The project contains the demo Shiny app in the app folder:

.

├── app

│ ├── global.R

│ ├── server.R

│ └── ui.R

├── .gitignore

├── .gitlab-ci.yml

├── Dockerfile

├── LICENSE

├── README.md

└── shinyproxy-hello.Rproj

**Working with an existing image**

First, you'll learn how to work with an existing image.

**Pull image**

You can pull the image made from the GitLab project's [container registry](https://gitlab.com/analythium/shinyproxy-hello/container_registry) using the docker pull CLI command (you need to have the [Docker Desktop](https://docs.docker.com/desktop/) running on our local machine):

docker pull registry.gitlab.com/analythium/shinyproxy-hello/hello

**Image tag**

The image is tagged as REGISTRY/USER/PROJECT/IMAGE:TAG. In this case the TAG is latest which is the default tag when not specified otherwise. (When the REGISTRY os not provided, Docker uses the Docker Hub as the default and you can follow the USER/IMAGE:TAG pattern).

**Authentication**

You don't need to authenticate for public images (like this one), but in case you are trying to pull a private image from a private GitLab project, you need to log into the GitLab Container registry as:

docker login registry.gitlab.com

This command will ask for our credentials interactively. If you want, you can provide your username and password. But it is usually recommended to use a personal access token (PAT) instead of your password because PAT can have more restricted scopes, i.e. only used to (read) access the container registry which is a lot more secure.

cat ~/my\_password.txt | docker login --username USER --password-stdin

where ~/my\_password.txt is a file with the PAT in it, USER is the GitLab username.

**Run container**

After pulling the image, you can use docker run to run a command in a new container based on the image. Use the -p 4000:3838 argument and the image tag:

docker run -p 4000:3838 registry.gitlab.com/analythium/shinyproxy-hello/hello

The -p is a shorthand for --publish, that instructs Docker to publish a container’s port to the host port. In our example, 3838 is the container's port which is mapped to port 4000 of the host machine. As a result, you can visit 127.0.0.1:4000 where you'll find the Shiny app. Hit Ctrl+C to stop the container.

Read all about the docker run command [here](https://docs.docker.com/engine/reference/commandline/run/). You'll learn about the 3838 port in a bit.

**Shiny host and port**

When we discussed [local hosting of Shiny apps](https://hosting.analythium.io/run-shiny-apps-locally/) and runApp, we did not review all the possible arguments for this R function. Besides the app location (app object, list, file, or directory) there are two other important arguments:

* host: this defines the IP address (defaults to 'localhost': 127.0.0.1),
* port: TCP port that the application should listen on; a random port when no value provided.

When you run the shiny app locally, you see a message Listening on http://127.0.0.1:7800 or similar, which is the protocol (HTTP), the host address, and the port number. The Shiny app is running in a web server that listens to client requests and provides a response.

**Build a new image**

So far you saw how to use the [basic docker commands](https://hosting.analythium.io/docker-basics-for-data-apps/) to pull and run images. Now you'll build a Docker image by recreating the simple Shiny app that we worked with before.

**Create the Dockerfile**

Create a file named Dockerfile (touch Dockerfile) then open the file and copy the following text into the file and save:

FROM rocker/r-base:latest

LABEL maintainer="USER <user@example.com>"

RUN apt-get update && apt-get install -y --no-install-recommends \

sudo \

libcurl4-gnutls-dev \

libcairo2-dev \

libxt-dev \

libssl-dev \

libssh2-1-dev \

&& rm -rf /var/lib/apt/lists/\*

RUN install.r shiny

RUN echo "local(options(shiny.port = 3838, shiny.host = '0.0.0.0'))" > /usr/lib/R/etc/Rprofile.site

RUN addgroup --system app \

&& adduser --system --ingroup app app

WORKDIR /home/app

COPY app .

RUN chown app:app -R /home/app

USER app

EXPOSE 3838

CMD ["R", "-e", "shiny::runApp('/home/app')"]

You can use the docker build command to build the image from the Dockerfile:

docker build -t registry.gitlab.com/analythium/shinyproxy-hello/hello .

The -t argument is the tag, the . at the end refers to the *context* of the build, which is our current directory (i.e. where the Dockerfile resides) with a set of files based on which the image is built. So this is where the app folder and the R scripts need to be placed.

Once the docker image is built, you can run the container as before to make sure the app is working as expected:

docker run -p 4000:3838 registry.gitlab.com/analythium/shinyproxy-hello/hello

**Push image**

Push the locally build Docker image to the container registry:

docker push registry.gitlab.com/analythium/shinyproxy-hello/hello

The image tag should start with the registry name unless you are pushing to Docker Hub. When the image tag is not specified, Docker will treat the new image as :latest automatically. Read more about Docker tags and semantic versioning [here](https://medium.com/@mccode/using-semantic-versioning-for-docker-image-tags-dfde8be06699).

**What's in the Dockerfile**

Let's review the Dockerfile line by line. The full Dockerfile reference can be found [here](https://docs.docker.com/engine/reference/builder/).

The [FROM instruction](https://docs.docker.com/engine/reference/builder/#from) initializes a new build stage and sets the *base image*.  
Take the latest r-base image from the rocker project (see on [Docker Hub](https://hub.docker.com/r/rocker/r-base)):

FROM rocker/r-base:latest

The LABEL instruction is optional, it adds metadata to an image, e.g. who to contact in case of issues or questions:

LABEL maintainer="USER <user@example.com>"

The [RUN instruction](https://docs.docker.com/engine/reference/builder/#run) executes the command in a new *layer* (a [layer](https://docs.docker.com/glossary/#layer) is a modification to the image) on top of the current image. The following command updates the base image with a couple of libraries that are required by Shiny and related R packages (system dependencies):

RUN apt-get update && apt-get install -y --no-install-recommends \

sudo \

libcurl4-gnutls-dev \

libcairo2-dev \

libxt-dev \

libssl-dev \

libssh2-1-dev \

&& rm -rf /var/lib/apt/lists/\*

The following RUN command uses the [littler](https://github.com/eddelbuettel/littler) command-line interface shipped with the r-base image to install the Shiny package and its dependencies:

RUN install.r shiny

The next command sets the options in the Rprofile.site file which are going to be loaded by the R session. These options specify the Shiny host and port that runApp will use.

Do not run containers as root in production. Running the container with root privileges allows unrestricted use which is to be avoided in production. Although you can find lots of examples on the Internet where the container is run as root, this is generally considered bad practice.

Switching to the root USER opens up certain security risks if an attacker gets access to the container. In order to mitigate this, switch back to a non privileged user after running the commands you need as root. – [Hadolint rule DL3002](https://github.com/hadolint/hadolint#rules)

The following command creates a Linux group and user, both called app. This user will have access to the app instead of the default root user:

RUN addgroup --system app \

&& adduser --system --ingroup app app

You can read more about security considerations [here](https://engineering.bitnami.com/articles/why-non-root-containers-are-important-for-security.html) and Dockerfile related code smells [here](https://arxiv.org/abs/2103.12298). Read about [best practices](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/) and [linting rules](https://github.com/hadolint/hadolint#readme) in general that can be helpful in reducing vulnerabilities of your containerized application.

The [WORKDIR instruction](https://docs.docker.com/engine/reference/builder/#workdir) sets the working directory for subsequent instructions. Change this to the home folder of the app user which is /home/app:

WORKDIR /home/app

The [COPY instruction](https://docs.docker.com/engine/reference/builder/#copy) copies new files or directories from the source (our app folder containing the R script files for our Shiny app) and adds them to the file system of the container at the destination path (. refers to the current work directory defined at the previous step):

COPY app .

The next command sets permissions for the app user:

RUN chown app:app -R /home/app

The [USER instruction](https://docs.docker.com/engine/reference/builder/#user) sets the user name (or UID) and optionally the user group (or GID) to use when running the image:

USER app

The [EXPOSE instruction](https://docs.docker.com/engine/reference/builder/#expose) tells Docker which ports the container listens on at runtime. Set this to the Shiny port defined in the Rprofile.site file:

EXPOSE 3838

Finally, the [CMD instruction](https://docs.docker.com/engine/reference/builder/#cmd) closes off our Dockerfile. The CMD instruction provides the defaults for an executing container. There can only be one CMD instruction in a Dockerfile (only the last CMD will take effect). Our CMD specifies the executable ("R") and parameters for the executable in an array. The -e option means you are running an expression that is shiny::runApp('/home/app'). The expression will run the Shiny app that we copied into the /home/app folder:

CMD ["R", "-e", "shiny::runApp('/home/app')"]

Build the image using docker build by specifying the tag (-t) and the context (. indicates the current directory):

docker build -t registry.gitlab.com/analythium/shinyproxy-hello/hello .

You can test and push the locally build Docker image to the container as before.

**Summary**

With the newfound ability to wrap any Shiny app in a Docker container, you'll be able to deploy these images to many different hosting platforms. Of course, there is a lot more to learn, e.g. about handling dependencies, persisting data across sessions and containers, and so on. We'll cover these use cases in due time. Until then, celebrate this milestone, check out further readings, and try to containerize some of your own Shiny apps.