfile in your preferred text editor and add the necessary commands. Here's an example Dockerfile for a simple Python based application:

- Dockerfile:

```
# Use an official Python runtime as a parent image
FROM python:3.9-slim

# Set the working directory in the container
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . /app

# Install IfcOpenShell and other necessary dependencies
RUN apt-get update && apt-get install -y \
   libboost-all-dev \
   cmake \
   build-essential \
   python3-dev \
   libpython3-dev \
   libxml2-dev \
   libgeos-dev \
   libgdal-dev \
   && rm -rf /var/lib/apt/lists/*

# Install Python dependencies
RUN pip install --upgrade pip

# copy ifcopenshell dir to python path - site packages
RUN cp -r /app/ifcopenshell /usr/local/lib/python3.9/site-packages/.

# install lark dependency
RUN pip install lark-parser

# Define environment variable
ENV NAME IfcOpenShellApp
```

 # Run the application
CMD ["python3"]

Building and running the application with Docker-compose

1. Create a docker-compose.yml File

    - Create a docker-compose.yml file in your project directory. This file defines the services, networks, and volumes for your application.
    - Example:

```yaml
version: '3.8'

services:
  elements_properties:
    build: /home/ec2-user/syyclops_docker_compose/elements_properties/.
    volumes:
      - elements_volume_compose:/app
    command: python3 elements_properties.py


  relationships:
    build: /home/ec2-user/syyclops_docker_compose/relationships/.
    volumes:
      - relationships_volume_compose:/app
    command: python3 relationships.py

volumes:
  elements_volume_compose:
    driver: local
  relationships_volume_compose:
    driver: local
```

    - The build directive tells Docker Compose to build an image using the Dockerfile located in /home/ec2-user/syyclops_docker_compose/elements_properties/.
    - The volumes directive mounts the named volume elements_volume_compose to the /app directory inside the container.
    - The command directive runs python3 elements_properties.py when the container starts.

2. Include scalability directive if required:
    - Write the following code in the docker-compose file for horizontal scaling.

      deploy:
       replicas: number_of_containers

3. Build to generate the docker images using docker-compose:

   docker-compose build

4. Run the Services using docker-compose:

   docker-compose up

5. Verify the Services:
    - Check the running containers:

      docker-compose ps

6. Access and Manage Volumes:
    - The data from your application will be stored in the Docker volume for persistent data retrieval.
    - To access the data on the host machine:

      cd /var/lib/docker/volumes

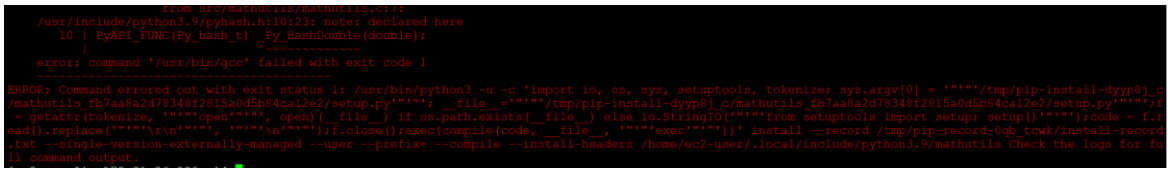
Challenges

1. Domain knowledge acquisition for proper implementation of the requirements:

   IFC files are complex and contain a wealth of information about building structures, including geometry, spatial relationships, and properties of building elements. To work effectively with IFC files, you need to have a basic understanding of both the technical aspects of the IFC format and the domain-specific knowledge related to building construction and architecture.

Solution:
Utilize online tutorials, guides, and example projects to see how others have approached similar challenges. Websites like BIMForum, buildingSMART, and YouTube channels dedicated to BIM maybe utilized.

2. Ifcopenshell dependency installation: encountered "'/user/bin/gcc' failure with exit status 1" error while installing ifcopenshell using pip.



Solution:

- Download the compatible pre-build package zipfile from: https://docs.ifcopenshell.org/ifcopenshell-python/installation.html#

- Unzip the downloaded file and copy the ifcopenshell directory into your Python path. To find the Python path, run the following code in Python:

  import sys
  print(sys.path)

  This will give a list of possible directories that can install the IfcOpenShell module into. Copy the ifcopenshell directory into one of these called site-packages.

- Test importing the module in a Python session or script to make sure it works.

  import ifcopenshell
  print(ifcopenshell.version)
  model = ifcopenshell.file()

3. Docker-compose volume mapping: Constantly encountered 'mapping values are not allowed here' errors due to incorrect YAML syntax in the docker-compose.yml file.

Solution:
Specified a separate volume directives for each services.
Ex.:

```
volumes:
  elements_volume_compose:
    driver: local
  relationships_volume_compose:
    driver: local
  tests_volume_compose:
    driver: local
```

4. Automated test scripts: Constantly encountered ‘tests failed due to incorrect configurations’ errors. Hence, the required JSON file output was not obtainable.

   Solution:
   Write the correct syntax for the respective test cases. An example for relationships.py script is as follows:

4.1. Importing required libraries:

```
# Import required libraries
import os
import json
import ifcopenshell
import pytest
```

4.2. Define the script to be tested as a function:

```
# Function to run the script
def run_ifc_relationships_script():
    import ifcopenshell
    import json

    # Import model
    model = ifcopenshell.open('/home/ec2-user/automated_test/sherier_place_arch.ifc')

    # Initialize data dictionary
    data = {}

    # Extract relationships
    data['relationships'] = []

    rels = model.by_type("IfcRelationship")
    for rel in rels:
        rel_data = {
            "type": rel.is_a(),
            "global_id": rel.GlobalId,
            "relating_element": rel.RelatingElement.GlobalId if hasattr(rel, "RelatingElement") else None,
            "related_elements": [elem.GlobalId for elem in rel.RelatedElements] if hasattr(rel, "RelatedElements") else None
        }
        data['relationships'].append(rel_data)

    # Save data to JSON file
    output_path = "/home/ec2-user/automated_test/ifc_relationships.json"
    with open(output_path, "w") as json_file:
        json.dump(data, json_file, indent=4)

    return data
```

4.3.Write test functions:

```python
def test_ifc_file_loading():
    # Test if the IFC file loads correctly
    model = ifcopenshell.open('/home/ec2-user/automated_test/sherier_place_arch.ifc')
    assert model is not None

def test_relationship_extraction():
    # Tests relationships extraction from the IFC file
    data = run_ifc_relationships_script()
    assert 'relationships' in data
    assert len(data['relationships']) > 0

def test_output_json():
    # Test if the output JSON file is created and has the expected structure
    run_ifc_relationships_script()
    output_path = "/home/ec2-user/automated_test/ifc_relationships.json"
    assert os.path.exists(output_path)
    with open(output_path) as json_file:
        data = json.load(json_file)
        assert 'relationships' in data
        assert len(data['relationships']) > 0

if __name__ == "__main__":
    pytest.main()
```