



Course Number	COE892
Course Title	Distributed Cloud Computing
Semester/Year	Winter 2025
Instructor	Khalid Abdel Hafeez
TA name	Amirhossein Razavi

Lab No.	4-5
----------------	-----

Section No.	05
Submission Date	March 30th, 2025
Due Date	March 30th, 2025

Student Name	Student ID	Signature*
Maria Labeeba	500960386	M.L

Student Name Student ID Signature*

*By signing above you attest that you have contributed to this submission and confirm that all work you have contributed to this submission is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a "0" on the work, an "F" in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct

1. Introduction

The objective of Lab 4-5 is a modification of Lab 2, where instead of using gRPC for communication, we were tasked with implementing RESTful APIs using a FastAPI HTTP server to simulate and control rover operation. The endpoints seen below in Table 1 were provided by the lab manual and were implemented.

Map		
Route	Method	Description
/map	GET	To retrieve the 2D array of the field.
/map	PUT	To update the height and width of the field.
Mines		
Route	Method	Description
/mines	GET	To retrieve the list of all mines, the response should include the serial number of the mines, and coordinates.
/mines/:id	GET	To retrieve a mine with the ":id", the response should include the serial number of the mine, and coordinates.
/mines/:id	DELETE	To delete a mine with the ":id"
/mines	POST	To create a mine. The coordinates (X and Y), along with the serial number should be required in the body of the request. The ID of the mine should be returned in the response upon successful creation.
/mines/:id	PUT	To update a mine. The coordinates (X and Y), along with the serial number should be optional in the body of the request. Only the included parameters should get updated upon receiving the request. The response must include the full updated mine object.
Rover		
Route	Method	Description
/rovers	GET	To retrieve the list of all rovers, the response should at least include the ID and rover status ("Not Started", "Finished", "Moving", or "Eliminated").
/rovers/:id	GET	To retrieve a rover with the ":id", the response should include the ID, status ("Not Started", "Finished", "Moving", or "Eliminated"), latest position and list of commands of the rover.
/rovers	POST	To create a rover. The list of commands as a String should be required in the body of the request. The ID of the rover should be returned in the response upon successful creation.
/rovers/:id	DELETE	To delete a rover with the ":id"
/rovers/:id	PUT	To send the list of commands to a rover as a String. Note if the rover status is not "Not Started" nor "Finished", a failure response should be returned.
/rovers/:id/dispatch	POST	To dispatch a rover with the ":id", the response should include the ID, status, latest position and list of the executed commands of the rover.

Table 1: Provided API Endpoints

We were tasked with creating, modifying and deleting rovers and mines, as well as dispatching the rover to traverse through the map using their given commands. The logic for the map traversal as well as disarming a mine was taken from lab 2. A responsive and user-friendly UI was designed using HTML, Tailwind CSS and Javascript for the rover navigation. The lab also involves the optional implementation of WebSocket-based real-time communication and the deployment of the server to Microsoft Azure using Docker containers.

2. Overall Implementation

The figure below showcases the structure of this Lab's project folder. The `main.py` file serves as the entry point, defining all RESTful endpoints and routing incoming API requests to appropriate helper functions. These helpers are organized within the `Utils` folder: `map_mine_utils.py` handles map and mine-related operations, while `rover_utils.py` contains logic for rover behavior and traversal. The static directory contains the frontend UI, with `dashboard.html` and `dispatch.html` providing the interfaces for rover management and live dispatch monitoring. JavaScript files `rover-dashboard.js` and `rover-dispatch.js` are linked to these HTML pages and implement dynamic behavior for CRUD actions and real-time updates. The `rover_info` folder, with `rover_data.txt`, stores persistent data for rovers such as IDs and states. `map1.txt` and `mines.txt` define the initial terrain and mine placements used by the simulation. The `Dockerfile` outlines how to containerize the app for deployment, and `requirements.txt` lists Python dependencies. Finally, `run.py` is an alternative script for running the application during development or testing. Each component works together to form a cohesive distributed system with both backend and frontend functionality.

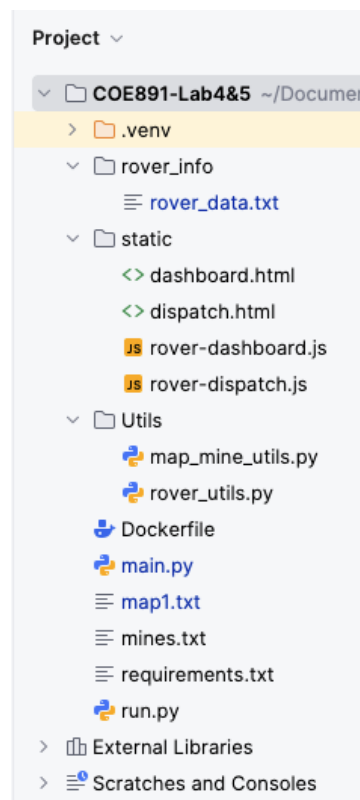




Figure 1: Lab % Pycharm Project Structure





3. Part 1: Server

In the first part of Lab 4-5, a FastAPI HTTP server was developed using Python to simulate rover and mine management within a distributed system. The server was designed to

expose all the RESTful API endpoints outlined in the lab specification, enabling operations such as retrieving and updating the map, and creating, dispatching, and monitoring rovers and mines. A suitable data structure was implemented to store key elements like the 2D map, list of rovers, and mine information, ensuring persistent and structured access during API calls. FastAPI's built-in support for automatic documentation via the /docs route was leveraged to test and validate the functionality of each endpoint through an interactive UI interface.

The project is structured around a main.py file, which handles all incoming CRUD requests. This file delegates specific operations to helper functions organized in map_mine_utils.py and rover_utils.py, both located within the Utils directory. The following code can be found in the main.py file of my project.

Map Endpoint Implementation		
/map GET	54 	<pre> @app.get("/map") # Maria def get_map(): return read_map() </pre>
/map PUT	58 	<pre> @app.put("/map") # Maria def update_map(dim: MapDimensions): old_grid = map_data.get("grid", []) old_height = len(old_grid) old_width = len(old_grid[0]) if old_height > 0 else 0 new_grid = [[0 for _ in range(dim.width)] for _ in range(dim.height)] # Copy existing bombs into new grid where possible for i in range(min(dim.height, old_height)): for j in range(min(dim.width, old_width)): new_grid[i][j] = old_grid[i][j] map_data["height"] = dim.height map_data["width"] = dim.width map_data["grid"] = new_grid # Save to file with open("map1.txt", "w") as f: f.write(f"{dim.height} {dim.width}\n") for row in new_grid: f.write(" ".join(map(str, row)) + "\n") return {"status": "success", "message": "Map resized and saved"} </pre>

Mines Endpoint Implementation		
/mines GET	<pre> 95  @app.get("/mines") # Maria 96 def get_mines(): 97 return mines </pre>	
/mines/:id GET	<pre> 99  @app.get("/mines/{mine_id}") # Maria 100 def get_mine(mine_id: int): 101 for mine in mines: 102 if mine["id"] == mine_id: 103 return mine 104 raise HTTPException(status_code=404, detail="Mine not found") </pre>	
/mines POST	<pre> 106  @app.post("/mines") # Maria 107 def create_mine(mine: Mine): 108 global mine_id_counter 109 mine.id = mine_id_counter 110 mine_id_counter += 1 111 mines.append(mine.dict()) 112 map_data["grid"][mine.x][mine.y] = 1 113 save_map() 114 return {"id": mine.id} </pre>	
/mines/:id PUT	<pre> 116  @app.put("/mines/{mine_id}") # Maria 117 def update_mine(mine_id: int, mine_update: UpdateMine): 118 for mine in mines: 119 if mine["id"] == mine_id: 120 if mine_update.x is not None: 121 mine["x"] = mine_update.x 122 if mine_update.y is not None: 123 mine["y"] = mine_update.y 124 if mine_update.serial_number is not None: 125 mine["serial_number"] = mine_update.serial_number 126 save_map() 127 return mine 128 raise HTTPException(status_code=404, detail="Mine not found") </pre>	

Rover Endpoint Implementation	
/rovers GET	<pre> 215 @app.get("/rovers") # Maria 216 def get_rovers(): 217 try: 218 with open("rover_info/rover_data.txt", "r") as f: 219 return [json.loads(line.strip()) for line in f if line.strip()] 220 except Exception as e: 221 print("Error loading rover data:", e) 222 return [] </pre>
/rovers/:id DELETE	<pre> 245 @app.delete("/rovers/{index}") # Maria 246 def delete_rover(index: int): 247 try: 248 with open("rover_info/rover_data.txt", "r") as f: 249 rovers = [json.loads(line.strip()) for line in f if line.strip()] 250 if 0 <= index < len(rovers): 251 del rovers[index] 252 with open("rover_info/rover_data.txt", "w") as f: 253 for rover in rovers: 254 f.write(json.dumps(rover) + "\n") 255 return {"status": "success"} 256 else: 257 return {"status": "error", "message": "Invalid index"} 258 except Exception as e: 259 return {"status": "error", "message": str(e)} </pre>

The main purpose of this part was to design the backend of the rover navigation system.

4. Part 2: Operator

For this part of the lab, we were required to design a user-friendly interface to allow an operator to have full control of a rover. The operator should have full control on picking the map size and mine placements the rover will traverse through. The operator should also be able to create and delete a rover as well as edit the rovers commands. The final functionality is that the operator should be able to dispatch the rover and have that rover traverse through the map with their commands and display the path. This was implemented using HTML, Tailwind CSS and Javascript.

The dashboard.html and rover-dashboard.js contain the design and logic for all the CRUD (Create, Read, Update, Delete) operations related to rovers and mines, allowing the operator to interact with the FastAPI backend to manage map dimensions, place or remove mines, create new rovers, edit their commands, delete them, and dispatch them to visualize their path across the map in real-time. The following figure displays the result of these two files.

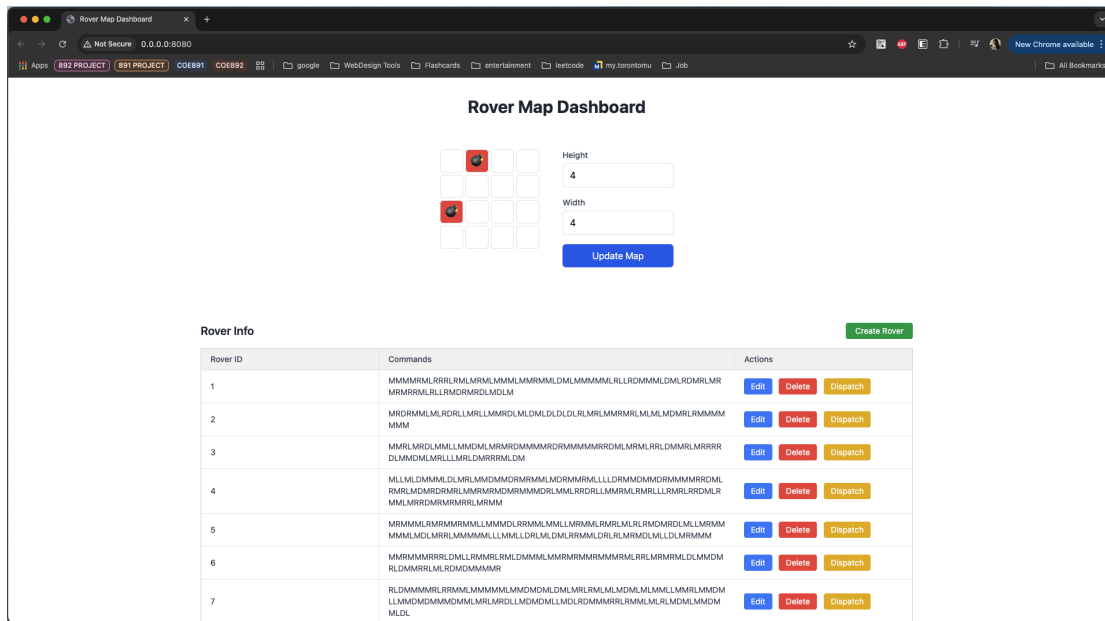


Figure 2: Rover Map Dashboard

When the operator clicks the Dispatch button for any rover, they are redirected to the Dispatch Rover ID page, where they can initiate and monitor the rover's movement. This interface includes Play and Pause controls, along with a visual representation of the map the rover will traverse. Below the map, the rover's current status and the command being executed are clearly displayed in real-time. For example, dispatching Rover 10 demonstrates this functionality, as shown in the figure below.

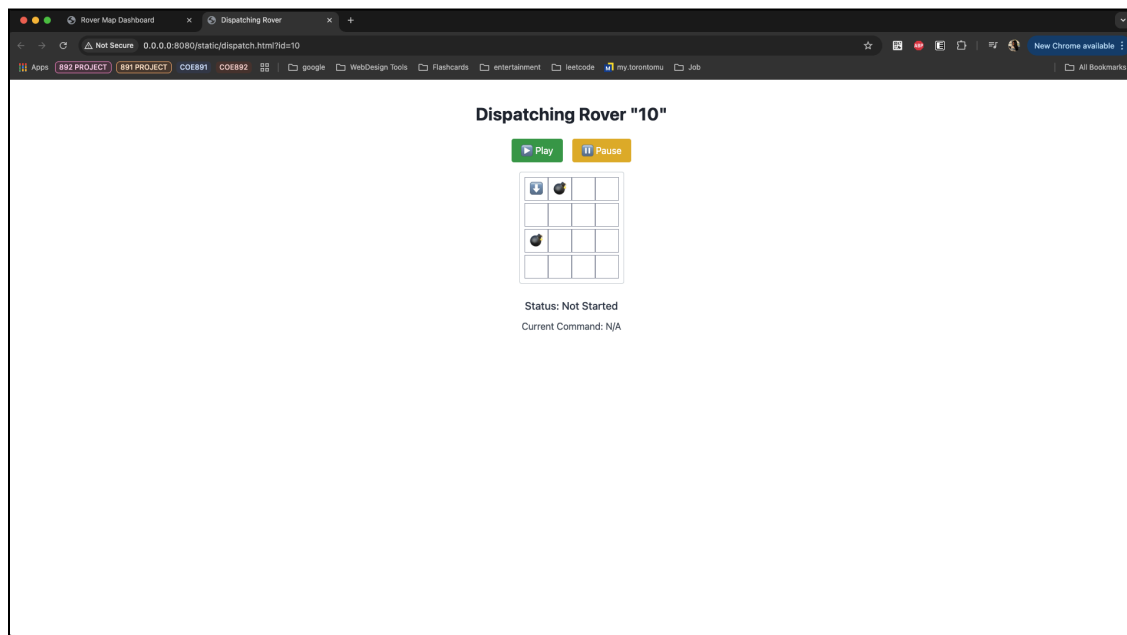


Figure 3: Dispatching rover 10 UI

When the rover lands on a mine and is followed by a dig command, it retrieves the mine's serial number from mines.txt and begins the disarming process. The rover attempts to find a valid PIN by hashing combinations of the serial number and potential PINs using the SHA-256 algorithm. A PIN is considered valid if the resulting hash has at least six leading zeros. Once a valid PIN is found, it is recorded, and the mine is considered disarmed, allowing the rover to safely continue executing its remaining commands. If the rover fails to dig or provide a valid PIN, the mine explodes and the rover is eliminated, stopping further command execution. A complete demo of the rover navigation system can be found in the video linked below.

Complete Demo: [🔥 COE892-Lab4/5-Project.mov](#)

5. Part 4: Deploying using Docker Container

The final part of the lab involved deploying the web application using Azure and Docker containers. While I was able to successfully create my Azure account, set up the necessary containers and resources, and configure the Dockerfile within my project as outlined in the lab manual, I encountered persistent challenges during deployment. Although the application was successfully deployed to Azure, it consistently displayed an "Application Error" as seen in the figure below. Despite thoroughly reviewing the console logs and making several adjustments to address the identified issues, the error persisted, and I was ultimately unable to resolve it.

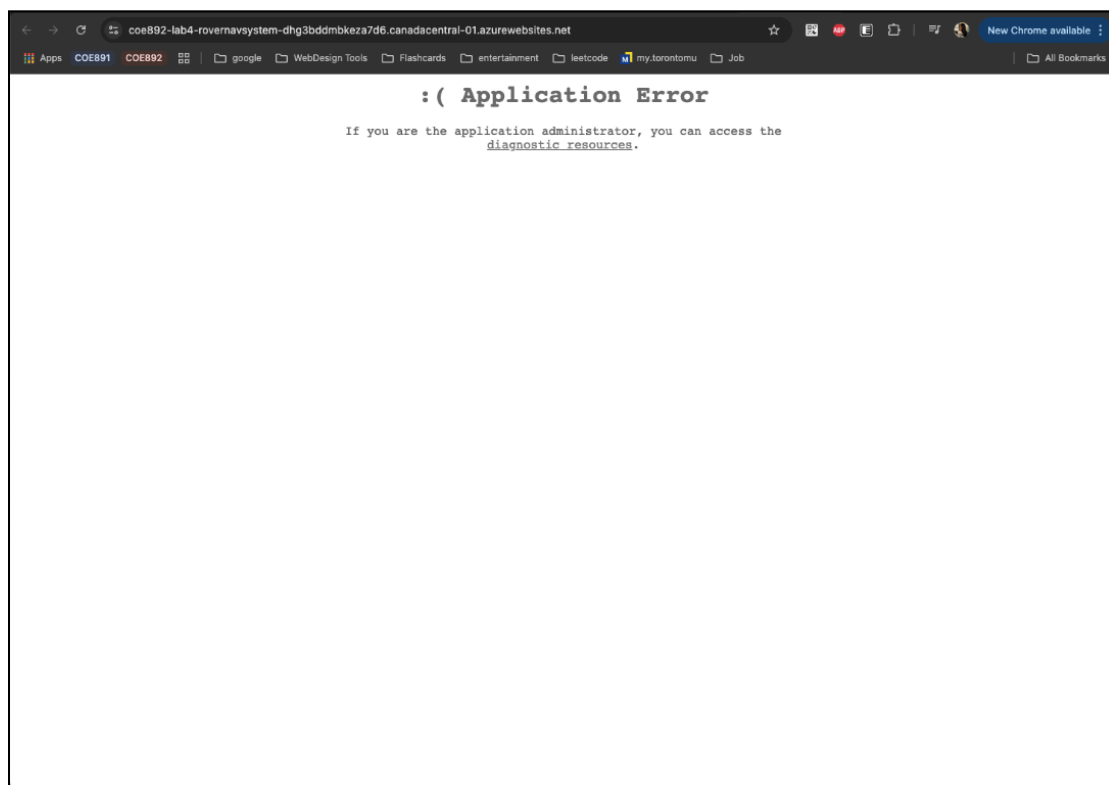


Figure 4: Application Error page

6. Conclusion

Lab 4/5 served as a culmination of concepts covered throughout the course by integrating RESTful API development with FastAPI, interactive frontend design, and container-based deployment in a distributed system context. By transitioning from gRPC to HTTP-based communication, I was able to deepen my understanding of REST principles and FastAPI's capabilities for building scalable and testable APIs. The modular design using helper utility files (`map_mine_utils.py`, `rover_utils.py`) promoted clean separation of logic and maintainability. The frontend, designed with Tailwind CSS and JavaScript, successfully offered a dynamic interface for real-time rover control and mine management, aligning with the user experience goals of distributed systems.

While the backend functionality and UI worked effectively in local development, deploying the system using Docker on Microsoft Azure presented challenges. Despite creating the required container and configuring the deployment pipeline correctly, the final deployment failed with an unresolved "Application Error." This highlighted the complexities and real-world difficulties of cloud deployment and emphasized the need for thorough debugging and logging strategies in distributed cloud environments.

Overall, the lab provided valuable hands-on experience in RESTful API design, UI development, rover simulation logic, and cloud-based deployment, tying together theoretical and practical knowledge in distributed and cloud computing systems.

References

[1] M. Jaseemuddin et al., "COE892 Lab Manual," *Dept. Elect., Comput., and Biomed. Eng., Ryerson Univ.*, Toronto, ON, Canada, pp. 1-16, Winter 2024.