WiFi: GSExpLab
Password: gs19boston
Cloud alternative: https://dockr.ly/intersystems

Containers Bootcamp

Presenter: Kerry Kirkham

Table of Contents

Introduction 2

Exercise 1 — Basic Container Operations 3

Exercise 2 — Building Your Container 7

Exercise 3 — Persisting Data with Bind Mounts 14

Exercise 4 — Persisting Data with Volumes 16

Exercise 5 — Running an InterSystems IRIS Docker Container 19

Exercise 6 — Using Durable %SYS for Data Persistence 22

Summary 25

- Next Steps 25
- Further Resources 26
- Recommended Global Summit Sessions 26
- Further Online Resources 26

Introduction

Welcome to the Containers Bootcamp at the 2019 Global Summit! By the time you complete this bootcamp, you will feel comfortable working with InterSystems IRIS in Docker containers.

Throughout this bootcamp, you will complete exercises and view accompanying presentations. Supporting materials will be linked in the provided slide deck in case you miss any of the presentations or wish to revisit the materials later.

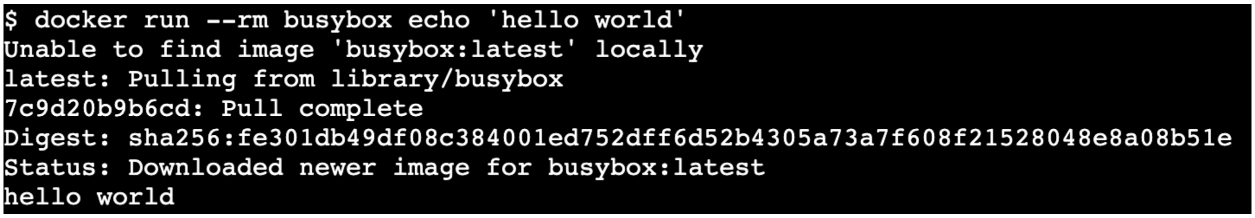
Enjoy the Containers Bootcamp!

Exercise 1 — Basic Container Operations

1.1 — Creating and Running Containers

Step 1. Create and start a new temporary BusyBox container to ensure that Docker is installed correctly. If you do not have the BusyBox image locally already, Docker will download it first.

```
$ docker run --rm busybox echo 'hello world'
```

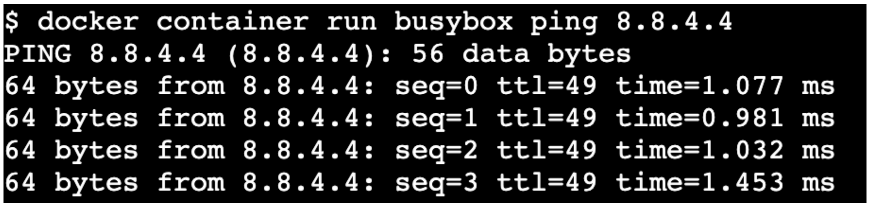


```
$ docker run --rm busybox echo 'hello world'
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
7c9d20b9b6cd: Pull complete
Digest: sha256:fe301db49df08c384001ed752dff6d52b4305a73a7f608f21528048e8a08b51e
Status: Downloaded newer image for busybox:latest
hello world
```

The `--rm` part of this command removes the container after the call was executed.

Step 2. Create and start a new BusyBox container. Include a `ping` to see that it is running and accessible.

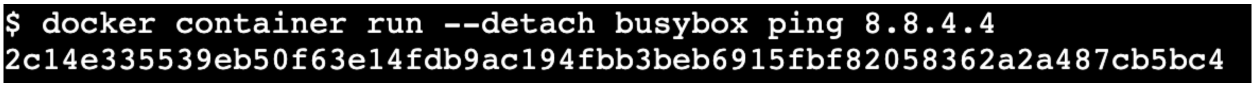
```
$ docker container run busybox ping 8.8.4.4
```



```
$ docker container run busybox ping 8.8.4.4
PING 8.8.4.4 (8.8.4.4): 56 data bytes
64 bytes from 8.8.4.4: seq=0 ttl=49 time=1.077 ms
64 bytes from 8.8.4.4: seq=1 ttl=49 time=0.981 ms
64 bytes from 8.8.4.4: seq=2 ttl=49 time=1.032 ms
64 bytes from 8.8.4.4: seq=3 ttl=49 time=1.453 ms
```

Step 3. Press **Ctrl+C** after a few pings to stop the pinging process and exit the container. Then create a second container, this time, in detached mode.

```
$ docker container run --detach busybox ping 8.8.4.4
```



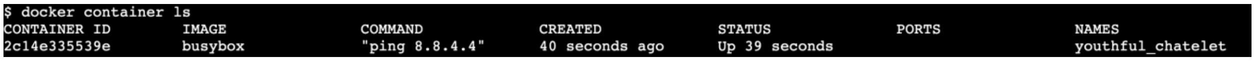
```
$ docker container run --detach busybox ping 8.8.4.4
2c14e335539eb50f63e14fdb9ac194fbb3beb6915fbf82058362a2a487cb5bc4
```

Docker displays a long, hexadecimal number; this is the full container ID of your new container. This container is now running detached, which means it is running as a background process. Thus, it is not printing the ping results to your terminal.

1.2 — Listing and Removing Containers

Step 1. To see all of your running containers, you can run the command below.

```
$ docker container ls
```

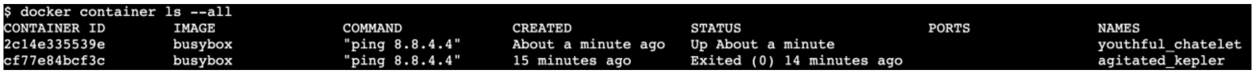


```
$ docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
2c14e335539e        busybox             "ping 8.8.4.4"      40 seconds ago      Up 39 seconds                           youthful_chatelet
```

As you can see, the `ls` command shows you the containers that are running on your machine. It includes useful information, such as the container name and ID and the image name. If you do not provide a name when initializing the container, Docker generates a random name.

Step 2. This command, however, only shows you the containers you have running. To see all containers, running or stopped, run the command below:

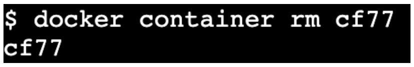
```
$ docker container ls --all
```



```
$ docker container ls --all
CONTAINER ID        IMAGE               COMMAND             CREATED              STATUS                      PORTS               NAMES
2c14e335539e        busybox             "ping 8.8.4.4"      About a minute ago   Up About a minute                               youthful_chatelet
cf77e84bcf3c        busybox             "ping 8.8.4.4"      15 minutes ago       Exited (0) 14 minutes ago                       agitated_kepler
```

Step 3. You will notice that the temporary BusyBox container from **Step 1.1** is shown here, with a status of *`Exited`*. You can remove this container with the `docker rm` command. In the command below, replace `<CONTAINER ID>` with the ID of your exited container. **Note:** You can typically just provide the first few characters of the container ID here.

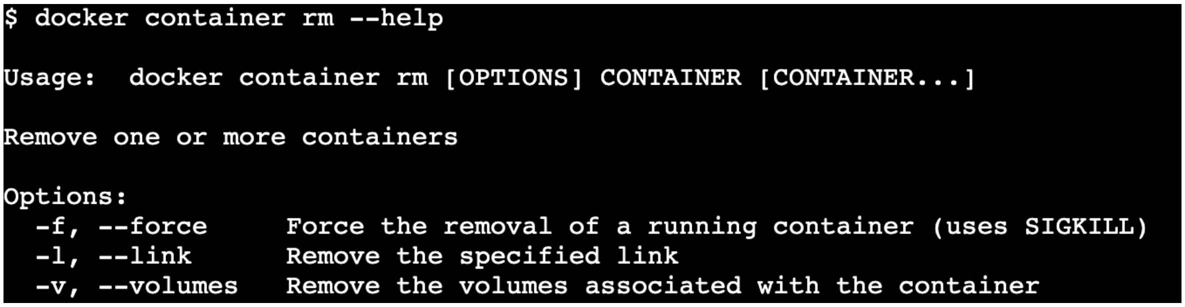
```
$ docker container rm <CONTAINER ID>
```



```
$ docker container rm cf77
cf77
```

Step 4. With this command, you cannot directly remove a running container, only a stopped one. To learn more about the `rm` command and your options for forcing the removal of a running container, enter the `help` command below.

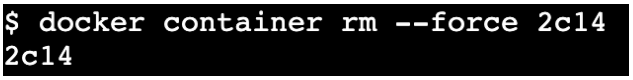
```
$ docker container rm --help
```



```
$ docker container rm --help

Usage:  docker container rm [OPTIONS] CONTAINER [CONTAINER...]

Remove one or more containers

Options:
  -f, --force     Force the removal of a running container (uses SIGKILL)
  -l, --link      Remove the specified link
  -v, --volumes   Remove the volumes associated with the container
```

Step 5. As you can see, adding `--force` will force the removal of a running container. Run a forced removal on your active BusyBox container, replacing `<CONTAINER ID>` with the first few characters from the ID of your container.

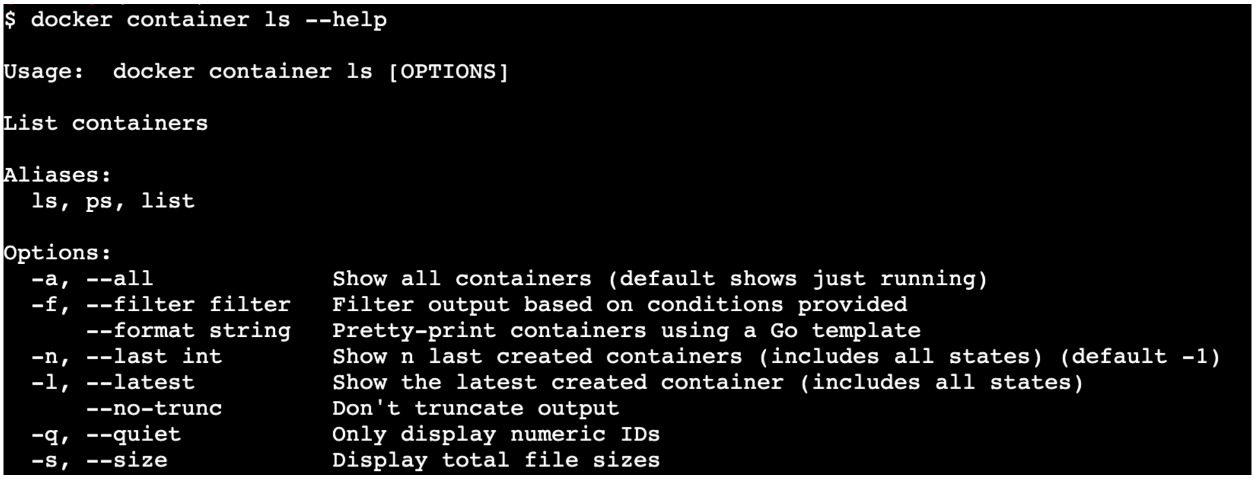
```
$ docker container rm --force <CONTAINER ID>
```



```
$ docker container rm --force 2c14
2c14
```

Step 6. The `--help` option can be applied to most commands within Docker if you want to see more information. Take a look at the options for the `ls` command by viewing the information under `help`.

```
$ docker container ls --help
```



```
$ docker container ls --help

Usage:  docker container ls [OPTIONS]

List containers

Aliases:
  ls, ps, list

Options:
  -a, --all             Show all containers (default shows just running)
  -f, --filter filter   Filter output based on conditions provided
      --format string   Pretty-print containers using a Go template
  -n, --last int        Show n last created containers (includes all states) (default -1)
  -l, --latest          Show the latest created container (includes all states)
      --no-trunc        Don't truncate output
  -q, --quiet           Only display numeric IDs
  -s, --size            Display total file sizes
```

Step 7. You can play around with some of these options for listing containers. For instance, you can list all containers — both started and stopped — by only their numeric IDs:

```
$ docker container ls --all --quiet
```

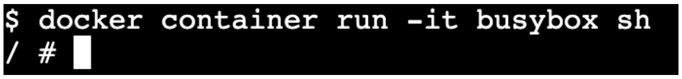
Step 8. You can also kill all containers, regardless of what state they are in. **Note:** If you already had containers on your machine before this lab, **carefully** remove this bootcamp's containers individually using their IDs.

```
$ docker container rm --force $(docker container ls --quiet --all)
```

1.3 — Writing to Containers

Step 1. Create another container using the BusyBox image, and connect to its shell in interactive mode. You can do this by using the `-i` flag (as well as the `-t` flag, which requests a TTY connection).

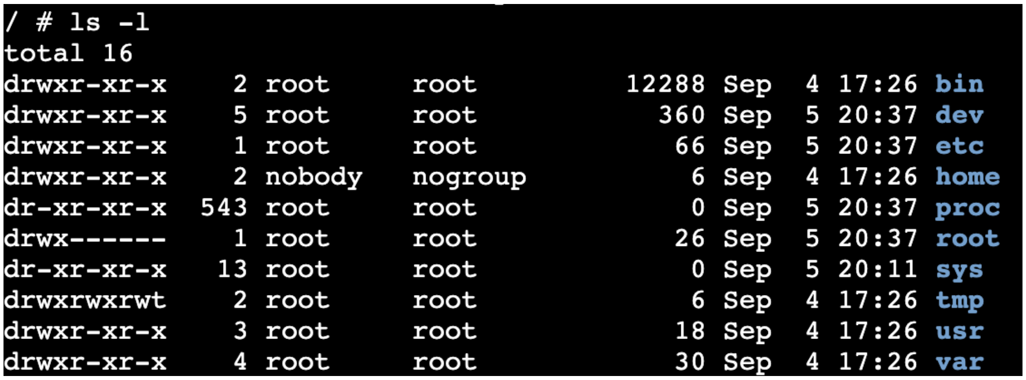
```
$ docker container run -it busybox sh
```



```
$ docker container run -it busybox sh
/ # █
```

Step 2. From here, you can explore your container's file system using the `ls` command.

```
# ls -l
```



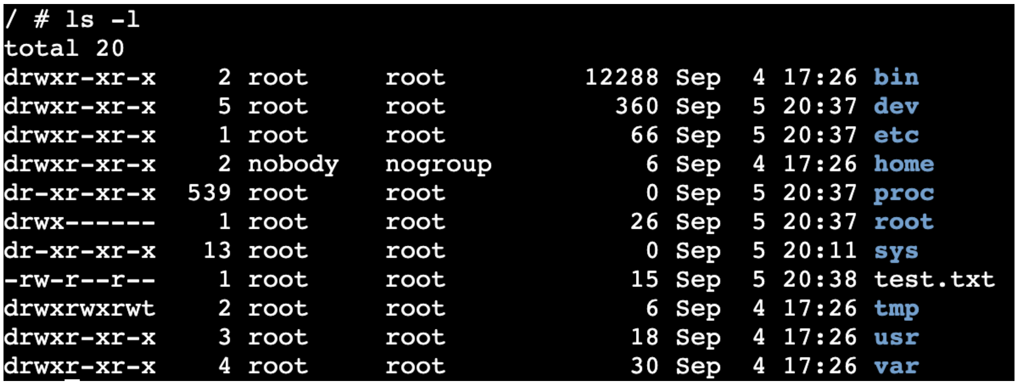
```
/ # ls -l
total 16
drwxr-xr-x    2 root     root         12288 Sep  4 17:26 bin
drwxr-xr-x    5 root     root           360 Sep  5 20:37 dev
drwxr-xr-x    1 root     root            66 Sep  5 20:37 etc
drwxr-xr-x    2 nobody   nogroup          6 Sep  4 17:26 home
dr-xr-xr-x  543 root     root             0 Sep  5 20:37 proc
drwx------    1 root     root            26 Sep  5 20:37 root
dr-xr-xr-x   13 root     root             0 Sep  5 20:11 sys
drwxrwxrwt    2 root     root             6 Sep  4 17:26 tmp
drwxr-xr-x    3 root     root            18 Sep  4 17:26 usr
drwxr-xr-x    4 root     root            30 Sep  4 17:26 var
```

Step 3. Create a new text file.

```
# echo 'Hello there…' > test.txt
```

Step 4. List your files again to see that a text file has been created.

```
# ls -l
```



```
/ # ls -l
total 20
drwxr-xr-x    2 root     root         12288 Sep  4 17:26 bin
drwxr-xr-x    5 root     root           360 Sep  5 20:37 dev
drwxr-xr-x    1 root     root            66 Sep  5 20:37 etc
drwxr-xr-x    2 nobody   nogroup          6 Sep  4 17:26 home
dr-xr-xr-x  539 root     root             0 Sep  5 20:37 proc
drwx------    1 root     root            26 Sep  5 20:37 root
dr-xr-xr-x   13 root     root             0 Sep  5 20:11 sys
-rw-r--r--    1 root     root            15 Sep  5 20:38 test.txt
drwxrwxrwt    2 root     root             6 Sep  4 17:26 tmp
drwxr-xr-x    3 root     root            18 Sep  4 17:26 usr
drwxr-xr-x    4 root     root            30 Sep  4 17:26 var
```

Step 5. Exit your container.

```
# exit
```

Step 6. Run the same command as before to start another container from the same image.

```
$ docker container run -it busybox sh
```

Step 7. Try to find your `test.txt` file inside this new container via the `ls -l` command. You will see that it is nowhere to be found. Exit this container.

```
# exit
```

Exercise 2 — Building Your Container

2.1 — Creating a New Container Image and Adding a New File

Step 1. Run another new BusyBox container and drop it into a shell on that container.

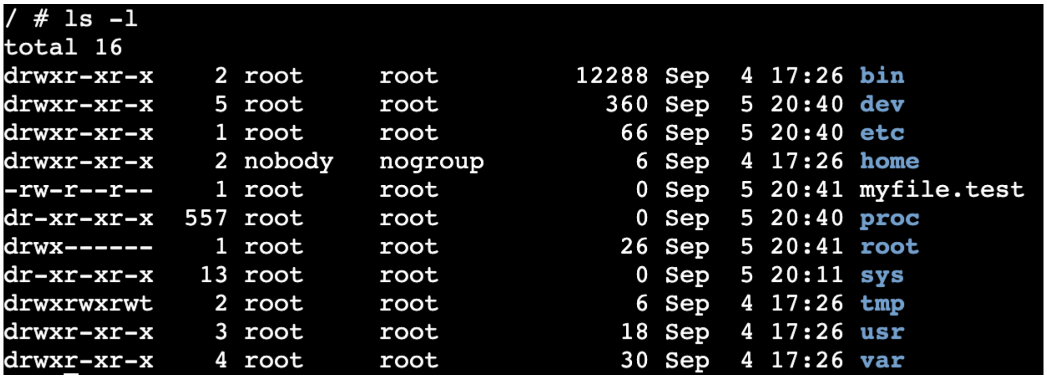
```
$ docker run -it busybox sh
```

Step 2. Create an empty file on this container.

```
# touch myfile.test
```

Step 3. List your files to confirm that `myfile.test` has been created.

```
# ls -l
```



```
/ # ls -l
total 16
drwxr-xr-x    2 root     root         12288 Sep  4 17:26 bin
drwxr-xr-x    5 root     root           360 Sep  5 20:40 dev
drwxr-xr-x    1 root     root            66 Sep  5 20:40 etc
drwxr-xr-x    2 nobody   nogroup          6 Sep  4 17:26 home
-rw-r--r--    1 root     root             0 Sep  5 20:41 myfile.test
dr-xr-xr-x  557 root     root             0 Sep  5 20:40 proc
drwx------    1 root     root            26 Sep  5 20:41 root
dr-xr-xr-x   13 root     root             0 Sep  5 20:11 sys
drwxrwxrwt    2 root     root             6 Sep  4 17:26 tmp
drwxr-xr-x    3 root     root            18 Sep  4 17:26 usr
drwxr-xr-x    4 root     root            30 Sep  4 17:26 var
```

Step 4. Exit your container.

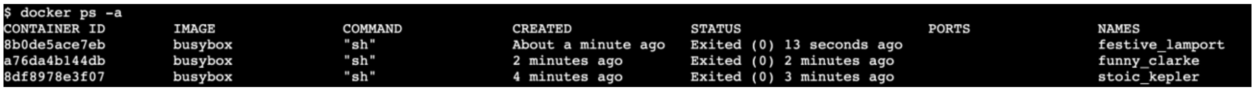
```
# exit
```

Step 5. List all of your containers, but this time use the `ps` – or *process status* – command. It works the same as `docker container ls` and is often a shorthand choice.

```
$ docker ps
```

Step 6. Notice that this only shows your running containers. Add the `-a` tag to see all containers, both running and stopped. Note that your results may not look exactly like the provided screenshot, but they should look similar.

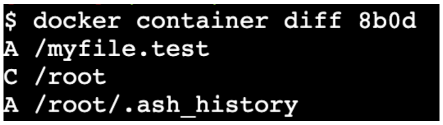
```
$ docker ps -a
```



```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED              STATUS                      PORTS               NAMES
8b0de5ace7eb        busybox             "sh"                About a minute ago   Exited (0) 13 seconds ago                       festive_lamport
a76da4b144db        busybox             "sh"                2 minutes ago        Exited (0) 2 minutes ago                        funny_clarke
8df8978e3f07        busybox             "sh"                4 minutes ago        Exited (0) 3 minutes ago                        stoic_kepler
```

Step 7. You will use the `diff` command to see what has changed about a container relative to its image. To do this, use the ID of your most recent container in the command below to see the difference between the container and its base image.

```
$ docker container diff <CONTAINER ID>
```

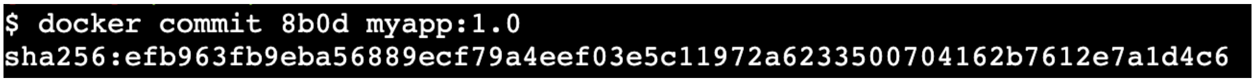


```
$ docker container diff 8b0d
A /myfile.test
C /root
A /root/.ash_history
```

The results of this command show you information about what has changed. Lines that begin with an *A* show that a file or directory was added. Lines beginning with a *C* show that a file or directory was changed. Though none are present in this example, lines beginning with a *D* would indicate that a file or directory was deleted.

Step 8. When you created `myfile.test`, you wrote information to the container's read/write layer. Now, you are going to save that read/write layer as a new read-only image layer. With this, you will create a new container image that reflects the additions you made. You can do this with the `commit` command, where *myapp* is the new image name and *1.0* is the image tag.

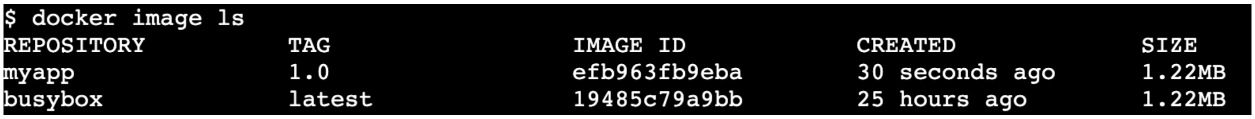
```
$ docker commit <CONTAINER ID> myapp:1.0
```



```
$ docker commit 8b0d myapp:1.0
sha256:efb963fb9eba56889ecf79a4eef03e5c11972a6233500704162b7612e7a1d4c6
```

Step 9. Verify that your new image has been created by listing all of your images.

```
$ docker image ls
```



```
$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
myapp               1.0                 efb963fb9eba        30 seconds ago      1.22MB
busybox             latest              19485c79a9bb        25 hours ago        1.22MB
```

2.2 — Create a New Container Image via Build Process

2.2.1 — Creating the Node Application and Container Definition

Step 1. Make sure you are in the `root` directory. You can verify your current directory with the `pwd` command.

```
$ pwd
```

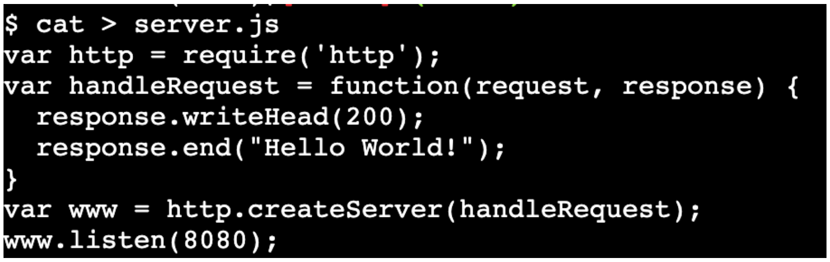
Step 2. Run the `cat` command to edit text for a new file called *server.js*.

```
$ cat > server.js
```

This will leave a prompt open for more text. Enter the contents of the `server.js` file below.

```
var http = require('http');
var handleRequest = function(request, response) {
  response.writeHead(200);
  response.end("Hello World!");
}
var www = http.createServer(handleRequest);
www.listen(8080);
```

Step 3. Press **Ctrl+D** to save the contents.



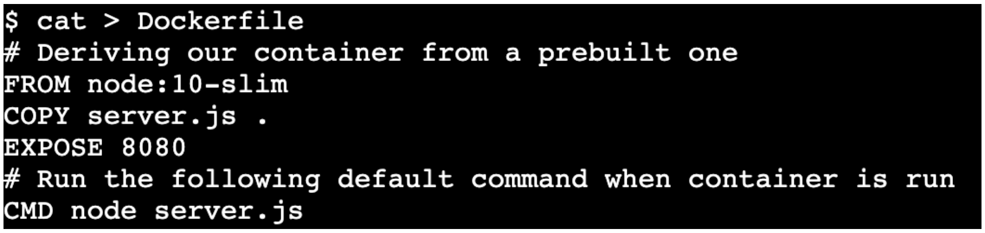
```
$ cat > server.js
var http = require('http');
var handleRequest = function(request, response) {
  response.writeHead(200);
  response.end("Hello World!");
}
var www = http.createServer(handleRequest);
www.listen(8080);
```

Step 4. Create a Dockerfile for your container. This Dockerfile outlines the steps to create the container, including copying in the file you just created. Start by running the `cat` command again, this time naming the file *`Dockerfile`*.

```
$ cat > Dockerfile
```

Then enter the contents below and press **Ctrl+D** when finished.

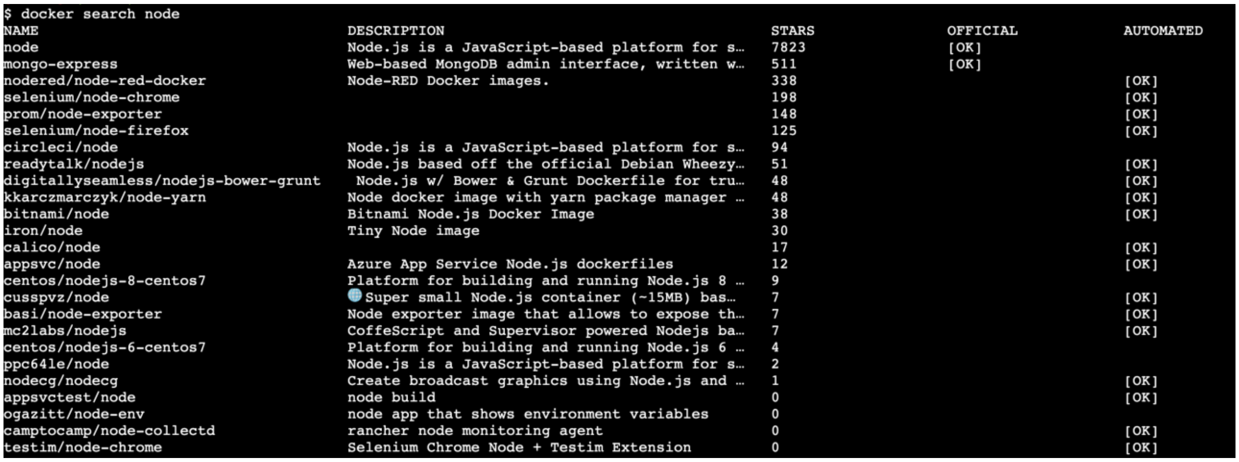
```
# Deriving our container from a prebuilt one
FROM node:10-slim
COPY server.js .
EXPOSE 8080
# Run the following default command when container is run
CMD node server.js
```



```
$ cat > Dockerfile
# Deriving our container from a prebuilt one
FROM node:10-slim
COPY server.js .
EXPOSE 8080
# Run the following default command when container is run
CMD node server.js
```

Step 5. Notice that in this Dockerfile, the container is being derived from a prebuilt one: `node:10-slim`. To search for all Docker Hub images that contain the word *node*, run the command below. The one you are using in this example is likely the most popular result.

```
$ docker search node
```

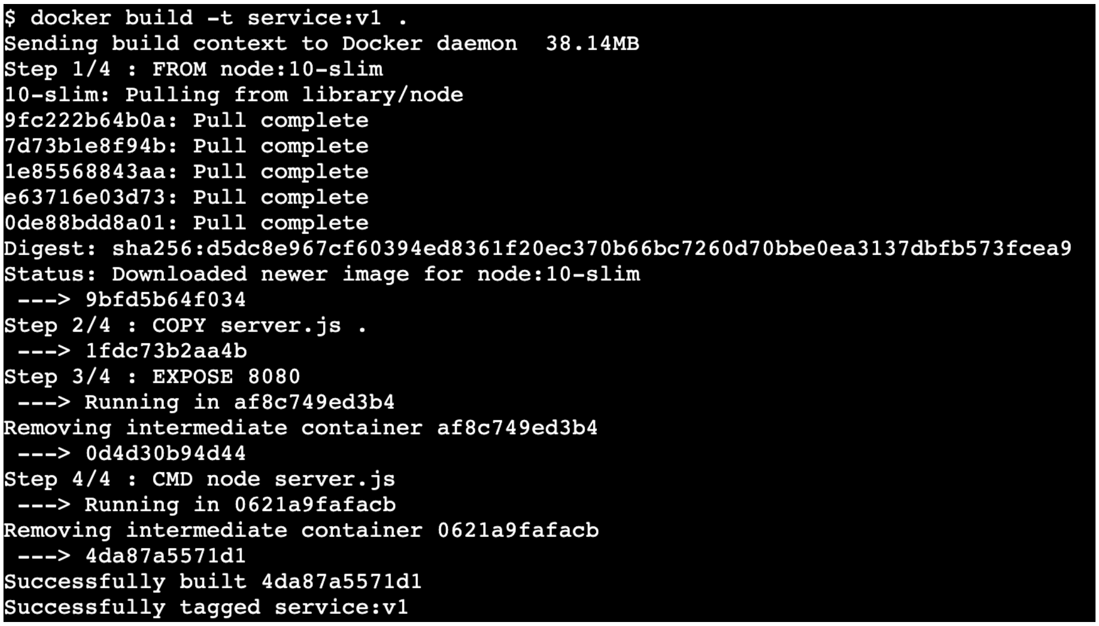


```
$ docker search node
NAME                                   DESCRIPTION                                     STARS               OFFICIAL            AUTOMATED
node                                   Node.js is a JavaScript-based platform for s…   7823                [OK]
mongo-express                          Web-based MongoDB admin interface, written w…   511                 [OK]
nodered/node-red-docker                Node-RED Docker images.                         338                                     [OK]
selenium/node-chrome                                                                   198                                     [OK]
prom/node-exporter                                                                     148                                     [OK]
selenium/node-firefox                                                                  125                                     [OK]
circleci/node                          Node.js is a JavaScript-based platform for s…   94
readytalk/nodejs                       Node.js based off the official Debian Wheezy…   51                                      [OK]
digitallyseamless/nodejs-bower-grunt    Node.js w/ Bower & Grunt Dockerfile for tru…   48                                      [OK]
kkarczmarczyk/node-yarn                Node docker image with yarn package manager …   48                                      [OK]
bitnami/node                           Bitnami Node.js Docker Image                    38                                      [OK]
iron/node                              Tiny Node image                                 30
calico/node                                                                            17                                      [OK]
appsvc/node                            Azure App Service Node.js dockerfiles           12                                      [OK]
centos/nodejs-8-centos7                Platform for building and running Node.js 8 …   9
cusspvz/node                           🌐Super small Node.js container (~15MB) bas…     7                                       [OK]
basi/node-exporter                     Node exporter image that allows to expose th…   7                                       [OK]
mc2labs/nodejs                         CoffeScript and Supervisor powered Nodejs ba…   7                                       [OK]
centos/nodejs-6-centos7                Platform for building and running Node.js 6 …   4
ppc64le/node                           Node.js is a JavaScript-based platform for s…   2
nodecg/nodecg                          Create broadcast graphics using Node.js and …   1                                       [OK]
appsvctest/node                        node build                                      0                                       [OK]
ogazitt/node-env                       node app that shows environment variables       0
camptocamp/node-collectd               rancher node monitoring agent                   0                                       [OK]
testim/node-chrome                     Selenium Chrome Node + Testim Extension         0                                       [OK]
```

2.2.2 — Building the Container

Step 1. Now that you have created a Dockerfile, you can use the `docker build` command to build an image according to the steps the file specifies. Run the command below from the same directory you have been working in. The `-t` option tags the new image, in this case with the tag *`v1`*. **Note:** The period at the end of the command is **required**.

```
$ docker build -t service:v1 .
```

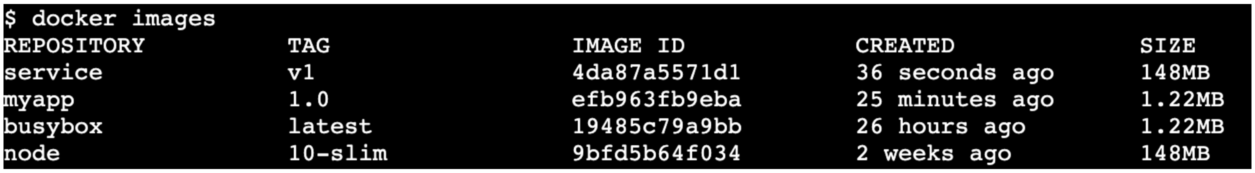


```
$ docker build -t service:v1 .
Sending build context to Docker daemon  38.14MB
Step 1/4 : FROM node:10-slim
10-slim: Pulling from library/node
9fc222b64b0a: Pull complete
7d73b1e8f94b: Pull complete
1e85568843aa: Pull complete
e63716e03d73: Pull complete
0de88bdd8a01: Pull complete
Digest: sha256:d5dc8e967cf60394ed8361f20ec370b66bc7260d70bbe0ea3137dbfb573fcea9
Status: Downloaded newer image for node:10-slim
 ---> 9bfd5b64f034
Step 2/4 : COPY server.js .
 ---> 1fdc73b2aa4b
Step 3/4 : EXPOSE 8080
 ---> Running in af8c749ed3b4
Removing intermediate container af8c749ed3b4
 ---> 0d4d30b94d44
Step 4/4 : CMD node server.js
 ---> Running in 0621a9fafacb
Removing intermediate container 0621a9fafacb
 ---> 4da87a5571d1
Successfully built 4da87a5571d1
Successfully tagged service:v1
```

When you review the output of this `build` command, you can see the steps outlined in your Dockerfile actually being executed here.

Step 2. By reviewing your images again, you will see that your new container image has been created.

```
$ docker images
```



```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
service             v1                  4da87a5571d1        36 seconds ago      148MB
myapp               1.0                 efb963fb9eba        25 minutes ago      1.22MB
busybox             latest              19485c79a9bb        26 hours ago        1.22MB
node                10-slim             9bfd5b64f034        2 weeks ago         148MB
```

Step 3. Using the `docker history` command, you can look at the history of a given image. Run the command below to look at the history of the new `service:v1` image you have created.

```
$ docker history service:v1
```

2.2.3 — Remove Testing Containers

Step 1. Take a look at your existing running containers with the `docker ps` command.

```
$ docker ps
```

Step 2. Now look at your list of all containers, both running and stopped.

```
$ docker ps -a
```

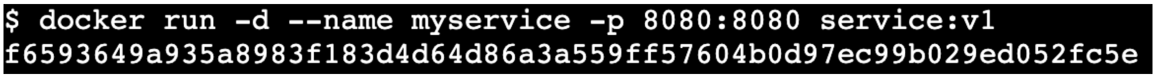
Step 3. Stop and remove all of the containers you have created to this point, so that the previously assigned ports are made available again. You can do this in batches by using the two commands below — these commands stop and remove the set of containers returned by the `ps -aq` command in parentheses. **Note:** If you already had containers on your machine before this lab, you should **carefully** remove the containers individually using their IDs.

```
$ docker stop $(docker ps -aq)

$ docker rm $(docker ps -aq)
```

2.2.4 — Run Your Container

Step 1. With your previous containers removed, you can now run your new container with the command below.

```
$ docker run -d --name myservice -p 8080:8080 service:v1
```

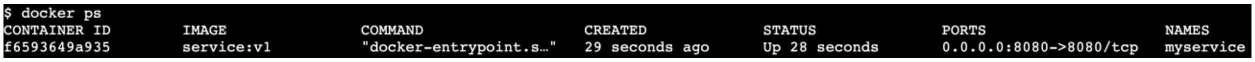


```
$ docker run -d --name myservice -p 8080:8080 service:v1
f6593649a935a8983f183d4d64d86a3a559ff57604b0d97ec99b029ed052fc5e
```

This command runs the container from the `service:v1` image in detached mode, names it *`myservice`*, and exposes the container's port 8080 on your host machine as port 8080. The written output is again your full container ID.

Step 2. Run `docker ps` again to see that your new container is running.

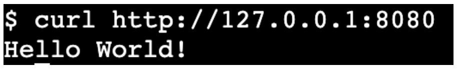
```
$ docker ps
```



```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                    NAMES
f6593649a935        service:v1          "docker-entrypoint.s…"   29 seconds ago      Up 28 seconds       0.0.0.0:8080->8080/tcp   myservice
```

Step 3. By running the `curl` command below, you can verify that this container is, indeed, running the application that you created in your `server.js` file.

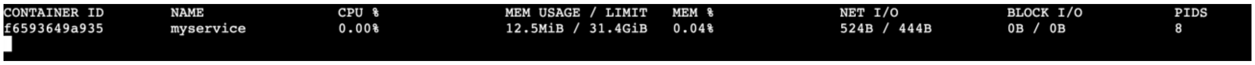
```
$ curl http://127.0.0.1:8080
```



```
$ curl http://127.0.0.1:8080
Hello World!
```

Step 4. You also have the ability to inspect the resource usage statistics associated with your container. To do this, use the `docker stats` command.

```
$ docker stats myservice
```

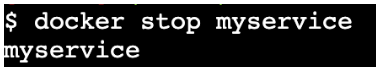


```
CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT     MEM %               NET I/O             BLOCK I/O           PIDS
f6593649a935        myservice           0.00%               12.5MiB / 31.4GiB     0.04%               524B / 444B         0B / 0B             8
```

Step 5. Press **Ctrl+C** to stop these statistics from writing to the console. These statistics provide a basic overview of the system resources that your container is using. As you can see, this particular container is very lightweight and has a rather small impact on the system.

Step 6. Stop your container using the `docker stop` command. This time, since you provided a name to your container, you can use its name instead of its ID.

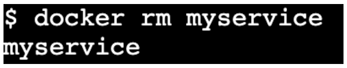
```
$ docker stop myservice
```



```
$ docker stop myservice
myservice
```

Step 7. Remove your container with the `docker rm` command.

```
$ docker rm myservice
```



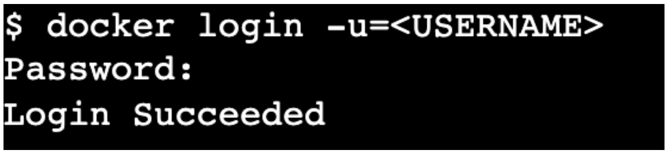
```
$ docker rm myservice
myservice
```

2.2.5 — Push Your Container to Docker Hub

Step 1. Now you can push your container to your personal repository on Docker Hub. If you did not already create a Docker Hub account, you can do so at https://hub.docker.com/.

Step 2. Log in to your Docker account with the command below, replacing `<USERNAME>` with your username. You will be prompted to enter your password, as well.

```
$ docker login -u=<USERNAME>
```



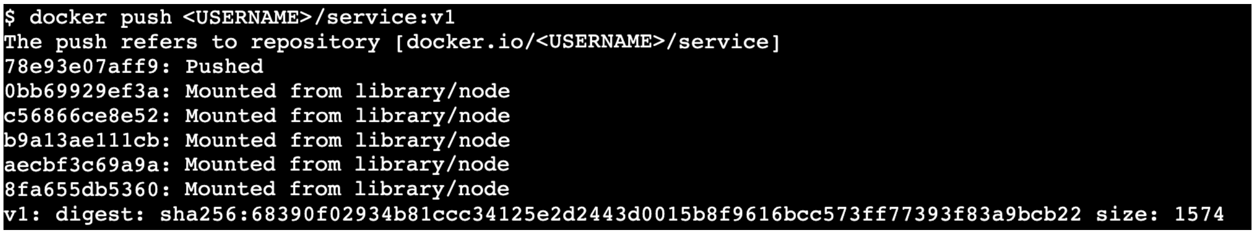
```
$ docker login -u=<USERNAME>
Password:
Login Succeeded
```

Step 3. Tag your image using your username as an identifier. In the command below, again replace `<USERNAME>` with your username.

```
$ docker tag service:v1  <USERNAME>/service:v1
```

Step 4. With your image tagged, you can push it to your repository.

```
$ docker push <USERNAME>/service:v1
```



```
$ docker push <USERNAME>/service:v1
The push refers to repository [docker.io/<USERNAME>/service]
78e93e07aff9: Pushed
0bb69929ef3a: Mounted from library/node
c56866ce8e52: Mounted from library/node
b9a13ae111cb: Mounted from library/node
aecbf3c69a9a: Mounted from library/node
8fa655db5360: Mounted from library/node
v1: digest: sha256:68390f02934b81ccc34125e2d2443d0015b8f9616bcc573ff77393f83a9bcb22 size: 1574
```

Step 5. You can see your image, pushed to your own docker repository, by replacing your username in the following link: https://hub.docker.com/u/*USERNAME*

Exercise 3 — Persisting Data with Bind Mounts

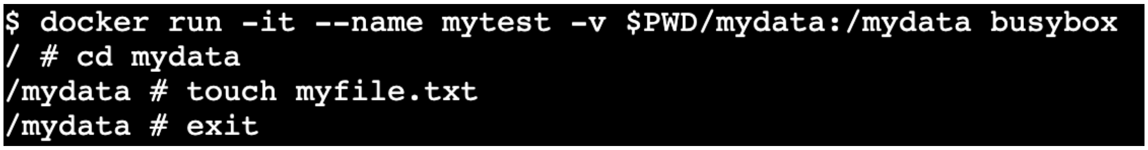
3.1 — Start a Container with a Bind Mount

Step 1. Run the following command to start a BusyBox container with a bind mount. This command specifies the directory of the bind mount on the host file system and mounts that directory into the `mydata` directory of the container file system.

```
$ docker run -it --name mytest -v $PWD/mydata:/mydata busybox
```

Step 2. At this point, you are in a bash shell at the container level. Run the following three commands to enter your `mydata` directory, add a file, and then exit the bash shell.

```
# cd mydata

# touch myfile.txt

# exit
```

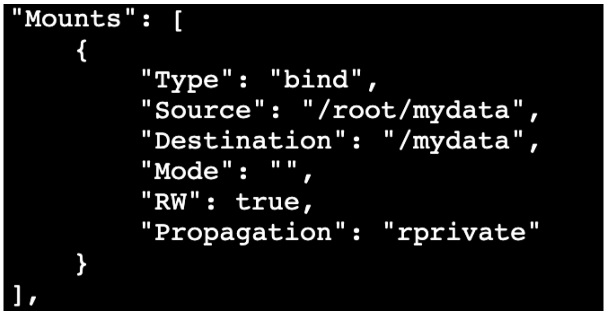


```
$ docker run -it --name mytest -v $PWD/mydata:/mydata busybox
/ # cd mydata
/mydata # touch myfile.txt
/mydata # exit
```

Step 3. Use the `docker inspect` command to verify that the bind mount was created correctly in your `mytest` container.

```
$ docker inspect mytest
```

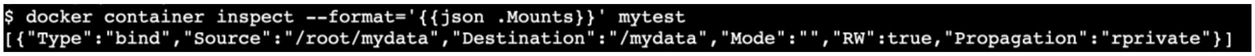
Observe the `Mounts` section of the output.



```
"Mounts": [
    {
        "Type": "bind",
        "Source": "/root/mydata",
        "Destination": "/mydata",
        "Mode": "",
        "RW": true,
        "Propagation": "rprivate"
    }
],
```

Step 4. You can filter the JSON results of the `inspect` command by using the `--format` flag. Run the command below to return filtered results.

```
$ docker container inspect --format='{{json .Mounts}}' mytest
```

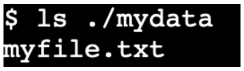


```
$ docker container inspect --format='{{json .Mounts}}' mytest
[{"Type":"bind","Source":"/root/mydata","Destination":"/mydata","Mode":"","RW":true,"Propagation":"rprivate"}]
```

Step 5. Remove all of your containers. **Note:** If you already had containers on your machine before this lab, you should **carefully** remove the containers individually using their IDs.

```
$ docker container rm -f $(docker container ls -aq)
```

Step 6. Verify your `myfile.txt` file still exists in the `mydata` folder on the host machine.



```
$ ls ./mydata
myfile.txt
```

Exercise 4 — Persisting Data with Volumes

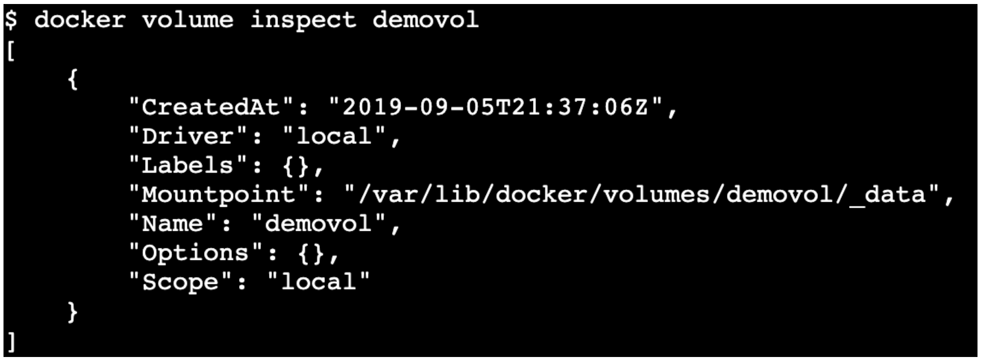
4.1 — Creating a New Volume

Step 1. Create a new volume in Docker called *demovol*.

```
$ docker volume create demovol
```

Step 2. Inspect this volume with the `volume inspect` command.

```
$ docker volume inspect demovol
```



```
$ docker volume inspect demovol
[
    {
        "CreatedAt": "2019-09-05T21:37:06Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/demovol/_data",
        "Name": "demovol",
        "Options": {},
        "Scope": "local"
    }
]
```

By default, named volumes are created under this path: *`/var/lib/docker/volumes/<name>/_data`*.

4.2 — Running a Container with a Mounted Volume

Now that you have created a volume, you can run a BusyBox container that mounts your `demovol` volume, starting the container with a mapping to this external volume.

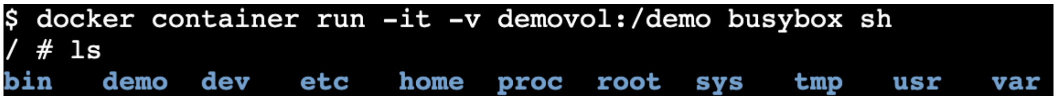
Note that this process is extremely useful when using Windows workstations/laptops. Mounting host volumes can be very problematic on Windows, so running containers with external named volumes is especially useful.

Step 1. Enter this command to run a container with the `demovol` volume mounted, and then open a shell on that container.

```
$ docker container run -it -v demovol:/demo busybox sh
```

Step 2. Run the `ls` command to list your file system.

```
# ls
```



```
$ docker container run -it -v demovol:/demo busybox sh
/ # ls
bin    demo   dev    etc    home   proc   root   sys    tmp    usr    var
```

Step 3. With the command below — which reads the `mountinfo` file and searches it for lines that contain *`demo`* — you can see relevant information about your volume.

```
# cat /proc/self/mountinfo | grep demo
```

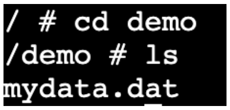
4.3 — Adding a File to Your Volume

Step 1. Add a file to your `demovol` volume. Use the command below to create and store `mydata.dat` inside the `demo` folder.

```
# echo 'my data' > /demo/mydata.dat
```

Step 2. By setting your current directory to the `demo` folder and then running `ls` again, you will see that your `mydata.dat` file has been created.

```
# cd demo
# ls
```



```
/ # cd demo
/demo # ls
mydata.dat
```

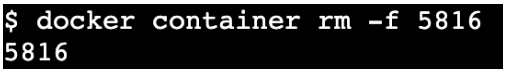
Step 3. Exit the bash shell of this container.

```
# exit
```

4.4 — Showing Data Persistence on a New Container

Step 1. Obtain the ID of the container you created in **Step 4.2**, and then delete this container.

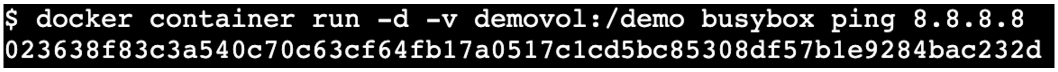
```
$ docker container rm -f <CONTAINER ID>
```



```
$ docker container rm -f 5816
5816
```

Step 2. Start a new container, once again using `demovol` as the mounted volume. This demonstrates the ability to have a persistent data source that can be used across multiple containers.

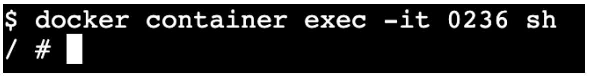
```
$ docker container run -d -v demovol:/demo busybox ping 8.8.8.8
```



```
$ docker container run -d -v demovol:/demo busybox ping 8.8.8.8
023638f83c3a540c70c63cf64fb17a0517c1cd5bc85308df57b1e9284bac232d
```

Step 3. Obtain the new ID of this container (by using `docker ps` or `docker container ls`), and then use the ID to open a shell on the container.

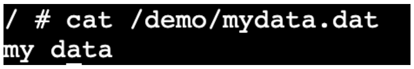
```
# docker container exec -it <CONTAINTER ID> sh
```



```
$ docker container exec -it 0236 sh
/ #
```

Step 4. Run the `cat` command below to read and print the contents of your `mydata.dat` file. Notice that the file, now from within this new container, is still accessible as it was before.

```
# cat /demo/mydata.dat
```



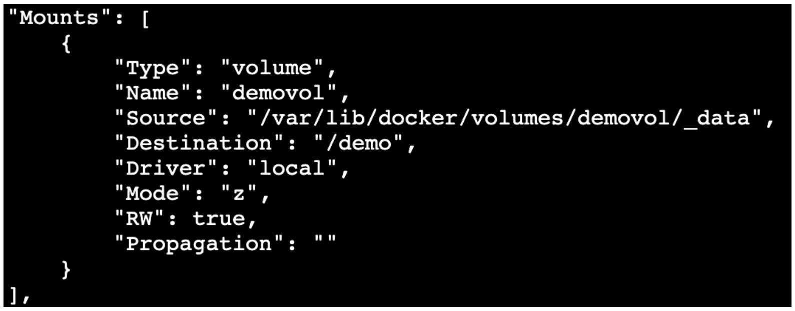
```
/ # cat /demo/mydata.dat
my data
```

4.5 — Inspect the Container's Mount Metadata

Step 1. Exit your container again by typing `exit`. Once you are back at your host machine level in the terminal, inspect this container with the inspect command.

```
$ docker container inspect <CONTAINER ID>
```

The results of this command are lengthy, but notice that the `"Mounts"` area contains information about your volume, which is this container mounted when initially run.



```
"Mounts": [
    {
        "Type": "volume",
        "Name": "demovol",
        "Source": "/var/lib/docker/volumes/demovol/_data",
        "Destination": "/demo",
        "Driver": "local",
        "Mode": "z",
        "RW": true,
        "Propagation": ""
    }
],
```

By going through this process with mounted volumes, there are two key points to understand:

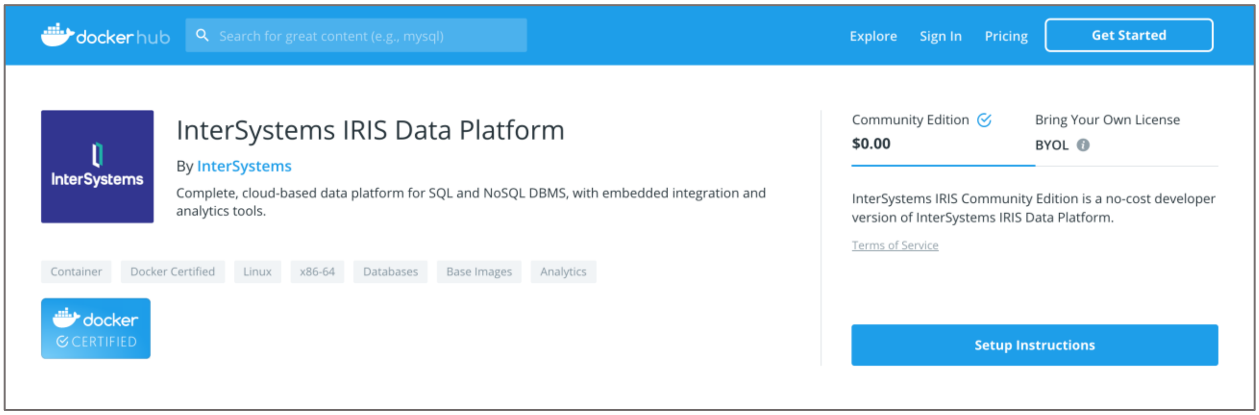
1. Volumes exist outside of the layered file system of a container. This means that they are not included in the usual copy-on-write procedure when manipulating files in the writable container layer.
2. You can manipulate files on the host machine and have those changes seamlessly propagate into a running container via a mounted volume. This is a popular technique for developers who containerize their runtime environment, but mount their in-development code. This way, you can edit your code using your host machine, and propagate those changes into running containers without rebuilding or restarting machines.

Exercise 5 — Running an InterSystems IRIS Docker Container

5.1 — Pull an InterSystems IRIS Community Edition Image

InterSystems is a certified Docker partner, and the Community Edition of InterSystems IRIS is available in the Docker Store: https://hub.docker.com/_/intersystems-iris-data-platform

Step 1. Click **Setup Instructions** on the Community Edition listing. Here, you could follow steps to set up an InterSystems IRIS image. For now, you can follow the steps in this bootcamp.



InterSystems IRIS Data Platform

By InterSystems

Complete, cloud-based data platform for SQL and NoSQL DBMS, with embedded integration and analytics tools.

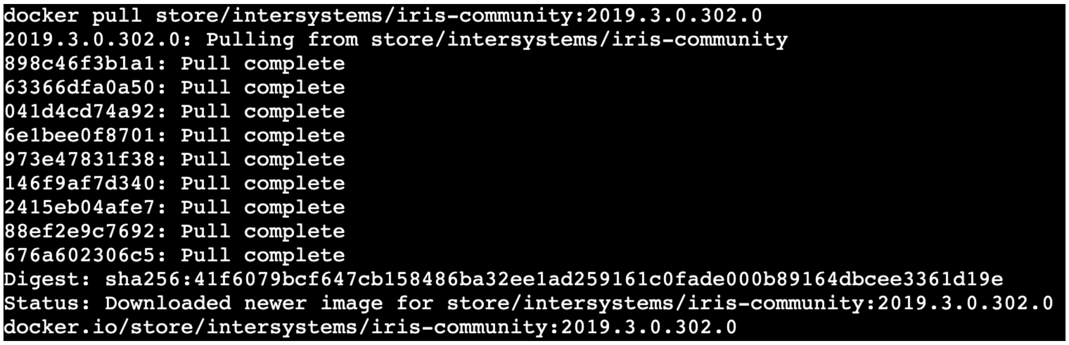
Community Edition $0.00

Bring Your Own License BYOL

InterSystems IRIS Community Edition is a no-cost developer version of InterSystems IRIS Data Platform.

Step 2. Run the command below in your terminal to pull the specified InterSystems IRIS image.

```
$ docker pull store/intersystems/iris-community:2019.3.0.302.0
```



```
docker pull store/intersystems/iris-community:2019.3.0.302.0
2019.3.0.302.0: Pulling from store/intersystems/iris-community
898c46f3b1a1: Pull complete
63366dfa0a50: Pull complete
041d4cd74a92: Pull complete
6e1bee0f8701: Pull complete
973e47831f38: Pull complete
146f9af7d340: Pull complete
2415eb04afe7: Pull complete
88ef2e9c7692: Pull complete
676a602306c5: Pull complete
Digest: sha256:41f6079bcf647cb158486ba32ee1ad259161c0fade000b89164dbcee3361d19e
Status: Downloaded newer image for store/intersystems/iris-community:2019.3.0.302.0
docker.io/store/intersystems/iris-community:2019.3.0.302.0
```

Pulling this image may take a few minutes, depending on network speeds. Once it completes, you will see a success message like the one shown here.

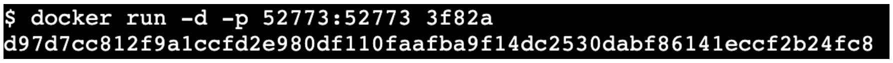
Step 3. To view your list of images and see that this image now exists on your machine, run the `docker images` command.

```
$ docker images
```

5.2 — Run an InterSystems IRIS Container

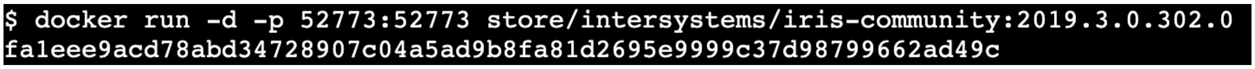
Step 1. You can now run a container using this InterSystems IRIS image. Note that you can specify the image in the command below with either the image's ID, or with the repository and the tag.

```
$ docker run -d -p 52773:52773 <IMAGE ID>
```



```
$ docker run -d -p 52773:52773 3f82a
d97d7cc812f9a1ccfd2e980df110faafba9f14dc2530dabf86141eccf2b24fc8
```

or



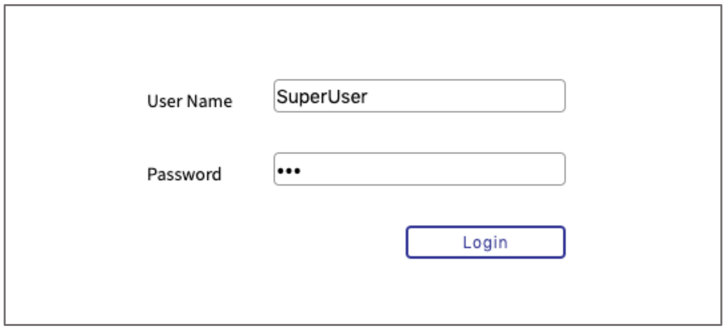
```
$ docker run -d -p 52773:52773 store/intersystems/iris-community:2019.3.0.302.0
fa1eee9acd78abd34728907c04a5ad9b8fa81d2695e9999c37d98799662ad49c
```

Step 2. By running `docker container ls`, you can see that your new container is running.

```
$ docker container ls
```

5.3 — Access InterSystems IRIS via the Management Portal

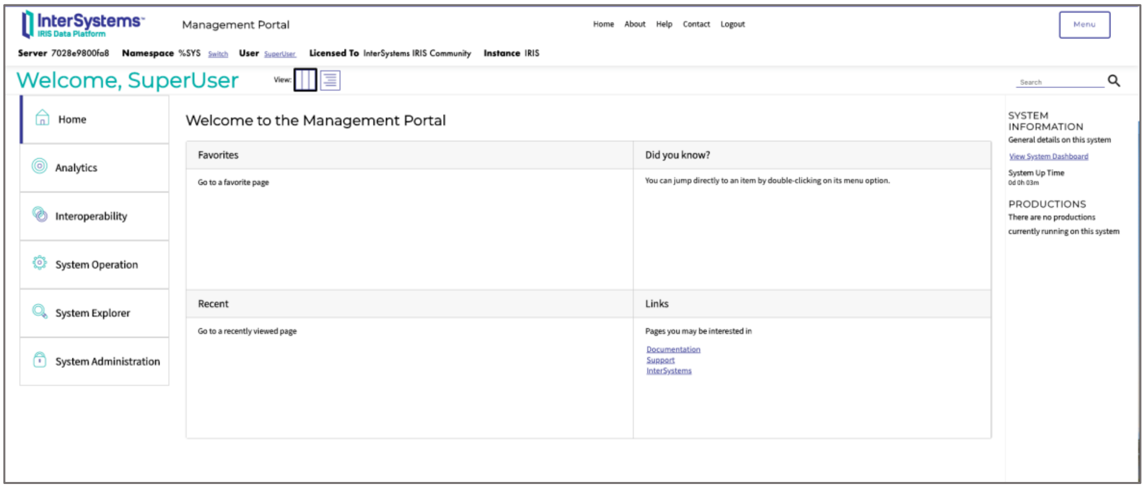
Step 1. Navigate directly to the InterSystems IRIS Management Portal by going to http://localhost:52773/csp/sys/%25CSP.Portal.Home.zen. You can log in with username *SuperUser* and password *SYS.* **Note:** If using *Play With Docker,* click the *52773* port on the interface and append */csp/sys/%25CSP.Portal.Home.zen* to the end of the URL.



User Name: SuperUser

Password: •••

Step 2. You will be prompted to change your password upon your first login. Once you do, you will be brought to the home page of the Management Portal. You are now using an instance of InterSystems IRIS Community Edition running in a Docker container.



Server 7028e9800fa8 Namespace %SYS Switch User SuperUser Licensed To InterSystems IRIS Community Instance IRIS

Welcome, SuperUser

Welcome to the Management Portal

Favorites

Go to a favorite page

Did you know?

You can jump directly to an item by double-clicking on its menu option.

Recent

Go to a recently viewed page

Links

Pages you may be interested in

SYSTEM INFORMATION

General details on this system

System Up Time
0d 0h 03m

PRODUCTIONS

There are no productions currently running on this system

5.4 — Access InterSystems IRIS via the Terminal

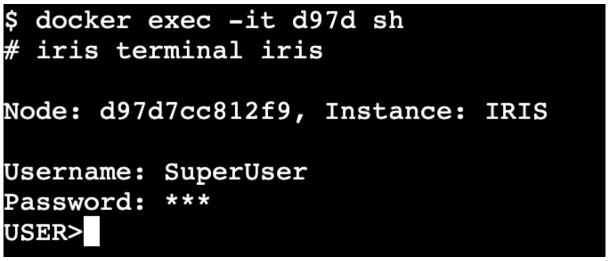
In addition to accessing InterSystems IRIS via the Management Portal, you can use your terminal to go directly into an InterSystems IRIS Terminal session.

Step 1. Obtain the ID for your InterSystems IRIS container, then enter the command below to open a shell within the container.

```
$ docker exec -it <CONTAINER ID> sh
```

Step 2. From within the container-level shell, run the command below to start an InterSystems IRIS Terminal session.

```
# iris terminal iris
```



```
$ docker exec -it d97d sh
# iris terminal iris

Node: d97d7cc812f9, Instance: IRIS

Username: SuperUser
Password: ***
USER>
```

You may be prompted for a username and password (remember that you reset this password in the previous step), and once entered, you will be in an InterSystems IRIS Terminal session. From this command line interface, you can interact with code and data in InterSystems IRIS.

Step 3. Stop and remove this container.

```
$ docker stop <CONTAINER ID>
$ docker rm <CONTAINER ID>
```

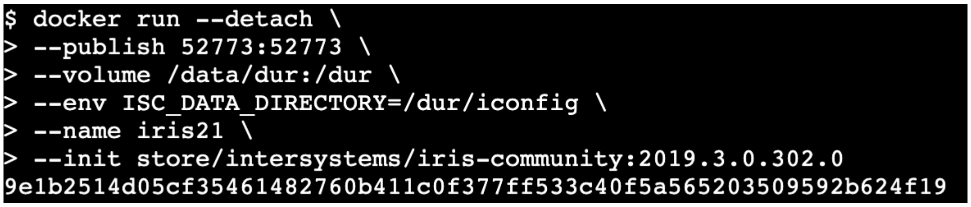
Exercise 6 — Using Durable %SYS for Data Persistence

In this exercise, you will create and run an InterSystems IRIS container that uses durable `%SYS`. This feature ensures that you can persist instance-specific data for containerized instances of InterSystems IRIS on durable storage.

6.1 — Run a Container with Durable %SYS Using Minimal Parameters

Step 1. Run the container with the command below. This command uses minimal parameters and is a basic use case for durable `%SYS`.

```
$ docker run --detach \
--publish 52773:52773 \
--volume /data/dur:/dur \
--env ISC_DATA_DIRECTORY=/dur/iconfig \
--name iris21 \
--init store/intersystems/iris-community:2019.3.0.302.0
```



```
$ docker run --detach \
> --publish 52773:52773 \
> --volume /data/dur:/dur \
> --env ISC_DATA_DIRECTORY=/dur/iconfig \
> --name iris21 \
> --init store/intersystems/iris-community:2019.3.0.302.0
9e1b2514d05cf35461482760b411c0f377ff533c40f5a565203509592b624f19
```

This command performs all of the following:

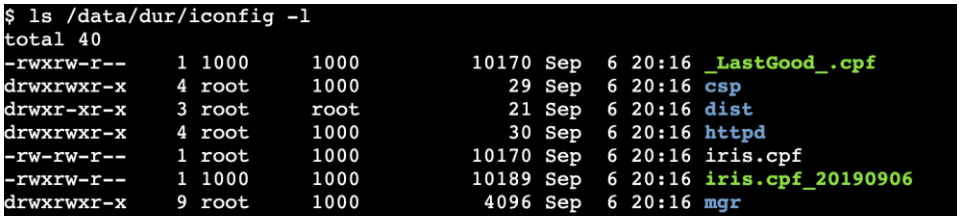
- Runs a container in detached mode.
- Exposes the container's port 52773 on the host's port 52773. (**Note:** If your local port 52773 is already in use, you will need to use a different port that is available and enter it before the colon.)
- Mounts the host machine's `/data/dur` directory to the `/dur` directory on the container's file system.
- Sets the InterSystems-specific environment variable `ISC_Data_Directory` (this specifies the durable `%SYS` directory, which is `/data/dur/iconfig` outside of the container and `/dur/iconfig` inside the container).
- Names the container *`iris21`* and runs it from the InterSystems IRIS Community Edition image.

Step 2. Check the status of your container with the `docker ps` command.

```
$ docker ps
```

Step 3. List the contents of the host machine's durable storage location.

```
$ ls /data/dur/iconfig -l
```



```
$ ls /data/dur/iconfig -l
total 40
-rwxrw-r--    1 1000      1000           10170 Sep  6 20:16 _LastGood_.cpf
drwxrwxr-x    4 root      1000              29 Sep  6 20:16 csp
drwxr-xr-x    3 root      root              21 Sep  6 20:16 dist
drwxrwxr-x    4 root      1000              30 Sep  6 20:16 httpd
-rw-rw-r--    1 root      1000           10170 Sep  6 20:16 iris.cpf
-rwxrw-r--    1 1000      1000           10189 Sep  6 20:16 iris.cpf_20190906
drwxrwxr-x    9 root      1000            4096 Sep  6 20:16 mgr
```

Step 4. Stop the container and remove it. You will need the container's ID from **Step 2** on the previous page.

```
$ docker container stop <CONTAINER ID>
$ docker container rm <CONTAINER ID>
```

Troubleshooting Note

If you had trouble with **6.1**, follow the steps below to ensure that Docker can see and access your durable %SYS directory:

Step 1. Create a new directory on your machine. In this example, the directory will be /data/dur, but your directory can be anything you choose.

```
$ mkdir data
$ mkdir data/dur
```

Step 2 (Mac). Share your durable directory with Docker. Under **Docker** > **Preferences**, choose **File Sharing**. Add your directory here, and then restart Docker.

Step 2 (PC). Make sure your local drive containing the durable directory is made available to your Docker containers in the Shared Drives area of your Docker settings. Then restart Docker.

6.2— Run a Container with Durable %SYS Using Additional Options

Step 1. Create a password file that your new container will use upon startup. This tells the iris-main program running in the background to use a specific password instead of prompting for an immediate password change. The command below creates this file in the /tmp directory.

```
$ echo GS2019! > /tmp/pwd.txt
```

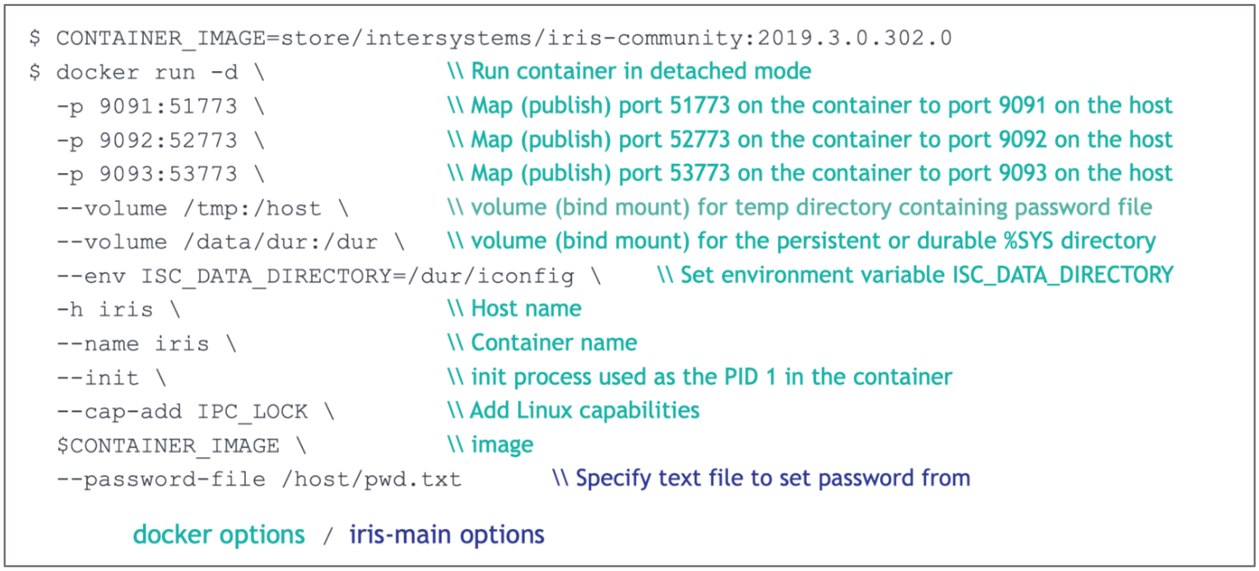
Step 2. Set the environment variable CONTAINER_IMAGE to specify what image will be used to run this container.

```
$ CONTAINER_IMAGE=store/intersystems/iris-community:2019.3.0.302.0
```

Step 3. Use the docker run command below to start an InterSystems IRIS container. Notice that the --password-file option specifies the password file to use. -iris-main will use the options that appear after the image name ($CONTAINER_IMAGE) during InterSystems IRIS startup. Options before the image name are specific to Docker.

```
$ docker run -d \
-p 9091:51773 \
-p 9092:52773 \
-p 9093:53773 \
--volume /tmp:/host \
--volume /data/dur:/dur \
--env ISC_DATA_DIRECTORY=/dur/iconfig \
-h iris \
--name iris \
--init \
--cap-add IPC_LOCK \
$CONTAINER_IMAGE \
--password-file /host/pwd.txt
```

The graphic below better outlines what each line of this command is doing.



```
$ CONTAINER_IMAGE=store/intersystems/iris-community:2019.3.0.302.0
$ docker run -d \                  \\ Run container in detached mode
  -p 9091:51773 \                  \\ Map (publish) port 51773 on the container to port 9091 on the host
  -p 9092:52773 \                  \\ Map (publish) port 52773 on the container to port 9092 on the host
  -p 9093:53773 \                  \\ Map (publish) port 53773 on the container to port 9093 on the host
  --volume /tmp:/host \            \\ volume (bind mount) for temp directory containing password file
  --volume /data/dur:/dur \        \\ volume (bind mount) for the persistent or durable %SYS directory
  --env ISC_DATA_DIRECTORY=/dur/iconfig \      \\ Set environment variable ISC_DATA_DIRECTORY
  -h iris \                        \\ Host name
  --name iris \                    \\ Container name
  --init \                         \\ init process used as the PID 1 in the container
  --cap-add IPC_LOCK \             \\ Add Linux capabilities
  $CONTAINER_IMAGE \               \\ image
  --password-file /host/pwd.txt           \\ Specify text file to set password from

          docker options / iris-main options
```

NOTE: While the Community Edition of InterSystems IRIS requires no license key, you could specify a license key for a full version of InterSystems IRIS by including a `--key` flag in the set of iris-main options in that command:

```
 --key $PWD/ISC/iris.key
```

Step 4. Check your container status again with the `docker ps` command. This time, add the `-l` flag to show only the latest container.

```
$ docker ps -l
```

Step 5. Take the ID of your container and use it in the `docker logs` command to view the information in the log for that container.

```
$ docker logs <CONTAINER ID>
```

6.3 — Access InterSystems IRIS

Step 1. Recall that in **Steps 5.3** and **5.4**, you accessed InterSystems IRIS via the Management Portal and Terminal. Do the same here, starting with the Management Portal. Use the URL below.

`http://localhost:9092/csp/sys/%25CSP.Portal.Home.zen`

Notice that instead of 52773, the local port is 9092. Recall that in the last example, you mapped your local port 52773 to the container's port 52773. In this command, you can see that it is your local port 9092 being mapped to the container's port 52773.

Step 2. Using the ID of your container, open a shell within the container.

```
$ docker exec -it <CONTAINER ID> sh
```

Step 3. Within this shell, use the `iris terminal` command to open a new InterSystems IRIS Terminal session.

```
# iris terminal iris
```

This time, you should not be prompted for a password — recall that it is *GS2019!*, based on the password file you specified.

Summary

Congratulations on completing the Containers Bootcamp! In this bootcamp, you:

- Learned the basics of using Docker containers
- Built container images
- Ran multiple containers
- Pushed your container to Docker Hub
- Persisted data across multiple containers
- Built and ran an InterSystems IRIS container
- Used the durable `%SYS` feature of InterSystems IRIS to persist instance-specific data

Next Steps

The materials from this bootcamp, as well as additional materials you may find interesting, are available in the GitHub repository linked below.

- https://github.com/intersystems/Samples-Containers-Bootcamp

Further Resources

Recommended Global Summit Sessions

- **The Value of Developing with Containers** (Joe Carroll)
    - Monday 1:30 – Fairfield/Exeter
    - Tuesday 2:30 – Fairfield/Exeter
- **InterSystems IRIS Containers for Developers** (Sean Klingensmith)
    - Monday 2:30 – Fairfield/Exeter
    - Tuesday 3:30 – Fairfield/Exeter
- **Durable Data Storage with Containers** (Mark Bolinsky)
    - Monday 3:30 – Fairfield/Exeter
    - Tuesday 4:30 – Fairfield/Exeter
- **Building Data-Driven Web Apps** (Sergei Shutov)
    - Wednesday 11:00 – Salon A/B
- **Introduction to Kubernetes** (Luca Ravazzolo)
    - Tuesday 1:30 – Arlington
    - Wednesday 11:00 – Arlington
- **The Basics and Benefits of Cloud Deployment** (Joe Carroll)
    - Monday 2:30 – Dartmouth/Clarendon

Further Online Resources

- First Look: InterSystems Products in Docker Containers (exercise)
- How Are Containers Different From Virtual Machines? (video)
- Docker Containers and InterSystems IRIS (video playlist)
- Docker for Windows and the InterSystems IRIS Data Platform (article)
- Using Package Manager with InterSystems IRIS in Docker Container (article)
- What is a Container? (article)
- Running InterSystems Products in Containers (documentation)
- Best Practices for Writing Dockerfiles (Docker documentation)