Project 1 : basic web application to display list of unix commands

High-level view:

1. Architecture & Design
2. Source Control and collaboration
3. Continuous Integration
4. Containerization
5. Basic Infrastructure as Code (IAC)
6. Basic monitoring and logging

Low-level view:

1. Architecture and design:

   > *Concept* : basic system architecture
   > *Task* : design a simple web app with a frontend and a backend that serve a list of Linux commands and their descriptions

2. Source control management:

   > *Concept* : Version control systems and collaboration tools
   > *Task* : Set up a Git repository to manage the codebase and collaborate with team members

3. Continuous Integration (CI):

   > *Concept* : CI pipelines
   > *Task* : Implement a basic CI pipeline using tools like Jenkins, GitLab CI, or GitHub Actions to automate building and testing the application.

4. Containerization:

   > *Concept*: Docker
   > *Task*: Containerize the frontend and backend components using Docker, making it easier to deploy and run the application.

5. Basic Infrastructure as Code (IaC):

   > *Concept* : IaC tools (e.g., Terraform, Ansible)
   > *Task* : Define the infrastructure components (e.g., virtual machines, load balancers) using IaC tools and automate the provisioning process.

6. Basic Monitoring and Logging:

> *Concept* : Basic monitoring and logging tools (e.g., log analyzers, simple monitoring solutions)
> *Task* : Implement a simple monitoring and logging system to keep track of the application's performance and logs.

Exercise 1: Design a simple web application

Goal:
Design a basic web application with a frontend and a backend (API server) that serves a list of Linux commands and their descriptions.

Components :

- Frontend: A simple web interface that displays the list of Linux commands and their descriptions fetched from the backend.
- Backend: An API server that serves the list of Linux commands and their descriptions.

Step 1: Choose technologies for frontend and backend

Frontend:

```
Framework: React, Angular, or Vue.js
CSS framework: Bootstrap or Tailwind CSS (for styling)
```

Backend:

```
Language: Python, Node.js, or Ruby
Framework: Flask (Python), Express (Node.js), or Sinatra (Ruby)
```

Step 2: Plan the communication between components

> The frontend will communicate with the backend through RESTful API calls. Define a simple API endpoint, such as /api/commands, to fetch the list of Linux commands and their descriptions from the backend. The frontend will call this API endpoint to retrieve the data and display it on the web interface.

Step 3: Design the frontend and backend components

Frontend:

```
Design a basic layout for the web interface using the chosen CSS
framework.Implement the functionality to fetch the list of Linux commands
from the backend and display it on the interface.
Handle any errors that might occur during the API call.
```

Backend:

```
Set up the chosen backend framework and create a simple RESTful API with
the /api/commands endpoint.
Define a data structure (e.g., a list of dictionaries or a JSON object) to
store the Linux commands and their descriptions.
Implement the functionality to serve the list of Linux commands and their
descriptions through the API endpoint.
```

Implementation

All components are hosted on a single server.

Exercise 1.1 : develop the backend

let's create a simple Flask backend API to serve the Unix commands data. We'll use Python for this implementation. Follow these steps:

1. First, make sure you have Python installed on your system.
2. Create a new directory for your Flask backend.

```
mkdir unix-commands-api
cd unix-commands-api
```

3. Create a virtual environment and activate it

```
sudo apt install python3-venv
python3 -m venv flask-env
source flask-env/bin/activate
```

4. Install Flask in the virtual environment

```
sudo apt install python3-pip
pip3 install flask
```

5. To handle CORS issues (Corss-Origin Resources Sharing), use flask-cors

```
pip3 install flask-cors
```

6. Create a new Python file, app.py, in the unix-commands-api directory with the following content:

```python
from flask import Flask, jsonify
from flask_cors import CORS

app = Flask(__name__)
CORS(app)

commands = [
    {
        'id': 1,
        'name': 'ls',
        'description': 'List directory contents',
    },
    {
        'id': 2,
        'name': 'cd',
        'description': 'Change the current directory',
    },
    # Add more commands as needed
]

@app.route('/api/commands', methods=['GET'])
def get_commands():
    return jsonify(commands)

if __name__ == '__main__':
    app.run(debug=True)
```

7. Start the backend server

```
python3 app.py
```

The Flask server will start running at http://127.0.0.1:5000.
You can now access the **/api/commands** endpoint at **http://127.0.0.1:5000/api/commands**.

Exercise 1.2 : develop the frontend