Containerized PostgreSQL Database

Using Docker

Stamatis Sideris
DATA MANAGEMENT & ANALYTICS CONSULTANT

Contents

Introduction

The following Report aims at clarifying the use of PostgreSQL DB to create Local Relational Databases for businesses with large volume of data. The procedure is supported by Docker for containerized development of the database.

About PostgreSQL Database

PostgreSQL is an open-source, highly stable database system that provides support to different functions of SQL, like foreign keys, subqueries, triggers, and different user-defined types and functions. It further augments the SQL language proffering up several features that meticulously scale and reserve data workloads. It's primarily used to store data for many mobile, web, geospatial, and analytics applications.

Reliability and Standards Compliance

PostgreSQL offers true ACID semantics for transactions and has full support for foreign keys, joins, views, triggers, and stored procedures, in many different languages. It includes most data types of SQL like that of INTEGER, VARCHAR, TIMESTAMP, and BOOLEAN. It also supports the storage of binary large objects, including pictures, videos, or sounds. It is reliable as it has a large built-in community support network. PostgreSQL is a fault-tolerant database thanks to its write-ahead logging.

Extensions

PostgreSQL boasts several robust feature sets including point-in-time recovery, Multi-Version Concurrency Control (MVCC), tablespaces, granular access controls, asynchronous replication, a refined query planner/optimizer, and write-ahead logging. Multi-Version Concurrency Control allows for concurrent reading and writing of tables, blocking for only concurrent updates of the same row. This way, clashes are avoided.

Scalability

PostgreSQL supports Unicode, international character sets, multi-byte character encodings, and it is locale-aware for sorting, case-sensitivity, and formatting. PostgreSQL is highly scalable — in the number of concurrent users, it can accommodate as well as the quantity of data it can manage. Furthermore, PostgreSQL is cross-platform and can run on many operating systems including Linux, Microsoft Windows, OS X, FreeBSD, and Solaris.

Dynamic Loading

The PostgreSQL server can also include user-written code into itself via dynamic loading. The user can specify an object code file; for example, a shared library that implements a new function or type and PostgreSQL will load it as required. The ability to modify its operation on the fly makes it uniquely suited for implementing new storage structures and applications rapidly.

About PgAdmin

PgAdmin is considered the most advanced open-source GUI tool designed for the most advanced relational database management tools. The value of this tool is that it provides a user-friendly data administration interface for you to handle SQL queries, maintenance, and other necessary processes without using command line prompts. Additionally, pgAdmin provides monitoring tools that let you see the status of operations at a glance, and it helps automate jobs with its scheduling agent. Overall, pgAdmin is a valuable addition to the workflows of most PostgreSQL users.

About Docker & Docker Compose

Docker is a set of platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers. The service has both free and premium tiers. The software that hosts the containers is called Docker Engine. It was first started in 2013 and is developed by Docker, Inc.

Docker is a tool that is used to automate the deployment of applications in lightweight containers so that applications can work efficiently in different environments. Docker can package an application and its dependencies in a virtual container that can run on any Linux, Windows, or macOS computer. This enables the application to run in a variety of locations, such as on-premises, in public (see decentralized computing, distributed computing, and cloud computing) or private cloud.[

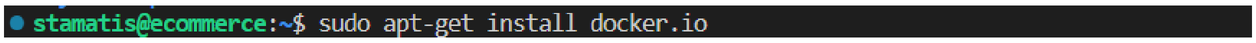
The Docker software as a service offering consists of three components:

- **Software**: The Docker daemon, called dockerd, is a persistent process that manages Docker containers and handles container objects. The daemon listens for requests sent via the Docker Engine API. The Docker client program, called docker, provides a command-line interface (CLI), that allows users to interact with Docker daemons.
- **Objects**: Docker objects are various entities used to assemble an application in Docker. The main classes of Docker objects are images, containers, and services. A Docker container is a standardized, encapsulated environment that runs applications. A container is managed using the Docker API or CLI. A Docker image is a read-only template used to build containers. Images are used to store and ship applications. A Docker service allows containers to be scaled across multiple Docker daemons. The result is known as a swarm, a set of cooperating daemons that communicate through the Docker API.
- **Registries**: A Docker registry is a repository for Docker images. Docker clients connect to registries to download ("pull") images for use or upload ("push") images that they have built. Registries can be public or private. The main public registry is Docker Hub. Docker Hub is the default registry where Docker looks for images. Docker registries also allow the creation of notifications based on events.

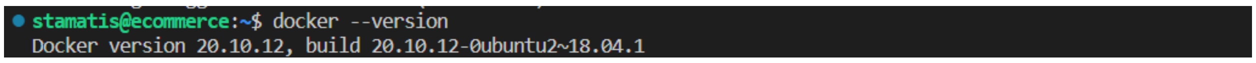
Docker Compose is a tool for defining and running multi-container Docker applications. It uses YAML files to configure the application's services and performs the creation and start-up process of all the containers with a single command. The docker-compose CLI utility allows users to run commands on multiple containers at once, for example, building images, scaling containers, running containers that were stopped, and more. Commands related to image manipulation, or user-interactive options, are not relevant in Docker Compose because they address one container. The docker-compose.yaml file is used to define an application's services and includes various configuration options. For example, the build option defines configuration options such as the Dockerfile path, the command option allows one to override default Docker commands, and more.

Docker & Docker Compose Installation

Firstly, we install the docker.io in our VM instance. To do so:

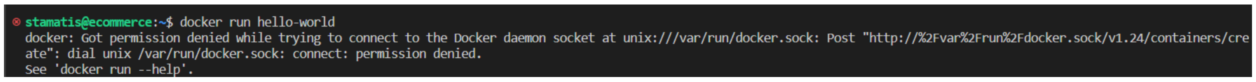


```
stamatis@ecommerce:~$ sudo apt-get install docker.io
```



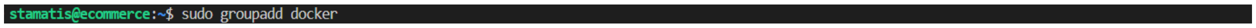
```
stamatis@ecommerce:~$ docker --version
Docker version 20.10.12, build 20.10.12-0ubuntu2~18.04.1
```

If we try and run docker an error occurs:

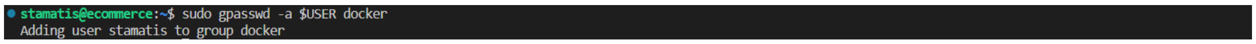


```
stamatis@ecommerce:~$ docker run hello-world
docker: Got permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Post "http://%2Fvar%2Frun%2Fdocker.sock/v1.24/containers/cre
ate": dial unix /var/run/docker.sock: connect: permission denied.
See 'docker run --help'.
```

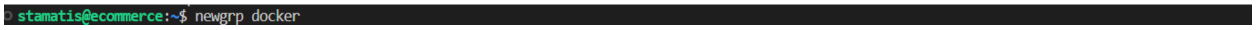
This happens because sudo rights are needed every time we run docker. In order to avoid running sudo each time, we add docker directory in sudo and then add our user in the directory. We activate the changes to groups with command newgrp and finally restart our instance and run docker again. Docker must be installed and ready for use.



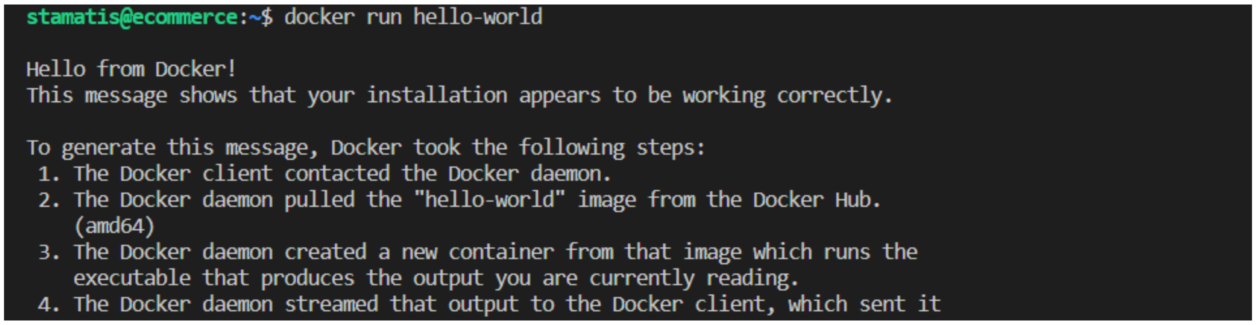
```
stamatis@ecommerce:~$ sudo groupadd docker
```



```
stamatis@ecommerce:~$ sudo gpasswd -a $USER docker
Adding user stamatis to group docker
```

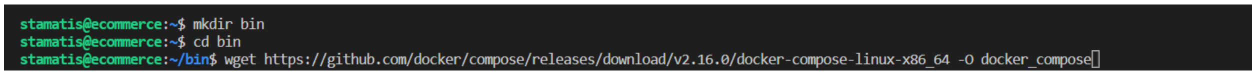


```
stamatis@ecommerce:~$ newgrp docker
```



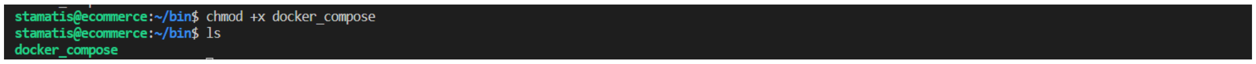
```
stamatis@ecommerce:~$ docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
```

Moreover, we download and install docker-compose that will help us easily manage and configure all the different containers we want to create. Firstly, create a bin directory to store your downloads. Then, visit the following link and choose the version of docker-compose you prefer for downloading. We download the docker-compose-linux-x86_64 version as our subsystem works in Ubuntu.



```
stamatis@ecommerce:~$ mkdir bin
stamatis@ecommerce:~$ cd bin
stamatis@ecommerce:~/bin$ wget https://github.com/docker/compose/releases/download/v2.16.0/docker-compose-linux-x86_64 -O docker_compose
```

You will observe that the system does not recognize docker_compose as an executable and so we use the following command to do so.

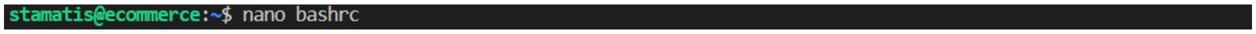


```
stamatis@ecommerce:~/bin$ chmod +x docker_compose
stamatis@ecommerce:~/bin$ ls
docker_compose
```

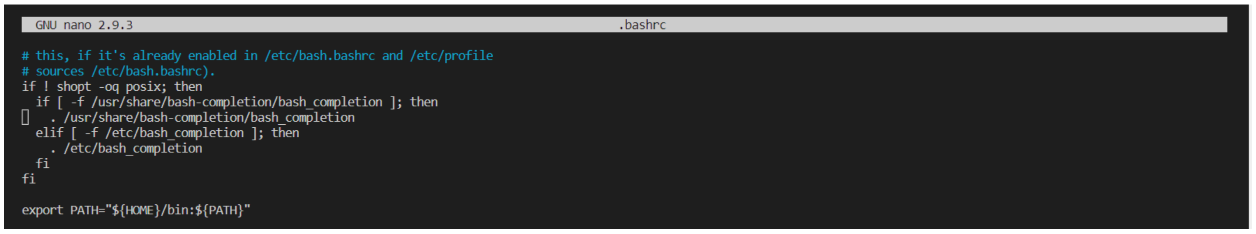
Finally, in order to make the executable visible from all directories, we visit the bashrc file using nano and we add the bin directory to the path using the following line of code:

export PATH=”${HOME}/bin:${PATH}”

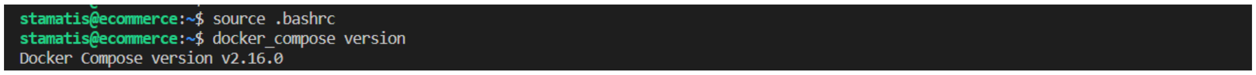
The bashrc file is the one storing the paths that are initialized when our instance starts. We write out the file and exit. Then, we use the source command to restart the file (we could also logout and login) and we are ready to use docker-compose.



```
stamatis@ecommerce:~$ nano bashrc
```



```
  GNU nano 2.9.3                                          .bashrc

# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
if ! shopt -oq posix; then
  if [ -f /usr/share/bash-completion/bash_completion ]; then
    . /usr/share/bash-completion/bash_completion
  elif [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
  fi
fi

export PATH="${HOME}/bin:${PATH}"
```

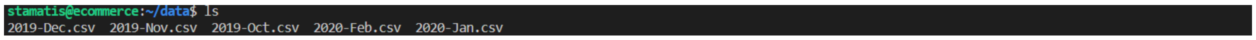


```
stamatis@ecommerce:~$ source .bashrc
stamatis@ecommerce:~$ docker_compose version
Docker Compose version v2.16.0
```

Deployment of Local PostgreSQL Database using Docker

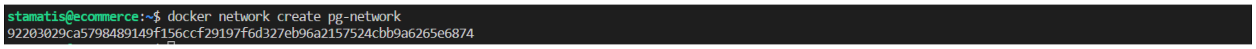
We create a local PostgreSQL database and manage it via pgAdmin. Moreover, we containerize the whole procedure using Docker in order to isolate it and have more freedom in experimenting with the pipeline deployment without affecting the base infrastructure.

We start by uploading our data to a data directory.



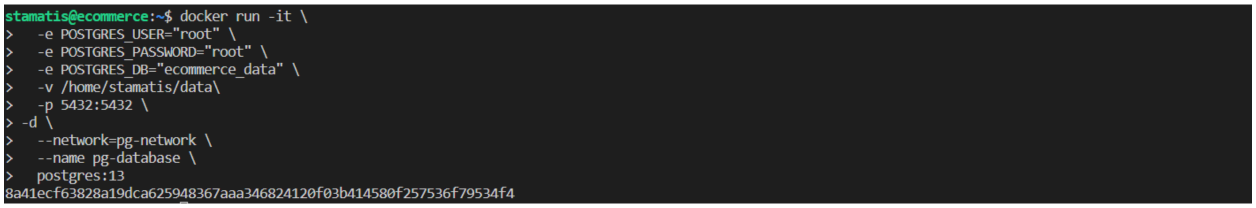
```
stamatis@ecommerce:~/data$ ls
2019-Dec.csv  2019-Nov.csv  2019-Oct.csv  2020-Feb.csv  2020-Jan.csv
```

Then, we create a Docker Network. The network will enable different containers in it to communicate with each other. After we run the following command, the name of our network is displayed.



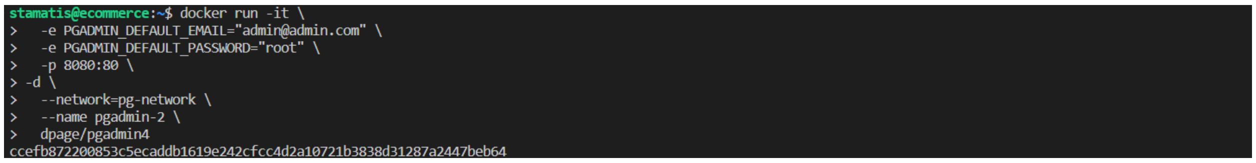
```
stamatis@ecommerce:~$ docker network create pg-network
92203029ca5798489149f156ccf29197f6d327eb96a2157524cbb9a6265e6874
```

To run PostgreSQL, we create a new container based on an PostgreSQL image from Docker Hub. We define the username, password and name of our database, the path to our data, we are mapping the port to which we want to have access locally, and we set the names of the network and the image. Variable -d runs the command in detach mode. In the end we can see the name of the container that was created.



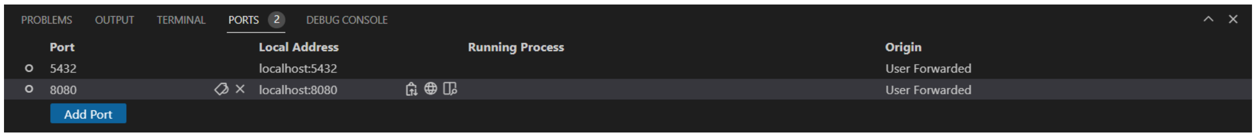
```
stamatis@ecommerce:~$ docker run -it \
>   -e POSTGRES_USER="root" \
>   -e POSTGRES_PASSWORD="root" \
>   -e POSTGRES_DB="ecommerce_data" \
>   -v /home/stamatis/data\
>   -p 5432:5432 \
> -d \
>   --network=pg-network \
>   --name pg-database \
>   postgres:13
8a41ecf63828a19dca625948367aaa346824120f03b414580f257536f79534f4
```

To run pgAdmin, we proceed by creating another container based on a pgAdmin image from Docker Hub. We set the username and password of our user, we map the ports and we set the network and name of the container. We run again in detach mode.



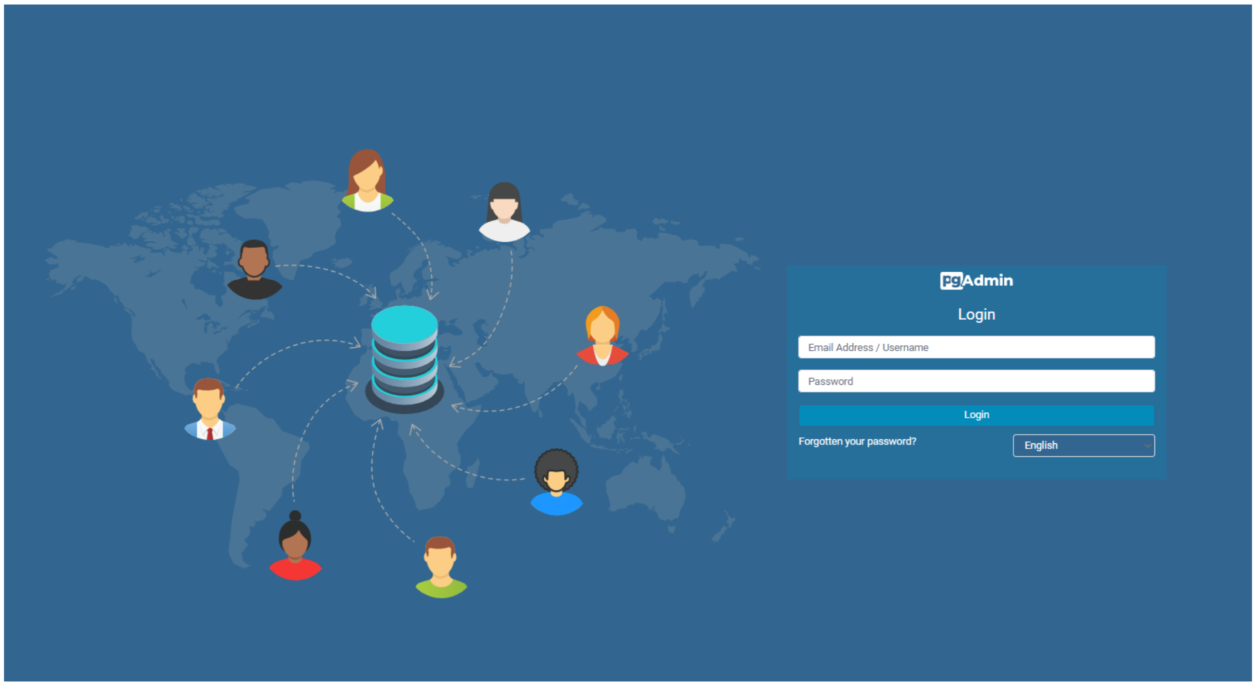
```
stamatis@ecommerce:~$ docker run -it \
>   -e PGADMIN_DEFAULT_EMAIL="admin@admin.com" \
>   -e PGADMIN_DEFAULT_PASSWORD="root" \
>   -p 8080:80 \
> -d \
>   --network=pg-network \
>   --name pgadmin-2 \
>   dpage/pgadmin4
ccefb872200853c5ecaddb1619e242cfcc4d2a10721b3838d31287a2447beb64
```

We finally forward a port to run our services locally. To do so, we choose the option PORTS in the terminal and we forward the ports 5432 and 8080 which we chose to map our services when configuring the containers.

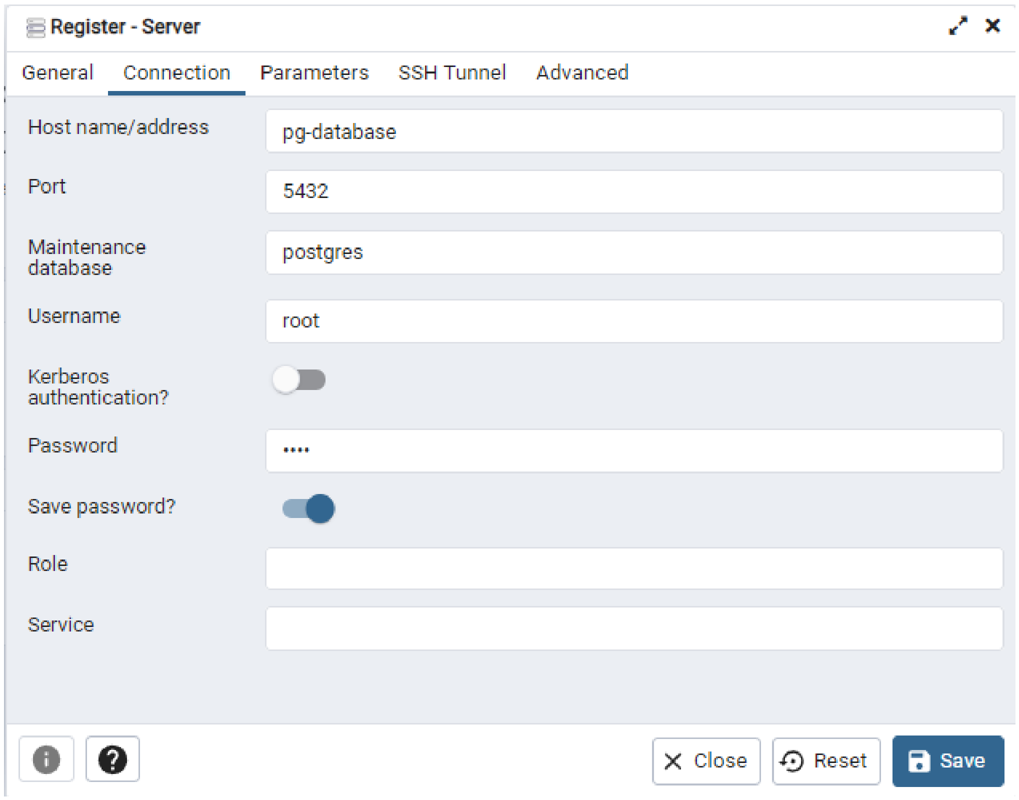


```
Port     Local Address     Running Process     Origin
5432     localhost:5432                        User Forwarded
8080     localhost:8080                        User Forwarded
```

We can then access pgAdmin from our browser by visiting the localhost:8080. We gain access by entering the username and password we set before.

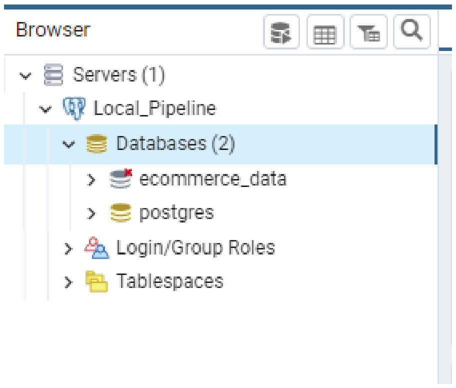


Afterwards, we choose to create a new server and we give it a name. I called it “Local_Pipeline”. Finally, we create a connection to the PostgreSQL server we created in the container by passing the credentials asked.



Host name/address	pg-database
Port	5432
Maintenance database	postgres
Username	root
Kerberos authentication?	off
Password	••••
Save password?	on
Role	
Service	

Our database is ready for use!



```
Servers (1)
  Local_Pipeline
    Databases (2)
      ecommerce_data
      postgres
    Login/Group Roles
    Tablespaces
```

References

https://kinsta.com/knowledgebase/what-is-postgresql/

https://www.adservio.fr/post/what-is-pgadmin

https://www.docker.com

https://docs.docker.com/compose/