Titanic Lost – A Docker Project

#### Pre-Amble

The purpose of this project is to bring together a number of web engineering products and then orchestrate them via Docker containers. The intent is to show how easily you can create an ecosystem from just a Git hub repo and some server resources.

Containers make it extremely easy to move entire systems to virtually anywhere, and expect the same operational results. For instance, I have developed all my code and deployed it to containers on my Mac, however I could just as easily clone down the repo on an Azure virtual machine (with Docker installed) and simply standup the project a with a few “docker up” commands.

#### The Project Ecosystem

Here are the major assemblies being driven by the Docker containers:

* Apache2 webserver with PHP7
* Postgres Database
* PGAdmin – a tool for working with Postgres databases
* PGWatcher – a tool for monitoring & alerting on Postgres databases

And a peek inside the wheelhouse, these are some of the technologies we will use to design, develop & deploy our full Titanic Lost web solution:

* Docker (of course!)
* Github – storing all our source code for backup & portability
* Liquibase – a tool for templating databases
* PHP – most of the working code is PHP7
* AnyCharts – these are web-plugins to create the charts
* JQWidgets – the JQXgrid pluging to hold the “Full List” web display
* EclipseIDE – Using Eclipse to writeup all the code and connect to Git

Charted Course, or how do we get there from here:

* Install Docker
* Initialize Git repo to safeguard the code and keep it portable/shareable
  + Note: the local repo will be used to “share” code with containers
* Create & stage the Liquibase file with database details
* Create & stage the HTML/PHP files for web front-end
* Pull Docker Images
* Create a named Docker Network resource
* Use Dockerfile to update & create custom images for the project
* Use Docker Compose to stand up the infrastructure
* Complete Database construction
* Use Docker run command to create PGWatch container
* Run through basic sanity checks

Alright, let’s get started, and mind the icebergs!

#### Docker Install

Just a quick note on this item. Docker install was super easy, I downloaded the software from the site (I am using a Mac) and just dragged the icon over to application folder. Windows users will undoubtably have a different experience, although I am sure, just as easy.

A picture containing sign, black, holding, player

Description automatically generated

A close up of a sign

Description automatically generated

#### Directory Setup & Git init

The directories are a critical part of making the whole system work. There are dependencies on various locations for everything from the Web calls to where the widget codes are.

**Directory Tree:**

$ tree -dL 3

TitanicLost/

|-- WebContent

| |-- META-INF

| |-- WEB-INF

| | `-- lib

| |-- css

| |-- fonts

| |-- jqwidgets

| | |-- globalization

| | `-- styles

| |-- js

| |-- scripts

| |-- src

| `-- styles

| |-- fonts

| `-- images

|-- assets

|-- build

| `-- classes

|-- database

| |-- mysql

| `-- postgres

| `-- liquibase

`-- docker

|-- compose-phpApache

| `-- webcontent -> ../../WebContent

`-- compose-postgres

|-- pgdb

`-- provision

28 directories

#### ****Git Setup****

Git is initialized inside the TitanicLost directory. Once files are added and the project is complete, Git will look like this:

A screenshot of a cell phone

Description automatically generated

**NOTE:**

**Careful attention must be paid to the .gitignore file. Because the project directories also serve as mount points for the containers, there are some items we don’t want to put in the remote repository (i.e. the web frontend for Github)**

**The complete. gitignore**

TitanicLost $ cat .gitignore

# Ignore the MacOS tracking files

.DS\_Store

.classpath

.project

.settings

/docker/compose-postgres/pgdb/data

/docker/compose-postgres/pgdb/logs

/docker/compose-postgres/pgadmin

**After deciding where to base the code, the top-level project directory is created and will be referenced as “project directory”. This assumes you already have a Github account, if not see the Source List at the end of this paper for where to go and ‘git’ one**

**$ mkdir dir: TitanicLost**

**$ cd TitanicLost**

**$ git init**

**$ git remote add origin** [git@github.com](mailto:git@github.com)**: <your repo>**

**That’s it for git, this is more of a reminder to do it than a step by step.**

**Let’s steam onward…**

#### ****Set Up Liquibase****

Liquibase is like “Git” for your database projects. It uses versioning, migration and rapid deployment features to support the Virtual/portable/container nature of TitanicLost. Installation methods vary however, they are fairly easy and are out of scope for this paper. See the Source List below if you need help with Liquibase install.

In the case of TitanicLost, Liquibase is used to rapidly provision the “titanic3” table in the DB after the containers are available

$ mkdir ./TitanicLost/database/postgres/liquibase

$ mkdir ./TitanicLost/database/postgres/liquibase/changelogs

To understand the setup, there are three critical pieces to the Liquibase

* Liquibase configure file to drive the work
* Liquibase driver for your DB brand (there are MANY!)
* A changelog file

TitanicLost is containerized and using Postgres for this project. Grab the necessary JAR driver and place it in the Liquibase directory.

The changelog directory will hold the change log for the TitanicLost DB table called “titanic3”

The liquibase.config file tells Liquibase where to find things, how to execute them and, of course, how to access the DB. In the case of this project, it is a fairly simple config:

driver: org.postgresql.Driver

classpath: postgresql-42.2.10.jar

url: jdbc:postgresql://localhost:5432/titanicdb

username: postgres

password: changeme

changeLogFile: changelogs/postgres/changelog-01.yml

As the source code for the changelog is lengthy, it can be found in Git for the project ( <https://github.com/harvash/titaniclost/tree/master/database/postgres/liquibase/changelogs/postgres>)

Essentially, this file communicates with the DB to describe the table columns and metadata. It is very easy to update this file and run it again as a new change set. This eliminates needing lengthy command line sessions or using a tool and have to retype the material every time.

The Liquibase changes will be introduced after the Docker containers are stood up and the DB is ready.

#### ****PHP/HTML Code & SQL Calls****

The heart of the project is the TitanicLost database on Postgres, however the front end is where it all comes together.

There are several pages of code for this to all work – a combination of PHP, HTML, Javascript & some Javascript – all the code can be seen in the

Github repo at <https://github.com/harvash/titaniclost>

The project also uses some interesting plugins like JQWidgets and AnyChart to interact between database and web browser. Please take a look in the Source List below for more information about these dynamic, easy to use plugins that make everyone looks like the World’s Best Web Engineer!

While the code itself verbose, it is worth a look in the Git before proceeding. The important bits here are the SQL calls from the Web container to the DB container (over the titanicnet) are worth showing here:

Top Ten Chart:

select destination, count(\*)

from titanic.titanic3 t

where destination !=''

group by destination

order by count(\*) desc

limit 10"

Passenger List: (this fills the JQXGrid)

SELECT \* FROM titanic.titanic3

Survival by Class Chart:

select

case t.pclass

when 1 then 'First'

when 2 then 'Second'

when 3 then 'Third'

end as pclass,

count(t.pclass)

from titanic.titanic3 t

where t.survived = 1

group by t.pclass

order by pclass;

Survival By Age and Gender Chart:

SELECT \* FROM (

SELECT CASE

when age between 0 and 17 then '0-17'

when age between 17 and 25 then '18-25'

when age between 25 and 35 then '26-35'

when age between 35 and 55 then '36-55'

when age between 55 and 69 then '56-69'

when age > 69 then '70+'

else 'Unknown'

END AS Age\_Range,

SUM(CASE WHEN Sex = 'female' THEN 1 ELSE 0 END) AS Males,

SUM(CASE WHEN Sex = 'male' THEN 1 ELSE 0 END) AS Females

FROM titanic.titanic3 t

WHERE survived = 1

GROUP BY CASE

when age between 0 and 17 then '0-17'

when age between 17 and 25 then '18-25'

when age between 25 and 35 then '26-35'

when age between 35 and 55 then '36-55'

when age between 55 and 69 then '56-69'

when age > 69 then '70+'

else 'Unknown'

END) as tmp\_table ORDER BY Age\_Range

All of these queries hit the Postgres DB and return a JSON element that allows the widgets to create some pretty interesting charts & graphs.

#### ****Docker work****

Ok, work starts on the containers – good times! Creating a simple network in docker. This allows you to create a network, define some networking parameters – and it is persistent, meaning it will NOT go away when you bring containers up/down (or do docker run). It allows a simple method to keep your containers talking to each other, as well as means to use internet and communicate with localhost.

##### Create titanicnet:

docker network create --attachable \

--driver=bridge \

--subnet=10.10.0.0/16 \

--ip-range=10.10.10.0/24 \

--gateway=10.10.10.1 \

titanicnet

**This setup allows it to connect to any container via –attachable, uses the localhost Network Interface Card (NIC) via “bridge” driver, defines the constraints of the network and finally, creates it with the name “titanicnet”**

**Listing Docker networks available by using command line:**

$ docker network ls

7ca900179dd2 titanicnet bridge local

**Inspecting existing named network:**

$ docker network inspect 7ca900179dd2

A screenshot of a cell phone

Description automatically generated

That is all for now with the network setup. The containers being created will hook into the network using Docker Compose files.

#### ****Pull Docker images****

Although not strictly required, I find that having the images “on-board” the localhost (i.e. your computer) helps when I need to customer configure something or want to test “bare” images for later Dockerfile work.

Images can be directly used with ‘docker run’ command or in a Dockerfile to build a custom image (as is the case with the titanicweb image)

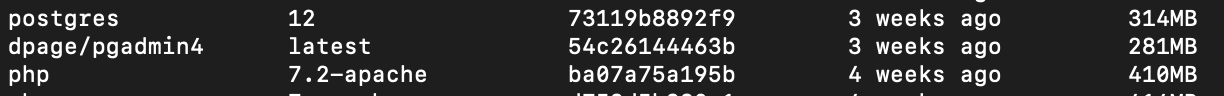
Here is an example of executing ‘docker pull’ to get the image from Docker and refresh the pgwatch image with the latest copy:

A screenshot of a cell phone

Description automatically generated

The images for the following were also pulled:

* Postgres:12
* Php:7.2-apache
* Dpage/pgadmin4:lastest



#### Creating images with Dockerfiles

##### Titanicweb Image:

First up is creating the web container images using a docker file. This is a fairly simple process with a small file for each setup. A great amount of customization is allowed in dockerfiles, please review the documentation for idea of everything one could put in this file.

This image is customized to include the necessary PDO database connection package used to interact with the backend Postgres DB

**Dockerfile**:

A screenshot of a cell phone

Description automatically generated

This file is in TitanicLost/docker/compose-phpApache. Notice we are using php:7.2-apache as the base for this image. You need not pull the image; Docker will handle that if necessary.

There is also a symlink to the WebContent file that has all the web files for displaying the website. This is necessary because the Docker Compose file will mount this as a named volume for use by the container, more on that later!

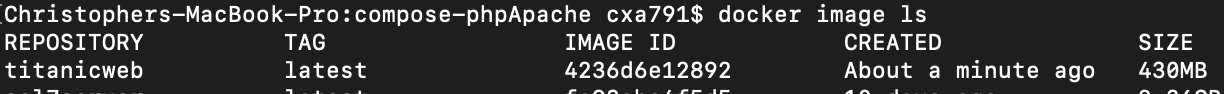
Build the image using Docker commands:

docker build -t titanicweb .

After a few minutes of Docker building the image, there is a nice brand-new image for use in the Compose file later:

A screenshot of a cell phone

Description automatically generated



##### Postgres DB Image:

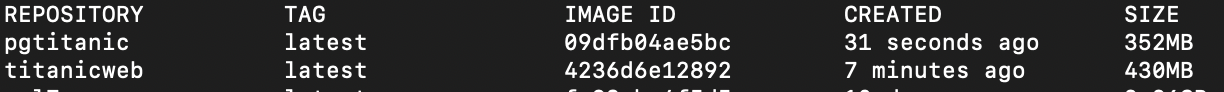
Steps similar to the above are used to create the database container.

The docker items for this container are in TitanicLost/docker/compose-postgres

This dockerfile creates the image from postgres using build command:

docker build -t pgtitanic:latest .

The build will commence and give output similar to building the web container image. Again, we have a shiny new image to use:



With custom image build done, docker compose can be used to create containers from these images, as we will see in the next section.

#### Docker-Compose

Fresh images in hand, containers will be built to give us working “servers” for the project.

Remember to change directories into the directory containing the docker-compose file for each container resource.

##### First up, the web container:

The web container will be built from a docker-compose file. This file dictates how the container is formed and what resources will be allocated to it. Note: need to be in the correct directory!

From TitanicLost/docker/compose-phpApache

A screenshot of a cell phone

Description automatically generated

Note that the container will hook into the network previously created and use a specific IP address (10.10.10.30).

It also directly mounts (or binds, in Docker-ese) the WebContent directory supplied via directory link in the container directory. This creates a connection between the host file system and the container. This means the content can be updated in real-time without re-building the container – very useful for testing code or patching an issue without bringing down the container

Creation is a simple docker compose command with the -d (detach) option set so as to start the container and leave it running.

$ docker-compose up -d

Creating webserver-php ... done

Checking the running container:

$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

cf7240f0a7a0 titanicweb "docker-php-entrypoi…" 2 minutes ago Up 2 minutes 0.0.0.0:80->80/tcp, 0.0.0.0:443->443/tcp webserver-php

##### Next passenger is database container:

This compose file has some additional complexity to it, chiefly the need to pass configuration options to the database and expose some ports so communication is possible.

A screenshot of a cell phone

Description automatically generated

A picture containing sitting, flower, white

Description automatically generated

Once again, the container hooks the titanicnet with specific IPs. It also incorporates named volumes for data persistence: pgdata,pglogs & pgadmin

This is a busy little guy; it sets up the database and also configures and deploys the administrative application PGAdmin. That’s two containers for the price of one ticket!

Further, it also mounts a few host directories to get the needed scripts and configuration data for everything to run.

Lastly, it slipstreams a vital piece of code for provisioning the database using the *docker-entrypoint-initdb.d*  hook. This tells the container to run anything in that directory after the container is stood up. In the case of this project, a host directory is mounted to a container volume that contains the provision.sh script, this sets up the database.

A screenshot of a cell phone

Description automatically generated

This script allows Postgres to use some extensions and also pre-creates the titanicdb schema. It also creates a role for the PGWatch DB monitoring app that will be running in the next section.

Let’s fire off the build command!

$ docker-compose up -d

Creating volume "compose-postgres\_pgdata" with default driver

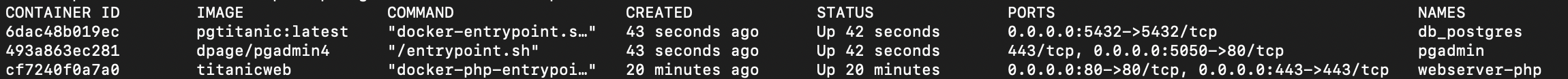
Creating volume "compose-postgres\_pglogs" with default driver

Creating volume "compose-postgres\_pgadmin" with default driver

Creating db\_postgres ... done

Creating pgadmin ... done

Now all three containers are up and cruising:



The additional named volumes are also full steam ahead:

A picture containing food

Description automatically generated

#### Complete Database Construction

Ok, time to stoke the boilers! Let’s finish up the DB construction and get the first look at the maiden voyage for Titanic Lost

While the DB schema and supporting objects were slipstreamed in using docker-entrypoint, the data itself still needs to be loaded. This is were Liquibase is going to help out and do some of the heavy lifting for creating the titanic3 DB table, after which a CSV file will serve to load the table with data.

##### Liquibase deliver

Using the Liquibase change set will demonstrate the first interactions with the container, namely creating the titanic3 table in the database. As the ports were exposed to the DB container, there is no need to log into the container. The file is simply run from the TitanicLost/database/postgres/liquibase directory right on the host. Two things are needed to complete this task – add the liquibase executable to the PATH and run the liquibase update command:

On this host, Liquibase is in the u01 directory above TitanicLost project folder

$ export PATH=$PATH:/Users/cxa791/u01/liquibase/liquibase-3.8.7

Run the update command:

$ liquibase --contexts="base" --defaultsFile=liquibase.postgres update

Liquibase Community 3.8.7 by Datical

Liquibase: Update has been successful.

The command has a few interesting bits:

* Contexts option allows the command to only run the parts of the change log that are tagged with “base” – that means changes can be staged selectively
* defaultsFile points to the configuration file that contains all the fuel to connect to the database and run the commands
* update simply means, push it to the DB

The changes can be observed by completing interactions directly through the browser using the PGAdmin container

##### Admining the PGAdmin

The PGAdmin container is alive and well at localhost:5050 (remember, the ports were shared during the compose actions)

A screenshot of a cell phone

Description automatically generated

Logging in will give an empty server set, but no worries – that’s the next step, add the DB Server!

Right click Servers, select Create and fill in the information for the DB server (which uses the static IP that was setup already)

A screenshot of a cell phone

Description automatically generated A screenshot of a cell phone

Description automatically generated

After successfully adding the DB server and drilling into the server menu, it is apparent that the dockerfiles, docker-compose, docker network, Liquibase & docker-entrypoint scripts have all done their duties and DB is born. There is even a bonus of some critical metrics beginning to show on the dashboards to the right:

A screenshot of a computer

Description automatically generated

The DB & PGAdmin containers are talking and interacting – great, now to add some data (or datas, if you prefer)!

##### Adding the data:

Now, remember that directory that held the dockerfile/docker-compose, it also had a directory that was mounted during the compose:

A close up of a logo

Description automatically generated

The one of specific interest is the scripts directory – you got it, that’s where the CSV file for the data lives. Docker being so clever, we don’t need to transfer anything from the host, it was done already during docker-compose! All that’s left is to login to the DB container and import the data with a few Postgres commands – too easy.

Docker containers that are running can be easily accessed using the docker exec command.

$ docker exec -it db\_postgres bash

After which, the command line can be used as normal. Let’s get that data imported!

* Change to postgres user:
  + # su – postgres
* Log to DB:
  + $ psql titanicdb postgres
* Set the path for using Titanicdb
  + # SET search\_path to titanic;
* Confirm login to correct DB by listing tables

A black sign with white text

Description automatically generated

* Lastly, import the data to the DB using postgres COPY command



* Now, PGAdmin container can be used to show the data in DB container DB

A screenshot of a cell phone

Description automatically generated

#### Using Docker-Run to start container from an image

So PGAdmin & the DB container are talking, that’s great but who’s listening? The answer is soon to be: “PGWatch” is listening.

Let’s take a different tact for running containers. Instead of custom images and docker-compose, you can just run a container from image.

Previously, the Docker image for PGWatch was pulled down from Docker repo – handy! Now, it can be instantiated by a simple run command for anywhere on the host (it is not directory specific):

$ docker run -d --network="titanicnet" -p 3000:3000 -p 8080:8080 -e PW2\_TESTDB=true -e PW2\_WEBPORT=8080 --name pw2 cybertec/pgwatch2

After which, a nice list of of running containers can be had by executing the trusty ‘docker ps’ command:

A picture containing bottle

Description automatically generated

#### Checking in with PGwatch

So, the project is near a close, let’s check in with PGWatch and setup monitoring for the titanicdb DB

Under localhost:8080, the configuration for PGwatch can be updated for the new DB

A screenshot of a cell phone

Description automatically generated

Now the dashboards can be viewed at localhost:3000 which uses Grafana and SQL to provide some nice metrics. NOTE: It will take a few mintues for PGwatch to “spin up” the new DB – be patient:

A screenshot of a video game

Description automatically generated

#### Finally, what’s it all look like!!

With all the containers chugging along full speed, the web front end is now open for business. Let’s take a look at two pages on the web project: The home page and the “full list” page:

##### HOME PAGE

A screenshot of a cell phone

Description automatically generated

##### Clicking “Full List” button

A screenshot of a cell phone

Description automatically generated

#### Closing Remarks

So that wraps up the Titanic Lost project – no icebergs were harmed in the making of this Docker Project. Please visit the sources listed below for information and ideas about how all this technology works.

#### Source List

AnyChart: <https://www.anychart.com/features/>

JQWidgets: <https://www.jqwidgets.com/bind-jquery-chart-to-mysql-database-using-php/>

Docker: <https://docs.docker.com/>

Docker Install (Mac): <https://docs.docker.com/docker-for-mac/install/>

Docker-Compose: <https://docs.docker.com/compose/>

Docker network create: <https://docs.docker.com/engine/reference/commandline/network_create/>

Liquibase install: <https://www.liquibase.org/documentation/installation/index.html>

PGwatch: <https://www.cybertec-postgresql.com/en/products/pgwatch/>

Git PGwatch: <https://github.com/cybertec-postgresql/pgwatch2>