DevOps – Program – 03 – Documentation

Title: Code a Dockerized Python Flask or Node.js Application

This image shows the Dockerfile used to containerize the Node.js application. The Dockerfile begins with the official lightweight Node.js 20 Alpine base image to ensure an optimized and stable runtime environment. Metadata such as the maintainer's name, version, and build date are added for documentation and traceability.
The working directory `/app` is defined inside the container, and the `package.json` file is copied to install necessary production dependencies using `npm install`. The remaining source code is then copied into the container. Port **3000** is exposed to allow external access to the application, and the command `CMD ["npm", "start"]` specifies how the Node.js application should be executed when the container starts.

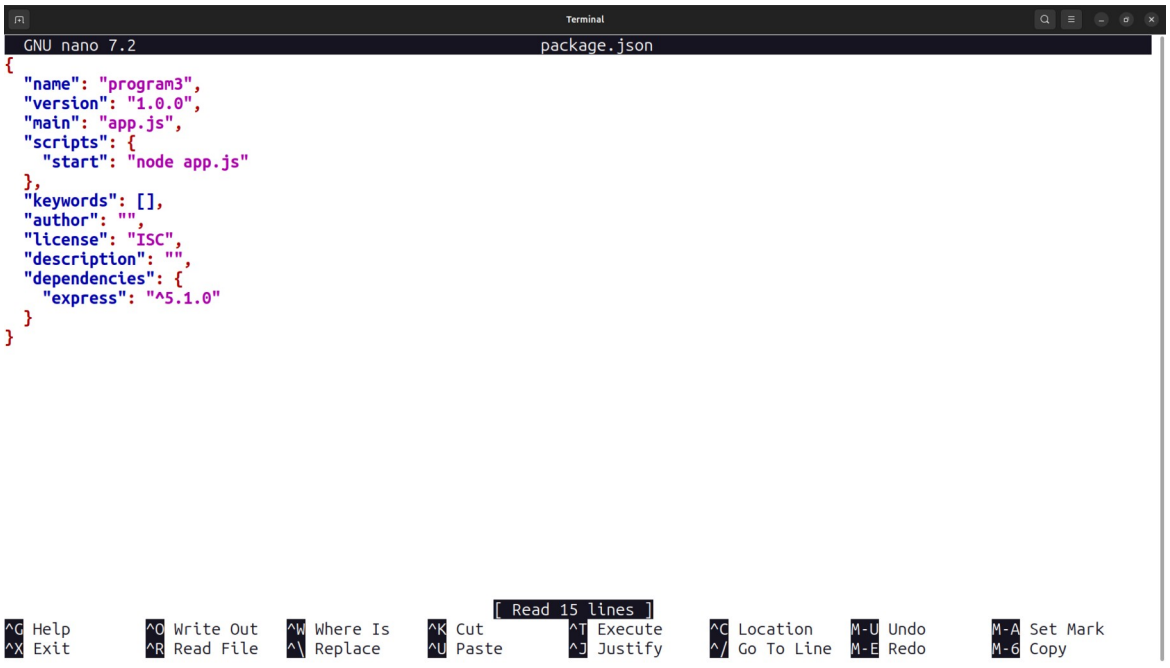
This Dockerfile forms the foundation for building the container image, encapsulating all dependencies and configurations needed to run the application consistently across environments.

This screenshot shows the implementation of a simple Express.js server in the `app.js` file. The code defines a minimal Node.js web application that runs inside a Docker container. It imports the Express framework and creates an application instance that handles HTTP requests.
Two routes are defined:

- GET '/' — returns a JSON response with a message confirming that the Node.js application is running inside Docker.
- POST '/echo' — receives data from the client and echoes it back in JSON format.

The server listens on port 3000, or on a custom port defined by the environment variable `PORT`. The message "Server is running on port ${PORT}" is displayed in the console upon successful startup.

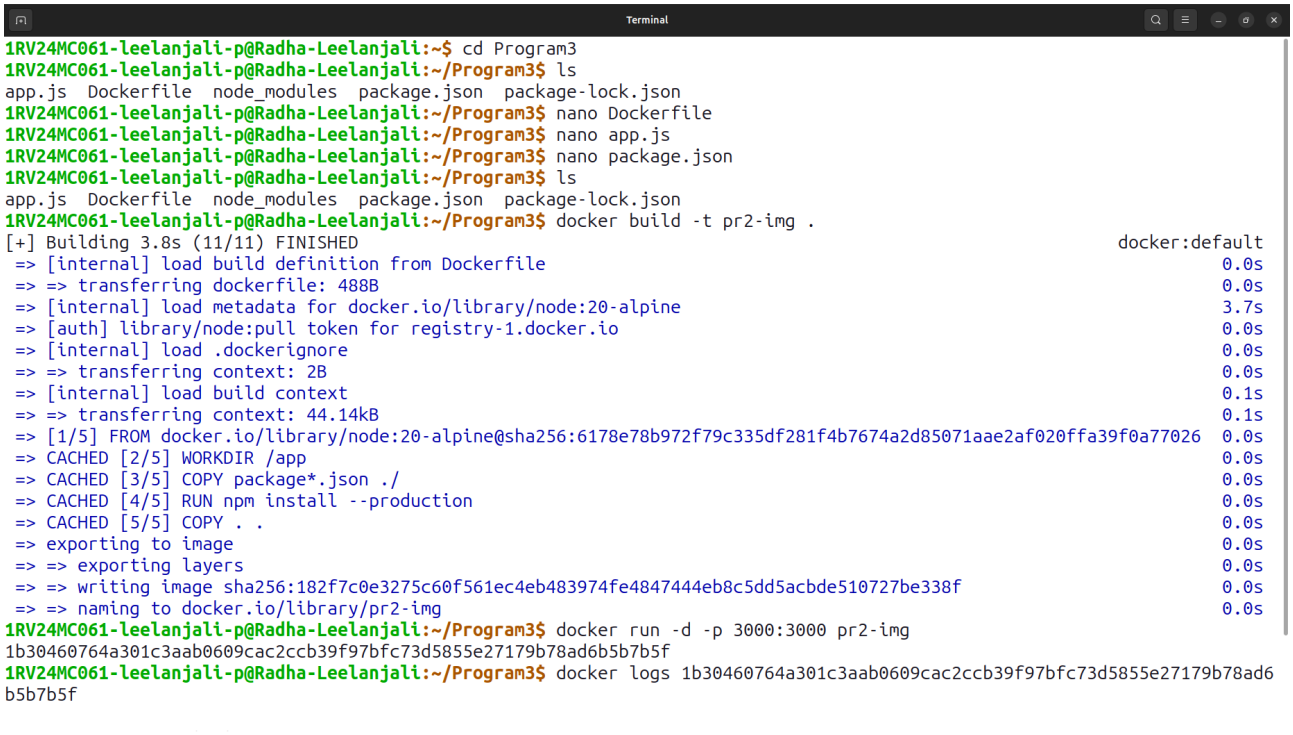
This file forms the core backend logic of the Dockerized application, demonstrating how a lightweight web server can be deployed in a containerized environment.



This image displays the `package.json` file, which defines the project metadata and dependencies for the Node.js application. It includes essential information such as the project name (`program3`), version, main entry point (`app.js`), and startup command (`node app.js`).

The `"dependencies"` section lists Express.js (version ^5.1.0) as the primary framework used to build and run the web server. The `"scripts"` section specifies the `"start"` command that is executed when the container runs the application.

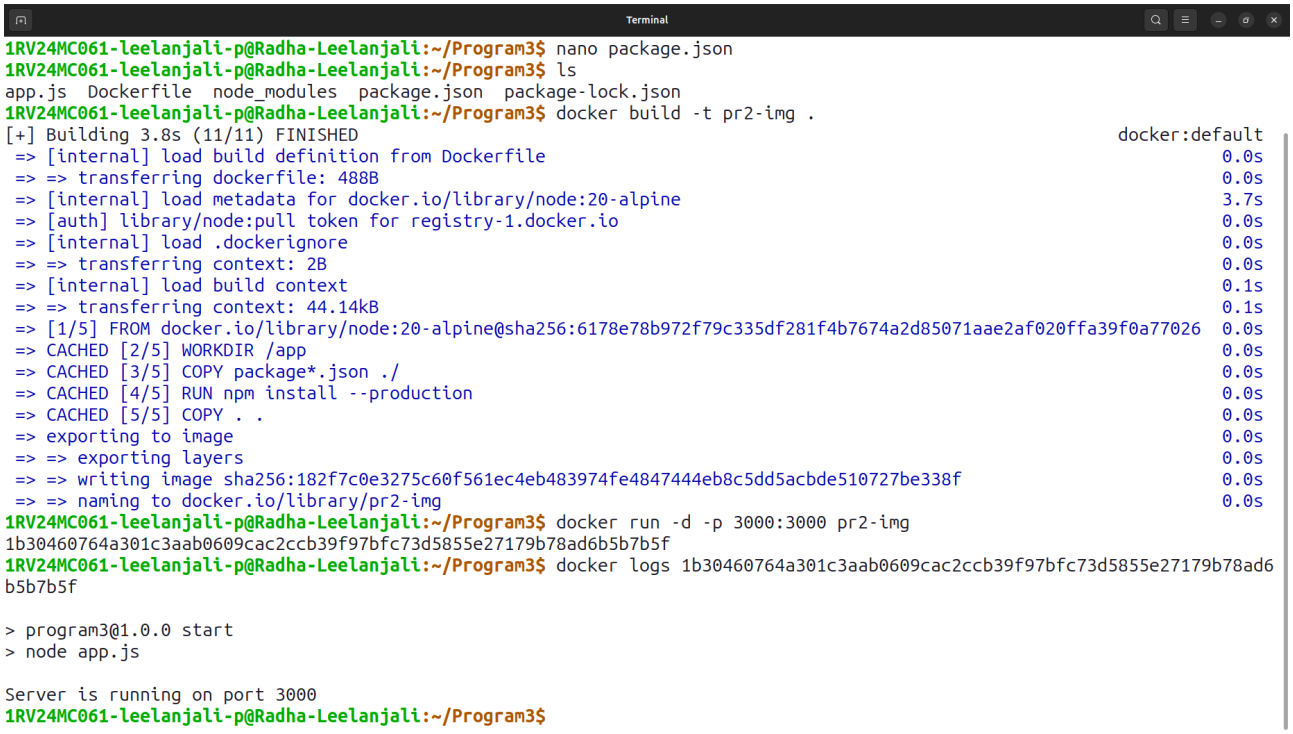
This file plays a crucial role in dependency management and automation of execution commands, ensuring that all required packages are installed and the application can be started seamlessly within the Docker environment.



This shows the steps to build the Docker image and start a container.

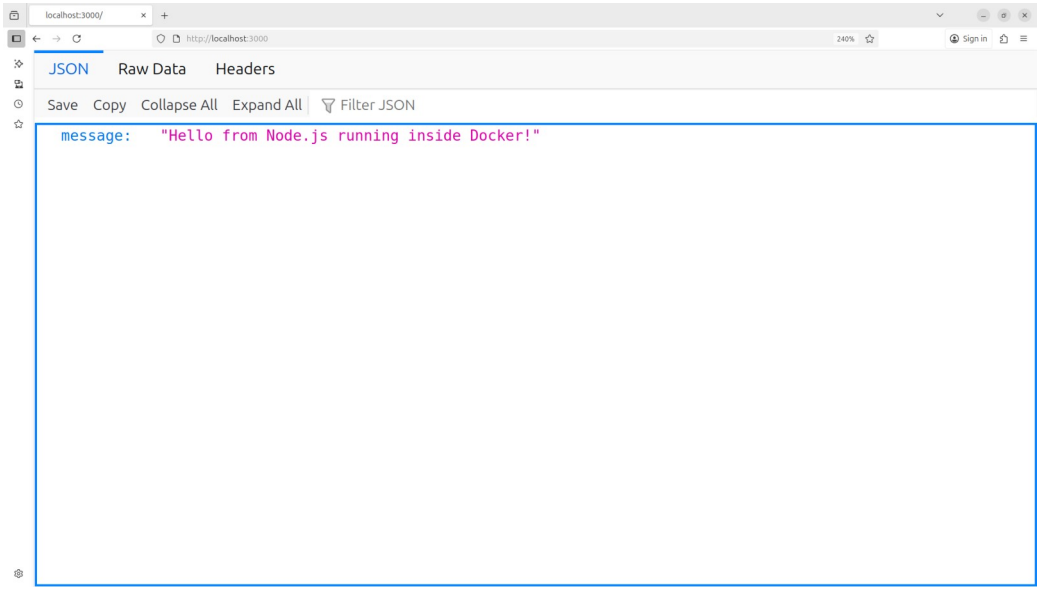
- Pre-commands: Shows directory change (`cd Program3`) and file listings (`ls`).
- Docker Build:
  - Command: `docker build -t pr2-img .`
  - Purpose: Builds the Docker image using the `Dockerfile` in the current directory, tagging it as `pr2-img`. The output shows the successful 11-step build process, leveraging cache (`CACHED`) for several steps.
- Docker Run:
  - Command: `docker run -d -p 3000:3000 pr2-img`
  - Purpose: Starts a container from the `pr2-img` image.
    - `-d`: Runs the container in detached mode (background).

        - `-p 3000:3000`: Maps the container's Port 3000 to the host machine's Port 3000, making the application accessible.
    - The command outputs the container ID (`1b30460...`).
- Docker Logs:
    - Command: `docker logs <container_id_prefix>`
    - Purpose: Retrieves the output logs from the running container.



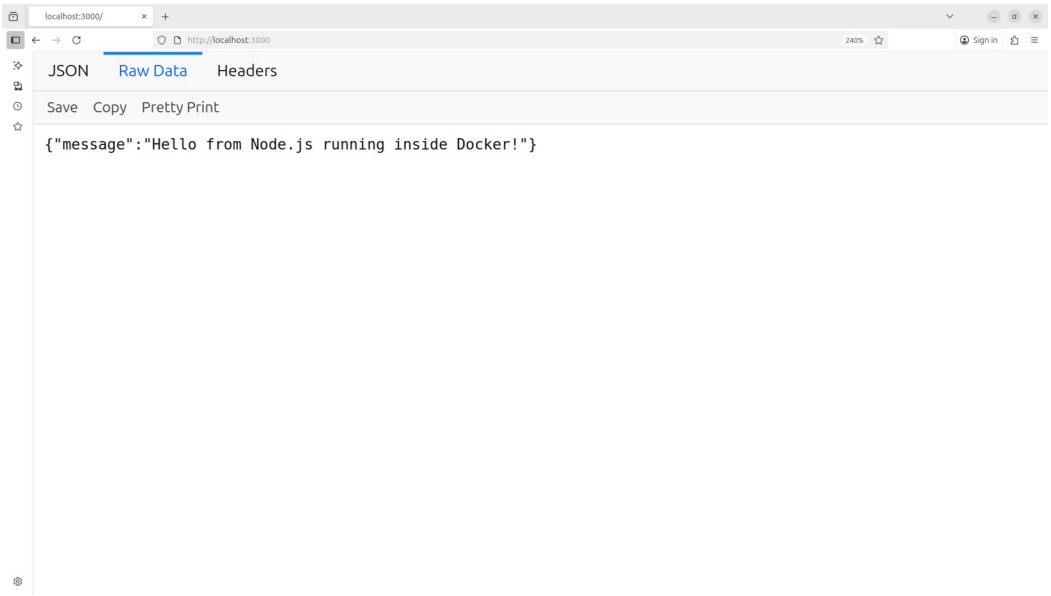
This continues from the previous screenshot, showing the output from the `docker logs` command.

- Logs Output: `Server is running on port 3000`
- Context: This log confirms that the Node.js application successfully started *inside* the Docker container, listening on its internal Port 3000, which has been mapped to the host's Port 3000. This is the final step before testing connectivity.



This shows the application being successfully accessed via a web browser.

- URL: `http://localhost:3000/`
- Result: The browser displays the JSON response from the Node.js application's root endpoint (`/`).
- Content: `{"message": "Hello from Node.js running inside Docker!"}`
- Verification: This confirms the entire Dockerization process was successful: the image was built, the container is running, the port forwarding is working, and the application is responding correctly.



This is an alternate view of the successful browser test.

- URL: `http://localhost:3000/`

- View: Displays the same successful JSON response but in the "Raw Data" format, showing the plain text output received from the server.
- Confirmation: Reinforces the successful deployment and access of the Dockerized Node.js application.