

# Using Docker Env Vars



How can Docker environment variables be used by a node.js application within the container?



To use Docker environment variables in a Node.js application running within a Docker container, you can follow these steps:

## 1. Define Environment Variables in Docker

Environment variables in Docker can be defined in various ways, such as:

- **Directly in the `Dockerfile`:** You can use the `ENV` keyword to define environment variables within the Dockerfile.
- **Using the `docker run` command:** You can pass environment variables when running the container using the `-e` flag.
- **With a `.env` file:** You can store environment variables in a `.env` file and reference them when running the container using Docker Compose or passing them via the `docker run` command.

## 2. Access Environment Variables in Node.js

Inside your Node.js application, you can access environment variables using `process.env`. This is part of Node.js's `process` object that exposes the environment to the application.

### Steps:

#### 1. Using the Dockerfile

Dockerfile

```
# Dockerfile

# Base image
FROM node:14

# Set environment variables
ENV PORT=3000
ENV NODE_ENV=production

# Copy app and install dependencies
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .

# Expose the port and start the app
EXPOSE $PORT
CMD ["npm", "start"]
```

In this case, `PORT` and `NODE_ENV` are set as environment variables and will be available within the container.

## 2. Using the docker run Command

You can also set environment variables when you run the Docker container:

```
bash

docker run -e PORT=3000 -e NODE_ENV=production -p 3000:3000 my-node-app
```

## 3. Using a `.env` File with Docker Compose

You can use a `.env` file for Docker Compose or Docker itself:

- **Create a `.env` file:**

```
bash

# .env
PORT=3000
NODE_ENV=production
```

- **Reference it in `docker-compose.yml`:**

```
yaml

version: '3'
services:
  nodeapp:
    image: my-node-app
    ports:
      - "3000:3000"
    env_file:
      - .env
```

## 4. Accessing Environment Variables in Node.js

In your Node.js application, you can access these environment variables using `process.env`:

```
js

// app.js

const port = process.env.PORT || 3000;
const nodeEnv = process.env.NODE_ENV || 'development';

console.log(`Server is running on port ${port} in ${nodeEnv} mode`);

const express = require('express');
const app = express();

app.get('/', (req, res) => {
```

```
res.send(`Environment: ${nodeEnv}`);
});

app.listen(port, () => {
  console.log(`App listening on port ${port}`);
});
```

## Summary

- **Set environment variables** in the Dockerfile (`ENV`), with `docker run` (`-e`), or using a `.env` file in Docker Compose.
- **Access these variables** inside your Node.js application using `process.env`.

This approach allows your Node.js app to be easily configurable for different environments like production, development, or testing.



How do the environment variables end up in the process object with this approach?



When you define environment variables in Docker (either in the `Dockerfile`, through the `docker run` command, or via a `.env` file), they become part of the environment for any processes running inside the container. Here's how these environment variables are passed into the Node.js application:

## How Environment Variables End Up in `process.env` in Node.js

### 1. Setting Environment Variables in Docker

- Docker sets environment variables at the operating system (OS) level within the container. These variables are part of the environment that the container's processes will inherit.

### 2. Process Inheritance

- When a new process is started (e.g., starting your Node.js application inside the container), it automatically inherits the environment variables from the parent process (in this case, the Docker container). This is a standard feature of operating systems: child processes inherit the environment from the parent.

### 3. Node.js and `process.env`

- In Node.js, the `process.env` object is a special part of the global `process` object, which represents the currently running Node.js process.
- `process.env` is automatically populated by Node.js with all environment variables that the process inherits when it starts. This includes environment variables set by Docker.
- When your Node.js app starts, it reads the environment variables from the OS environment (in this case, the Docker container) and makes them available via `process.env`.

## The Flow of Environment Variables

Here's the step-by-step flow of how environment variables end up in the `process.env` object:

### 1. Environment Variables Defined in Docker

- **In the Dockerfile** (e.g., `ENV PORT=3000`), the `ENV` directive sets environment variables at build time.
- **Using `docker run -e`** (e.g., `docker run -e PORT=3000`), Docker passes environment variables directly to the container when it starts.
- **With `.env` files and Docker Compose**, variables defined in the `.env` file (e.g., `PORT=3000`) are loaded into the environment of the container through the `docker-compose.yml` file.

## 2. Container Starts

- When the Docker container starts, these environment variables are part of the container's runtime environment, similar to how environment variables work on a local machine.

## 3. Node.js Application Launches

- The Node.js application is launched as a process inside the container (typically via `CMD ["node", "app.js"]` or `CMD ["npm", "start"]` in the Dockerfile).
- When Node.js starts, it automatically reads the environment from the OS (the container in this case) and makes it accessible through `process.env`.

## 4. Access in Node.js

- You can now access these variables in your Node.js application via `process.env`. For example:

```
js
const port = process.env.PORT || 3000;
console.log(`Server will run on port ${port}`);
```

## Example: Using Docker and Environment Variables

### 1. Dockerfile:

#### Dockerfile

```
# Base image
FROM node:14

# Set environment variable in Dockerfile
ENV PORT=3000

# Copy and install application
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .

# Expose the port defined in environment variable
EXPOSE $PORT
CMD ["node", "app.js"]
```

### 2. Running the Container

You can override or define additional environment variables at runtime using `docker run`:

bash

```
docker run -e NODE_ENV=production -e PORT=8080 -p 8080:8080 my-node-app
```

In this case, `PORT` is set to `8080`, and the Node.js app will access this via `process.env.PORT`.

### 3. Accessing in Node.js:

js

```
// app.js
const express = require('express');
const app = express();

const port = process.env.PORT || 3000;
const nodeEnv = process.env.NODE_ENV || 'development';

app.get('/', (req, res) => {
  res.send(`App running in ${nodeEnv} mode on port ${port}`);
});

app.listen(port, () => {
  console.log(`Server is running on port ${port} in ${nodeEnv} mode`);
});
```

When the container starts, it passes the environment variables (`PORT` and `NODE_ENV`) to the Node.js app, and you can access them using `process.env.PORT` and `process.env.NODE_ENV` respectively.

## Conclusion

Environment variables set in Docker are passed to the Node.js app through the container's environment. The Node.js process, running within the container, inherits these variables and makes them available through `process.env`, allowing the application to access and use them.



So from this, all environment variables passed in via Docker are set at the OS Level?

Therefore, other processes/applications running on that OS would also be able to access them?



In the context of Docker, environment variables set for a container **do not exist at the host OS level** but are specific to the container's environment. Let's break this down:

### 1. Environment Variables in Docker Are Container-Specific

- Each Docker container has its **own isolated environment**. When you pass environment variables to a Docker container (whether via a `Dockerfile`, `docker run`, or Docker Compose), these variables are set **only within the container**.

- The environment variables are accessible only to the processes running inside that particular container, not the host operating system (OS) or other containers.

## 2. Container Isolation (Namespaces and Cgroups)

- Docker uses Linux kernel features like **namespaces** and **cgroups** to isolate containers. These isolation mechanisms ensure that:
  - **Namespaces:** Each container has its own isolated view of the filesystem, process tree, network stack, and environment. This means that processes running in different containers or on the host OS have separate environments, and they cannot see each other's environment variables.
  -



Correction - I meant within the container only, not the host OS Level.



Got it! Yes, within the Docker container, the environment variables passed in are set at the **container's OS level**, which means:

### 1. All Processes Inside the Container Share the Environment

- Any environment variables passed into the container (via the `Dockerfile`, `docker run`, or `.env` file) are part of the container's OS environment.
- Since all processes inside the container share the same environment, any application or process running within the container can access these environment variables.

For example:

- If you run a Node.js app and a shell (`bash`, `sh`, etc.) inside the same container, both can access the same environment variables via `process.env` in Node.js or `echo $VARIABLE_NAME` in a shell.

### 2. Environment Variables Are Isolated to the Container

- The environment variables are scoped only to the container. Other containers or processes running outside the container (on the host or in other containers) cannot access them unless explicitly shared (e.g., via Docker networks or volumes, but not directly as environment variables).

#### Example:

Suppose you set environment variables inside a Docker container:

```
bash
```

```
docker run -e MY_ENV_VAR=myvalue -it node:14 bash
```

- Inside the container, you can start multiple processes (e.g., Node.js app, bash, or other tools).

- All processes inside the container can access the `MY_ENV_VAR` environment variable:
  - In **Node.js**, you can access it via `process.env.MY_ENV_VAR`.
  - In a **bash shell**, you can access it via `echo $MY_ENV_VAR`.

## Summary:

- All environment variables passed into a Docker container are available to **all processes running within that container**.
- These variables are isolated to the container and cannot be accessed by other containers or the host OS unless explicitly shared through other mechanisms.

ChatGPT can make mistakes. Check important info.