

**Qwitter**

**Qwitter**

**DevOps Phase1**

**Prepared By**

**Fares Atef**

**Qwitter DevOps Engineer**

**1/11/2023**

Contents

[Introduction 2](#_Toc149707129)

[1) Simple Node.js App Deployment 3](#_Toc149707130)

[2) Simple React App Deployment 5](#_Toc149707131)

[3) To-Do list App Deployment with EC2 6](#_Toc149707132)

[4) To-Do list App Deployment with EC2 and RDS instances 8](#_Toc149707133)

# Introduction

In this phase, I have completed my Docker learning and have also obtained the one-year free tier of AWS. In this document, I will record all the work I have accomplished, including the projects I chose to deploy, the process of dockerizing them before deployment, and my interaction with various AWS services and Docker Hub.

# 1) Simple Node.js App Deployment

I've developed a straightforward Node.js application that I've containerized using Docker. This application comprises multiple services, including MongoDB, Mongo-Express, Redis, and Nginx, to handle various aspects of its functionality.

This app includes Dockerfile, docker-compose.yml, docker-compose.dev.yml, docker-compose.prod.yml and .dockerignore file.

|  |  |
| --- | --- |
| File | Usage |
| Dockerfile | Used to define a Docker image. It contains a series of instructions that specify how to create the image, what base image to use, what files to copy into the image, which commands to run, and more. |
| docker-compose.yml | Used to define and run multi-container Docker applications. It allows you to define a group of services (containers) that work together.  It specifies the containers, their configurations, and the networks they should connect to. It's a convenient way to manage multi-container applications as a single unit. |
| docker-compose.dev.yml | These are extension files for docker-compose.yml used for different environments, typically development and production.  docker-compose.dev.yml might include settings that make development easier, such as enabling debugging or auto-reloading of application code.  docker-compose.prod.yml might include settings optimized for production, such as load balancing or scaling.  By using separate files for development and production, you can maintain different configurations and easily switch between environments. |
| docker-compose.prod.yml |

**Note:** For all upcoming apps, I will provide only a Dockerfile and a docker-compose.yml file. This is because I do not require separate development and production environments; I will use one environment for production, testing deployment, and AWS services.

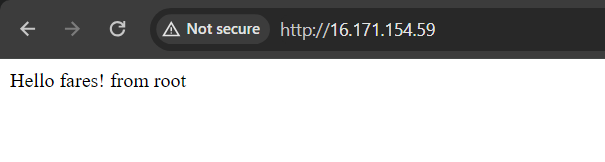
For this project, I deployed an EC2 instance from AWS with the following configuration options:

|  |  |
| --- | --- |
| Field | Option |
| Amazon Machine Image (AMI) | Ubuntu server |
| Instance Type | t3.micro |
| Network Settings | Allow SSH traffic from; so, I can access this server from my machine for example.  Allow SSH traffic from; so, anyone can access it from its IP.  Allow SSH traffic from; so, anyone can access it from its IP. |

I accessed the cloud server from my local machine using SSH and a Key Pair with the following command:

Once inside the server, I executed the following command:

The second command is used for building images and containers. After running it, the website can be accessed using the server's IP address.



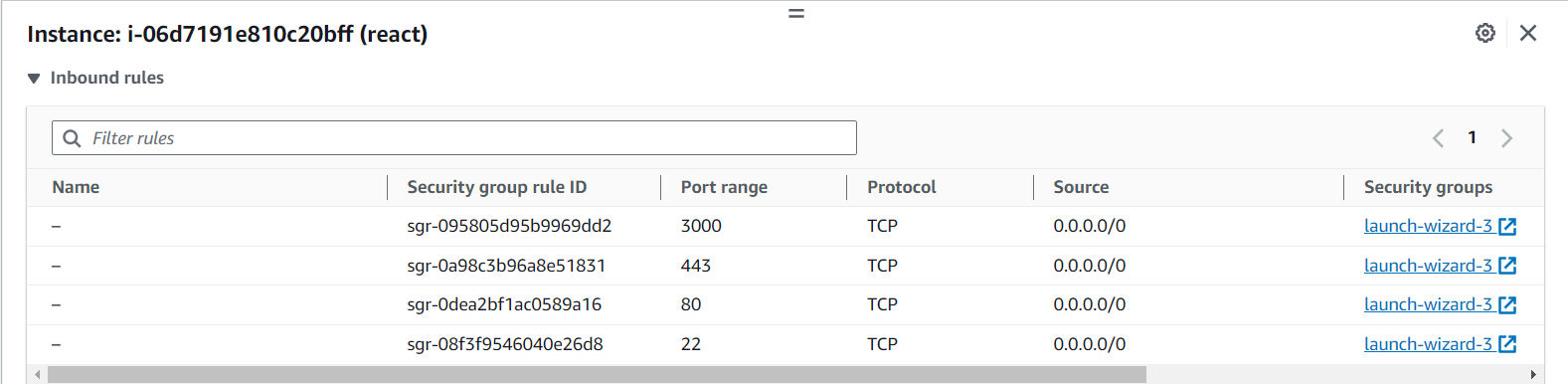
*Fig1: First app*

# 2) Simple React App Deployment

I've developed a straightforward React application that I've containerized using Docker.

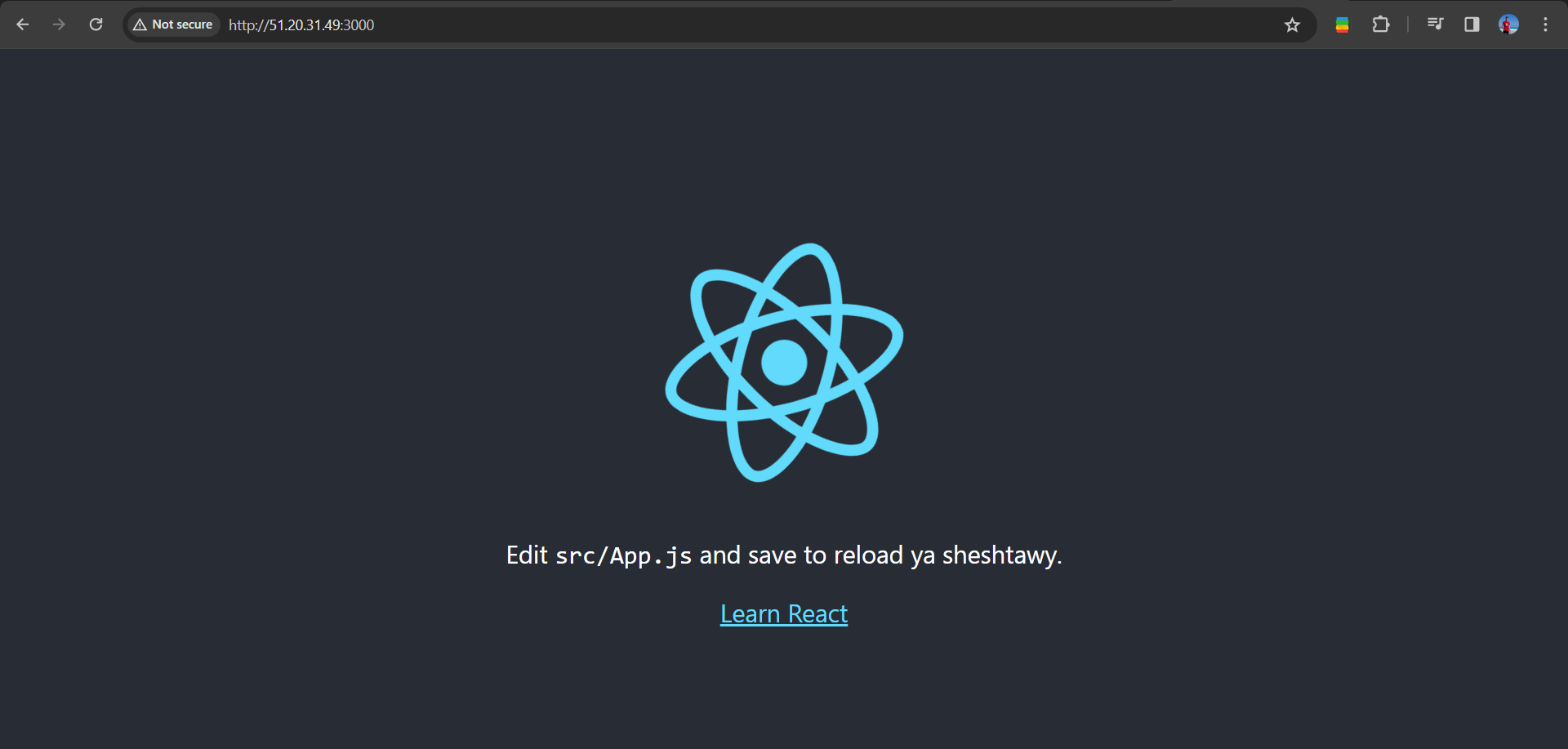
This app includes Dockerfile, docker-compose.yml and .dockerignore file.

For this project I used an EC2 instance from AWS with the same options of the first instance, but in EC2 instances, the firewall is enabled and all ports are closed, and since we access the react app through port 3000(default), I needed to make this port accessible from the outside world, so, I created an inbound rule in the firewall.



*Fig2: Inbound Rules*

Accessing the server followed a similar process to the first project. Once inside the server, I executed the following command:



*Fig3: Second app*

# 3) To-Do list App Deployment with EC2

It is a PERN (Postgres, Express, React, Node.js) app, in this app we have a To-Do list that we can insert, edit or delete from it.

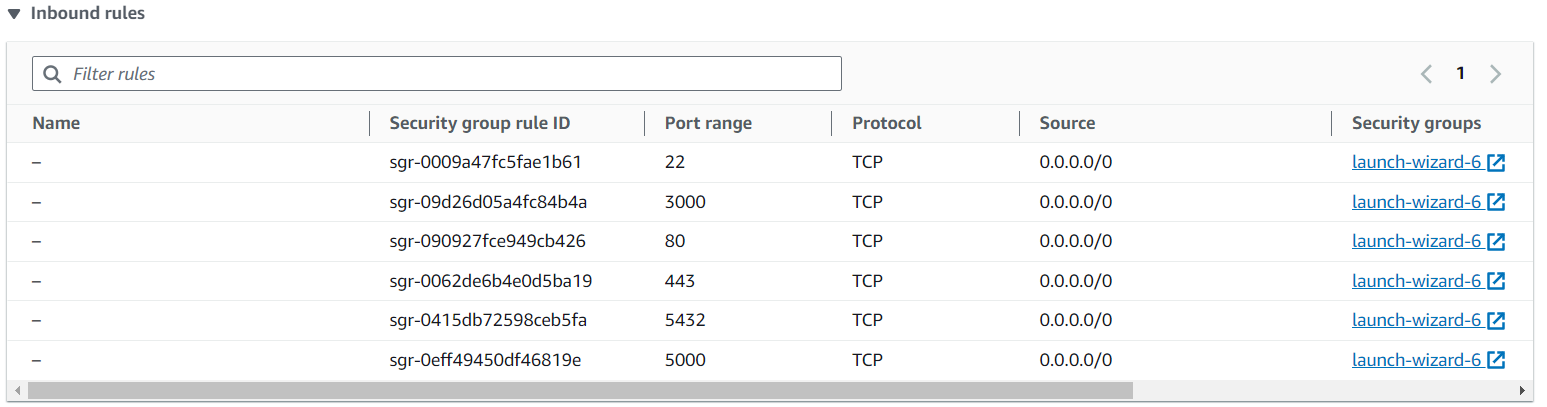
This application comprises multiple services, including Postgres, React and Node.js.

The app includes Dockerfile for the client side and another one for the server side, docker-compose.yml and .dockerignore file.



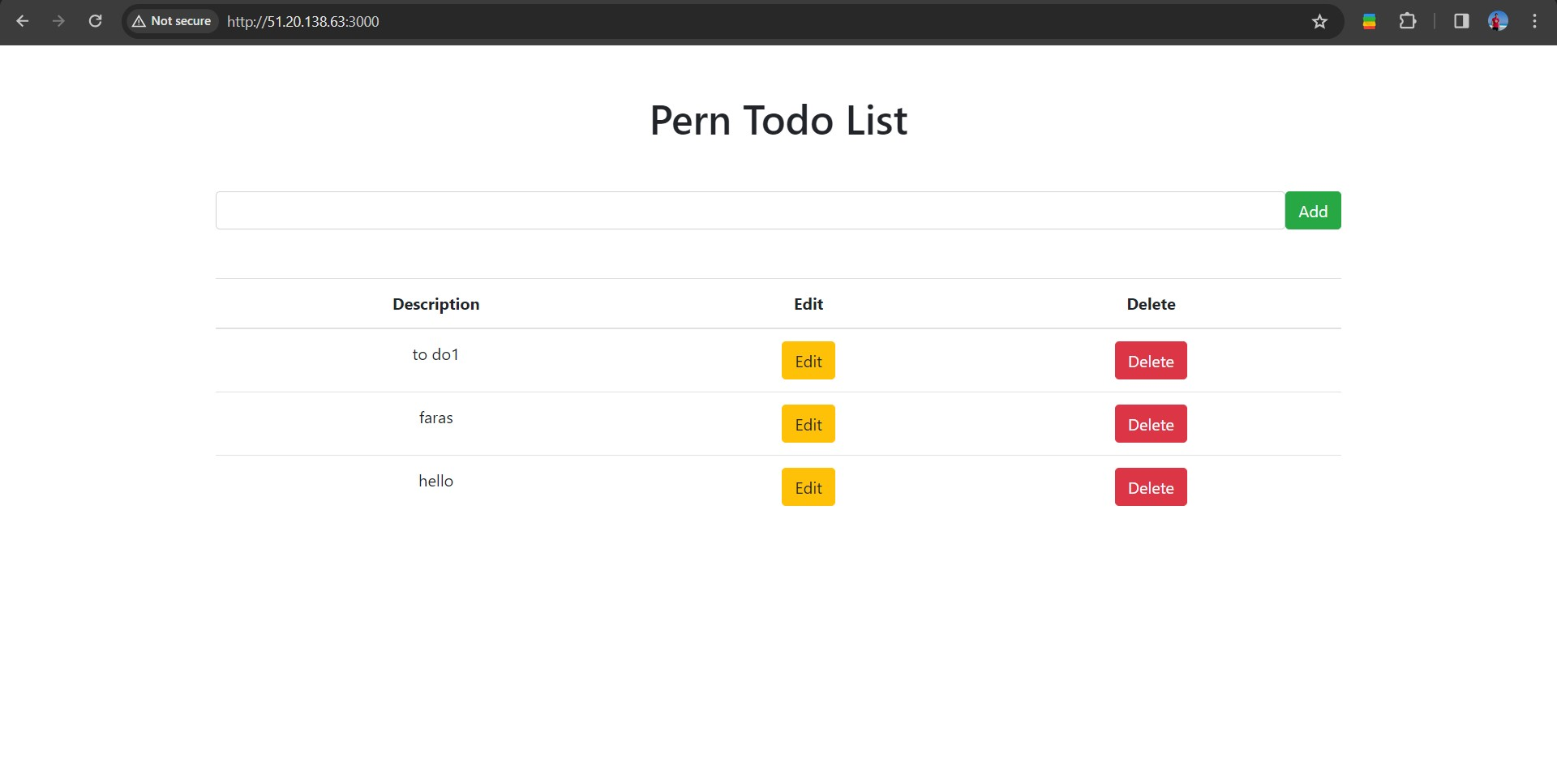
*Fig4: docker-compose.yml file to handle app services*

For this project I used an EC2 instance from AWS with the same options of the first instance, and for the ports problem mentioned in the second project, here we need to enable 3 ports, port 3000 for React, port 5000 for Node.js and port 5432 for Postgres.



*Fig5: Inbound Rules*

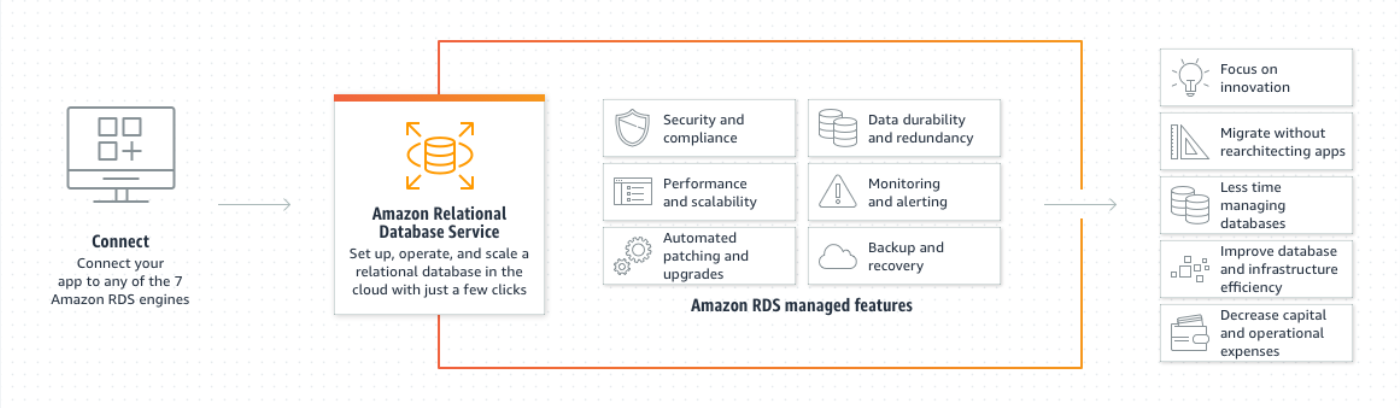
Accessing the server followed a similar process to the first project.



*Fig6: Third app*

# 4) To-Do list App Deployment with EC2 and RDS instances

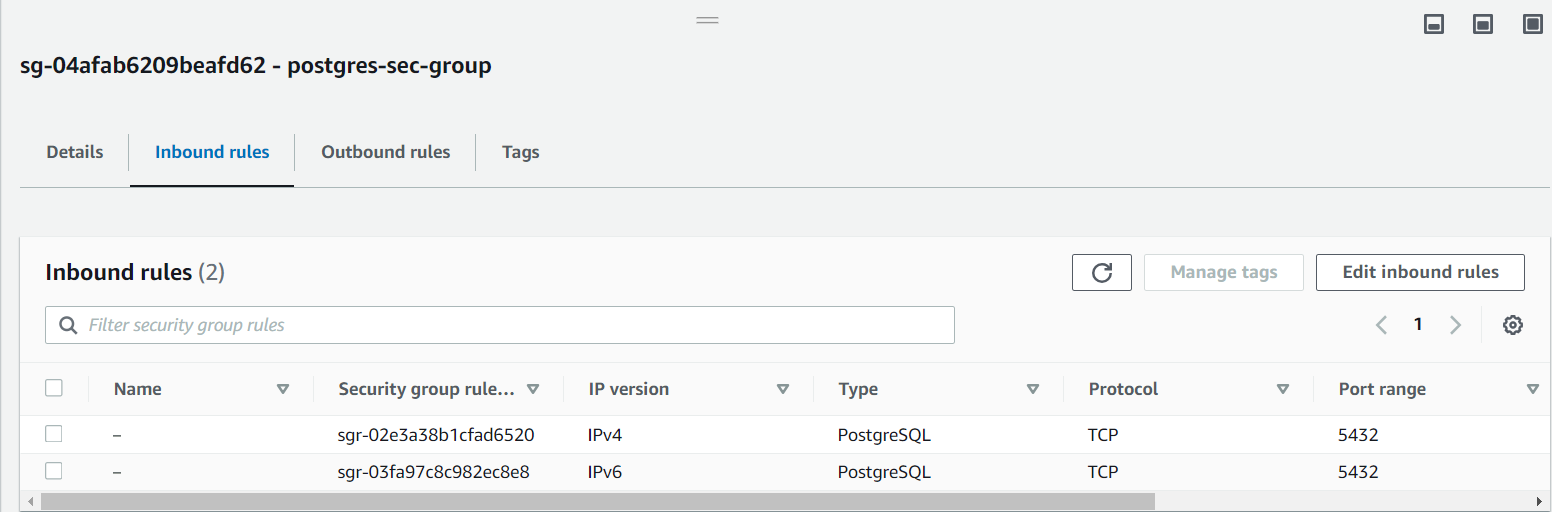
Since we will use relational database in our project, so I will use RDS instance to handle our database.



*Fig7: RDS*

It’s the same project in section3 but with RDS instance connected with the EC2 to handle the database.

I created a security group that will be used in the options for the RDS instance, this is for accessing it from port 5432.

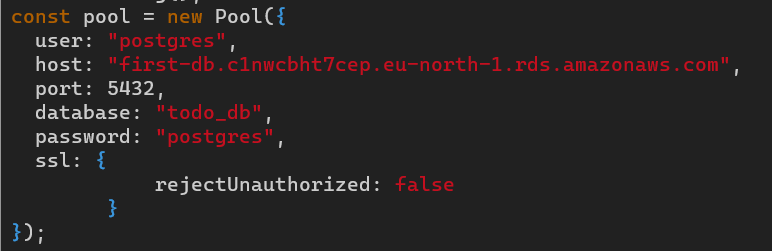


*Fig8: postgres-sec-group*

For the RDS instance from AWS, I used the following configuration options:

|  |  |
| --- | --- |
| Field | Option |
| Engine option | PostgreSQL |
| VPC security group (firewall) | postgres-sec-group |
| Database port | 5432 |

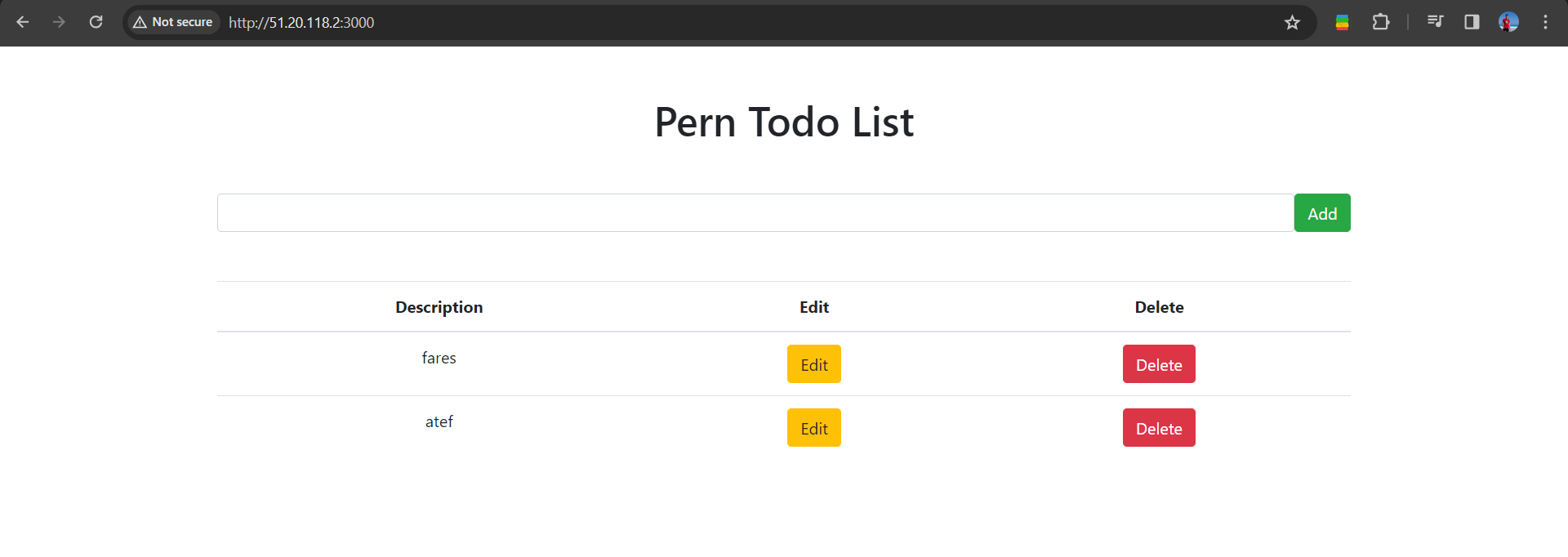
Now inside the EC2 instance I modified the pool:



*Fig9: New pool*

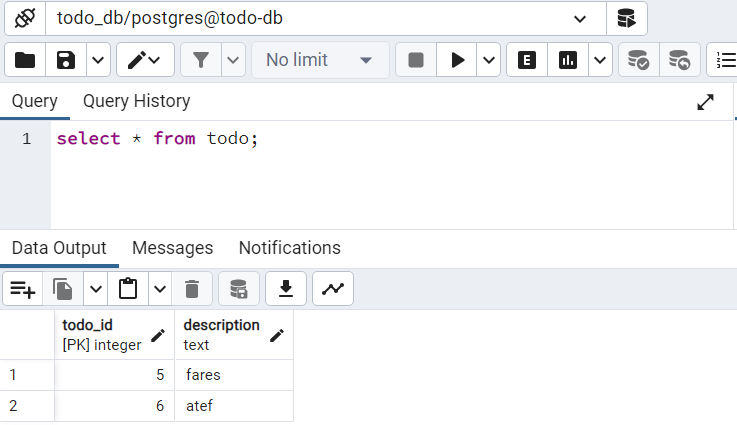
**NOTE:** host field is the Endpoint of the RDS instance.

Finally, everything is ready:



*Fig10: Fourth app*

And we will see the same data from the pgAdmin:



*Fig11: todo relation*