1. ***Docker: Create** two applications in two different docker containers. Push those applications and run to show the communications between two Dockers. (Hint IPC)

```
mkdir A32

nano app1.js


const http = require('http');

const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello from App 1!');
});

const PORT = 3000;
server.listen(PORT, () => {
  console.log(`Server 1 running at http://localhost:${PORT}`);
});



nano app2.js



const http = require('http');

const options = {
  hostname: 'localhost',
  port: 3000,
  path: '/',
  method: 'GET',
};

const req = http.request(options, (res) => {
  let data = '';

  res.on('data', (chunk) => {
        data += chunk;
  });
  res.on('end', () => {
          console.log('Received from App 1:', data);
  });
});
```

```
nano Dockerfile1
```

```
FROM node:14

WORKDIR /app

COPY app1.js /app

CMD ["node", "app1.js"]
```

```
nano Dockerfile2
```

```
FROM node:14

WORKDIR /app

COPY app2.js /app

CMD ["node", "app2.js"]
```

```
docker build -t app1_image -f Dockerfile1 .
docker build -t app2_image -f Dockerfile2 .
```

```
docker run -d --name app1_container -p 3000:3000 app1_image
docker run --network host app2_image
```

```
docker logs app1_container
```

```
docker logs app2_container
```

Certainly! Let's break down the experiment of creating two Node.js applications within Docker containers to demonstrate communication between them using HTTP requests.

Experiment Overview:

1. **Objective:**
   - Showcase communication between two Docker containers using Node.js applications.
   - Use one container as a server (`app1`) and the other as a client (`app2`), communicating over HTTP.

2. **Components:**
   - **App 1 (Server):** A Node.js application (`app1.js`) that acts as an HTTP server, listening on a specific port (e.g., 3000).
   - **App 2 (Client):** Another Node.js application (`app2.js`) that makes an HTTP request to App 1 to retrieve data.

3. **Docker Configuration:**

- Dockerfiles: Configuration files specifying instructions to build Docker images for each application.
- Docker Image for App 1: Image named `app1_image` built from `Dockerfile1`.
- Docker Image for App 2: Image named `app2_image` built from `Dockerfile2`.

Experiment Steps:

1. **Create Node.js Applications:**
   - **App 1 (`app1.js`):** A simple HTTP server that responds with a greeting message.
   - **App 2 (`app2.js`):** An HTTP client that makes a request to App 1 and displays the received response.

2. **Create Dockerfiles for Each Application:**
   - **Dockerfile 1 (`Dockerfile1`):** Instructions to build an image for App 1.
   - **Dockerfile 2 (`Dockerfile2`):** Instructions to build an image for App 2.

3. **Build Docker Images:**
   - Use Docker commands (`docker build`) to create Docker images for both applications based on their respective Dockerfiles.

4. **Run Docker Containers:**
   - Start Docker containers for both applications:
     - **App 1 Container (`app1_container`):**
     - Run the container for App 1 (`app1`) based on the `app1_image`.
     - Expose port 3000:3000 to allow external access to the server.
     - **App 2 Container (`app2_container`):**
     - Run the container for App 2 (`app2`) based on the `app2_image`.

5. **Check Logs and Communication:**
   - View logs of both containers (`app1_container` and `app2_container`) using `docker logs` command:

    - Verify if App 1 (`app1`) is running and listening for incoming requests.
    - Confirm if App 2 (`app2`) successfully receives the response from App 1.

6. **Interpretation:**
   - Observe the log outputs to ensure successful communication between the Docker containers:
     - Confirmation of App 1 serving requests.
     - Logs from App 2 showing the data received from App 1, indicating successful communication between the containers.

Conclusion:

The experiment demonstrates communication between two Docker containers through HTTP requests, showcasing how two separate Node.js applications hosted within isolated Docker environments can interact over the network. Although not based on traditional Inter-Process Communication (IPC), the example highlights Docker's networking capabilities for inter-container communication, serving as a practical demonstration of microservices interaction within a Dockerized environment.