A Project Report On

3-Body Runge-Kutta Solver Infrastructure

For

Virtualization II (WiSe 21/22)

Candidates:

1. Mohammad Tohin Bapari ID:1836950
2. Agyapong Prince ID: 1737596
3. Anselem Okeke ID: 1943585

Supervised by:

1. Torsten Harenberg
2. Marisa Sandhoff

Bergische Universität Wuppertal

Gaußstraße 20, 42119 Wuppertal

Table of Content

Chapter 0: Introduction

0.1: What is Docker

Docker is an open platform for developing, shipping, and running applications. Docker enables us to separate our applications from our infrastructure so we can deliver software quickly.

A container is a runnable instance of an image. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image

Compose is a tool for defining and running multi-container Docker applications. With Compose, we use a YAML file to configure our application's services. Then, with a single command, you create and start all the services from your configuration.

0.2: Solving Method

To build our project infrastructure we use multi-container docker system.

Firstly we create a container solver-api for solving the 3-body problem by Runge-Kutta method and insert the output value in a predefined database. We use it as a server for our frontend client.

Secondly we create another container frontend to visualize and setting coordinates three body problem

Thirdly we use traefik to secure our frontend

Chapter 1:Solver

1.1: Functions

We define 5 main functions and build a solver API.

These functions are :

i) f - A function to calculate the forces of interaction between the bodies

ii) ODE45 - A function to estimate errors and determine step sizes and to solve the ordinary differential equation initial value problem.

**3 asynchronous python functions (save_to_db, send_id_to_solver, backend_service)*

iii) Save_to_db - This is an asynchronous function that creates a database of (time, mass, position (x,y,z) and velocities in the (x,y,z) plane)

iv) Send_id_to_solver - This is another asynchronous function that connects the return values of the database to the solver api

v) Backend_service – This functions connects to the database based on incoming requests from the front end. Uses to ODE45 function to to solve the initial value problem and save the results back to the database

We employ asynchronous functions to handle the network and I/O database tasks to prevent the waiting times for tasks to finish.

1.2: Inserting data in database

We implement a SQLite database which is called (Solver_DB.db). It consists of tables of time, mass, position and velocity values. It receives input values from the frontend. A connection is created from the database to the solver-api in backend it solves the ordinary differential equation using Runge-Kutta and the results are saved back to the database and appears graphically in the frontend

1.3: Dockerfile

```
FROM python:latest                          #Pull latest python image as Base
RUN pip install websockets pandas           #install websockets and pandas
WORKDIR /server/                            #Create working directory
ADD solver-api.py .                          #Copy solver-api.py to WORKDIR
EXPOSE 8001
CMD [ "python3", "-u", "solver-api.py"]     # To run the container
```

Chapter 2: Frontend

2.1: Javascript libraries

We have open source JavaScript libraries for 3-body problem. So we took three source file (body.js, OrbitControls.js, three.js) and put it in frontend folder.

2.2: Visualization

For visualizing our project we create index.html file. The keynotes are as follows

- Importing Javascript files.

```
<script src="lib/three.js"></script>
 <script src="lib/OrbitControls.js"></script>
<script src="js/body.js"></script>
```

- Create a form to setup mass and Coordinate data and start simulation.

```
<input id="play-sim" type="button" value="Start" disabled>
<input id="setup-sim" type="button" value="Setup">
```

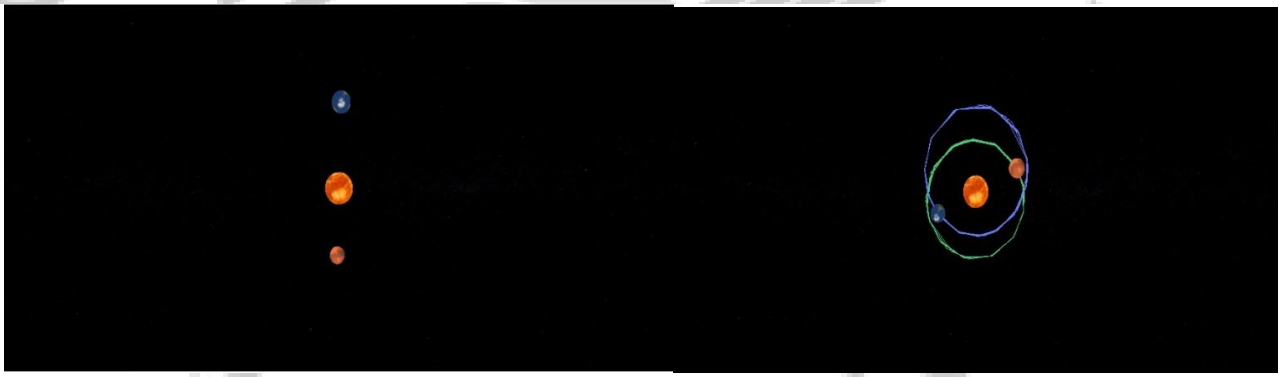
- For setup simulation

```
document.querySelector('#setup-sim').onclick = function(e)
```

- For playing simulation

```
document.querySelector('#play-sim').onclick = function(e)
```

- Graphical representation:



Before simulation

After simulation

2.3: Dockerfile

```
FROM python:latest
WORKDIR /var/www/html
COPY frontend/ .
EXPOSE 8000
CMD python -m http.server 8000
```

Chapter 3: Docker Compose

Using Compose is basically a three-step process:

- Define our project's environment with a Dockerfile so it can be reproduced anywhere.
- Define the services that make up our project in docker-compose.yml so they can be run together in an isolated environment.
- Run docker compose up and runs our entire project.

Keynotes of our Compose file:

- We create three services (solver-api, solver-frontend and traefik)
  - Build: It helps us to build the container services.
  - Image: For naming the image.
  - Volume: To save persistent data
  - Labels:
    - ✓ traefik.enable=true
      Explicitly tell Traefik to expose container
    - ✓ traefik.http.routers.mywebsocket.entrypoints=server1
    - ✓ traefik.http.routers.mywebsocket.entrypoints=frontend1
      Two entrypoint server1 and frontend1 declared
    - ✓ entrypoints.frontend1.address=:8000"
    - ✓ entrypoints.server1.address=:8001
      Port number for the entrypoints through traefik
    - ✓ providers.docker=true
      Enabling docker provider
    - ✓ providers.docker.exposedbydefault=false
      Do not expose containers unless explicitly told so

Chapter 4: Alternative Solutions

Other ways	Reason for not choosing
Multi-stage build: We could use multi-stage Dockerfile to build our Containers.	Since these Dockerfiles have multiple stages to produce the production-grade image, our **cache will not consist of the images built** in the previous steps leading to your final output.
External network: we can build our containers separately and connect them by creating a external network.	By default Compose sets up a single network for our containers. Each container for a service joins the default network and is both *reachable* by other containers on that network, and *discoverable* by them at a hostname identical to the container name.
Docker Swarm: Docker swarm is a container orchestration tool that allows you to run and connect containers on multiple hosts.	In our project we try to build a container infrastructure that run and connect container on single host which is dode by docker compose file.
NGINX: For load balancing and security of our services we could also use NGINX	Activating simple features with Traefik **does not require multiple complex settings** as it does with nginx, and the configuration itself tends to be a lot quicker and more concise as well. While nginx settings end up in huge config maps that are hard to read and manage, it's not an issue with Traefik

Chapter 5: Who did which part

Mohammad Tohin Bapari	1. Solver-api.py 2. Backend Dockerfile 3. Docker compose file 4. Presentation slides 5. Project Report
Agyapong Prince	1. Solver-api.py 2. Solver_DB.db 3. Project Report.
Anselem Okeke	1. JavaScript Libraries 2. HTML file 3. Frontend Dockerfile