Configure and deploy below application on Linux server (3 tier application)

Frontend server(apache webserver) -> backend application (Node app) - database (mongo)

Create .env file in you project to pass environment =>

```
MONGODB_URL=mongodb://localhost:27017/demo
PORT=3000
```

Application source - https://github.com/BL-AniketChile/NodeJs-API

Aim
The main aim of this task is to **deploy and configure a 3-tier web application** (frontend, backend, and database) on a Linux server. The focus is on ensuring seamless integration between:

1. **Frontend (Apache)**: Serving as the web interface and reverse proxy for routing user requests.
2. **Backend (Node.js App)**: Handling business logic and communicating with the database.
3. **Database (MongoDB)**: Storing and managing application data.

This task develops skills in system configuration, environment setup, application deployment, and troubleshooting, simulating real-world production scenarios.

Database (MongoDB):
Why database layer is been configured first?
The database layer serves as the backbone of the application. Setting it up first ensures that subsequent layers, like the backend and frontend, have a stable and fully functional foundation to interact with. This sequential approach minimizes errors and accelerates the deployment process.

Why MongoDB?
The application likely deals with semi-structured data and requires rapid development.
Its compatibility with Node.js simplifies the integration between the backend and database.

MongoDB's flexibility and performance make it a natural fit for the scalability and responsiveness needed in MongoDB is chosen for its **flexibility, scalability, performance, and ease of use**, making it ideal for modern, dynamic applications like this Node.js-based project. While other databases have their strengths, MongoDB aligns best with the project's goals and technologies.

Basic commands to deal with Mongodb

Show All Databases
show dbs

Use/Create a Database
use database_name

Drop a Database

db.dropDatabase()

Show Current Database
Db

Connect to a MongoDB Server

Mongosh

Show All Users
db.getUsers()

2. Setting Up the Backend (Node.js Application)

Node.js is chosen as a backend technology because of its speed, scalability, ease of use with JavaScript, and suitability for modern application requirements like real-time features, microservices, and high concurrency.

The .env file is a simple way to manage configuration for your application, making it secure, flexible, and easier to maintain. The nano .env command allows you to edit or create this file conveniently.

1. MONGODB_URL=mongodb://localhost:27017/demo

This line defines the connection string for MongoDB, a NoSQL database. It tells the application how to connect to a MongoDB database. Let's break it down:

- **MONGODB_URL**:
  - This is the name of the environment variable.
  - It holds the MongoDB connection string, which the application uses to connect to the database.
- **mongodb://**:
  - The protocol used for connecting to the MongoDB server.
  - Similar to http:// for web traffic, mongodb:// specifies the connection type for MongoDB.
- **localhost**:
  - The hostname of the MongoDB server.
  - localhost indicates that the MongoDB server is running on the same machine as the Node.js application.
- **27017**:
  - The default port for MongoDB.
  - The application connects to the MongoDB server using this port.
- **demo**:
  - The name of the database to which the application connects.
  - If the demo database doesn't exist, MongoDB will create it when data is first inserted.

2. PORT=3000

This line defines the port on which the Node.js application will listen for incoming HTTP requests.

- **PORT**:
    - The name of the environment variable specifying the port number for the server.
- **3000**:
    - The port number on which the application's backend server will run.
    - When the application starts, it will listen for HTTP requests on http://localhost:3000 (or the server's IP/hostname if deployed remotely).
    -

What is nodeapp.service?

The nodeapp.service file is a **systemd service file**. Systemd is a system and service manager for Linux, widely used to manage services (applications or processes) that run in the background. This file defines how a Node.js application should be started, stopped, restarted, and managed by systemd.

By creating this service file, you enable the Node.js application to start automatically at boot time, handle unexpected crashes, and run in the background as a managed service.

Breakdown of the nodeapp.service Content

[Unit]

- **Description**: Provides a short description of the service. Here, it's described as a "Node.js Application Service."
- **After**: Ensures the service starts only after the specified target (network.target), which ensures that networking is up before the application starts.

[Service]

- **User**: Specifies the user under which the service runs (ubuntu in this case). This ensures security by limiting permissions to the ubuntu user.
- **Group**: Specifies the group under which the service runs.
- **WorkingDirectory**: Sets the directory where the application resides, ensuring the service starts from the correct location (/home/ubuntu/NodeJs-API).
- **ExecStart**: Defines the command to start the service. Here, it runs the Node.js application (server.js) using Node (/usr/bin/node).
- **Restart**: Ensures the service restarts automatically if it crashes.
    - always: The service will always restart after failure or termination.
- **RestartSec**: Specifies the delay (in seconds) before restarting the service (10 seconds).

[Install]

- **WantedBy**: Specifies the target in the boot sequence when the service should be started. multi-user.target is commonly used for non-graphical (server) environments.

sudo npm install pm2 -g

- **What It Does**:
  - Installs **PM2**, a process manager for Node.js, globally on your system.
  - PM2 makes it easier to manage Node.js applications by handling processes, ensuring uptime, and providing features like logging, monitoring, and load balancing.
- **Why We Need It**:
  - **Keeps Node.js Applications Running**: Automatically restarts your app if it crashes.
  - **Process Management**: Simplifies managing multiple Node.js apps with commands like start, stop, restart, etc.
  - **Monitoring**: Offers built-in monitoring tools (pm2 monit).
  - **Cluster Mode**: Supports scaling your application across multiple CPUs.

2. pm2 start server.js

- **What It Does**:
  - Starts your Node.js application (server.js) using PM2.
  - PM2 takes control of the process, managing it in the background and ensuring it keeps running.
- **Why We Need It**:
  - This command ensures the application runs as a managed background process.
  - It allows you to monitor and manage the process with PM2's tools.
  - Unlike running node server.js manually, PM2 ensures better fault tolerance and easier process management.

3. node server.js

- **What It Does**:
  - Directly runs your Node.js application (server.js) in the foreground using the Node.js runtime.
- **Why We Need It**:
  - Useful for development and debugging.
  - Not ideal for production since the app will stop if the terminal session ends or the process crashes.

PM2 vs node:

- PM2 is a better choice for production because it runs apps in the background, restarts them on failure, and provides monitoring tools.
- Running node server.js directly is more suitable for development or quick testing.

4. sudo systemctl status nodeapp.service

- **What It Does**:
  - Checks the status of the nodeapp.service managed by **systemd**.
- **Why We Need It**:
  - Verifies whether the Node.js application (configured as a systemd service) is running.
  - Shows information like whether the service is active, when it started, and logs of the application.
  - Ensures that the systemd-managed service is working correctly.

1. Setting Up MongoDB (Database Layer)

Update the package list
#sudo apt update

Install MongoDB
#sudo apt-get install -y mongodb-org

Start and enable MongoDB service:

#sudo systemctl start mongod

#sudo systemctl enable mongod

Check if MongoDB is running correctly by status

#sudo systemctl status mongod

Connect to the MongoDB shell
#mongosh

2. Setting Up the Backend (Node.js Application)

Install Node.js and npm (Node package manager)

sudo apt install -y nodejs

Cloning the Application Source

```
#git clone https://github.com/BL-AniketChile/NodeJs-API.git
```

Open .env file

```
#nano .env
```

Add the following environment variable in .env file

```
MONGODB_URL=mongodb://localhost:27017/demo
PORT=3000
```

Creating .service file in /etc/systemd/system

```
#cd /etc/systemd/system
#ls
#nano nodeapp.service
```

Add the following content in nodeapp.service file

```
[Unit]
Description=Node.js Application Service
After=network.target

[Service]
User=ubuntu
Group=ubuntu
WorkingDirectory=/home/ubuntu/NodeJs-API
ExecStart=/usr/bin/node server.js
Restart=always
RestartSec=10

[Install]
WantedBy=multi-user.target
```

Starting the node application as a system service

```
#sudo systemctl start nodeapp
#sudo systemctl enable nodeapp
#sudo systemctl status nodeapp
```

```
#cd NodeJs-API
```

We can also use a process manager like pm2 to keep the app running in the background

```
#sudo npm install pm2 -g
#pm2 start server.js
#node server.js
#sudo systemctl status nodeapp.service (It should be active: running)
```

3. <u>Setting Up the Frontend (Apache Web Server)</u>

Install Apache web server

```
#sudo apt update
#sudo apt install apache2
```

Start and enable Apache

```
#sudo systemctl start apache2
#sudo systemctl enable apache2
```

Configure Apache as a Reverse Proxy (Enable proxy modules)

- The Apache web server will act as a reverse proxy to route frontend traffic to Node.js backend application.
- Apache web server will forwards the user requests to backend Node application

```
#sudo a2enmod proxy
#sudo a2enmod proxy_http
```

Configure the Apache virtual host

- Create a new configuration file nodeapp.conf in /etc/apache2/sites-available/ directory

```
#sudo nano /etc/apache2/sites-available/nodeapp.conf
```

Add the following proxy settings inside the <VirtualHost *:80> block:

```
<VirtualHost *:80>
    ServerName default

    ProxyRequests Off
    ProxyPass / http://localhost:3000/
    ProxyPassReverse / http://localhost:3000/

    ErrorLog ${APACHE_LOG_DIR}/error.log
    CustomLog ${APACHE_LOG_DIR}/access.log combined
</VirtualHost>
```

Note:

- We have to use 'default' as a server name instead of IP address of server in nodeapp.conf file.
- If we use IP address as Server name, we need to change every time after restarting the Ec2 instance, because Public IP address of instance will change everytime

After changing the nodeapp.conf file, we need to restart the apache

```
#sudo systemctl restart apache2
```

We need to add a Rule for HTTP (Port 80) in security group (Inbound Rules) of current Ec2 instance

- **Type**: HTTP
- **Protocol**: TCP
- **Port Range**: 80
- **Source**: My IP (This will allow access only from your current IP address)

http://Public IP address of current Ec2 instance/route

- ❖ This setup allows Apache to forward requests from the specified port (80) to Node.js application running on port 3000, enabling smooth integration of your frontend and backend
- ❖ Port 80 is default port of http

http://3.109.32.53/hello_world