If you haven’t heard of Docker, it is a system that allows projects to be split into discrete units (i.e. containers) that each operate within their own virtual environment. Each container has a blueprint written in its Dockerfile that describes all of the operating parameters including operating system and package dependencies/requirements. Docker images are easily distributed and, because they are self-contained, will operate on any other system that has Docker installed, include servers.

When multiple instances/users attempt to start a Shiny App at the same time, only a single R session is initiated on the serving machine. This is problematic. For example, if one user starts a process that takes 10 seconds to complete, all other users will need to wait until that process has completed before any other tasks can be processed.

One of the benefits of deploying a containerised Shiny App is that each new instance will run in its own R session.

In this article, I will go through the steps that I took to deploy an app developed in Shiny onto a fully-functional web server. All the code here, plus some basic web-server configuration

**Getting Started**

Docker can be installed following the instructions [here](https://docs.docker.com/install/linux/docker-ce/ubuntu/), making sure to also follow the post-installation instructions [here](https://docs.docker.com/install/linux/linux-postinstall/). Docker-compose will also need to be installed by following these instructions [here](https://docs.docker.com/compose/install/). For the example presented here, Docker was installed on an Ubuntu 18.04 OS in a Virtual Box; the same approach will be used for other apps running on an [AWS EC2 instance](https://aws.amazon.com/ec2/).

I will be using the words *image* and *container* throughout this article. A Docker *image* is a functioning snapshot of the blueprint. A running image is called a *container*, which is operating by receiving, processing and sending information to the client or other containers.

**Docker**

The Dockerfile contains the schematics of a Docker container, that is it is used to call a base image and define and customisations that need to be made for the specific application to run correctly.

This is the Dockerfile that describes our Shiny App (i.e. the Default app when a new “Shiny Web App …” is created in RStudio):

FROM rocker/shiny:3.5.1  
  
RUN apt-get update && apt-get install libcurl4-openssl-dev libv8-3.14-dev -y &&\  
 mkdir -p /var/lib/shiny-server/bookmarks/shiny  
  
# Download and install library  
RUN R -e "install.packages(c('shinydashboard', 'shinyjs', 'V8'))"  
  
# copy the app to the image COPY shinyapps /srv/shiny-server/  
# make all app files readable (solves issue when dev in Windows, but building in Ubuntu)  
RUN chmod -R 755 /srv/shiny-server/  
  
EXPOSE 3838  
  
CMD ["/usr/bin/shiny-server.sh"]

This blueprint is using the base image rocker/shiny built on R version 3.5.1. For some packages to run properly on an Ubuntu OS (the container’s base operating system) I needed to install libcurl4 and libv8 by adding the following lines to our Dockerfile.

RUN apt-get update &&\  
 apt-get install libcurl4-openssl-dev libv8-3.14-dev -y &&\  
 mkdir -p /var/lib/shiny-server/bookmarks/shiny

The packages that the app depends on are also installed *via.* the Dockerfile with the RUN statement:

RUN R -e "install.packages(c('shinydashboard', 'shinyjs', 'V8'))"

Our app.R file (i.e. the Shiny App) is copied from the shinyapps directory on the host PC to a folder inside the container image with the COPY statement.

I had some file permission issues while building on Ubuntu *via.* a Windows VM. To overcome this, all copied file permissions were changed with chmod -R 755 to ensure they were readable and executable inside the container.

# Copy the app to the image  
COPY shinyapps /srv/shiny-server/  
  
# Make all app files readable  
RUN chmod -R +r /srv/shiny-server/

The final two lines expose port 3838 to receive incoming traffic and tell docker how to start the app. With the Dockerfile complete, images are easily built with the following one-liner:

docker build -t telethonkids/new\_shiny\_app ./path/to/Dockerfile/directory

and can be run as a standalone container with:

docker run --name=shiny\_app --user shiny --rm -p 80:3838 telethonkids/new\_shiny\_app

and accessed in a browser at [http://127.0.0.1](http://127.0.0.1/).

**ShinyProxy**

The story doesn’t end there. ShinyProxy can be easily set up in another container to facilitate container creation, this is an example from ShinyProxy’s for a containerised deployment:

FROM openjdk:8-jre  
  
RUN mkdir -p /opt/shinyproxy/  
RUN wget https://www.shinyproxy.io/downloads/shinyproxy-2.1.0.jar -O /opt/shinyproxy/shinyproxy.jar  
COPY application.yml /opt/shinyproxy/application.yml  
  
WORKDIR /opt/shinyproxy/  
CMD ["java", "-jar", "/opt/shinyproxy/shinyproxy.jar"]

The apps that are to be hosted are defined in the application.yml file. This contains the information for ShinyProxy to launch Shiny Apps and needs to be copied into the container from the host with COPY application.yml /opt/shinyproxy/application.yml.

The following code snippet has only one Shiny App, but multiple app configurations can be added to specs.

proxy:  
 title: Telethon Kids Institute Shiny Apps  
 hide-navbar: false  
 landing-page: /  
 heartbeat-rate: 10000  
 heartbeat-timeout: 600000  
 port: 8080  
 docker:  
 internal-networking: true  
 specs:  
 - id: shiny\_app  
 display-name: New Shiny App  
 description: The default app when initiating a new shiny app via. RStudio  
 container-cmd: ["/usr/bin/shiny-server.sh"]  
 container-image: telethonkids/new\_shiny\_app  
 container-network: tki-net  
 container-env:  
 user: "shiny"  
 environment:  
 - APPLICATION\_LOGS\_TO\_STDOUT=false

This ShinyProxy set up listens for traffic on port 8080 and will time-out after a period of inactivity, which has been set to 10 minutes.

heartbeat-rate: 10000  
heartbeat-timeout: 600000

internal-networking: true must be set because ShinyProxy is being run on a container on the same host as the Shiny App. The configuration for the each app is listed under specs, which give instructions on how to launch the container, what Docker image to use to start the container and what network it should be listening on (note that the container-network must be the same as the network listed in the docker-compose.yaml file, more to come soon).

container-cmd: ["/usr/bin/shiny-server.sh"]  
container-image: telethonkids/new\_shiny\_app  
container-network: tki-net

We also have some environment variables that we want to set so the R session is being run for the user *shiny* and logs are not recorded.

container-env:  
 user: 'shiny'  
 environment:  
 - APPLICATION\_LOGS\_TO\_STDOUT=false

**docker-compose.yml**

The docker-compose.yml file contains a series of instructions that can be used to build and launch all the containers needed for deployment in a single or multi-container project. The benefit of this is that a complex network of containers and networks can be version controlled, and launched with one line of code.

version: "3.6"

services:

shinyproxy:

depends\_on:

- nginx

- influxdb

image: telethonkids/shinyproxy

container\_name: tki\_shinyproxy

restart: on-failure

networks:

- tki-net

volumes:

- ./shinyproxy/application.yml:/opt/shinyproxy/application.yml

- /var/run/docker.sock:/var/run/docker.sock

ports:

- 8080:8080

networks:

tki-net:

name: tki-net

Docker uses networks to enable communication between containers. For ShinyProxy to communicate properly with the Shiny App, the network specified in docker-compose.yml must be the same as the same as that listed in application.yml. (Tip, if you’re using a docker-compose file to launch the app, don’t set up the docker network manually).

Each container listed in services: is named in container\_name, with the Dockerfile being pointed to *via.* the context and Dockerfile variables.   
If the container was terminated with an error code, the restart line instructs Docker what action is to be taken . The networks variables lists the Docker network that the container has access to. Finally, the app is exposed on port 80.

The network used by all applications in this docker-compose must match the network specified in the ShinyProxy application.yml.

networks:  
 telethonkids-net:  
 name: telethonkids-net

With everything in place, the ShinyProxy image is built and the container(s) started with docker-compose build and docker-compose up -d. (**Important**, the app defined in application.yml also be built). Containers are stopped and removed with docker-compose down.

**Bonus Tip**

Another app that I made used CSS to do some custom formatting (only 4 tags). For some reason the style sheet was ignored by Chrome when run with ShinyProxy, but not when running standalone. The problem did not exist when tested on Firefox and Internet Explorer browsers. I think this had something to do with the iframe used by ShinyProxy. This was resolved by putting the style in app.R within a head tag and removing the reference to the external style.css.