CI/CD with Docker

Prepared by:
Ms. Avita Katal
Assistant Professor (SG)
School of Computer Science
UPES, Dehradun

Real Time Case

- Imagine you're creating a website. And you want to host it to a server because you want to reach it to the people. So, after finishing the website, you upload that to the server. Then you see a **few glitches on your website**. Also, you are thinking of **updating your website at a particular time**. In both cases, website **users will experience downtime,** which will be a negative sight for your reputation.

- Now, you must follow a **CI/CD Docker approach for the code repository**. Here you can push the changes on the repository, and you can test, run and deploy from it to your web and fix issues. This CI/CD pipeline doesn't affect the server downtime.

CI/CD refers to **Continuous Integration/Continuous Delivery.**

- *It is **not a tool**.*
- *It is a **continuous methodology for SDLC**(Software Development Life Cycle).*
- *If you **run CI/CD with docker containers**, you can call it CI/CD Docker.*
- *With the perfect application of the docker container or hub, you can improve the overall experience of CI/CD workflows without reaching its limit.*

The main focus of the CI/CD pipeline is to deliver and integrate changes continuously.

- *First, you must **create a workflow** and run it on every push to the master branch.*
- *It **triggers a process that runs on Docker**.*
- *Then you can **tell the docker container what to do***

Before CI/CD Pipeline

Before the automation process of CI/CD exists, **developers develop the entire code and then deploy it on the repository**. They have to write the entire code, and only then can they push it to the repository. After that, the code is put to the test. If any bugs appear, the developers must track down and correct the problem from the entire coding. So, the problem arises like:

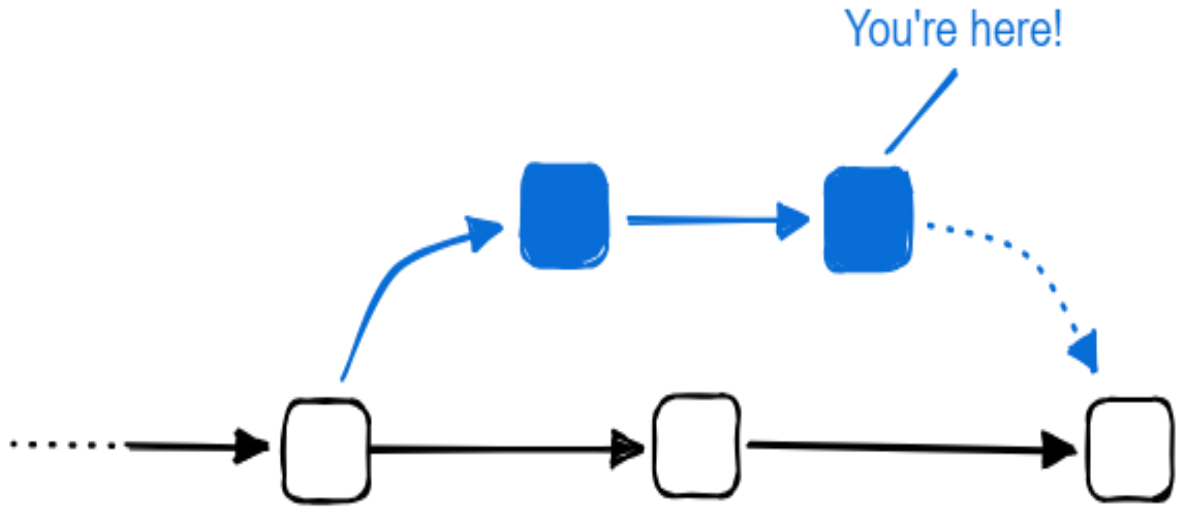
- *It consumes* ***a lot of time and resources for fixing***.
- *It takes* ***longer to go from design to testing***.
- ***Not convenient for developers***.

Thus, this disadvantage will increase the hassle of deploying the codes at the production level.

Continuous integration with Docker

- Continuous Integration (CI) is the part of the **development process where you're looking to get your code changes merged with the main branch of the project.** At this point, development teams run tests and builds to vet that the code changes don't cause any unwanted or unexpected behaviors.

- There are several uses for Docker at this stage of development, even if you don't end up packaging your application as a container image.



Continuous integration with Docker

Docker as a build environment

- Containers are reproducible, isolated environments that yield predictable results. **Building and testing your application in a Docker container makes it easier to prevent unexpected behaviors from occurring.** Using a ***Dockerfile, you define the exact requirements for the build environment, including programming runtimes, operating system, binaries, and more***.

- **Using Docker to manage your build environment also eases maintenance**. For example, updating to a ***new version of a programming runtime can be as simple as changing a tag or digest in a Dockerfile.*** No need to SSH into a pet VM to manually reinstall a newer version and update the related configuration files.

- Additionally**, just as you expect third-party open source packages to be secure, the same should go for your build environment**. You can scan and index a builder image, just like you would for any other containerized application.

GitHub Actions

GitHub Actions is a popular CI/CD platform for automating your build, test, and deployment pipeline. Docker provides a set of official GitHub Actions for you to use in your workflows. These official actions are reusable, easy-to-use components for building, annotating, and pushing images.

The following GitHub Actions are available:

Build and push Docker images: build and push Docker images with **BuildKit.**

Docker Login: sign in to a **Docker registry.**

Docker Setup Buildx: initiates a **BuildKit builder.**

Docker Metadata action: extracts metadata from **Git reference and GitHub events**.

Docker Setup QEMU: installs QEMU static binaries for multi-arch builds.

Docker Buildx Bake: enables using **high-level builds with Bake**.

Docker Scout: analyze Docker images for security vulnerabilities.

Get started with GitHub Actions

You will complete the following steps:

1. Create a new repository on GitHub.

2. Define the GitHub Actions workflow.

3. Run the workflow.

Step one: Create the repository

1. Create a GitHub repository and configure the Docker Hub secrets.
2. Create a new GitHub repository using this template repository. *(https://github.com/new?template_name=clockbox&template_owner=dvdksn)*
3. The repository contains a simple Dockerfile, and nothing else.
4. Open the repository Settings, and go to Secrets and variables > Actions.
5. Create a new secret named DOCKERHUB_USERNAME and your Docker ID as value.
6. Create a new secret named DOCKERHUB_PASSWORD.
7. Create a new Personal Access Token (PAT) for Docker Hub. You can name this token clockboxci.
   (https://docs.docker.com/docker-hub/access-tokens/)
7. Add the PAT as a second secret in your GitHub repository, with the name DOCKERHUB_TOKEN.
8. With your repository created, and secrets configured, you're now ready for action!

Step two: Set up the workflow

1. Set up your GitHub Actions workflow for building and pushing the image to Docker Hub.

2. Go to your repository on GitHub and then select the Actions tab.

3. Select set up a workflow yourself.

4. This takes you to a page for creating a new GitHub actions workflow file in your repository, under **.github/workflows/main.yml** by default.

5. In the editor window, copy and paste the following YAML configuration.

```
name: ci

on:
  push:
    branches:
      - "main"

jobs:
  build:
    runs-on: ubuntu-latest
```

name: the name of this workflow.
on.push.branches: specifies that this workflow should run on every push event for the branches in the list.
jobs: creates a job ID (build) and declares the type of machine that the job should run on.

For more details :https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions

Dockerfile

```
FROM node:latest
WORKDIR /usr/src/app
COPY package.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD ["node", "index.js"]
```

Step three: Define the workflow steps

Now the essentials: what steps to run, and in what order to run them.

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      -
        name: Checkout
        uses: actions/checkout@v3
      -
        name: Login to Docker Hub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}
      -
        name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v2
      -
        name: Build and push
        uses: docker/build-push-action@v4
        with:
          context: .
          file: ./Dockerfile
          push: true
          tags: ${{ secrets.DOCKERHUB_USERNAME }}/clockbox:latest
```

The previous YAML snippet contains a sequence of steps that:

- Checks out the repository on the build machine.
- Signs in to Docker Hub, using the Docker Login and your Docker Hub credentials.
- Creates **a BuildKit builder instance using the Docker Setup Buildx.**
- Builds the **container image and pushes it to the Docker Hub repository**, using Build and push Docker images.
- The with key lists a number of input parameters that configures the step:
  - **context:** the build context.
  - **file:** filepath to the Dockerfile.
  - **push:** tells the action to upload the image to a registry after building it.
  - **tags:** tags that specify where to push the image.

Add these steps to your workflow file. The full workflow configuration should look as follows:

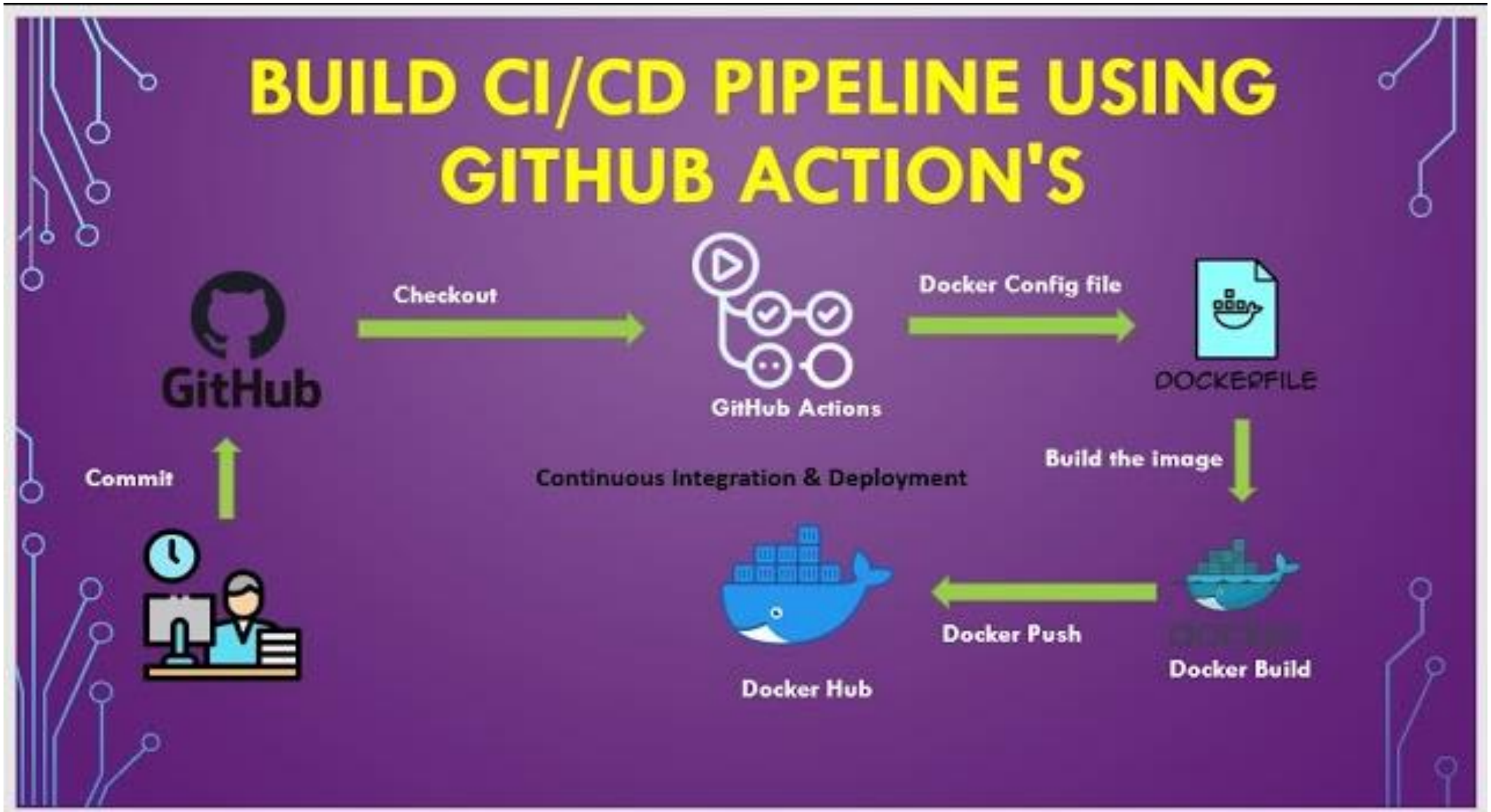
```
name: ci

on:
  push:
    branches:
      - "main"

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      -
        name: Checkout
        uses: actions/checkout@v3
      -
        name: Login to Docker Hub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}
      -
        name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v2
      -
        name: Build and push
        uses: docker/build-push-action@v4
        with:
          context: .
          file: ./Dockerfile
          push: true
          tags: ${{ secrets.DOCKERHUB_USERNAME }}/clockbox:latest
```

Run the workflow

1. Save the workflow file and run the job.
2. Select Commit changes... and push the changes to the main branch.
3. After pushing the commit, the workflow starts automatically.
4. Go to the Actions tab. It displays the workflow.
5. Selecting the workflow shows you the breakdown of all the steps.
6. When the workflow is complete, go to your repositories on Docker Hub
7. If you see the new repository in that list, it means the GitHub Actions successfully pushed the image to Docker Hub!


BUILD CI/CD PIPELINE USING GITHUB ACTION'S
GitHub
Checkout
GitHub Actions
Docker Config file
DOCKERFILE
Build the image
Commit
Continuous Integration & Deployment
Docker Hub
Docker Push
Docker Build

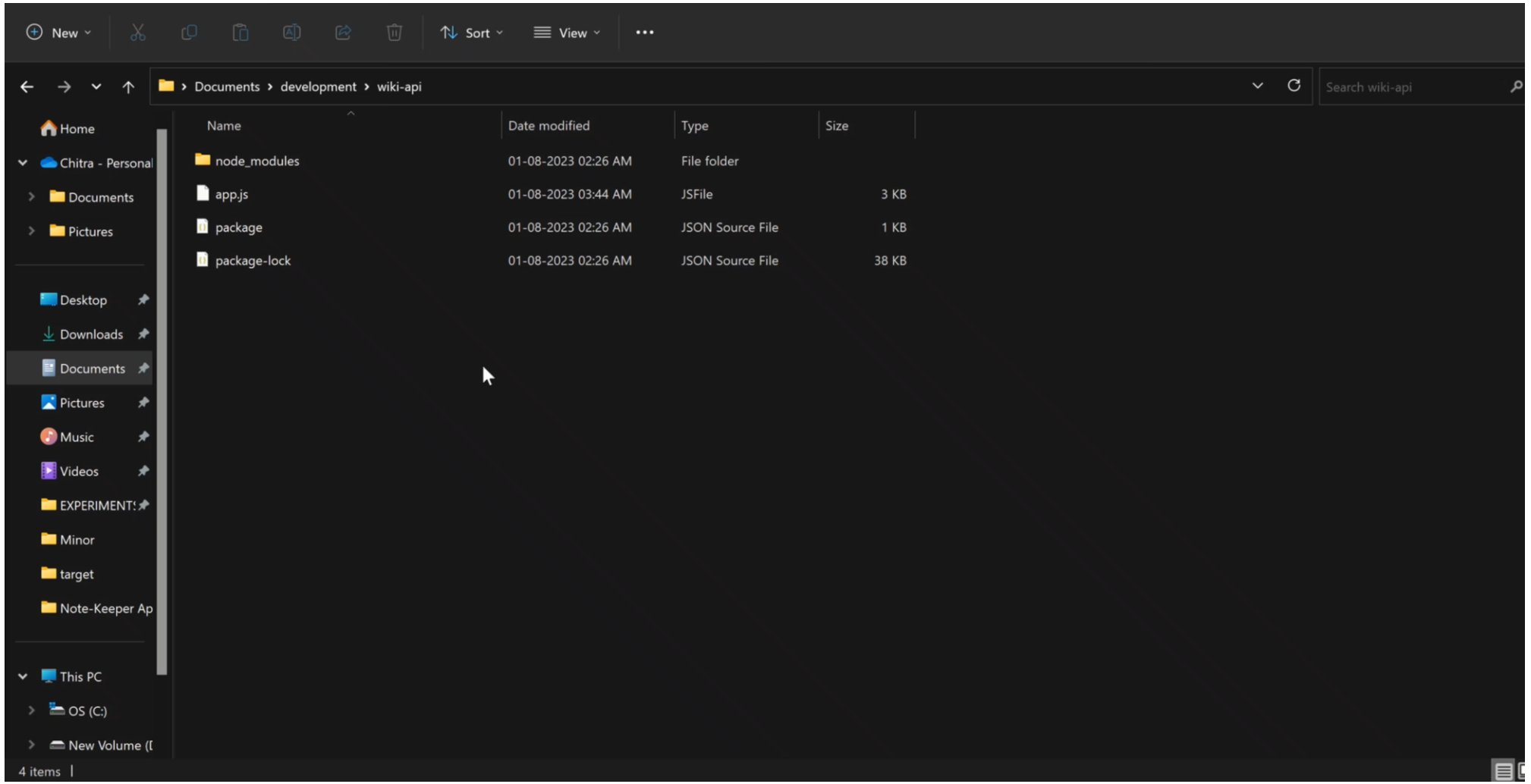
DEMO

References:

If you don’t know about GitHub yet, please go through this video tutorial:
https://www.youtube.com/watch?v=vfTBfDsEuIM

CI/CD Pipeline using Docker and GitHub
https://www.youtube.com/watch?v=euEkYEFCrI8 (Node js)
https://docs.docker.com/language/java/configure-ci-cd/

DEMO:


New
Sort
View
Documents › development › wiki-api
Search wiki-api
Home
Chitra - Personal
Documents
Pictures
Desktop
Downloads
Documents
Pictures
Music
Videos
EXPERIMENT
Minor
target
Note-Keeper Ap
This PC
OS (C:)
New Volume (
Name
Date modified
Type
Size
node_modules
01-08-2023 02:26 AM
File folder
app.js
01-08-2023 03:44 AM
JSFile
3 KB
package
01-08-2023 02:26 AM
JSON Source File
1 KB
package-lock
01-08-2023 02:26 AM
JSON Source File
38 KB
4 items