**Docker project from the scratch:**

Here I'm going to build a small application using Node.js, Nest.js and typescript to run a small HTTP request to print “hello docker” on a web server.

Steps that we should follow:

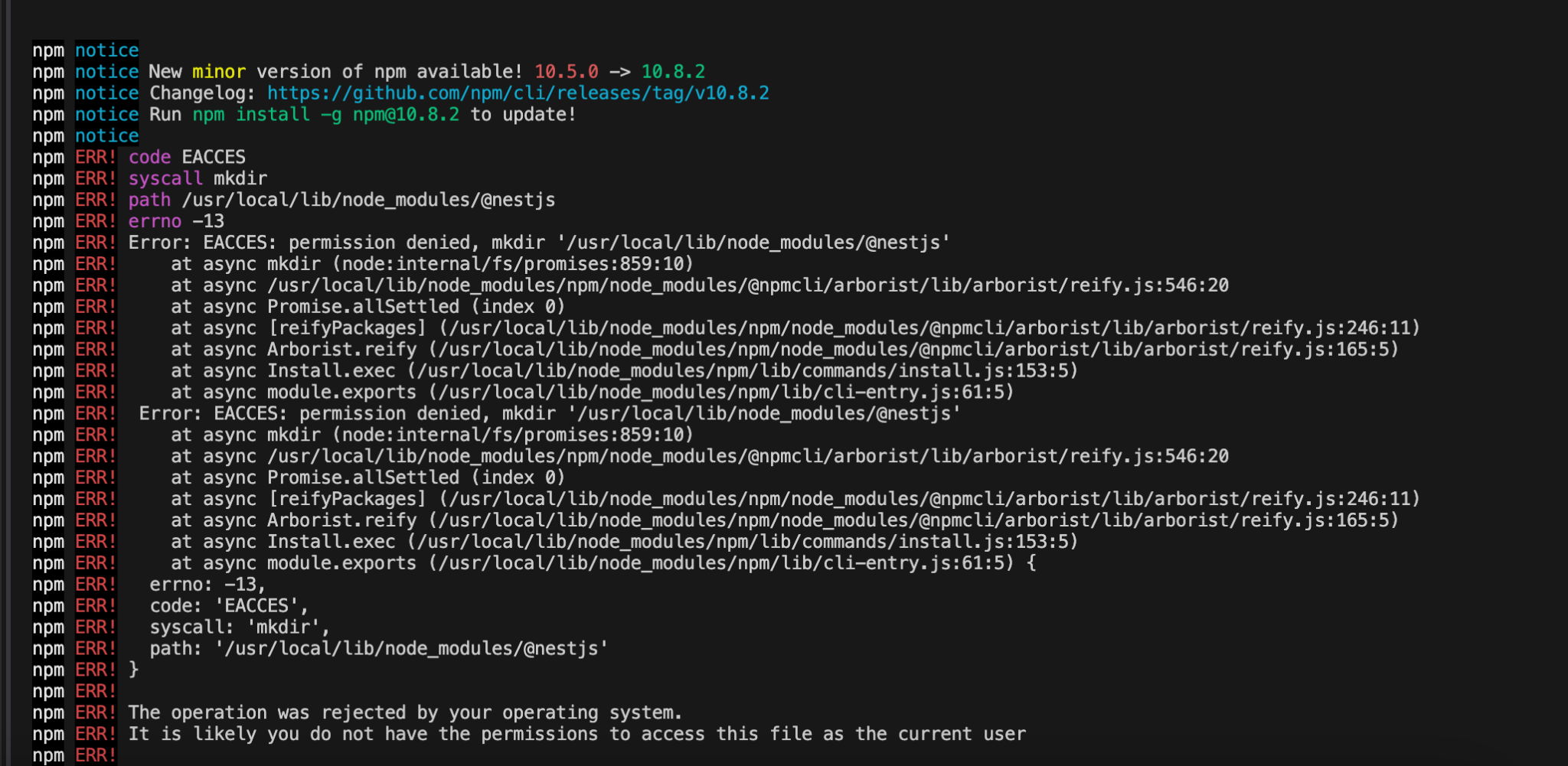
1. Install all the dependencies we need to run this application.

First we need nestCLI

$ *npm install -g @nestjs/cli*

In that above command -g represents global which means this will be installed globally in the system instead of installing it only for the current project.

While installing nestjs/cli i faced an issue



How i resolved it and what does it means

1. The above error is related to the permissions.
2. Permission Error: The main issue is that npm doesn't have the necessary permissions to create directories in the global npm installation location (/usr/local/lib/node\_modules/@nestjs).

**To resolve this issue, you have a few options:**

1. Use sudo (not recommended for security reasons): You could run the command with sudo to give it the necessary permissions:  
    Copy  
   sudo npm install -g @nestjs/cli  
    However, this is generally not recommended as it can lead to permission issues in the future.
2. **Safer way to resolve this issue is :**

Change npm's default directory: A safer approach is to change npm's default directory to one that your user owns:

1. Create a directory for global installations:

$ mkdir ~/.npm-global

2. Configure npm to use the new directory path:

$npm config set prefix '~/.npm-global'

3. Add the new path to your ~/.profile (or ~/.zshrc if you're using Zsh):

Before doing this we need to check in which shell you are in

$echo $SHELL

Based on the result:

* If it shows /bin/bash, you should use .bash\_profile or .bashrc
* If it shows /bin/zsh, you should use .zshrc

4. Let's assume you're using Zsh (which is the default on newer macOS versions). In this case, you should: a. Open your .zshrc file:

Copy

nano ~/.zshrc

Add the following line at the end of the file:

Copy

export PATH=~/.npm-global/bin:$PATH

Save and exit (in nano, you can do this by pressing Ctrl+X, then Y, then Enter)

5. Now, source your updated .zshrc file:

source ~/.zshrc

Finally Then install nest cli

$ *npm install -g @nestjs/cli*

Step 2: To create a nest project using the following command

$ nest new docker-nestjs-hello-world

After entering this command

Which package manager would you ❤️ to use? (Use arrow keys)

❯ npm

yarn

pnpm

This prompt is from the NestJS CLI, which is asking you to choose a package manager for your project. You have three options:

npm (Node Package Manager):

This is the default package manager that comes with Node.js.

It's the most widely used and has the largest ecosystem.

yarn:

Developed by Facebook as an alternative to npm.

Known for its speed and deterministic installations.

pnpm:

A newer alternative that aims to be faster and more efficient with disk space.

Uses a unique approach to manage dependencies.

The choice depends on your preferences and project requirements:

If you're familiar with npm and don't have specific needs, it's a safe choice.

If you want potentially faster installations and like its features, you might choose yarn.

If you're working on projects with many shared dependencies or are concerned about disk space, pnpm could be a good option.

You can use the arrow keys to select your preferred option. The "❯" indicates your current selection.

After this command was successfully ran then it will create a project with the name which we have provided along with the project structure contains files which are required to run an application.

Step 3: move to the project folder

$ cd docker-nestjs-hello-world

This will take you to the project directory

Then write the code in the controller and service files.

Controller contains all the routes created. In this project i have created one get request method t display hello world.

So the code for the controller class is

import { Controller, Get } from '@nestjs/common';

import { AppService } from './app.service';

@Controller()

export class AppController {

constructor(private readonly appService: AppService) {}

@Get()

getHello(): string {

return this.appService.getHello();

}

}

In the above code

@controller annotation is used to specify that this class is a controller class.

And @get annotation indicates that it is the get method.

Inside the Constructor there is a class mentioned that means that this class is a dependency class to the controller. When we declare it like this the nestjs framework will take care of creating instances for this class when there is a need for the methods of that class or the variables.

Below is the get method to call the service class method which is getHello().

Service class code

import { Injectable } from '@nestjs/common';

@Injectable()

export class AppService {

getHello(): string {

return 'Hello World!';

}

}

The @Injectable() decorator marks this class as a **provider** that can be injected as a dependency.

All the providers are singleton which means only once the instance will be created . whenever it requires it uses the same instance for all the places.

The @Injectable() decorator is applied to the AppService class.

This decorator marks the class as a provider in NestJS.

By doing this, you're telling NestJS that this class can be managed by its dependency injection system.

Now, let's discuss how this works in the NestJS ecosystem:

1. Registration:
   * When NestJS bootstraps your application, it scans for classes decorated with @Injectable().
   * It registers AppService in its dependency injection container.
2. Instantiation:
   * NestJS will create a single instance of AppService (by default, providers are singletons).
   * This instance will be reused across your entire application.
3. Injection:
   * When another class (like a controller) needs to use AppService, it can declare it as a dependency in its constructor.
   * NestJS will automatically provide the instance of AppService.

Usage example:  
 typescript  
Copy  
@Controller()

export class AppController {

constructor(private readonly appService: AppService) {}

@Get()

getHello(): string {

return this.appService.getHello();

}

1. }  
   * Here, AppController declares AppService as a dependency.
   * NestJS injects the AppService instance when creating AppController.
   * The controller can then use methods from AppService, like getHello().
2. Benefits:
   * Separation of concerns: AppService can focus on business logic, while AppController handles HTTP requests.
   * Testability: You can easily mock AppService when testing AppController.
   * Modularity: You can replace the implementation of AppService without changing the classes that depend on it.
3. Lifecycle management:
   * NestJS manages the lifecycle of AppService.
   * If you implement lifecycle hooks (like onModuleInit), NestJS will call them at the appropriate times.

The @Injectable() decorator is a key part of making your NestJS application modular, maintainable, and adhering to the dependency inversion principle. It allows NestJS to handle the complex task of managing dependencies, letting you focus on writing business logic.

The next step is to create dockerfile by mentioning all the instructions to build an image

Then .dockerignore to include the files or folders which should be ignored while containerization.

Then build the image using the below command

$docker build -t hello-world-app .

Then run the image to an instance of that which is container

docker run -p 3000:3000 hello-world-app

Test your application by opening a web browser and navigating to http://localhost:3000. You should see the message "Hello World!".

This project demonstrates:

* Setting up a basic Nest.js project with TypeScript
* Creating a simple REST API endpoint
* Writing a basic Dockerfile
* Building a Docker image
* Running a Docker container

Key Docker concepts you're learning:

* FROM: Specifies the base image (Node.js in this case)
* WORKDIR: Sets the working directory in the container
* COPY: Copies files from your local system to the container
* RUN: Executes commands in the container during the build process
* EXPOSE: Informs Docker that the container listens on the specified port
* CMD: Specifies the command to run when the container starts