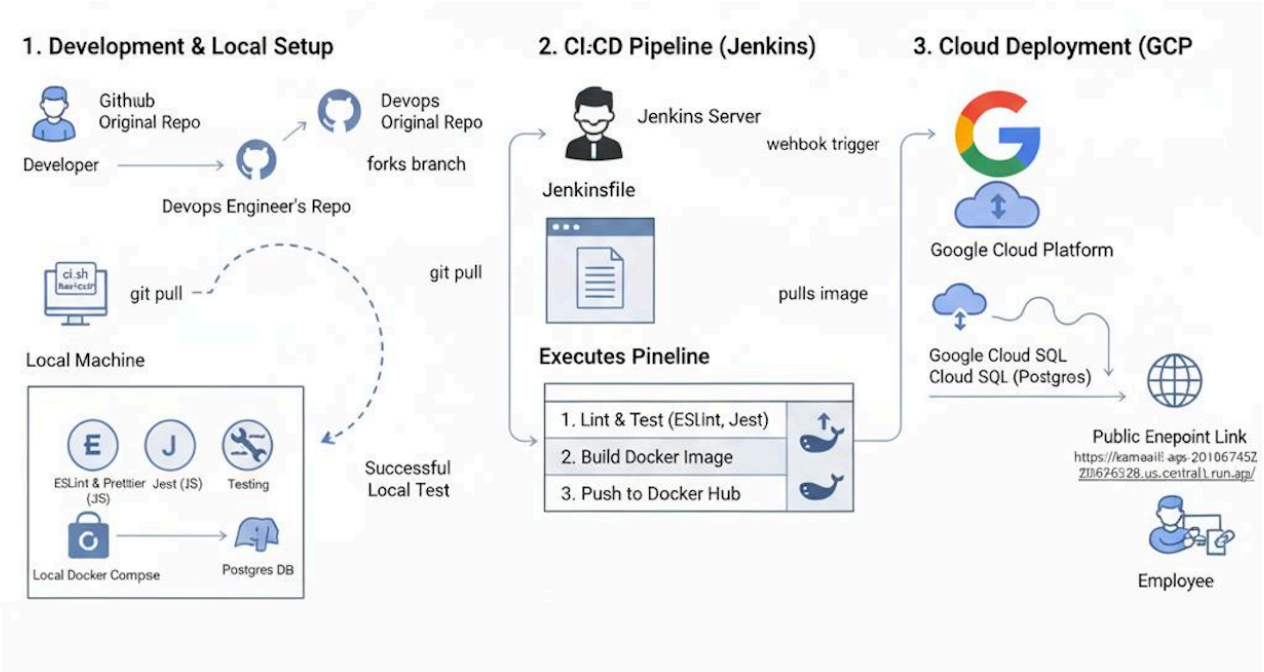
Project 1 Hands-on experience

Team Availability – CI/CD Project

Welcome to my **first CI/CD project**! 👋
This documentation walks you through how I built, automated, and deployed the **Team Availability Tracker** application using modern DevOps practices and cloud-native tools.

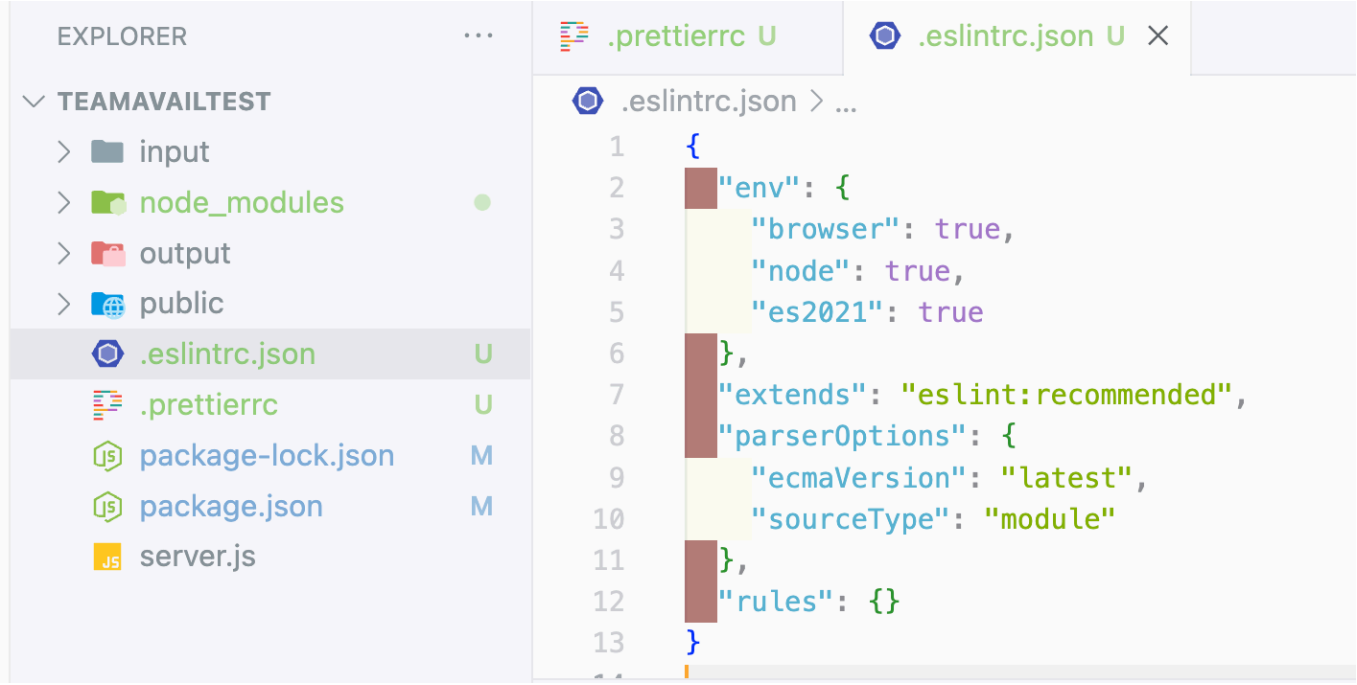
Along the way, I'll share not only the setup and implementation but also the **real errors I faced, how I solved them, and what I learned**.

Let's dive in! 🚀



- I started by **forking the project** to my own GitHub account.
- Then, I **cloned the repository** to my local machine, so I had the full project files available for development.
- Inside the project, I added the following configuration files to manage code quality:
  - `.eslintrc.json` – for JavaScript linting rules.
  - `.prettierrc` – to enforce consistent code formatting.

This setup ensures that both linting and formatting rules are applied consistently as I work on the project locally.



Prettier Test

After setting up the `.prettierrc` file, I ran a quick check to see if the existing code follows the formatting rules:

```
npx prettier --check .
```

The command scanned the project files and returned **errors**, indicating that some files did not follow the Prettier formatting rules.

> *This helped me identify areas in the code that needed formatting fixes before continuing with the pipeline setup.*

```
→  TeamavailTest git:(main) ✗ npx prettier --check .

Checking formatting...
[warn] input/selection.json
[warn] input/status.json
[warn] public/index.html
[warn] public/script.js
[warn] public/styles.css
[warn] Code style issues found in 5 files. Run Prettier with --write to fix.
```

Fixing Prettier Errors

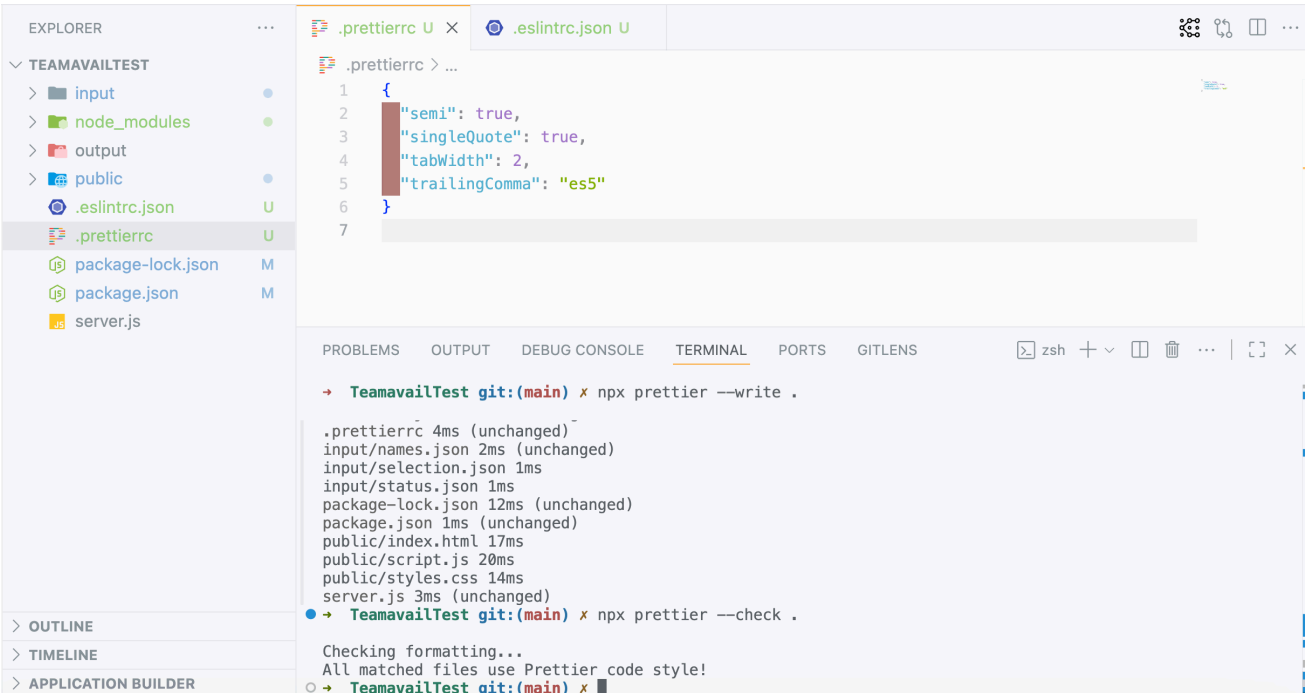
After seeing the formatting errors, I ran the following command to automatically fix them:

```
npx prettier --write .
```

This command **rewrites the code files** so that they comply with the rules defined in `.prettierrc`. Essentially, it saves time by automatically formatting everything instead of fixing

each file manually.

The command successfully updated the files, and the terminal output confirmed that all formatting issues were resolved.

> *This ensured that the project had a consistent code style, which is crucial before moving on to linting, testing, and building the CI/CD pipeline.*
>
> 
>
>
> Next, I ran **ESLint** to check for potential code errors or issues:

`npx eslint .`

- I had to **install an older version** of ESLint because the latest version wasn't working properly on my machine.
- Using the older version worked perfectly, and **no errors were reported** in the project.

```
npm install --save-dev eslint@8

npm warn deprecated rimraf@3.0.2: Rimraf versions prior to v4 are no longer supported
npm warn deprecated @humanwhocodes/config-array@0.13.0: Use @eslint/config-array instead
npm warn deprecated eslint@8.57.1: This version is no longer supported. Please see https://eslint.org/vers
ion-support for other options.

added 94 packages, and audited 162 packages in 4s

39 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
→  TeamavailTest git:(main) ✗ npx eslint .

→  TeamavailTest git:(main) ✗
```

Creating the CI Script (`ci.sh`)

The next and most important step was to **automate everything locally** using a Bash script (`ci.sh`).

- Up to this point, I had been running **Prettier** and **ESLint** manually to check for formatting and code errors.
- The task required automating these checks, along with installing dependencies and starting the project, so I could run everything with a single command.

My Approach

Before writing the full CI/CD script with Docker and Docker Compose, I first created a **minimal version** to test the automation.

The script performs the following steps:

1. **Running Prettier** – automatically formats the code.
2. **Running ESLint** – checks the code for errors or issues.
3. **Installing dependencies** – ensures all required packages are available.
4. **Starting the project** – launches the app locally.

Making the Script Executable

To run the script, I had to make it executable using:

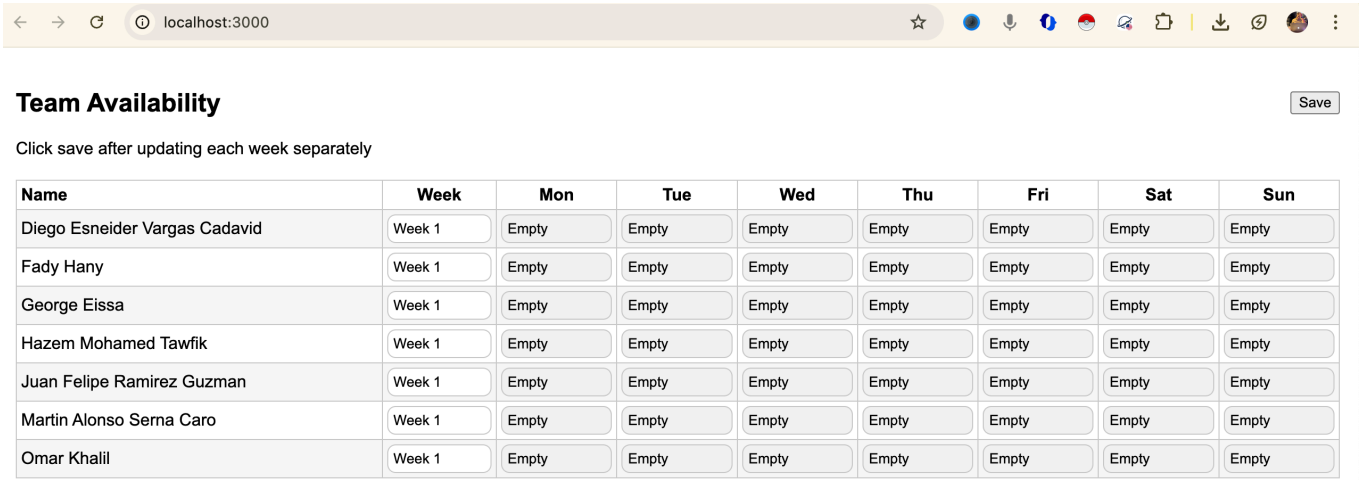
```
chmod +x ci.sh
```

Once the permissions were set, I ran the script:

```
./ci.sh
```

The command executed all the steps automatically, and the terminal output confirmed that everything worked as expected.

```
➜  TeamavailTest git:(main) ✗ chmod +x ci.sh
➜  TeamavailTest git:(main) ✗ ./ci.sh

=== Running Prettier (formatting) ===
.eslintrc.json 17ms (unchanged)
.prettierrc 4ms (unchanged)
input/names.json 2ms (unchanged)
input/selection.json 1ms (unchanged)
input/status.json 1ms (unchanged)
package-lock.json 12ms (unchanged)
package.json 1ms (unchanged)
public/index.html 15ms (unchanged)
public/script.js 21ms (unchanged)
public/styles.css 15ms (unchanged)
server.js 3ms (unchanged)
=== Running ESLint (code check) ===
=== Installing dependencies ===

up to date, audited 162 packages in 970ms

39 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
=== Starting the project ===

> version-1@1.0.0 start
> node server.js

Server running at http://localhost:3000
```



Dockerizing the Application

After confirming that all previous steps worked correctly (I like to **verify each step** to easily spot any errors), the next step was to **containerize the Node.js application**.

- I needed to create a **Dockerfile** that would build the image for my app.
- The Dockerfile, along with all the additional files I added (like `.eslintrc.json` and `.prettierrc`), was already pushed to my GitHub repository.

Building the Docker Image

I built the Docker image using the following command:

```
docker build -t teamavail-app .
```

- This created a Docker image named `teamavail-app`.
- Once the build was complete, I was ready to run the app in a container.
  -Screenshot while building the image :

```
➜  TeamavailTest git:(main) ✗ docker build -t teamavail-app .

[+] Building 119.4s (5/10)                                                  docker:desktop-linux
 => [internal] load build definition from Dockerfile                                        0.0s
 => => transferring dockerfile: 161B                                                        0.0s
 => [internal] load metadata for docker.io/library/node:20-alpine                           9.1s
 => [auth] library/node:pull token for registry-1.docker.io                                 0.0s
 => [internal] load .dockerignore                                                           0.0s
 => => transferring context: 2B                                                             0.0s
 => [1/5] FROM docker.io/library/node:20-alpine@sha256:eabac870db94f7342d6c33560d6613f188bbcf4  110.2s
 => => resolve docker.io/library/node:20-alpine@sha256:eabac870db94f7342d6c33560d6613f188bbcf4bb  0.0s
 => => sha256:127c05f5df6b075d5c314444a05d3de523268e2de5e9b235e7ba72aa60ed3c61 446B / 446B        0.7s
 => => sha256:c149c7c96aa9b49de8c5de3415d3692e81ced50773077ea0be1a0f3f36032234 1.26MB / 1.26MB   48.9s
 => => sha256:6e174226ea690ced550e5641249a412cdbefd2d09871f3e64ab52137a54ba606 0B / 4.13MB      110.1s
 => => sha256:5a8e8228254a218fbf68fbf8d7093ea99dfce63dd0a72ad0cc8be65e6da3f7c 1.05MB / 42.43MB  110.1s
 => [internal] load build context                                                           0.4s
 => => transferring context: 22.80MB                                                        0.3s
```

And it works !

```
➜  TeamavailTest git:(main) ✗ docker run -p 3000:3000 teamavail-app

Server running at http://localhost:3000
```

Setting Up Docker Compose

The next step was to create a `docker-compose.yml` file.

- This allows me to **run the project along with any additional services** easily (for example, Redis or PostgreSQL if we add them later).
- Docker Compose simplifies managing multiple containers and their dependencies, volumes, and ports in one place.

Running the Project with Docker Compose

Initially, I ran:

```
docker-compose up
```

However, I got the following error:

```
Error response from daemon: failed to set up container networking: driver failed programming external connectivity on endpoint teamavailtest-app-1 (c1ef5e851464165e7be32074c505ddc8a9331945f933905c1fd6da05b44746e0): Bind for 0.0.0.0:3000 failed: port is already allocated
```

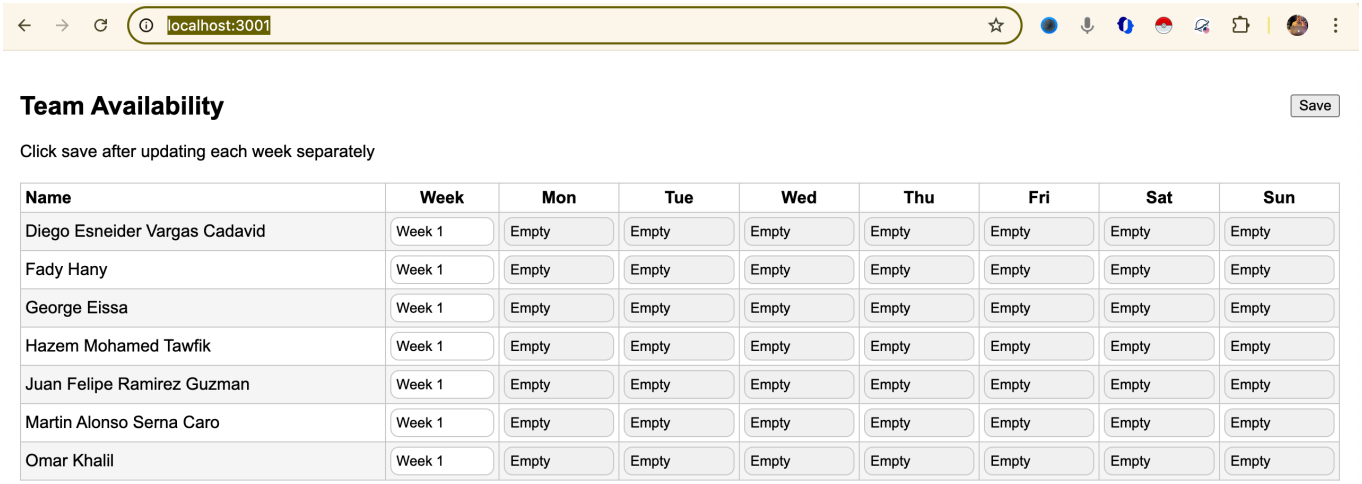
- This was expected because I already had the application running locally on **port 3000**.
- To solve this, I configured Docker to run the container on **port 3001**, while keeping the local app on port 3000.

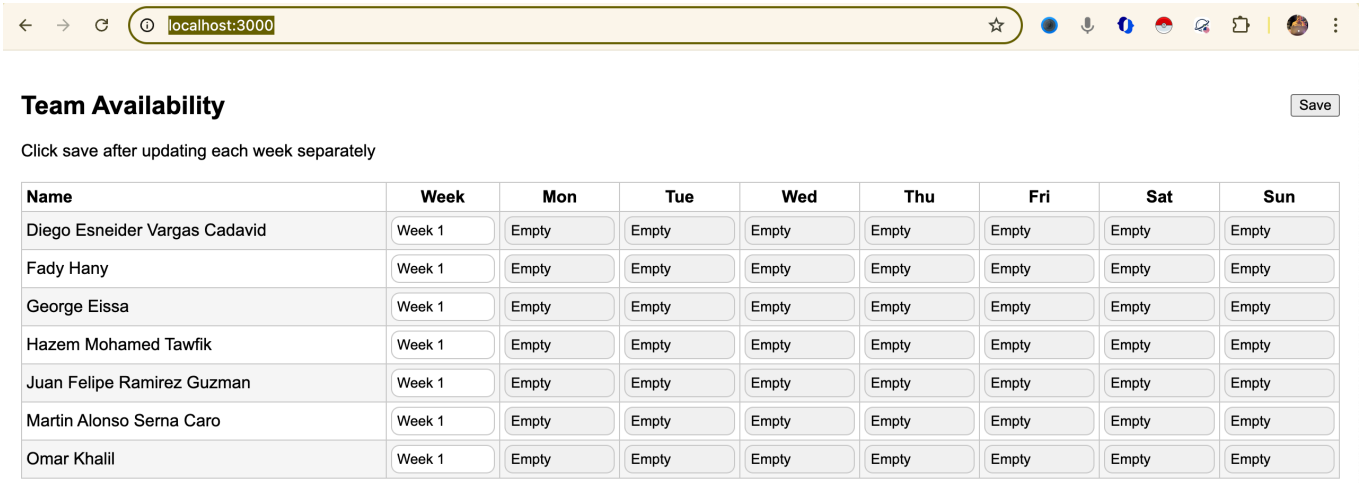
Final Setup

- Local app: `http://localhost:3000`
- Docker container: `http://localhost:3001`

Both instances ran **simultaneously without conflicts**, confirming that the Docker Compose setup works as intended.

_(Screenshots showing both the local app on port 3000 and the Docker container on port 3001 go here)





Adding Tests and Updating `ci.sh`

To make sure the project meets **100% of the task requirements**, I needed to **add automated testing**.

Adding Tests

1. I updated the `package.json` to include a test script:

```
"scripts": { "test": "jest", "start": "node server.js" }
```

2. Created a separate `tests` **folder** to hold test files.
   - For example, `server.test.js` contains tests for the application.

This ensures that the project can be tested automatically as part of the CI/CD pipeline.

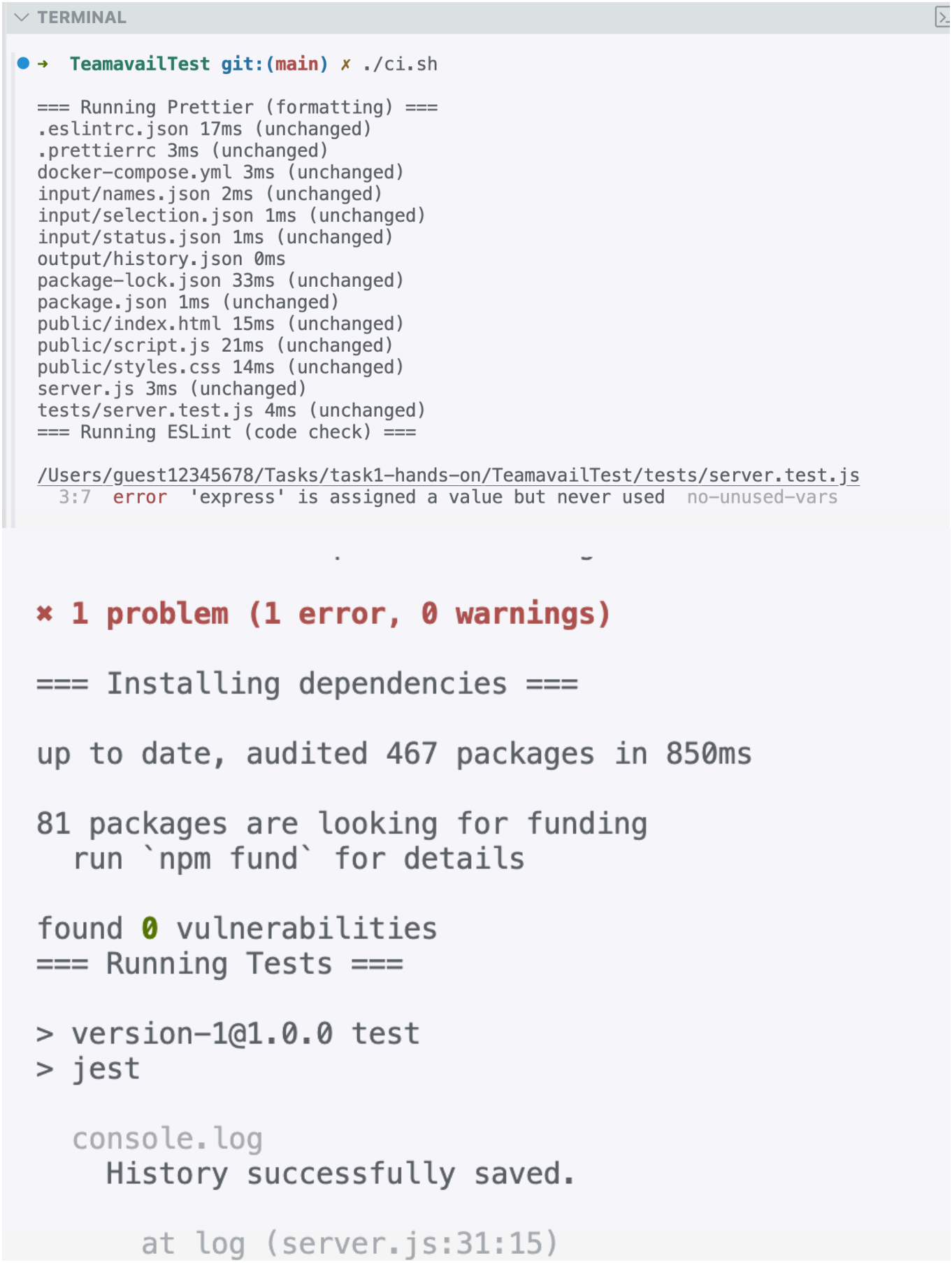
Updating `ci.sh`

After confirming that all previous steps worked correctly, I updated the `ci.sh` **script** to run the **full automated workflow**, including tests.

The updated script now runs:

1. **Prettier** – format the code automatically.
2. **ESLint** – check for linting issues.
3. **Tests** – run `jest` to verify the application works as expected.
4. **`npm install`** – install all dependencies.
5. **Docker build** – build the Docker image.
6. **Docker Compose up** – start the app along with any configured services.

> *Now, running `./ci.sh` executes the full workflow automatically, including formatting, linting, testing, dependency installation, and starting the app in Docker. This makes the project fully compliant with the task requirements.*


TERMINAL
→ TeamavailTest git:(main) ✗ ./ci.sh
=== Running Prettier (formatting) ===
.eslintrc.json 17ms (unchanged)
.prettierrc 3ms (unchanged)
docker-compose.yml 3ms (unchanged)
input/names.json 2ms (unchanged)
input/selection.json 1ms (unchanged)
input/status.json 1ms (unchanged)
output/history.json 0ms
package-lock.json 33ms (unchanged)
package.json 1ms (unchanged)
public/index.html 15ms (unchanged)
public/script.js 21ms (unchanged)
public/styles.css 14ms (unchanged)
server.js 3ms (unchanged)
tests/server.test.js 4ms (unchanged)
=== Running ESLint (code check) ===
/Users/guest12345678/Tasks/task1-hands-on/TeamavailTest/tests/server.test.js
3:7 error 'express' is assigned a value but never used no-unused-vars
✖ 1 problem (1 error, 0 warnings)
=== Installing dependencies ===
up to date, audited 467 packages in 850ms
81 packages are looking for funding
run `npm fund` for details
found 0 vulnerabilities
=== Running Tests ===
> version-1@1.0.0 test
> jest
console.log
History successfully saved.
at log (server.js:31:15)

```
➜  TeamavailTest git:(main) ✗ ./ci.sh

 PASS  tests/server.test.js
  API Tests
    ✓ GET / should return 200 (12 ms)
    ✓ POST /save-history should save JSON (26 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        0.274 s, estimated 1 s
Ran all test suites.
=== Building Docker image ===
[+] Building 4.6s (10/10) FINISHED                                  docker:desktop-li
 => [internal] load build definition from Dockerfile                               0
 => => transferring dockerfile: 161B                                               0
 => [internal] load metadata for docker.io/library/node:20-alpine                  0
 => [internal] load .dockerignore                                                  0
 => => transferring context: 2B                                                    0
 => [internal] load build context                                                  0
 => => transferring context: 1.09MB                                                0
 => [1/5] FROM docker.io/library/node:20-alpine@sha256:eabac870db94f7342d6c33560d  0
 => [5/5] COPY . .                                                               1.1s
 => exporting to image                                                           2.4s
 => => exporting layers                                                          1.8s
 => => exporting manifest sha256:05fafd390e86dc3dd9d006f0d8a9b31ab8634a5df802d4  0.0s
 => => exporting config sha256:1e229f55209efec3c297dac4fc1f3b4c4cbb007583390a281  0.0s
 => => exporting attestation manifest sha256:d7512add570238d386b1da6610648212796  0.0s
 => => exporting manifest list sha256:7a9e4f9210071e118f3c67ffb7923b14cfa2c25a6a  0.0s
 => => naming to docker.io/library/teamavail-app:latest                          0.0s
 => => unpacking to docker.io/library/teamavail-app:latest                       0.6s
=== Starting Docker Compose ===
WARN[0000] /Users/guest12345678/Tasks/task1-hands-on/TeamavailTest/docker-compose.yml: the attribute `v
ersion` is obsolete, it will be ignored, please remove it to avoid potential confusion
[+] Running 1/1
 ✔ Container teamavailtest-app-1  Running                                        0.0s
```

Test Warnings

While running the tests, everything worked fine except for **one warning** in `tests/server.test.js`:

```
3:7 error 'express' is assigned a value but never used no-unused-vars
```

- This is just a **linting warning** indicating that the `express` variable is imported but not used in the test file.
- It **does not affect the application** or any of the CI/CD workflow steps.

> *Warnings like this are common in test files, especially when placeholders or sample tests are added.*

Test Warnings and Fixes

While running the tests, everything worked fine except for **one warning** in `tests/server.test.js`:

```
3:7 error 'express' is assigned a value but never used no-unused-vars
```

- This was simply a **linting warning** indicating that the `express` variable was imported but never used in the test file.
- Such warnings are common in test files, especially when placeholder code or unused imports are left behind.
- Importantly, it **did not affect the application** or the execution of the CI/CD workflow.

How I Resolved It

To fix the issue, I opened the `server.test.js` file and **removed the unused `express` import**. After this change, the tests ran successfully without any warnings or errors.

Outcome

With the fix in place, the automation script (`ci.sh`) now runs smoothly, performing the following steps locally:

- Runs code linting and testing.
- Builds the Docker image of the application.
- Spins up the application with its dependencies using Docker Compose.

This confirms that the **local CI pipeline is fully functional**. ✅

Choosing the Right Database for Cloud Deployment

Since **Cloud Run** containers are stateless, using local volumes is not reliable—data would be lost when the container stops or scales. For persistence and reliability, the application needs an **external managed database** instead of local storage.

SQL vs NoSQL

- **SQL (e.g., PostgreSQL, MySQL):** Structured, relational data, strong consistency, ideal for history and availability tracking.
- **NoSQL (e.g., MongoDB, Redis):** Flexible schema, faster for caching or unstructured data, but not suited for complex relational queries.

Why PostgreSQL?

- **vs Redis:** Redis stores data in RAM, great for caching but not for persistent storage. My app needs reliable, long-term data storage.

- **vs MySQL:** PostgreSQL offers stronger features (e.g., JSONB, complex queries) and better fits hybrid relational + semi-structured needs.
- **vs MongoDB:** My data is relational and structured, which aligns more naturally with SQL.

Final Decision

✅ **PostgreSQL** ensures persistence, supports relational data, and integrates smoothly with **Google Cloud SQL**, making it the best fit for deploying the application on Cloud Run.

Setting Up PostgreSQL with Docker Compose

The next step was to integrate a **PostgreSQL container** into my environment.

- I updated the `docker-compose.yml` file to include a **Postgres service** with a mounted volume for persistent storage.
- I also created a folder `db/init/` containing an `init.sql` script to initialize the database schema.

Why This Step Matters

- Using a dedicated container ensures the database runs consistently across environments.
- The `init.sql` script automates schema creation, so every time the database container starts, it creates the required tables (in this case, the **history** table).
- This guarantees that the application always has the correct structure to store and query data.

Connecting the App to the Database

Instead of connecting to `localhost`, the application now connects to the **service name** (`db`) defined in Docker Compose.

Since I wanted to avoid changing too much in `server.js`, I created a separate file `db.js` responsible for handling the database connection:

```
const { pool, waitForDb } = require('./db');
```

Inside `db.js`, I required the official PostgreSQL library:

`const { Pool } = require("pg");`

This clean separation allowed me to:

- Keep `server.js` focused on the application logic.
- Manage all database-related setup and connections in a single place (`db.js`).

✅ With this, the application is now **fully set up to use PostgreSQL** as its backend database.

Troubleshooting Issues During Database Integration

While setting up PostgreSQL with Docker Compose, I ran into a couple of issues that were valuable learning points.

---

1. Docker Daemon Not Running

The first issue happened when I tried to build and run the services:

`docker-compose up --build`

I immediately got this error:

```
unable to get image 'postgres:15-alpine': Cannot connect to the Docker daemon at unix:///Users/guest12345678/.docker/run/docker.sock. Is the docker daemon running?
```

👉 This simply meant the **Docker daemon was not running** on my machine.
After starting Docker Desktop and retrying the command, the problem was resolved.

⚡ **Lesson:** Always make sure the Docker daemon is up before building or running containers.

---

2. Missing `pg` Module

After fixing the daemon issue, I ran into a more significant error:

`Error: Cannot find module 'pg'`

This indicated that the **Postgres client library (`pg`)** for Node.js was missing inside the container.

Root Cause

- In the `Dockerfile`, `npm install` had correctly installed dependencies into `/app/node_modules`.
- But in the `docker-compose.yml`, I had: `volumes: - ./:/app` This bind-mounted my local project folder over `/app` inside the container, **overwriting everything** in the image—including `node_modules`.

⚠️ **Highlight:** Bind mounts replace what's in the container with what's on the host. This wiped out the installed dependencies, including `pg`.

Solution

To solve this, I introduced a **named volume** for `node_modules`:

```
volumes: - ./:/app - node_modules:/app/node_modules
```

And defined it at the bottom of the file:

```
volumes: node_modules:
```

This way:

- Application code is still bind-mounted for development.
- Dependencies are stored in a persistent Docker-managed volume, no longer overwritten.

After this adjustment, the `pg` module was available and the application connected to PostgreSQL successfully. ✅

Key Takeaways

1. **Always start Docker daemon** before building/running containers.
2. **Be careful with bind mounts** — they can overwrite dependencies installed in the image.
3. Use **named volumes** for `node_modules` to keep dependencies consistent and isolated.

3. Port Conflict on 5432

After solving the dependency issue, another problem appeared when repeatedly running `docker-compose up`.

The **Postgres container** would start normally, initialize the database, and listen on port `5432`. However, Docker threw this error:

```
Error response from daemon: ports are not available: exposing port TCP
0.0.0.0:5432 -> 127.0.0.1:0: listen tcp 0.0.0.0:5432: bind: address already in
```

`use`

👉 This meant that **something else was already using port 5432** on the host machine.

```
 Container teamavailtest-db-1  Stopping
Error response from daemon: ports are not available: exposing port TCP 0.0.0.0:5432 ->
127.0.0.1:0: listen tcp 0.0.0.0:5432: bind: address already in use
➜  TeamavailTest git:(main) ✗
➜  TeamavailTest git:(main) ✗
➜  TeamavailTest git:(main) ✗ lsof -i :5432

➜  TeamavailTest git:(main) ✗ pgrep postgres

366
387
388
389
391
392
393
➜  TeamavailTest git:(main) ✗ kill -9 366 387 388 389 391 392 393
```

Investigation

1. I first tried checking which process was bound to port `5432` using:

   `lsof -i :5432`

   Surprisingly, no results came back—even though the error persisted.
2. To dig deeper, I checked for any running PostgreSQL processes with:

   `pgrep postgres`

   This revealed several leftover Postgres instances still running in the background.

Solution

I terminated those processes using `kill` and retried `docker-compose up`.
This time, the container started successfully without the port conflict. ✅

⚡ **Lesson Learned:**

- Port conflicts can occur if a local Postgres service (or a previously launched container) is still running.
- Tools like `lsof` and `pgrep` help identify hidden processes.
- Always make sure to clean up or stop existing services before binding the same port in Docker.

4. Init Script Conflict

Finally, I encountered another error during database initialization:

```
psql:/docker-entrypoint-initdb.d/init.sql: error: could not read from input file: Is a directory
```

```
db-1   | psql:/docker-entrypoint-initdb.d/init.sql: error: could not read from input file: Is a directory
```

Root Cause

While running `docker-compose up`, I noticed that a new **empty folder** named `init` was being created inside the `db` directory. This conflicted with my intended initialization file `init.sql`.

Because Docker mounted the directory instead of the actual file, Postgres tried to read `init.sql` but found a **directory** with the same name instead—leading to the error.

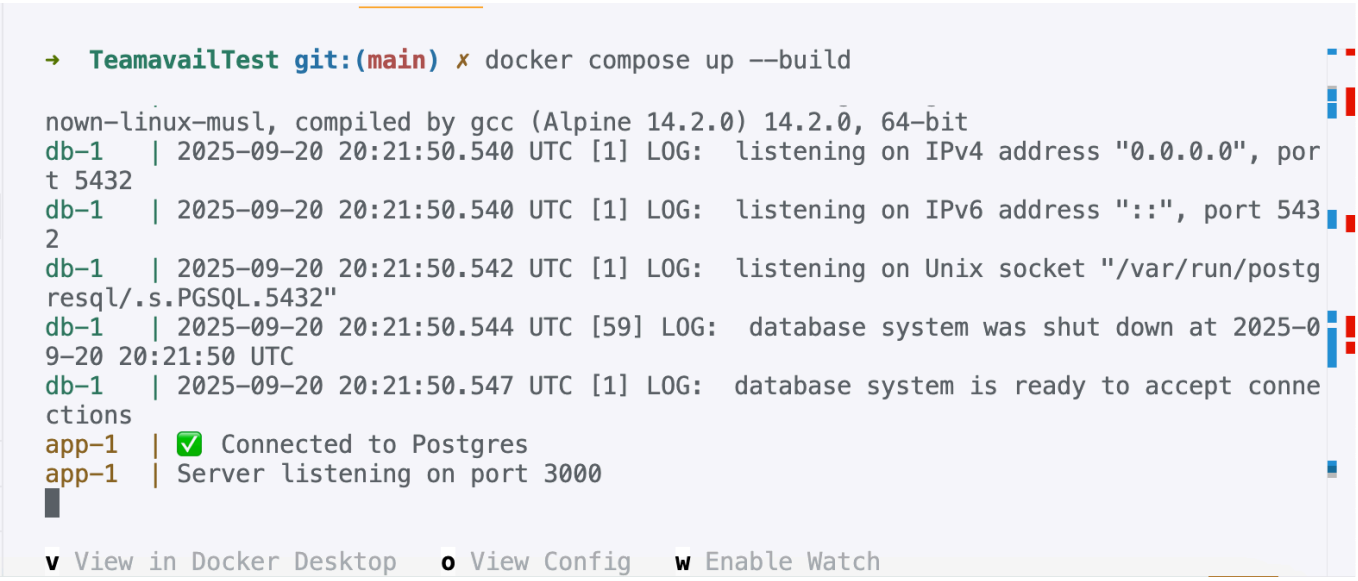
Solution

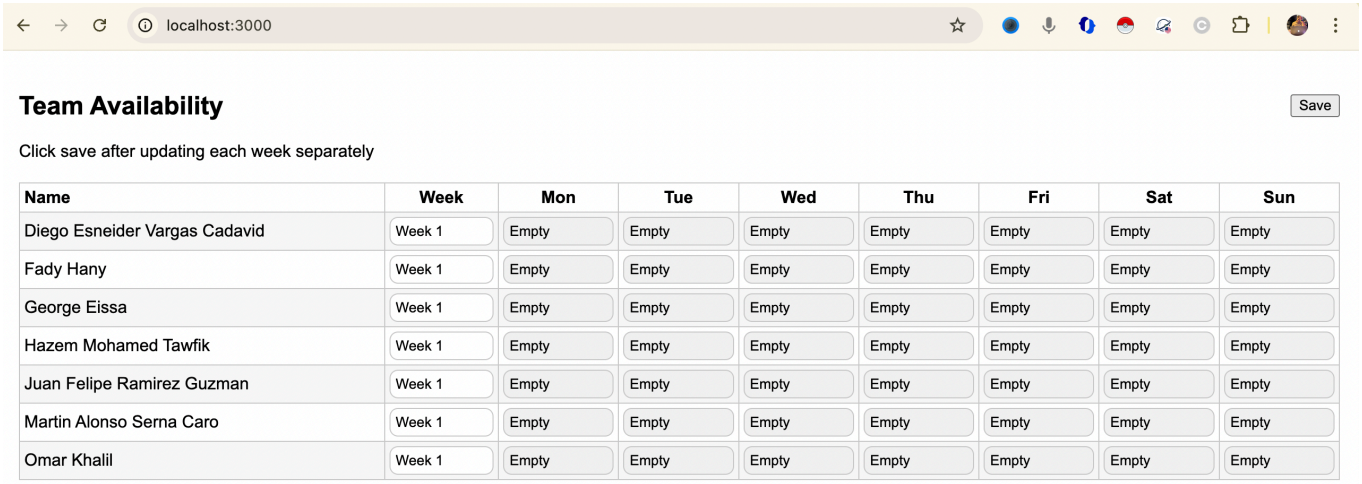
I removed the mistakenly created `init` folder and retried the setup. After that, Postgres correctly read the `init.sql` file, and the initialization worked as expected. ✅

⚡ **Lesson Learned:**

- When mounting SQL init scripts into `docker-entrypoint-initdb.d/`, always double-check that the path points to the **file** (e.g., `./db/init.sql:/docker-entrypoint-initdb.d/init.sql`) and not a directory.
- Any unexpected directory/file conflict will cause Postgres to fail at startup.

*And we are DONE !!!





Verifying Database Integration

After successfully setting up PostgreSQL and connecting the application, I wanted to **verify that the `history` table works as expected**.

Steps

1. Access the running database container:

```
docker exec -it teamavailtest-db-1 sh
```

2. Connect to the database using `psql`:

```
psql -U teamavail -d teamavail
```

You should see a prompt like:

```
teamavail=#
```

3. Run an SQL query to display the stored history in a readable format:

```
SELECT id, jsonb_pretty(payload) AS payload, created_at FROM history;
```

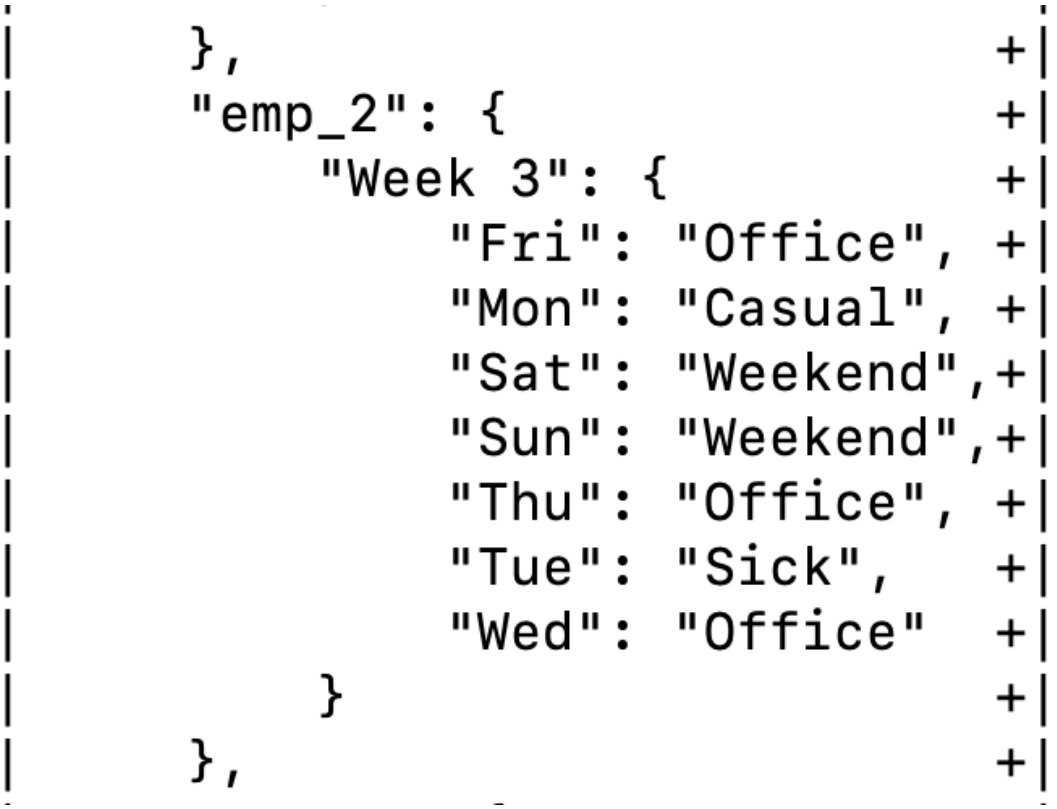
✅ The query returned the expected data, confirming that:

- The application successfully writes to PostgreSQL.
- The `history` table is correctly structured.
- Data can be retrieved and displayed in a formatted way.

Team Availability Save

Click save after updating each week separately

Name	Week	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Diego Esneider Vargas Cadavid	Week 2	Office	Remote	Empty	Empty	Empty	Empty	Empty
Fady Hany	Week 3	Casual	Sick	Office	Office	Office	Weekend	Weekend
George Eissa	Week 1	Empty	Empty	Empty	Empty	Empty	Empty	Empty
Hazem Mohamed Tawfik	Week 1	Empty	Empty	Empty	Empty	Empty	Empty	Empty
Juan Felipe Ramirez Guzman	Week 1	Empty	Empty	Empty	Empty	Empty	Empty	Empty
Martin Alonso Serna Caro	Week 1	Empty	Empty	Empty	Empty	Empty	Empty	Empty
Omar Khalil	Week 1	Office	Office	Office	Office	Annual	Weekend	Weekend



Moving from Bash Automation to Jenkins

Initially, I automated the pipeline using a **Bash script (`ci.sh`)**, which worked perfectly in a local setup.
However, while Bash scripts are useful for **quick automation**, tools like **Jenkins** are considered best practice because they provide:

- **Scalability**: Easily manage complex workflows across multiple environments.
- **Visibility**: Centralized dashboard for builds, logs, and results.
- **Extensibility**: Large ecosystem of plugins and integrations.
- **Collaboration**: Enables team-wide CI/CD processes instead of local-only scripts.

For these reasons, I decided to implement a **Jenkins Pipeline** that replicates the same automation as my local script.

Jenkins and Docker Permissions Issue

When running the pipeline inside the Jenkins Docker container, I faced a critical issue:

`permission denied while trying to connect to the Docker daemon socket`

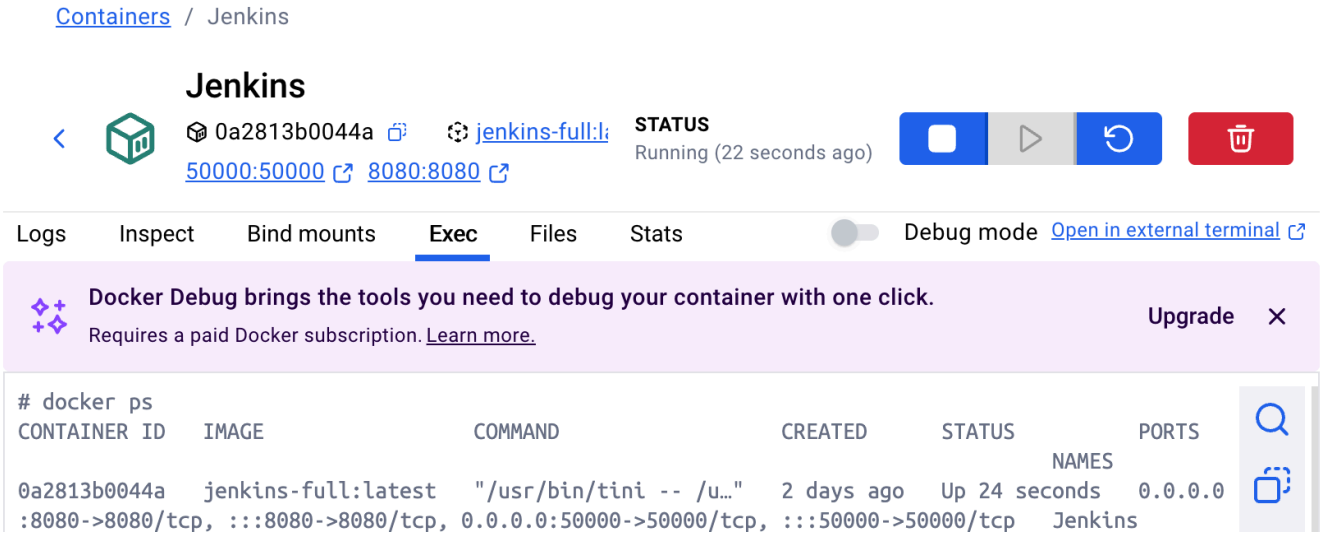
Although the Docker socket (`/var/run/docker.sock`) was mounted into the container, the default `jenkins` user inside the container did not have sufficient permissions to access the Docker daemon.

Solution

To solve this, I rebuilt the Jenkins container and explicitly ran it as **root** to ensure Docker commands could be executed inside:

```
docker run -d \ --name Jenkins \ -u root \ -p 8080:8080 -p 50000:50000 \ -v /Users/guest12345678/jenkins_home:/var/jenkins_home \ -v /var/run/docker.sock:/var/run/docker.sock \ jenkins-full:latest
```

After this change, running `docker ps` inside the Jenkins container worked as expected, and the pipeline was able to build and run Docker containers successfully. ✅



Lesson Learned

⚡ **Container permissions matter**: Without the right user privileges, even if volumes are mounted correctly, CI/CD workflows that depend on Docker will fail. Running Jenkins as `root` (or configuring proper group access) is essential for enabling Docker-in-Docker operations.

Jenkins Pipeline Execution

Before integrating with GitHub, I wanted to test the pipeline **locally**.

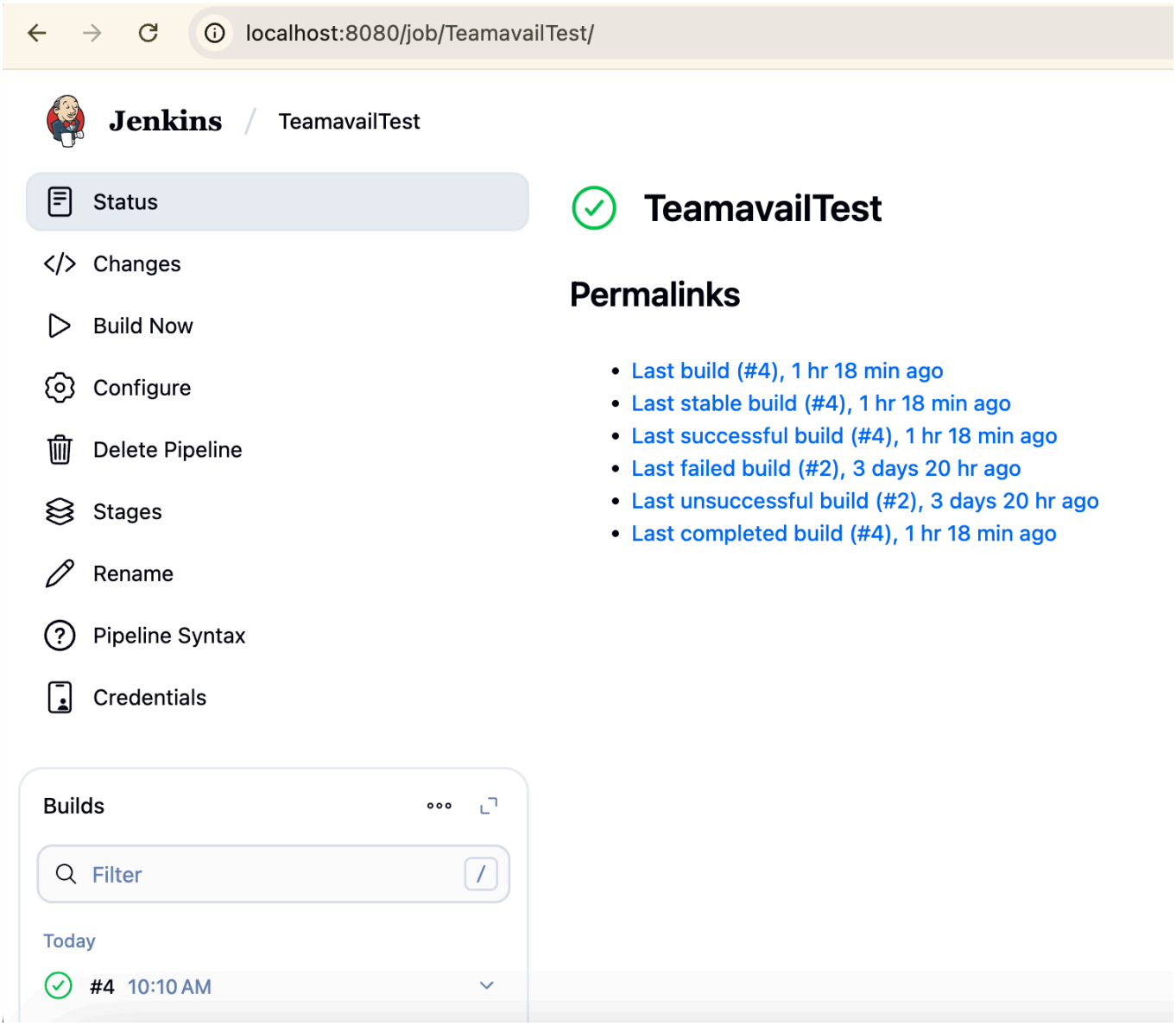
- I copied all project files directly into the **Jenkins workspace** under `/var/jenkins_home/workspace/`.
- From there, I triggered a build to make sure everything was working as expected.

The pipeline executed successfully:

- A **Docker image** for the application was built.
- A **container** was launched from that image.
- I verified the result on Docker Desktop, where both the image and container appeared.

After confirming the local run, I connected the pipeline to my GitHub repository to enable automated builds from version control.

📸 *Here's the screenshot of the successful local build result.*



GitHub Integration with Jenkins Pipeline

After verifying the pipeline locally, I created a new Jenkins pipeline and integrated it with **GitHub**:

- I had already configured my GitHub **credentials** inside Jenkins.
- Once the connection was established, I triggered a new pipeline run.

Issue Encountered

The first attempt failed with an error indicating that the port was already in use.

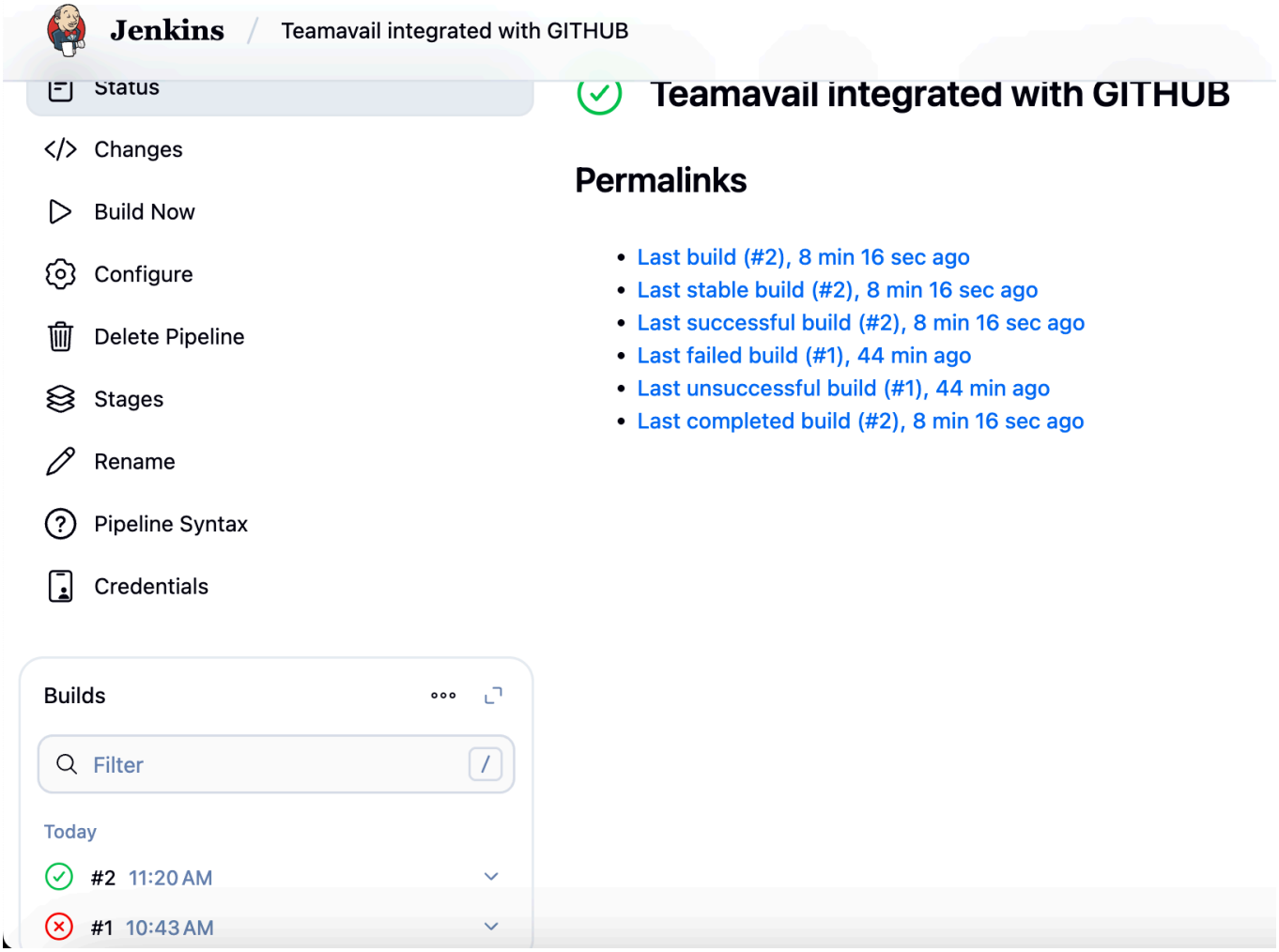
- This happened because a previous **Postgres process** was still running in the background, occupying port `5432`.
- I confirmed this by running: `pgrep postgres`
- After killing the leftover processes, I re-ran the pipeline.

Successful Execution

On the second attempt, Jenkins was able to:

- Pull the **Jenkinsfile** directly from GitHub.
- Build the Docker image.
- Launch the container successfully.

📸 *The screenshot clearly shows both runs: the first failed (bottom left) due to the port conflict, while the second run succeeded.*


Jenkins / Teamavail integrated with GITHUB
Status
Changes
Build Now
Configure
Delete Pipeline
Stages
Rename
Pipeline Syntax
Credentials
Builds
Filter
/
Today
#2 11:20 AM
#1 10:43 AM
Teamavail integrated with GITHUB
Permalinks
Last build (#2), 8 min 16 sec ago
Last stable build (#2), 8 min 16 sec ago
Last successful build (#2), 8 min 16 sec ago
Last failed build (#1), 44 min ago
Last unsuccessful build (#1), 44 min ago
Last completed build (#2), 8 min 16 sec ago

Containers

Give feedback

Container CPU usage
2.57% / 800% (8 CPUs available)

Container memory usage
1.03GB / 3.74GB

Show charts

Search

Only show running containers

	Name	Container ID	Image	Actions
☐	Jenkins	0a2813b0044a	jenkins	
☐	teamavailintegratedwithgithub	-	-	

teamavailintegratedwithgithub
/var/jenkins_home/workspace/Teamavail integrat...

View configurations

db
postgres:15-a
5432:5432

app
teamavailinte
3000:3000

```
2025-09-22 11:23:15.466 UTC [27] LOG:
checkpoint starting: end-of-recovery
immediate wait
2025-09-22 11:23:15.500 UTC [30] FATAL:  the
database system is not yet accepting
connections
2025-09-22 11:23:15.500 UTC [30] DETAIL:
Consistent recovery state has not been yet
reached.
app  DB not ready yet (attempt 1/30) — retrying
     in 1000ms
db   2025-09-22 11:23:15.508 UTC [27] LOG:
     checkpoint complete: wrote 921 buffers
     (5.6%); 0 WAL file(s) added, 0 removed, 0
     recycled; write=0.015 s, sync=0.023 s,
     total=0.044 s; sync files=301, longest=0.005
     s, average=0.001 s; distance=4238 kB,
     estimate=4238 kB
     2025-09-22 11:23:15.510 UTC [1] LOG:
     database system is ready to accept
     connections
app  ✅ Connected to Postgres
     Server listening on port 3000
```

```
➜  TeamavailTest git:(main) docker compose down -v

[+] Running 2/2
 ✔ Container teamavailtest_app_1  Removed
 ✔ Network teamavailtest_default  Removed
➜  TeamavailTest git:(main) lsof -i :5432

➜  TeamavailTest git:(main) pgrep postgres
334
357
358
359
361
362
363
➜  TeamavailTest git:(main) kill -9 334
kill: kill 334 failed: operation not permitted
➜  TeamavailTest git:(main) sudo kill -9 334
```

Setting Up Webhooks for CI/CD

To follow **best practices** in CI/CD, I decided to configure a **GitHub webhook** so that:

- Any time a developer pushes changes to the repository,
- Jenkins will automatically trigger the pipeline without requiring manual intervention.

Challenge

Since my Jenkins server is running **locally**, GitHub cannot directly reach it. For webhooks to work, Jenkins needs a publicly accessible URL.

Solution: Ngrok

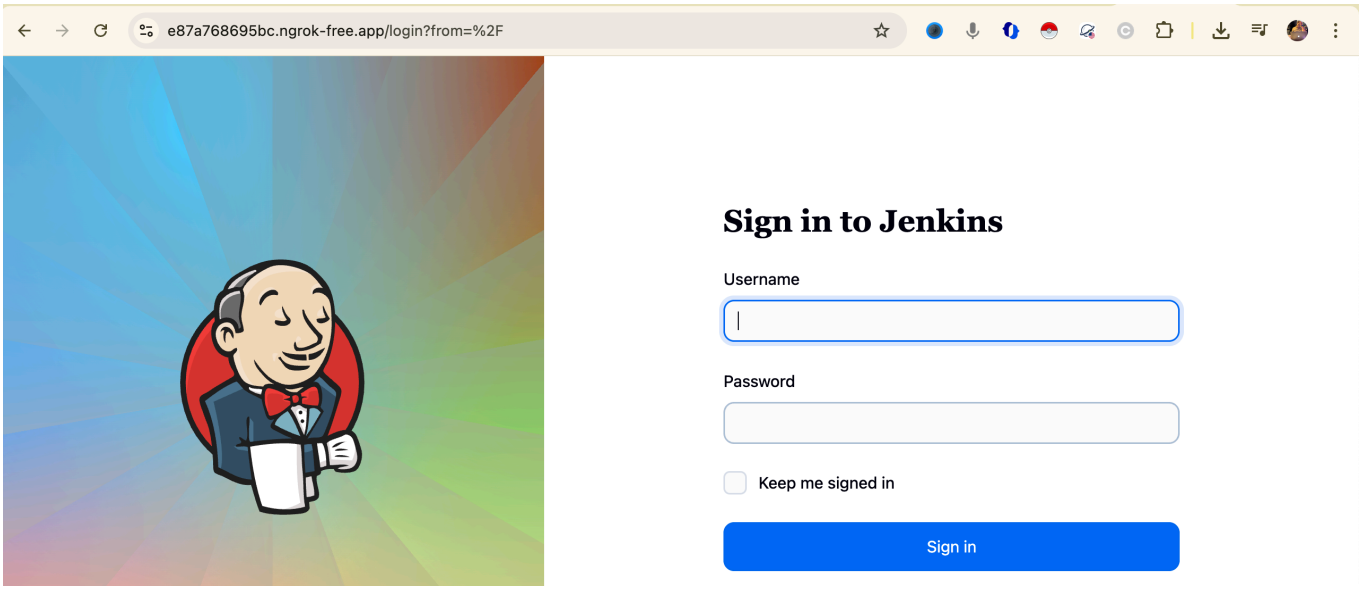
I used **ngrok**, a tunneling service that securely exposes local services to the internet.

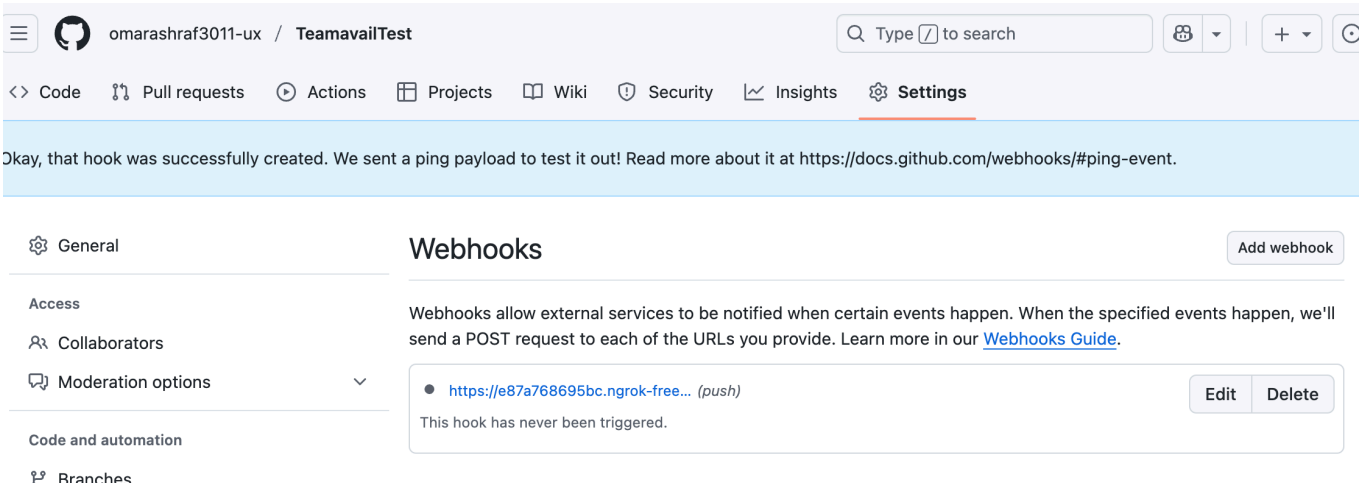
- It created a temporary public URL (e.g., `https://xyz123.ngrok.io`) that maps to my local Jenkins instance (`localhost:8080`).
- I configured this URL in the **GitHub webhook settings**.

With this setup, every **push event** to GitHub now automatically triggers the Jenkins pipeline, ensuring a smooth and automated CI/CD workflow.

ngrock — ./ngrok http 8080 — ./ngrok — ngrok http 8080 — 80×24

```
ngrok                                                          (Ctrl+C to quit)

Block threats before they reach your services with new WAF actions → https://

Session Status                online
Account                       omar.ashraf3011@gmail.com (Plan: Free)
Version                       3.29.0
Region                        Europe (eu)
Latency                       122ms
Web Interface                 http://127.0.0.1:4040
Forwarding                    https://e87a768695bc.ngrok-free.app -> http://loca

Connections                   ttl     opn     rt1     rt5     p50     p90
                              5       0       0.00    0.00    30.07   30.82

HTTP Requests
-------------

16:22:20.080 EEST GET /login
15:47:32.877 EEST GET /static/231b2735/favicon.ico
15:47:32.548 EEST GET /static/231b2735/jsbundles/simple-page.css
15:47:32.103 EEST GET /static/231b2735/scripts/redirect.js
15:47:32.549 EEST GET /adjuncts/231b2735/io/jenkins/plugins/thememanager/header/
15:47:32.634 EEST GET /adjuncts/231b2735/io/jenkins/plugins/thememanager/header/
```



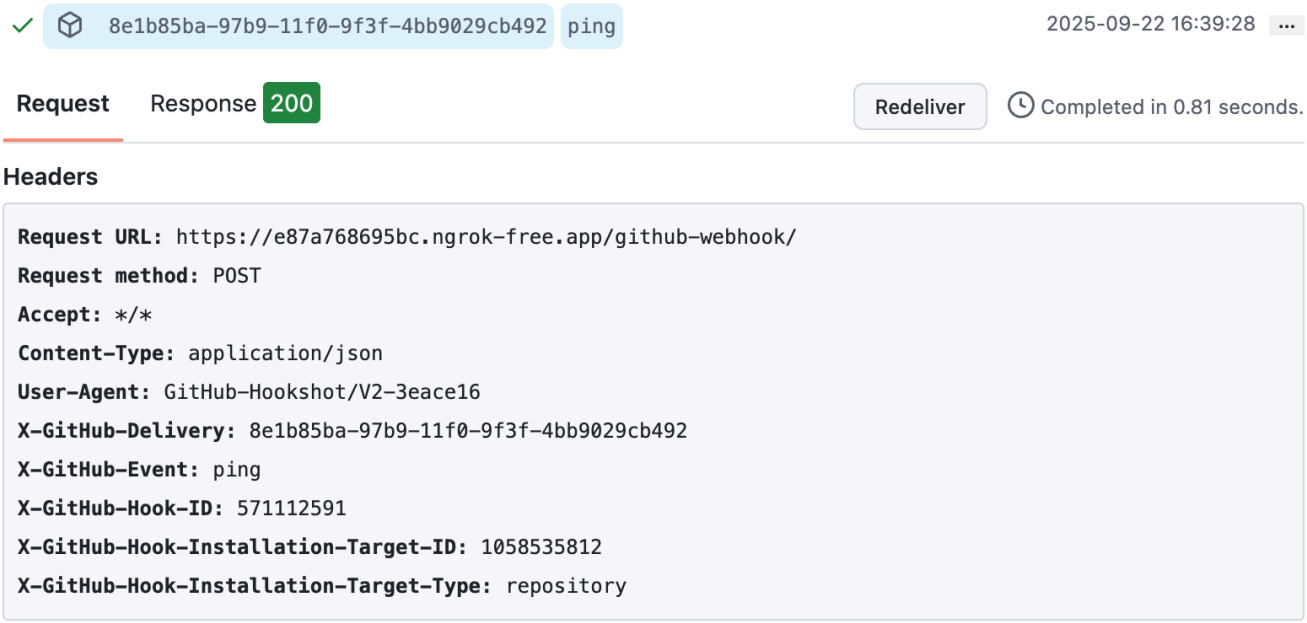


First Webhook Trigger

After configuring the webhook with **ngrok**, I pushed a commit to the GitHub repository.

- GitHub successfully sent the webhook event to Jenkins.
- Jenkins received the payload and automatically triggered the pipeline.

📸 *The screenshot below shows the webhook delivery marked as **successful** after the very first push.*



GitHub webhook troubleshooting — professional write-up

Below is a clear, professional, and detailed summary of what happened, why, and how you fixed it — ready to paste into your documentation.

---

1) What happened (short summary)

- I configured a GitHub webhook (via an ngrok public URL) to trigger the Jenkins pipeline on every `push`.
- GitHub showed the webhook delivery as **successful** on its side, but Jenkins did **not** start a build — there was no attempt recorded.
- I investigated the Jenkins job configuration and discovered the **"GitHub hook trigger for GITScm polling"** option was **not enabled**.
- After enabling that trigger, Jenkins received the webhook and tried to start a build. The first build attempt failed because a background PostgreSQL process was already using port `5432`. I killed the leftover Postgres processes and re-ran the build — the second run succeeded.

---

2) Step-by-step timeline, actions and results

A. Push → GitHub webhook delivery

- Action: I pushed a commit to the repository.
- GitHub: The webhook delivery was reported as **successful** (the webhook endpoint returned HTTP 200).
- Jenkins: No build was started — nothing appeared in the job history despite the webhook delivery.

Interpretation: GitHub successfully delivered the payload, but Jenkins was not configured to react to that payload.

B. Jenkins configuration fix

- Investigation: Opened the Jenkins job configuration → *Build Triggers* section.
- Problem: **"GitHub hook trigger for GITScm polling"** was **unchecked**. This is the checkbox that instructs Jenkins to start a build when GitHub posts a hook for the job.
- Action: Checked **"GitHub hook trigger for GITScm polling"** and saved the job.
- Result: Jenkins started reacting to incoming GitHub webhooks and attempted to run the pipeline.

How to enable (short reminder):

1. Jenkins job → *Configure*.
2. Scroll to *Build Triggers*.
3. Check **GitHub hook trigger for GITScm polling**.
4. Save.

C. First automatic build — failure due to Postgres port conflict

- Behavior: Jenkins attempted a build but failed during startup of services.
- Diagnosis: Console output showed an error indicating the database port was unavailable — a Postgres process on the host was already occupying `5432`. This is the same port the job's Docker Compose tried to bind.
- Commands I used to confirm and resolve: `# find postgres processes pgrep postgres # or list processes listening on port 5432 lsof -i :5432 # kill the leftover postgres processes (use responsibly) kill <pid1> <pid2> ...`
- Result: After killing the leftover Postgres processes, I triggered the Jenkins job again. The pipeline ran and completed successfully.

---

3) Why this happened (technical explanation)

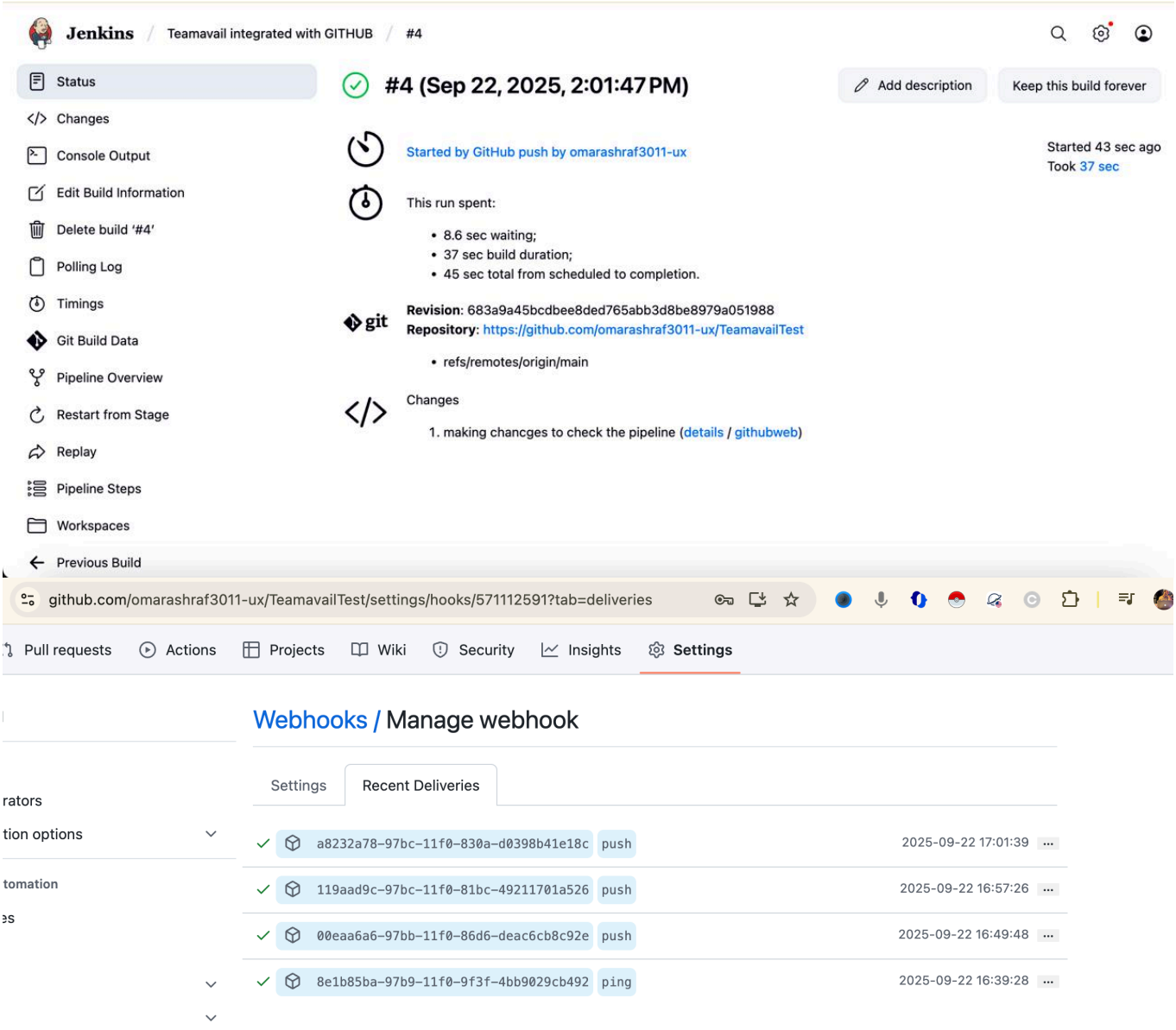
- GitHub → Jenkins webhook delivery succeeds does **not** guarantee Jenkins will start a build. Jenkins must be configured to *listen* and *react* to those webhook payloads — that is the job of the **GitHub hook trigger for GITScm polling** option. If it's disabled, Jenkins ignores the webhook even though GitHub reports a successful delivery.
- Even when Jenkins **does** try to run the pipeline, the job can fail during the environment setup phase if the CI job's Docker Compose tries to bind a host port that is already in use (e.g., Postgres listening on `5432`). The bind fails and the job aborts. This is common when local services or previously started containers/processes are still running.

---

4) Final result

- After enabling the GitHub hook trigger and removing the stray Postgres process that occupied port `5432`, Jenkins successfully received pushes from GitHub and ran the pipeline automatically.

- The second attempt completed the build (image built, container started).



Docker Hub Integration

After confirming that the CI/CD pipeline was working end-to-end and successfully triggered on every `git push`, the next logical step was to make the application image shareable with the rest of the team. For this, I decided to integrate **Docker Hub** into the pipeline.

Steps Completed

- Created a **Docker Hub repository** named `teamavail-app`.
- Generated a **Docker Hub Access Token** and securely stored it in Jenkins as credentials.
- Linked Jenkins with Docker Hub using these credentials.

Next Step: Pipeline Update

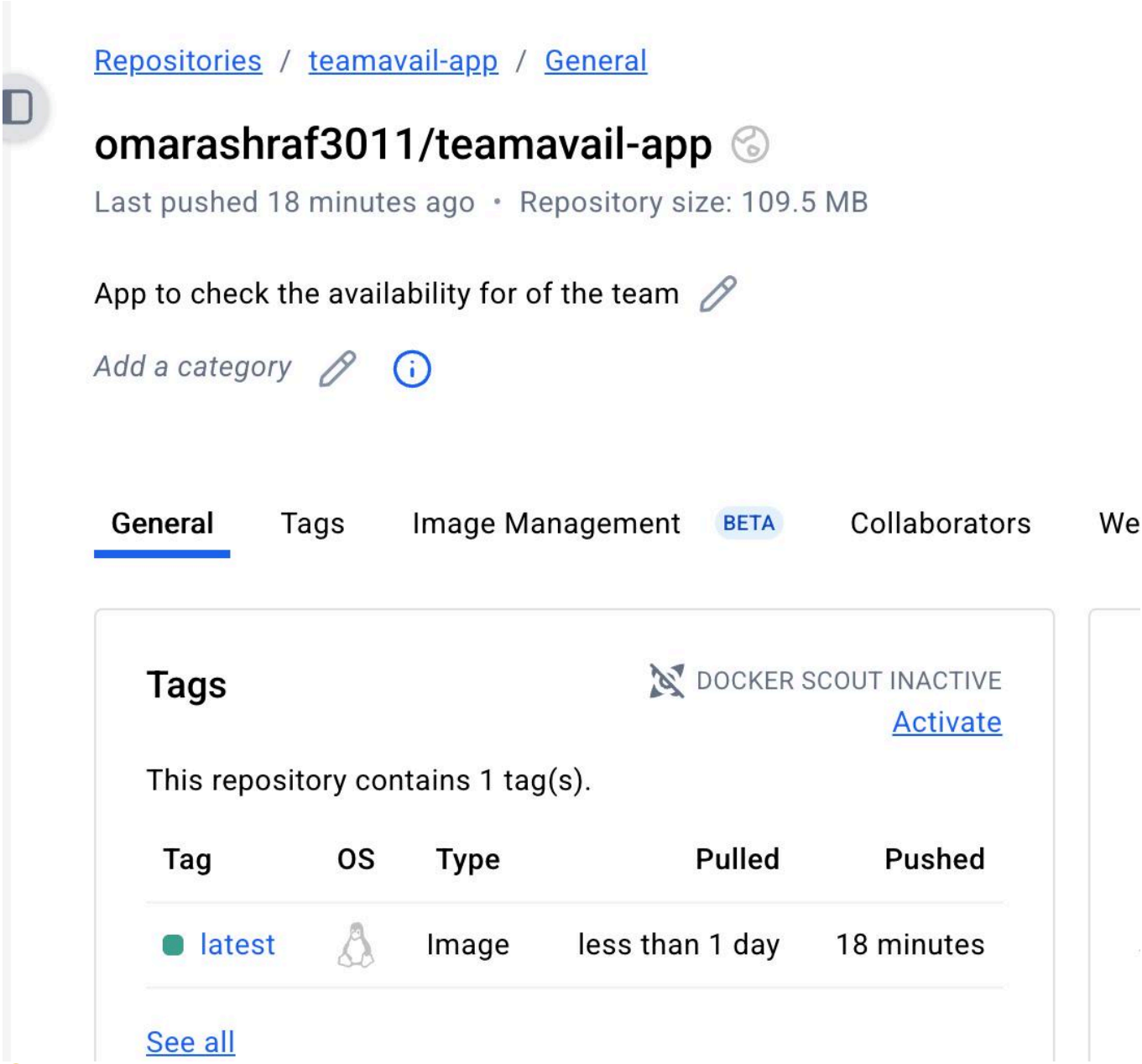
To extend the pipeline, I updated the `Jenkinsfile` so that after building and testing the application, Jenkins also:

1. Builds the Docker image for the application.
2. Authenticates with Docker Hub using the stored credentials.
3. Pushes the image to the `teamavail-app` repository on Docker Hub.

Source Code Reference

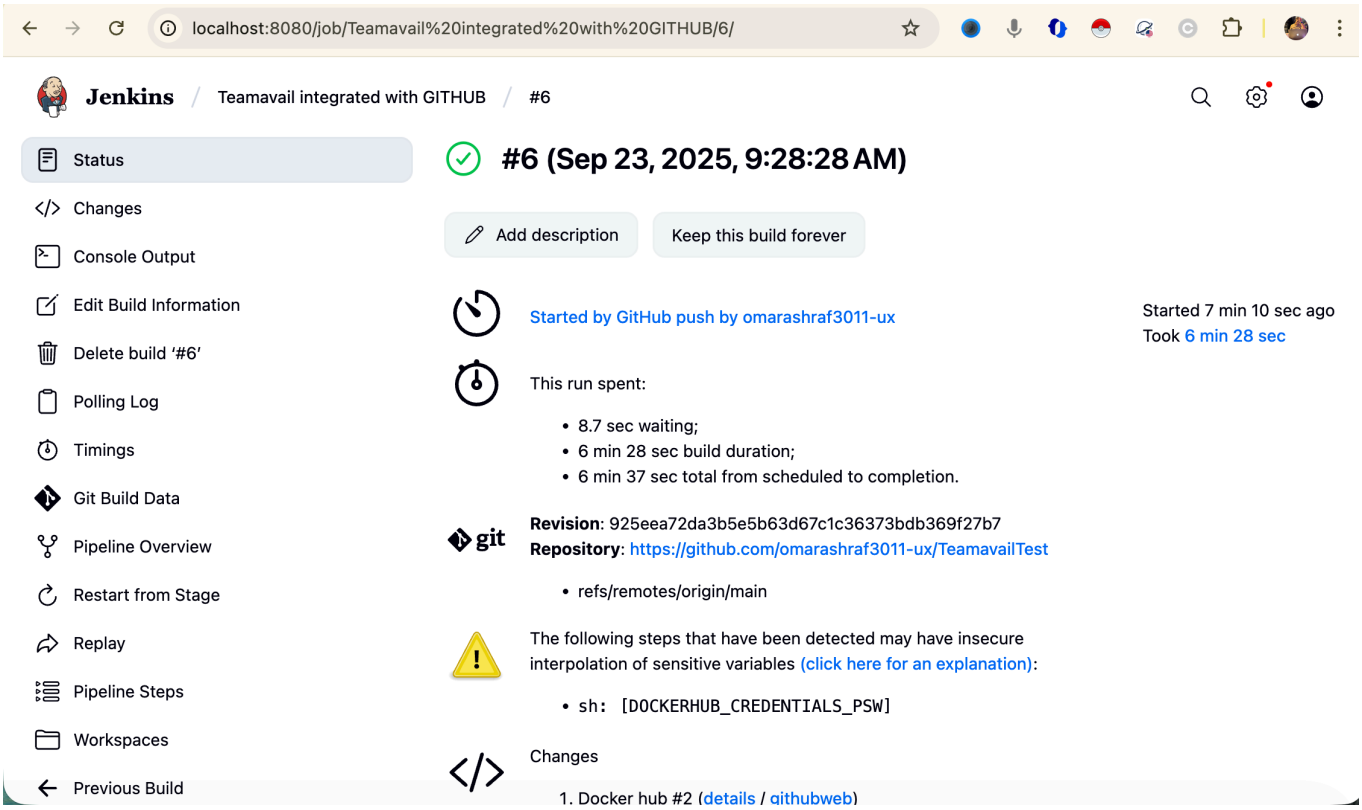
All pipeline scripts, the `Jenkinsfile`, and the full application source code are available in the project GitHub repository:
🔗 **https://github.com/omarashraf3011-ux/TeamavailTest

The repository contains the complete, up-to-date codebase (including CI configuration and Dockerfiles) and is updated regularly. Jenkins is configured to pull the `Jenkinsfile` directly from this repo, so the pipeline logic is versioned and maintained together with the application code.



💡 **Hint:** Initially, the GitHub webhook did not trigger the Jenkins pipeline because I forgot to start **ngrok** and update the webhook URL with the new public link. Once I started ngrok and updated the URL, the webhook worked immediately.

**And we are DONE !!



Deploying the Application on Google Cloud Run

After completing the local CI/CD pipeline and Docker image setup, the next step was to **deploy the application on Google Cloud Run**.

Why Deploy to Cloud Run?

- Running the app on **Cloud Run** allows it to be **accessible from anywhere**, not just locally.
- It provides **fully managed serverless hosting**, which means Google handles scaling, load balancing, and infrastructure management.
- This step is crucial for **sharing the application with other team members** and for demonstrating the end-to-end DevOps workflow in a real cloud environment.

APIs Enabled

Before deploying, I enabled the following Google Cloud APIs:

1. **Cloud Run API**
   - Required to deploy and manage serverless containerized applications.
   - Without it, Cloud Run cannot receive or run the Docker image.
2. **Cloud Build API**
   - Used to build Docker images directly in Google Cloud.
   - Ensures that the pipeline can push and build images in a cloud-native environment.
3. **Artifact Registry API**

- Provides a secure place to **store Docker images**.
- Necessary so Cloud Run can pull the container image from a reliable, managed registry.

> Enabling these APIs ensures the pipeline can fully automate deployment: building the image, storing it in Artifact Registry, and running it on Cloud Run.

Setting Up the Database for Cloud Run

The next crucial step for deploying the application to **Cloud Run** was to ensure the database was fully configured, as the team needs a **publicly accessible endpoint** for the app to function correctly.

Why Cloud Run

- Cloud Run is cost-efficient because it only runs when there is traffic, so there are **no charges when the service is idle**.
- This makes it ideal for internal applications that don't require 24/7 uptime.

Database Setup

1. Created a **PostgreSQL instance** on Google Cloud SQL named `teamavail-db`.
2. Created the **database** inside the instance: `teamavaildb`.
3. Created a **user** named `teamavail` and ensured it had **sufficient privileges** to read, write, and manage the database.
4. Verified the user's permissions by accessing the SQL shell and performing test operations.

> With the database ready and the user correctly configured, the application is now fully prepared for deployment on Cloud Run.

```
teamavaildb=> \du
                                   List of roles
           Role name            |                         Attributes
--------------------------------+------------------------------------------------------------
 cloudsqladmin                  | Superuser, Create role, Create DB, Replication, Bypass RLS
 cloudsqlagent                  | Create role, Create DB
 cloudsqlconnpooladmin          |
 cloudsqliamgroup               | Cannot login
 cloudsqliamgroupserviceaccount | Cannot login
 cloudsqliamgroupuser           | Cannot login
 cloudsqliamserviceaccount      | Cannot login
 cloudsqliamuser                | Cannot login
 cloudsqliamworkforceidentity   | Cannot login
 cloudsqlimportexport           | Create role, Create DB
 cloudsqlinactiveuser           | Cannot login
 cloudsqllogical                | Cannot login, Replication
 cloudsqlobservability          |
 cloudsqlreplica                | Replication
 cloudsqlsuperuser              | Create role, Create DB
 postgres                       | Create role, Create DB
 teamavail                      | Create role, Create DB
```

Deploying the Application on Google Cloud Run with Cloud SQL

Deploying the application on **Google Cloud Run** was a critical part of this project, as it allows the application to be accessible externally while following best practices for security and scalability. Since Cloud Run does **not support running multiple containers per service**, connecting the application to the database required the use of **Cloud SQL**.

Cloud SQL Setup

1. **Database Creation:**
   - Created a PostgreSQL instance and a database named `teamavaildb`.
2. **User Creation:**
   - Added a user `teamavail` and ensured it had the necessary privileges to manage the database.

Secret Management

- To follow **best practices**, I stored the database username and password in **Google Secret Manager** instead of hardcoding them in the application.
- This ensures sensitive credentials are securely stored and accessed only when needed.

Service Account Permissions

- Cloud Run services use a **Compute Engine default service account** (`201067245208-compute@developer.gserviceaccount.com`) to interact with other Google Cloud resources.
- I granted this service account the following permissions:

1. **Cloud SQL Client** – to allow the application to connect to the Cloud SQL instance.
2. **Secret Manager Admin** – to read the database credentials securely.

Cloud Run Service Deployment

1. Created a new **Cloud Run service** in the **same region as the Cloud SQL instance**.
2. Enabled **public access** so team members could reach the application via the endpoint.
3. Added **environment variables and secrets** to pass configuration and credentials to the application securely.
4. Configured **Cloud SQL connections** by linking the service to the `teamavail-db` instance.
5. Finally, clicked **Deploy** to launch the service.

> With these steps, the application is now fully deployed on Cloud Run, securely connected to Cloud SQL, and accessible via a public endpoint while following best practices for secret management and permissions.

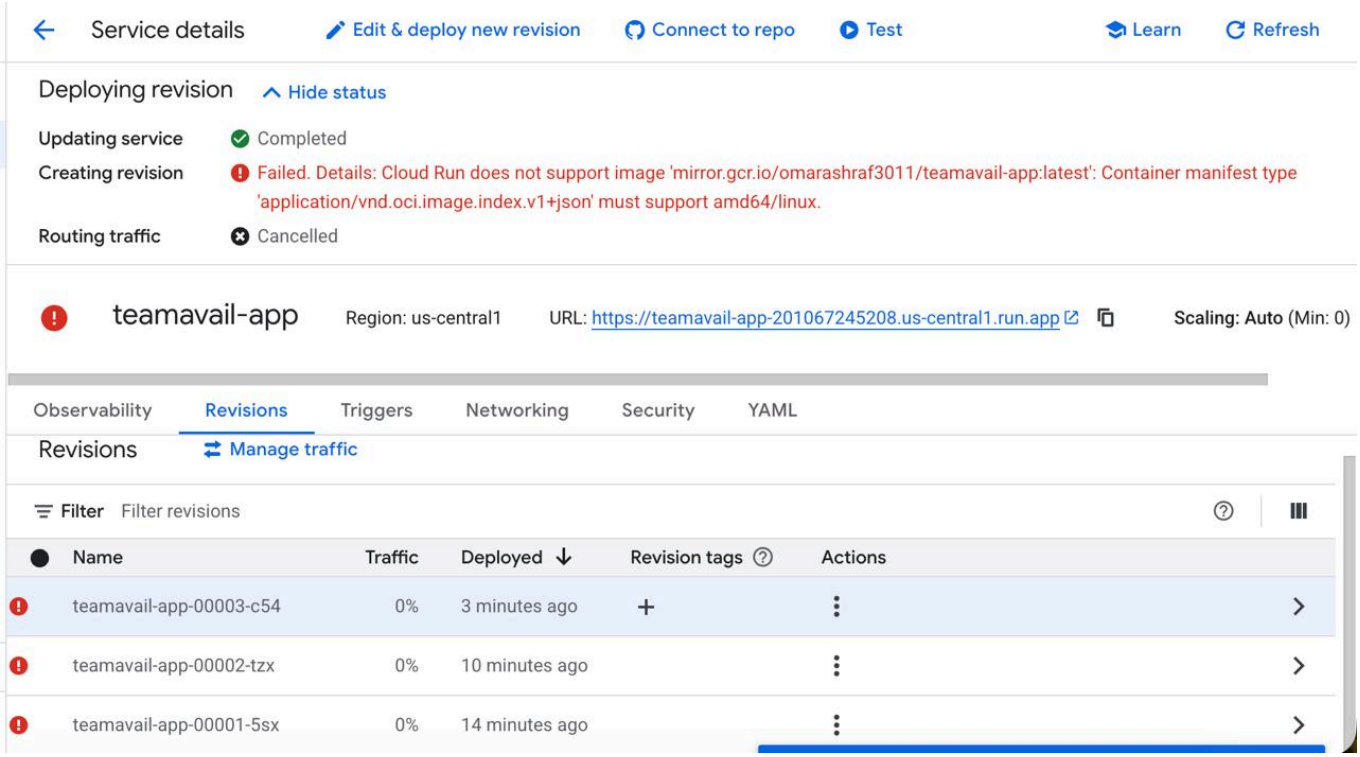
Architecture Compatibility Issue: ARM64 vs AMD64

During the Cloud Run deployment, I encountered an unusual but important issue related to **CPU architecture**.

The Problem

- I was building the Docker image on a **MacBook with M1 chip**, which uses **ARM64 architecture**.
- The resulting Docker image was therefore **ARM64-based**.
- Google Cloud Run, however, **only supports AMD64 (x86_64) architecture** for containers.

This caused repeated errors during deployment, such as:



`Container image architecture ARM64 is not supported on Cloud Run (requires AMD64)`

Technical Explanation

Architecture	Description	Compatibility
ARM64 (aarch64)	64-bit ARM processors, used in Apple M1/M2 chips and many mobile devices.	Efficient for local M1 builds but **not compatible with Cloud Run**, which uses x86_64 servers.
AMD64 (x86_64)	64-bit Intel/AMD processors, standard for most cloud servers.	Required by Google Cloud Run; ensures containers run reliably in the cloud.

- **ARM64** and **AMD64** are **binary-incompatible**, meaning an ARM64 image cannot run on an AMD64 machine without emulation.
- Cloud Run does **not provide ARM64 emulation**, so the image must be **built for AMD64** to deploy successfully.

Lesson Learned

- When developing on M1 Macs, always **use a multi-platform build** or **explicitly target AMD64**:

```
docker buildx build --platform linux/amd64 -t your-image:tag .
```

- This ensures the Docker image is compatible with Cloud Run's x86_64 environment and avoids architecture-related deployment failures.

> Understanding the difference between ARM64 and AMD64 and building for the correct architecture is crucial when deploying Mac-built Docker images to cloud platforms like Cloud Run.

Multi-Platform Docker Build Command

I used the following command to build and push the Docker image for Cloud Run:

```
docker buildx build --platform linux/amd64 -t omarashraf3011/teamavail-app:latest --push .
```

Explanation

1. **`docker buildx build`**
   - Advanced version of `docker build` that supports **multi-platform builds**, allowing images to run on different OS and CPU architectures (e.g., ARM64, AMD64).
2. **`--platform linux/amd64`**
   - Specifies the target platform for the image: Linux OS on AMD64 (Intel/AMD 64-bit) architecture.
   - Necessary because M1/M2 Macs are ARM64, but Cloud Run requires AMD64.
3. **`-t omarashraf3011/teamavail-app:latest`**
   - Tags the image with a name (`omarashraf3011/teamavail-app`) and version (`latest`) for pushing to Docker Hub.
4. **`--push`**
   - Automatically pushes the image to Docker Hub after building.
   - Without this, the image would remain only on the local machine.
5. **`.` (dot at the end)**
   - Defines the **build context**, i.e., the current folder containing the Dockerfile and application files.

> This command ensures the Docker image is built for the correct architecture and immediately uploaded to Docker Hub, ready for deployment on Cloud Run.

Cloud Run Deployment Issue: Port Mismatch

When I first deployed the container to **Cloud Run**, the service failed to start with the following error:

```
The user-provided container failed to start and listen on the port defined provided by the PORT=8080 environment variable
```

Cause

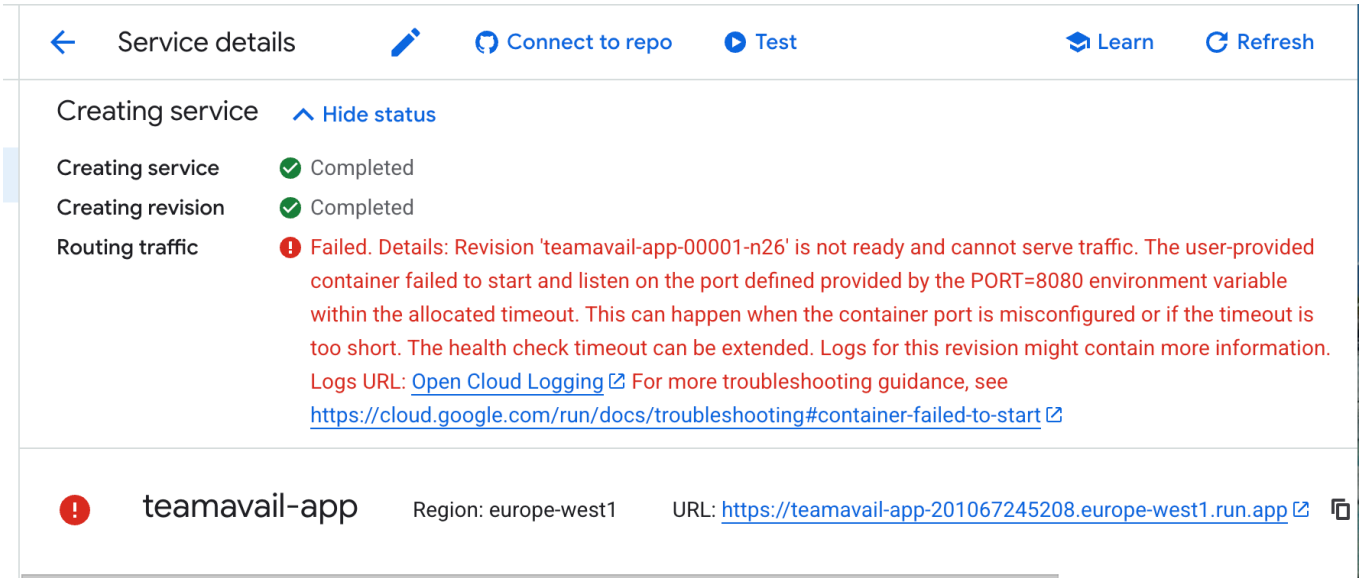
- **Cloud Run expects the container to listen on the port defined by the `PORT` environment variable**, which is automatically set (usually `PORT=8080`).
- My application, however, was **configured to run on port 3000** locally.
- As a result, when Cloud Run started the container, the app did not listen on the expected port. This caused the container to fail immediately.

Lesson Learned

- When deploying to Cloud Run (or any platform that injects a `PORT` environment variable), **the application must read the port from the environment variable**, not hardcoded values.
- In Node.js, this can be done using:

```
const PORT = process.env.PORT || 3000; app.listen(PORT, () =>
console.log(`Server running on port ${PORT}`));
```

This ensures the container works both **locally** (default `3000`) and on **Cloud Run** (`PORT=8080`).



Cloud Run Deployment Issue: Database Dependency

After fixing the architecture issue and rebuilding the Docker image, I encountered a **repeated deployment failure** on Cloud Run. Initially, it looked like a **port issue** (Cloud Run expects the container to listen on `PORT=8080`), but after checking the logs, the real problem became clear:

Logs Insight

```
Failed to start server (DB not ready): Error: Unable to connect to Postgres
```

- The application tried to connect to the **PostgreSQL database**.

- It failed multiple times (30 retries) and then **exited with `process.exit(1)`**.
- Because the server never started, **port 8080 was never opened**, causing Cloud Run to report a port-related error.

Root Cause

- The server was configured to **wait for the database (`waitForDb()`)** before starting the Express server.
- If the database is not ready, `waitForDb()` throws an error → the server **does not call `app.listen(PORT)`** → Cloud Run sees no open port → deployment fails.

Correct Approach

- **Do not block the server startup on database readiness.**
- Start the server immediately on `PORT=process.env.PORT`.
- Implement **retry logic or database connection checks internally**:
  - If the database is not ready, the endpoints return an **HTTP 503 Service Unavailable**.
  - This prevents the whole application from failing just because the database isn't ready.

Code Adjustment

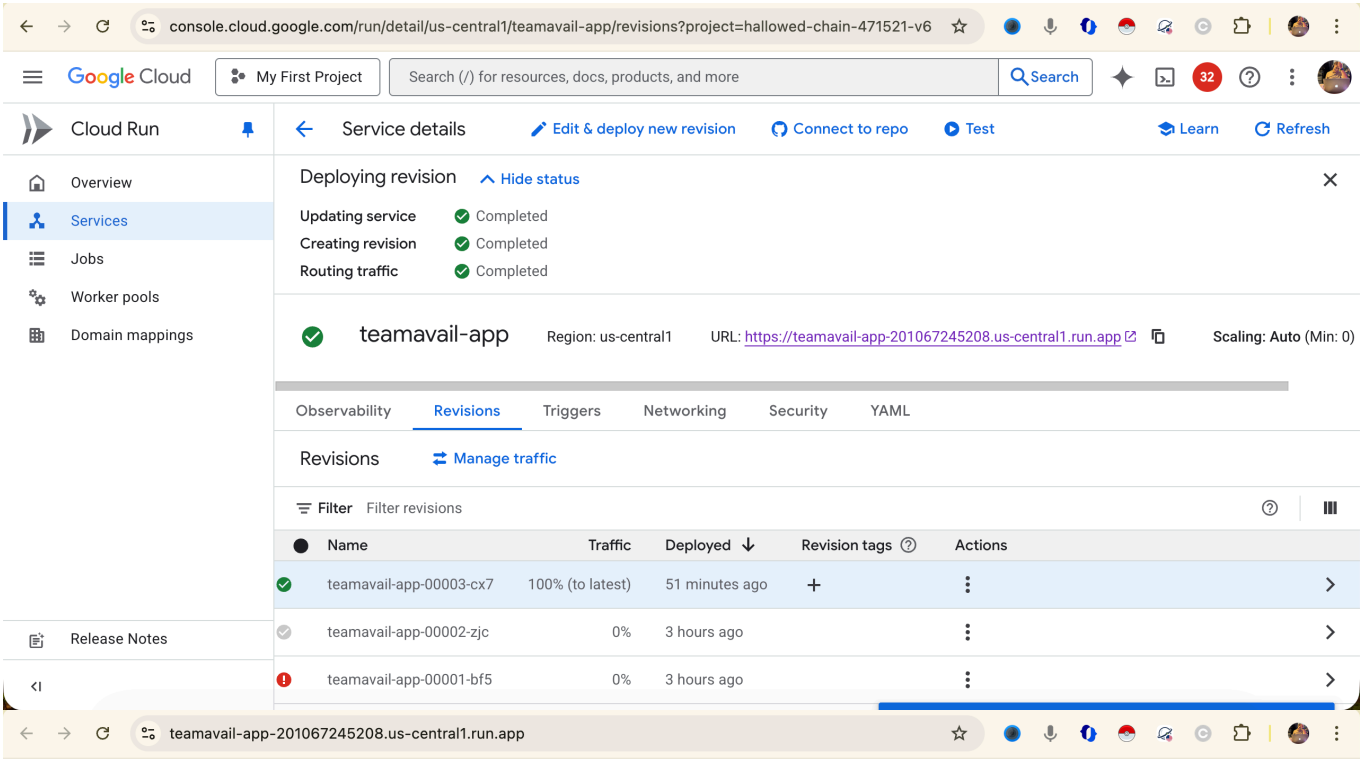
```
// server.js app.listen(PORT, () => { console.log(`Server running on port ${PORT}`); // Connect to DB asynchronously waitForDb().catch(err => { console.error('DB connection failed, but server is running', err); }); });
```

- This ensures the server **starts and listens on the required port** even if the database is temporarily unavailable.

Result

- After making these changes, I **rebuilt and pushed the Docker image**, then deployed to Cloud Run.
- Success was achieved **after 32 failed attempts** due to the previous DB dependency issue.

> **Key Lesson:** Cloud Run errors about ports are often **secondary symptoms**. The real issue may be internal application logic that prevents the server from opening the listening port. Always start the server first, handle DB connectivity asynchronously, and return proper error responses.



Team Availability Save

Click save after updating each week separately

Name	Week	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Diego Esneider Vargas Cadavid	Week 1	Office	Office	Office	Remote	Remote	Weekend	Weekend
Fady Hany	Week 1	Remote	Remote	Casual	Office	Office	Weekend	Weekend
George Eissa	Week 1	Empty	Empty	Empty	Empty	Empty	Empty	Empty
Hazem Mohamed Tawfik	Week 1	Empty	Empty	Empty	Empty	Empty	Empty	Empty
Juan Felipe Ramirez Guzman	Week 1	Empty	Empty	Empty	Empty	Empty	Empty	Empty
Martin Alonso Serna Caro	Week 1	Empty	Empty	Empty	Empty	Empty	Empty	Empty
Omar Khalil	Week 1	Empty	Empty	Empty	Empty	Empty	Empty	Empty

Conclusion

At the end of the journey, **Cloud Run successfully deployed the application and connected it to the Cloud SQL (Postgres) database**.

- A public endpoint was generated, allowing team members to access the service directly.
- In real-world scenarios, this endpoint would typically be mapped to a **custom domain** for production use.

With this setup:

- The application now checks **team availability** seamlessly.
- CI/CD best practices were applied through **GitHub, Jenkins, and Docker**.
- Finally, the service was deployed on a **serverless cloud environment (Google Cloud Run)**, ensuring scalability and cost efficiency.

✅ **Result:** Everything is working end-to-end, and the project achieved its original goal successfully.

Final Note

If you've made it this far, I want to sincerely thank you for taking the time to go through my project and the challenges I faced.

I know this documentation turned out a bit long, but I wanted to carefully **document every error, how I solved it, and what I learned from it**—because that's where the real growth happens.

I truly appreciate your time, and I hope to see you again in my **next project**! 🚀

— Omar Ashraf ✍️