InterConnect
2017

March 19–23
MGM Grand &
Mandalay Bay
Las Vegas, NV

IBM

Hands-on Lab

Session 6381

Using DataPower in a DevOps Toolchain

Harley Stenzel, IBM

Krithika Prakash, IBM

This document is current as of the initial date of publication and may be changed by IBM at any time.

Table of Contents

Introduction

Modern application development is fast and getting faster. Manual processes are necessarily being eliminated in favor of automation. Automation itself is moving one-off tasks maintained by a person or an organization to automation regimes that are hosted elsewhere. How can I use DataPower in this world of DevOps, Continuous Integration, and Continuous Deployment?

This lab will help you answer that question. In this lab, you will use DevOps, CI, and CD methodology on an existing DataPower application. We'll use Docker, Docker Compose, Git, and Jenkins – a few of the favorite tools of the DevOps world – to demonstrate the general techniques of bringing DevOps to DataPower. Don't worry if the tools you use with your organization are different than the ones we use in this lab. The techniques trump the technology.

You will start with an existing DataPower configuration; turn it into a Docker image; and save the configuration as files human-readable "source" files that we will place under version control. From there, we'll fully containerize the application by putting both the test client and the other servers into containers and the containers into a composed application with Docker Compose.

Once we have the composed application, we'll set up two different ways we can work with it. The first way will be as a developer would – making small changes to configuration and source and testing them live. The second way we want to run our composed application is as a standalone test.

While the developer may use the standalone test as his unit test prior to check-in, he has additional needs as well. For instance, he needs to save modify DataPower configuration. When he modifies DataPower configuration and saves it, he needs it to keep it because he may eventually check it in to version control.

The standalone test serves multiple roles as well. In addition to being run prior to check-in by the developer, it is also used by the continuous integration regime to validate the application on every check-in.

Then we will add continuous integration. The application is prepared, so we will use Jenkins to perform CI for our Docker Compose application.

From there, we can iterate quickly. We make a change, test a change check in, automatically validate the results, and repeat. With this, the continuous integration process is established. Now you can decide how the application should change.

But what of continuous deployment? Perhaps your organization uses physical appliances for production and staging. You're probably wondering "How does it help me if I'm using IBM DataPower Gateway (IDG) for Docker for development and physical appliances in production?" The artifacts that I have – configuration in source control – aren't the artifacts that I need for deployment into staging and production. Is there a new manual step that is required? If so, isn't that "development?"

While it is true that we don't yet have the required artifacts, it is also true that we can get them programmatically whenever the test succeeds. To do that, you can extend the composed application to gather the deployment artifacts that can then be promoted and used in the existing deployment automation. You can build on CI success to build a continuous delivery pipeline.

With the course of our shared journey set, it's time to take our first steps.

Preparation

For this lab, we will use the virtual machine labeled `6381DataPowerDevOps`. This image is based on Ubuntu and it has all the tools required to complete the lab.

The virtual machine is configured to auto-login as `localuser`. This user can run privileged commands with the `sudo` command, no password required.

Additionally, Docker and `docker-compose` are installed, and the official DataPower Docker image is pre-loaded. Both Chrome and Firefox are configured to open a tab on the GitHub `ibm-datapower/interconnect-labs` page. This repository contains, among other things, the DataPower application that we will use as the basis for the lab.

To get the DataPower configuration from GitHub, use the command:

```
localuser@ubuntu-base:~$ \
  git clone https://github.com/ibm-datapower/interconnect-labs.git
```

Bootstrapping a DataPower for Docker project

Now that you have `export.zip` from the lab's GitHub repository, you have all the artifacts that are required to bring the existing DataPower configuration into IDG for Docker.

Setting up the project

Before running a command, we first have to figure out the lay of the new land. We will use the IDG for Docker container to create the artifacts that we will put into version control. Because of that, it pays to think up front about how we want that version control arranged. While nothing is set in stone – there is no need to be overly concerned about getting it wrong on the first iteration – it is nevertheless less work to get it right the first time.

At the very least, we'll want a directory for use for our project – a project that will eventually include containers for DataPower, the load driver, and backend server. Because of that, let's create an appropriate directory structure. At the top level, we'll create `datapower-devops-lab`. Under that, we'll create directories for `client`, `datapower`, and `backend`.

```
localuser@ubuntu-base:~$ mkdir -p datapower-devops-lab/client \
  datapower-devops-lab/datapower \
  datapower-devops-lab/backend
```

There is no magic here, just creation of a few directories.

Running DataPower

Now that we've created the directory structure, let's get to work on `datapower`. Issue the command:

```
$ cd ~/datapower-devops-lab/datapower.
```

From within the `datapower` directory, we will use DataPower running inside Docker to create its own configuration. The configuration is contained entirely within DataPower's `config:` and `local:` directories, so we will treat as Docker volumes in `config` and `local` subdirectories of `datapower`.

Run IDG for Docker with the following command:

```
localuser@ubuntu-base:~/datapower-devops-lab/datapower$ \
  docker run -it \
   -v $PWD/config:/drouter/config \
   -v $PWD/local:/drouter/local \
   -e DATAPOWER_ACCEPT_LICENSE=true \
   -e DATAPOWER_INTERACTIVE=true \
   -e DATAPOWER_WORKER_THREADS=2 \
   -p 9090:9090 -p 80:80 -p 443:443 \
   --name datapower \
```

```
   ibmcom/datapower
```

Please also study the other `docker run` options. There are environment variables passed to control the behavior of DataPower, ports that we want Docker to map, and a name for our new container. For more details about the DataPower specific options, the Docker Hub landing page (https://hub.docker.com/r/ibmcom/datapower/) is a great resource. The `docker` options are documented https://docs.docker.com/engine/reference/commandline/docker/ .

Now you will see the DataPower default log interleaved with the color-coded default log:

```
20170221T153228.036Z [0x8040006b][system][notice] logging target(default-log): Logging started.
20170221T153228.074Z [0x804000fe][system][notice] : Container instance UUID: bc117eb3-2255-45d2-93a9-43a
efd7ba2c2, Cores: 2, vCPUs: 2, CPU model: Intel(R) Core(TM) i7-4870HQ CPU @ 2.50GHz, Memory: 7983.7MB, P
latform: docker, OS: dpos, Edition: developers-limited, Up time: 0 minutes
20170221T153228.079Z [0x8040001c][system][notice] : DataPower IDG is on-line.
20170221T153228.079Z [0x8100006f][system][notice] : Executing default startup configuration.
20170221T153228.434Z [0x8100006d][system][notice] : Executing system configuration.
20170221T153228.435Z [0x8100006b][mgmt][notice] domain(default): tid(8031): Domain operational state is
up.
f5fcfa1ac88d
Unauthorized access prohibited.
login: 20170221T153229.950Z [0x806000dd][system][notice] cert-monitor(Certificate Monitor): tid(399): En
abling Certificate Monitor to scan once every 1 days for soon to expire certificates
20170221T153236.055Z [0x8100003b][mgmt][notice] domain(default): Domain configured successfully.
```

You now have a completely unconfigured DataPower gateway running!

DataPower Initialization

DataPower is easier to work with using web management, but web management is off by default. We also don't want to enable web management while DataPower still has the default password. In this step, we log in, change the `admin` password, then enable web management.

Log in with user `admin` and password `admin`.

```
login: admin
Password: *****

Welcome to IBM DataPower Gateway console configuration.
Copyright IBM Corporation 1999-2017

Version: IDG.7.5.2.2 build 283762 on Jan 11, 2017 7:43:36 PM
Serial number: 0000001

idg#
```

Since we don't want to turn on remote administration with the default password, we'll change the password for `admin` first. To do that first enter `configure` mode then use the `user-password` command. Choose whatever you like for your new password, but be sure to remember it.

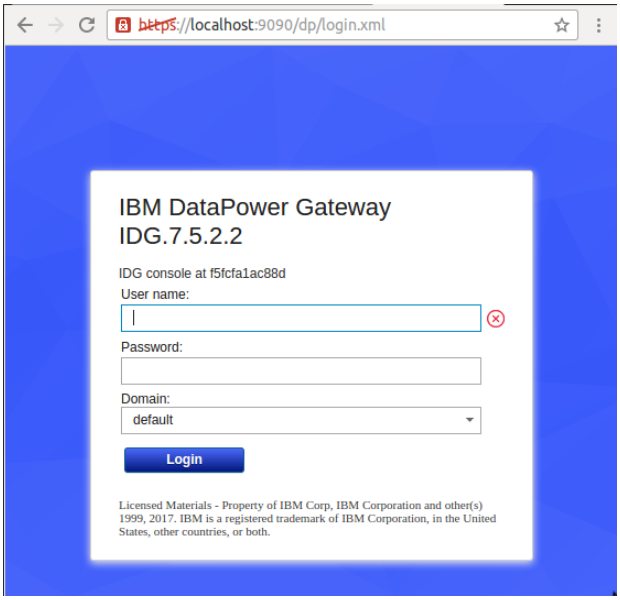
```
idg# configure
Global configuration mode
idg(config)# user-password
Enter old password: *****
Enter new password: ********
Re-enter new password: ********
Password for user 'admin' changed
Cleared RBM cache
20170221T161125.709Z [0x8100000c][mgmt][notice] : tid(111): Saved current
configuration to 'config:///auto-user.cfg'
```

Next enable web management with the commands `web-mgmt`, `admin-enabled`, `exit`.

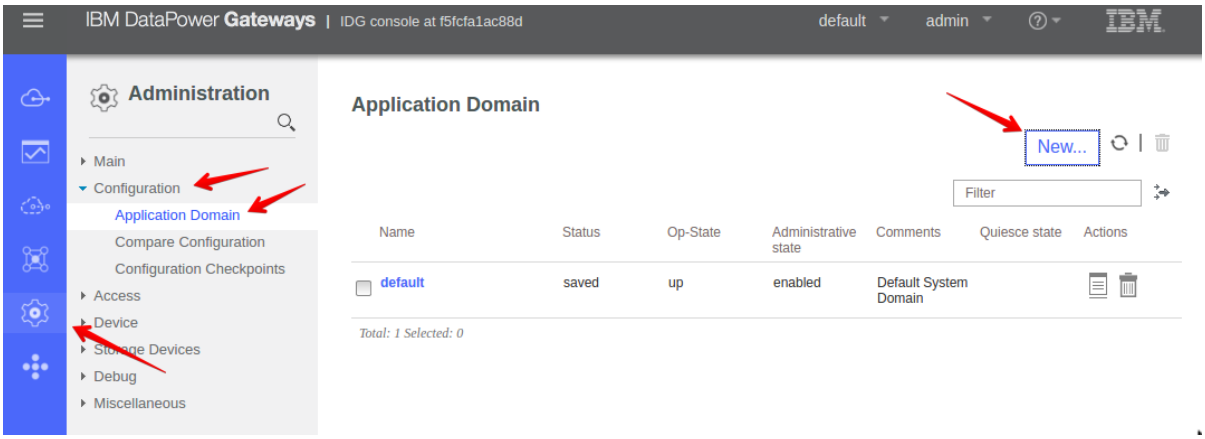
```
idg(config)# web-mgmt
Modify Web Management Service configuration

idg(config web-mgmt)# admin enabled
idg(config web-mgmt)# exit
idg(config)#
20170221T161604.565Z [0x8100003f][mgmt][notice] domain(default): tid(303):
Domain configuration has been modified.
20170221T161604.566Z [0x00350014][mgmt][notice] web-mgmt(WebGUI-Settings):
tid(303): Operational state up
```

Importing DataPower Configuration

Now that web management is enabled, point your favorite browser at https://localhost:9090 and log in as `admin` with your new password. You will have to add a security exception because presently `web-mgmt` is using self-signed SSL keys and certs.

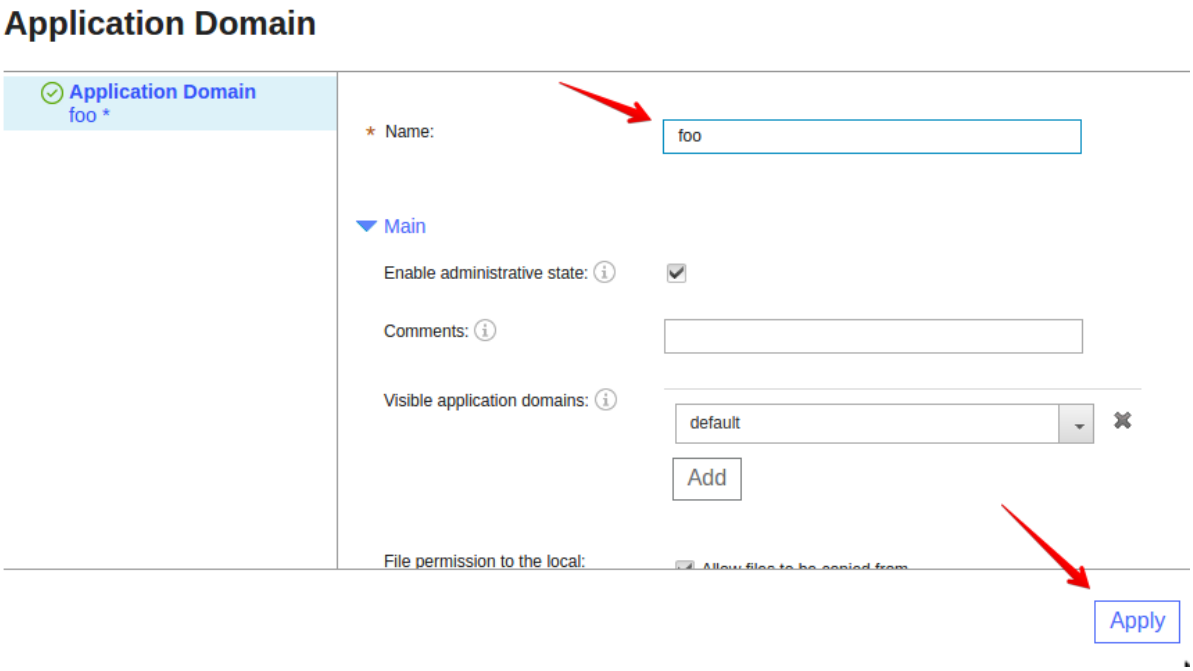


After logging in, navigate to Administration → Configuration → Application Domain → New…

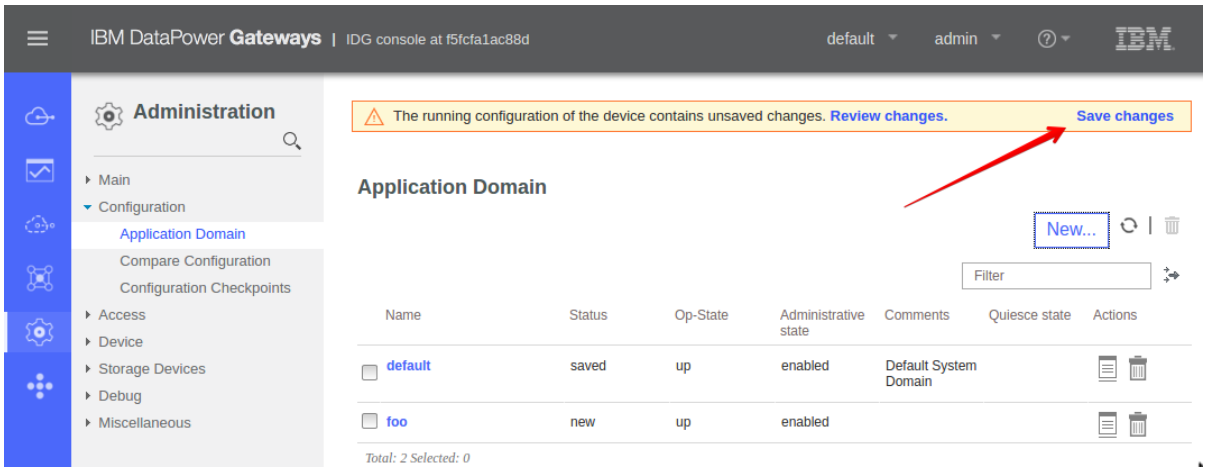


We want to create the DataPower application in its own Application Domain since this will both keep the focus entirely on the application and it will be more like the eventual deployment target. The DataPower default domain contains many configuration points that are for the platform, not for the application. By using an application domain, we keep our focus on the application.

Name the Application Domain `foo` then click `Apply`.

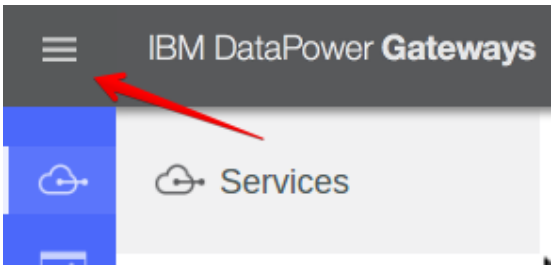


After the application domain is created, `Save changes`.

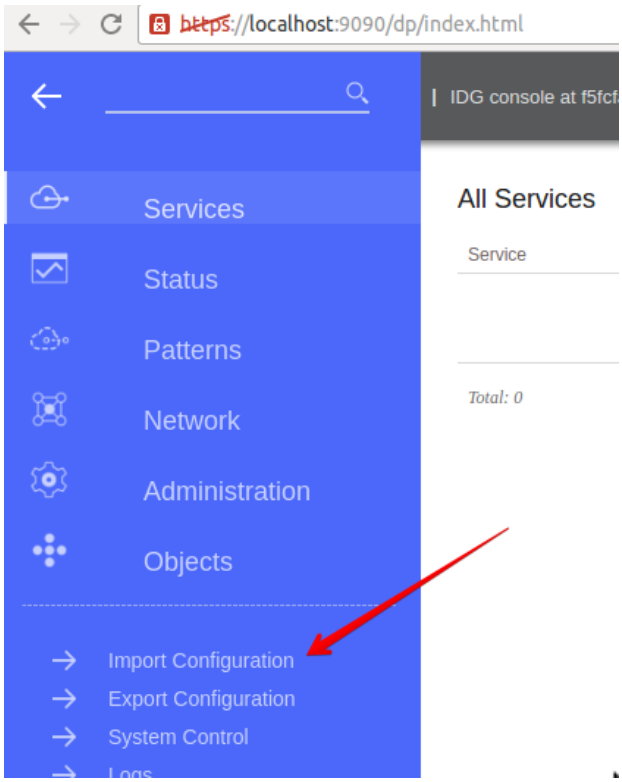


Since we want to import the configuration into the `foo` domain, we must first switch into that domain. To do that, pull down the domain drop-down where you see `default` then select `foo`.

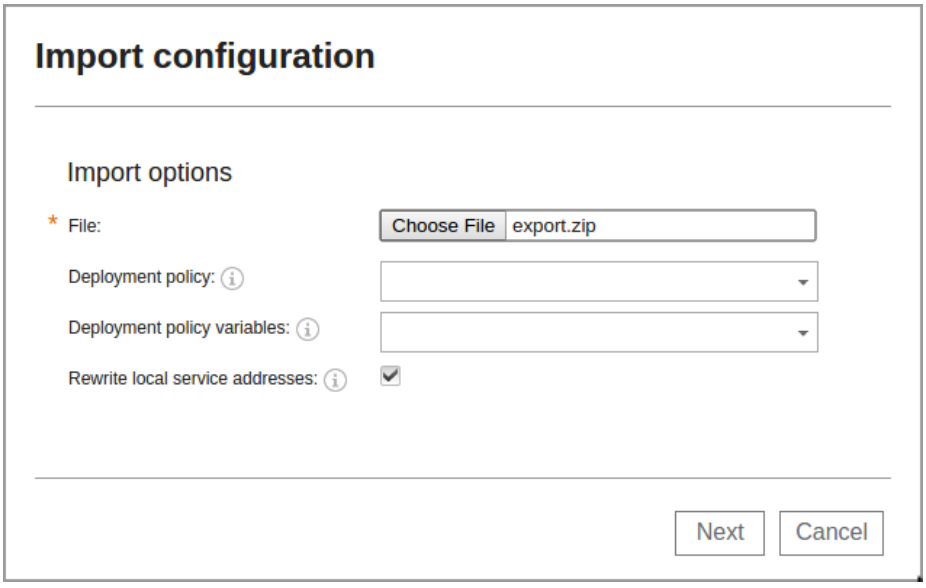
Now that we're in the `foo` domain, we can import the configuration that is in the `interconnect-labs` GitHub repository. To import the configuration, first click on the "hamburger" icon in the upper left hand corner of the management screen:



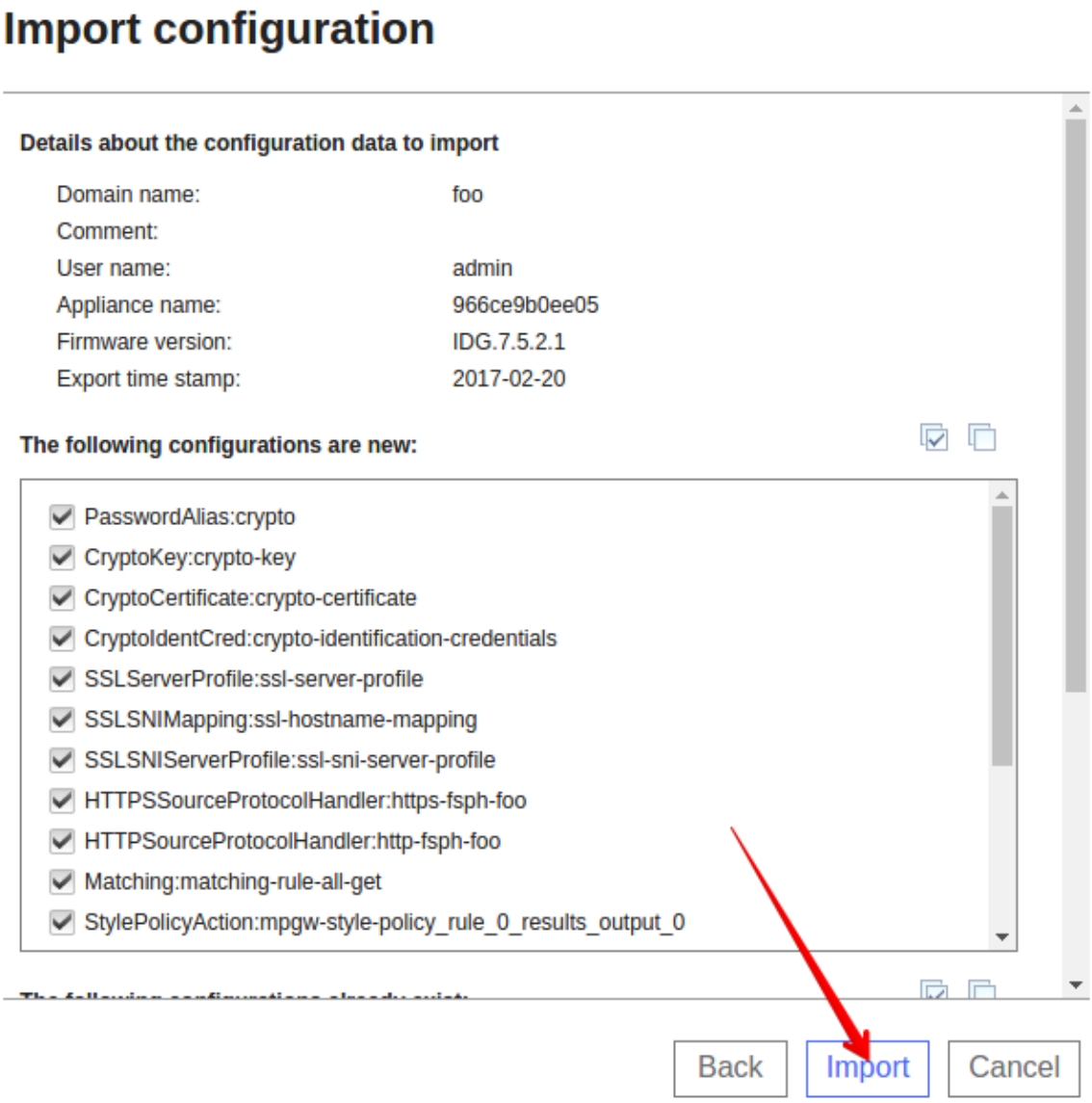
Then press Import Configuration on the resulting pane:



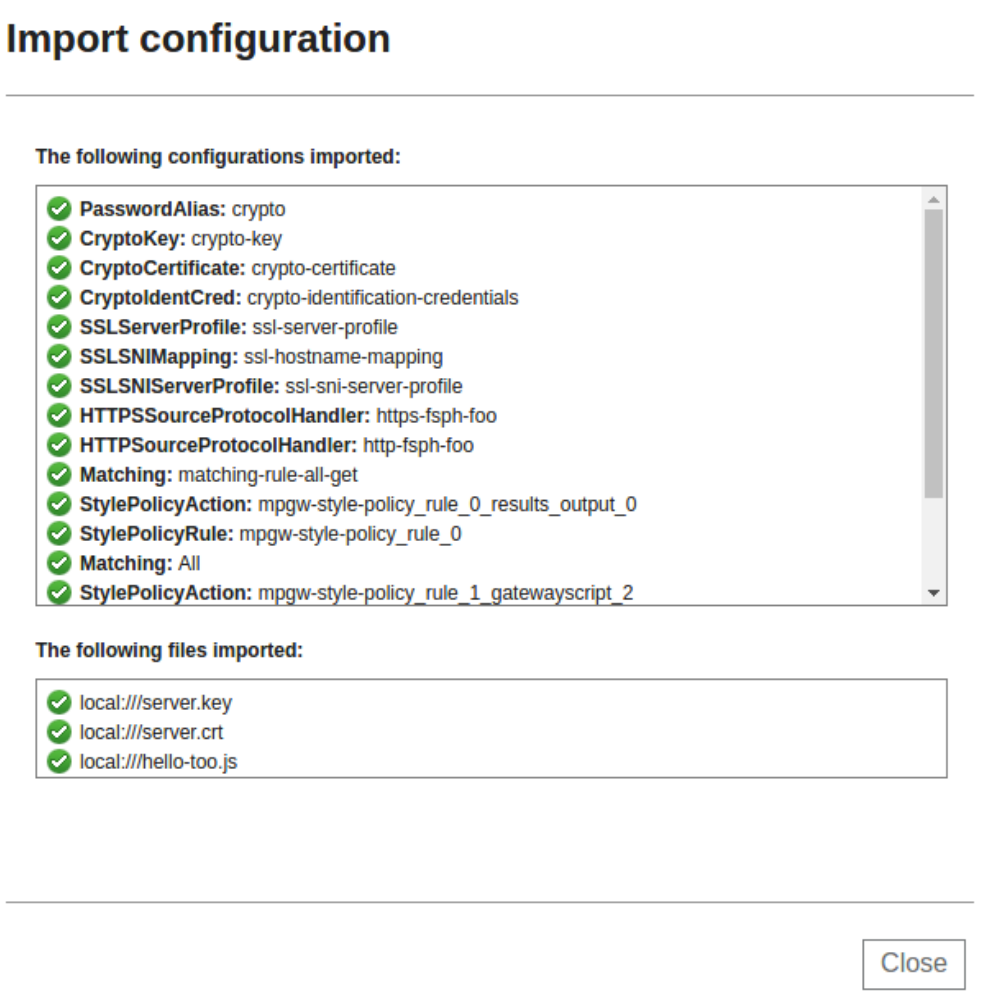
On the Import configuration screen, press the `Choose File` button and navigate to the `localuser` home directory. From there, select `interconnect-labs/IC17-6381-DataPower-DevOps/export.zip` and choose `Next`.



After you have reviewed the configuration and files that will be imported, click `Import`.



Then see that all the configuration objects and files were successfully imported and close the import screen.



Foo has been successfully imported.

Testing the Imported Application

We have the application imported, but we do not yet know if it works. Let's test it by pointing `curl` at the application. It happens to run on both port 80 and 443, and since we mapped both of those ports on our `docker run` command, the ports are available the lab VM. So back in Ubuntu, run the command:

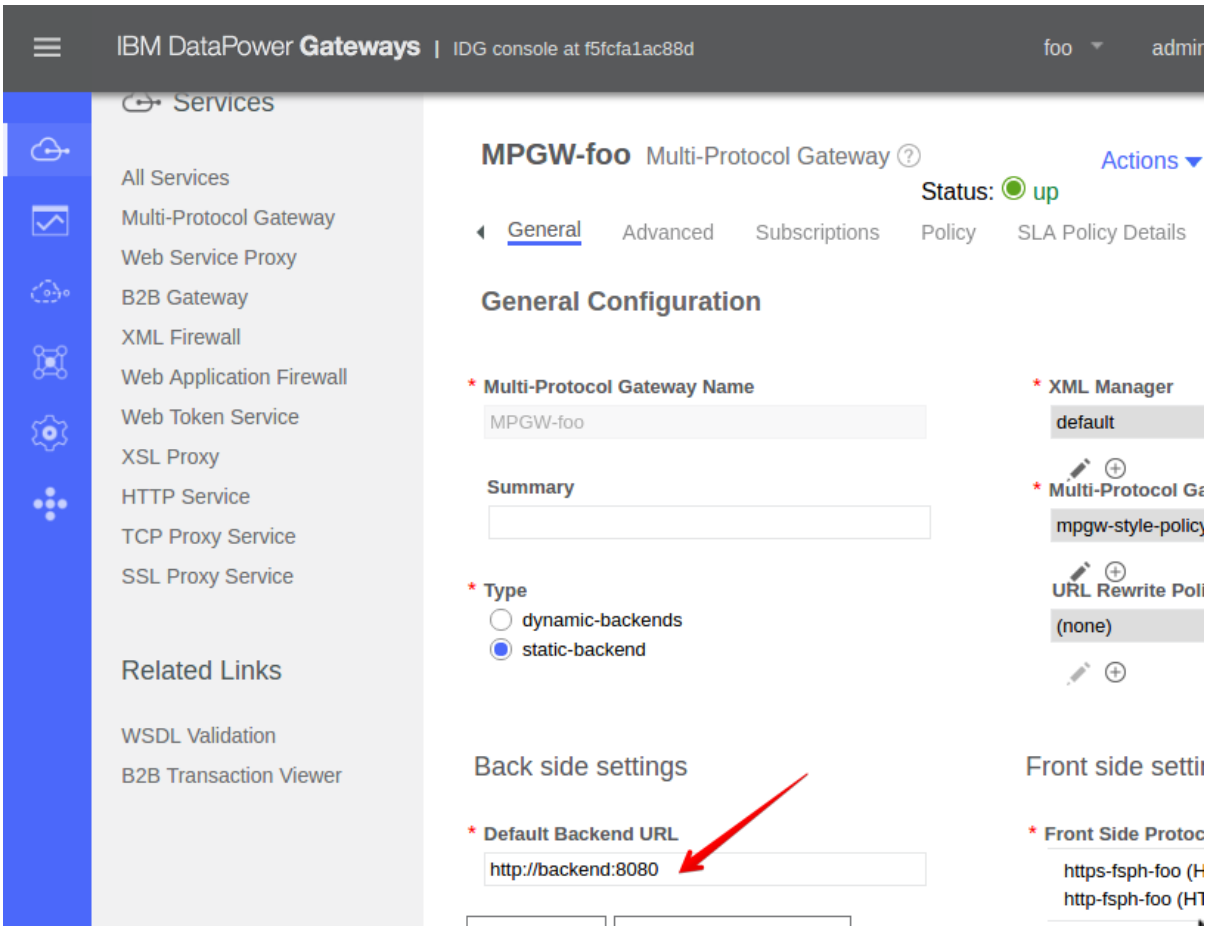
```
localuser@ubuntu-base $ curl http://127.0.0.1
```

And it didn't work:

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'>
<env:Body>
<env:Fault>
```

```
<faultcode>env:Client</faultcode>
<faultstring>Internal Error</faultstring>
</env:Fault>
</env:Body>
</env:Envelope>
```

What went wrong? Well, we did not change the backend. It had the value that was correct for the exported system, but it is not correct here. To fix it, drill into `MPGW-foo` in the web management interface and choose a better value for the `Default Backend URL`. A better value might be `http://www.ibm.com/`. After you make the change, click `Apply`.



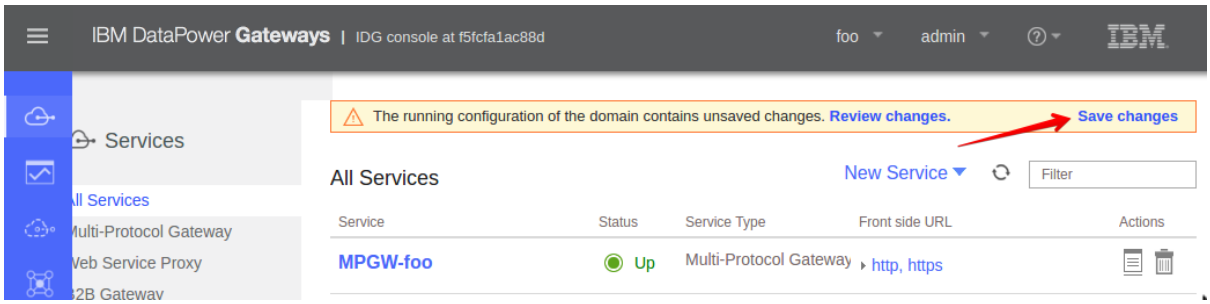
Now test again. This time we'll use a bit of a smarter test – we'll look for the breadcrumbs left by the DataPower configuration, and we'll try both http and https:

```
localuser@ubuntu-base:~/datapower-devops-lab/datapower$ \
  curl -k -s http://localhost https://localhost/ \
  | grep "DataPower Proxied"
DataPower Proxied: <!DOCTYPE html>
```

```
DataPower Proxied: <!DOCTYPE html>
```

Success!

All we must do now to have all the DataPower configuration saved to the volumes we specified at run time is `Save changes`.



We don't need this container running any more. One way we can stop it is to go to the console, `exit` from the `idg(config)#` prompt, `exit` from the `idg#` prompt, then press ctrl+c to stop the container running DataPower.

```
idg(config)# exit
idg# exit
Goodbye.
f5fcfa1ac88d
Unauthorized access prohibited.
login: ^C
localuser@ubuntu-base:~/datapower-devops-lab/datapower$
```

Inspecting the Source Artifacts

The stated goal of this process was to build a DataPower configuration suitable for use in Docker. Have we done that? How do we know?

Let us check two ways. First, we'll look at the files we have created. Second, we'll use a new DataPower container with the configuration we have just created. Let us do each in turn.

First, check the files we made:

```
localuser@ubuntu-base:~/datapower-devops-lab/datapower$ \
```

```
  find . -type f | xargs ls -ld
-rw------- 1 root root 13611 Feb 21 16:26 ./config/auto-startup.cfg
-rw------- 1 root root   260 Feb 21 16:26 ./config/auto-user.cfg
-rw------- 1 root root 15294 Feb 22 11:44 ./config/foo/foo.cfg
-rw------- 1 root root   479 Feb 21 16:56 ./local/foo/hello-too.js
-rw------- 1 root root  1801 Feb 21 16:56 ./local/foo/server.crt
-rw------- 1 root root  3311 Feb 21 16:56 ./local/foo/server.key
```

The good news is that we have the files we wanted to create. The file `auto-startup.cfg` contains the default domain configuration. The file `auto-user.cfg` contains the hash of the `admin` password. The file `foo.cfg` contains the domain `foo` configuration. In `local` we have the GatewayScript and crypto material.

In our scenario, this crypto material is only suitable for developer use. It is self-signed and not valuable; that is why the developer of the application saw it suitable to use in this `export.zip`. You can be sure that a proper regime is in place for the higher value key material! In Docker, there are many ways to manage secrets, so we won't discuss secrets management in depth in this lab. Nevertheless, it is important to think about this subject both from the DataPower and Docker points of view. This is discussed more in session 6310 where secrets management in DataPower in Docker is discussed.

The files we have are not what we want in at least one respect. They are all owned by `root`. This is a side effect of the ways the container and Linux host are managing the filesystem; such effects may not be seen for other ways of handling Docker volumes. But we still must counter the side effect. Fortunately, a single command is all it takes: `sudo chown --reference=$HOME --recursive .`. After we change the ownership, check to ensure that it worked:

```
localuser@ubuntu-base:~/datapower-devops-lab/datapower$ \
    sudo chown --reference=$HOME --recursive .
localuser@ubuntu-base:~/datapower-devops-lab/datapower$ find . -type f |
xargs ls -ld
-rw------- 1 localuser localuser 13611 Feb 21 16:26 ./config/auto-
startup.cfg
-rw------- 1 localuser localuser   260 Feb 21 16:26 ./config/auto-user.cfg
-rw------- 1 localuser localuser 15294 Feb 22 11:44 ./config/foo/foo.cfg
-rw------- 1 localuser localuser   479 Feb 21 16:56 ./local/foo/hello-
too.js
-rw------- 1 localuser localuser  1801 Feb 21 16:56 ./local/foo/server.crt
-rw------- 1 localuser localuser  3311 Feb 21 16:56 ./local/foo/server.key
-rw------- 1 localuser localuser   479 Feb 22 11:36 ./local/hello-too.js
-rw------- 1 localuser localuser  1801 Feb 22 11:36 ./local/server.crt
-rw------- 1 localuser localuser  3311 Feb 22 11:36 ./local/server.key
```

All the files are owned by `localuser` – we're all set. All that remains is a final test using a completely clean container and to put the whole project into version control!

Validation in Another Container

We started out using the `datapower` container. In fact, we still have it, although it is not running. You can tell because `docker ps` does not show the container but `docker ps -a` does. This is important because if there was anything left in the `datapower` container, we could still get it back, we'd just have to start the container again with `docker start datapower`. Hopefully there will not be a need!

To ensure that the configuration from `datapower` works in a new container, which we will call `validate`, we just need to run `ibmcom/datapower` again with the same volumes and test the application with `curl`. After it passes the test, we'll stop the container. All this can be done in just 3 steps:

First, we'll use a `docker run` command similar to our first. There are some differences. The first is that this time we will also use the `-d` flag to have the container start detached from the terminal. The second is that we will make our volumes read only so DataPower will not be able to change anything in the `config`: or `local:` directories. The `:ro` appended to each of the volumes indicates read only.

```
localuser@ubuntu-base:~/datapower-devops-lab/datapower$ \
  docker run -itd \
   -v $PWD/config:/drouter/config:ro \
   -v $PWD/local:/drouter/local:ro \
   -e DATAPOWER_ACCEPT_LICENSE=true \
   -e DATAPOWER_INTERACTIVE=true \
   -e DATAPOWER_WORKER_THREADS=2 \
   -p 9090:9090 -p 80:80 -p 443:443 \
   --name validate \
   ibmcom/datapower
6e42e0ccde23e61c8bdf53ae9f2e360e24d01cdb8a88e648642988bf93110611
```

Then we will wait just a bit for DataPower to start – it doesn't happen instantly! We'll use `curl` to test the application. As soon as a request to both the http and https URLs succeeds, we know the application is working.

```
localuser@ubuntu-base:~/datapower-devops-lab/datapower$ \
  curl -k -s http://localhost https://localhost/ \
    | grep "DataPower Proxied"
DataPower Proxied: <!DOCTYPE html>
```

```
DataPower Proxied: <!DOCTYPE html>
```

Since we see the two lines with `DataPower Proxied` we know that the test worked.

Lastly, we'll stop the `validate` container. It does not need to run any more.

```
localuser@ubuntu-base:~/datapower-devops-lab/datapower$ \
    docker stop validate
```

While we're at it, we don't need either the `datapower` or `validate` containers at this point. They have both served their purpose and can be disposed of. It's time to `docker rm` them.

```
$ docker rm datapower validate
datapower
validate
```

The code is validated, so it is time to check in to version control.

Into Version Control

First, a word about how we use version control in this lab. The lab is written using `git` examples. Later in the lab we will use a local GitLab server to host our repository, but we don't need to introduce that complexity yet. Instead, we will check in to a local repository along the way. Later you will push the local repository to a remote repository which we will integrate with Jenkins. The bottom line is that for now, we'll check in to a local repository and later we'll push to a remote location where continuous integration tests are running.

To that end, let's set up Git on the virtual machine. Substitute your own email address and name – you may want to push your lab repository to GitHub at the end of the session.

```
$ git config --global user.email "you@example.com"
$ git config --global user.name "Your Name"
$ git config --global push.default simple
```

And create a local Git repository and add our files:

```
$ cd ~/datapower-devops-lab/
localuser@ubuntu-base:~/datapower-devops-lab$ git init
Initialized empty Git repository in /home/localuser/datapower-devops-
lab/.git/
localuser@ubuntu-base:~/datapower-devops-lab$ git add .
localuser@ubuntu-base:~/datapower-devops-lab$ git status
```

```
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

      new file:   datapower/config/auto-startup.cfg
      new file:   datapower/config/auto-user.cfg
      new file:   datapower/config/foo/foo.cfg
      new file:   datapower/local/foo/hello-too.js
      new file:   datapower/local/foo/server.crt
      new file:   datapower/local/foo/server.key
```

Since the new files all look correct, time to commit:

```
localuser@ubuntu-base:~/datapower-devops-lab$ git commit -m 'First section
complete'
[master (root-commit) cef12be] First section complete
 6 files changed, 1512 insertions(+)
 create mode 100644 datapower/config/auto-startup.cfg
 create mode 100644 datapower/config/auto-user.cfg
 create mode 100644 datapower/config/foo/foo.cfg
 create mode 100644 datapower/local/foo/hello-too.js
 create mode 100644 datapower/local/foo/server.crt
 create mode 100644 datapower/local/foo/server.key
```

And check look at the git log:

```
localuser@ubuntu-base:~/datapower-devops-lab$ git log
commit cef12becd656b616c22f74efd4e4cadd2881388f
Author: Your Name <you@example.com>
Date:   Wed Feb 22 16:25:00 2017 -0500

    First section complete
```

We don’t have an origin to add yet; we’ll just use the local repository. Frequently the flow with Git would be to `git remote add origin <url>` followed by `git push –u origin master`. No need for us to do that now though!

Conclusion

The journey from `export.zip` to a dockerized, version-controlled configuration is straightforward.

1. Docker run
2. Import DataPower configuration
3. Test configuration and resolve any issues
4. Save configuration
5. “Deployment Test” – how cool is that?
6. Check in to version control

Already we can begin to see the benefits of a human readable configuration, although these benefits will become more obvious in later sections. Already we can begin to see the benefits of rapid iteration, but these benefits too will become more pronounced in later sections.

Docker Build and the DataPower Application

Up until now we have only seen one of the properties of containerized DataPower. We’ve seen that we can keep the configuration in the developer’s local file system where it can be reused by other DataPower instances. Now we want to bundle our configuration with the existing DataPower into a new image that can be distributed and run independently of the developer’s or the builder’s computer.

Creating the Dockerfile

In short, we need a `Dockerfile` and we need to `docker build`. In your `datapower` directory, create a file named Dockerfile

```
localuser@ubuntu-base:~/datapower-devops-lab/datapower$ atom Dockerfile
```

Then add the text:

```
FROM ibmcom/datapower:latest
ENV  DATAPOWER_ACCEPT_LICENSE=true \
     DATAPOWER_WORKER_THREADS=2 \
     DATAPOWER_INTERACTIVE=true
COPY config/ /drouter/config/
COPY local/ /drouter/local/
EXPOSE 80 443 9090
```

Take a moment to see how this compares to both our application – the `foo` service in DataPower – and to the `docker run` commands we’ve been using.

The `FROM` line indicates the starting point for the new image. We are staring from `ibmcom/datapower:latest` since that is what we have been running so far.

The `ENV` line allows us to build in environment variables instead of having to pass them in every time.

Next we COPY the `./config` and `./local` directories into the new image. Their destination is the documented location where DataPower expects these directories to be, and these are the same directories that we used as volumes when we were bootstrapping the project.

Lastly, we `EXPOSE` the ports that make up our application. These ports will be automatically mapped by Docker if the `-P` option is used.

Building the Customized DataPower Docker Image

Now that we have both the `Dockerfile` and our source, it's time to build our customized version of our DataPower Docker image. This customized version contains the application called Foo, so we'll tag the resulting image as `foo`.

```
localuser@ubuntu-base:~/datapower-devops-lab/datapower$ \
    docker build -t foo .
Sending build context to Docker daemon 43.01 kB
Step 1/5 : FROM ibmcom/datapower:latest
 ---> 754dcb0115d0
Step 2/5 : ENV DATAPOWER_ACCEPT_LICENSE true DATAPOWER_WORKER_THREADS 2
DATAPOWER_INTERACTIVE true
 ---> Running in b32ada243b5c
 ---> 3e573251b5b2
Removing intermediate container b32ada243b5c
Step 3/5 : COPY config/ /drouter/config/
 ---> 95af38e20ff9
Removing intermediate container 01f4d54cf466
Step 4/5 : COPY local/ /drouter/local/
 ---> 9dcab1b7d507
Removing intermediate container d306de7f654a
Step 5/5 : EXPOSE 80 443 9090
 ---> Running in e8895819c752
 ---> f8d1cf7e96d1
Removing intermediate container e8895819c752
Successfully built f8d1cf7e96d1
```

Check `docker images` and you'll see `foo` in the list.

Testing the Build Image

Now test. This will be just a bit different because now we will let Docker map the ports for us. That means the curl command will have to change to account for the NAT as we enter the Docker network space.

```
$ docker run -itdP --name foo foo
b25bb467734f011296ef6139cd92336d09e0a4031cc5b94ecc9d712840c1122a
```

At this point, the container is running but we don't know which ports have been mapped. We use the `docker port` command to find out:

```
$ docker port foo
443/tcp -> 0.0.0.0:32772
80/tcp -> 0.0.0.0:32773
9090/tcp -> 0.0.0.0:32771
```

Now that we know which ports are mapped, we can use the ports in the URLs in the curl command to test the application:

```
$ curl -k -s http://localhost:32773 https://localhost:32772 | grep
"DataPower Proxied"
DataPower Proxied: <!DOCTYPE html>
DataPower Proxied: <!DOCTYPE html>
```

Docker Challenge Round

Here are a few Docker questions, see if you can answer them. Try not to peek! Some can be answered with the content we've covered so far, others are base Docker commands. If the material is new, please try it out. How will you do?

Questions:

1. If you wanted to run a second container with a second copy of the Foo application, how would you do it?
2. How would you look at the logs of the `foo` container?
3. How would you access web management of the `foo` container? What user and password would you use?
4. If you wanted to access the DataPower CLI of container `foo`, how would you do it?
5. How can you detach from the console?
6. How can you list the Docker images present on your host?

7. How can you list **all** the Docker containers, even those not running? What about only those that are running right now?

Answers:

1. Change the name and use different ports
2. `docker logs <container>`
3. Point your browser to the port `docker port` says is mapped to 9090 with user `admin` and the password set initially. The password persists because its hash is saved in `config`.
4. `docker attach <container>`
5. With the ctl+p, ctl+q keyboard sequence.
6. With the `docker images` command.
7. With the `docker ps –a` and `docker ps` commands respectively.

Checking in to Version Control

The minimal invocation is `git add Dockerfile`, `git commit –m “Added Dockerfile”`. But it is common to check `git status` first and to see the list of changes with `git log`.

```
localuser@ubuntu-base:~/datapower-devops-lab/datapower$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

      Dockerfile

nothing added to commit but untracked files present (use "git add" to
track)
localuser@ubuntu-base:~/datapower-devops-lab/datapower$ git add Dockerfile
localuser@ubuntu-base:~/datapower-devops-lab/datapower$ git commit -m
"Added Dockerfile"
[master 115af92] Added Dockerfile
 1 file changed, 7 insertions(+)
 create mode 100644 datapower/Dockerfile
localuser@ubuntu-base:~/datapower-devops-lab/datapower$ git log
commit 115af9282cdaba7c08cbc18d4932bf8405e26ee8
Author: Your Name <you@example.com>
Date:   Thu Feb 23 17:39:57 2017 -0500

    Added Dockerfile
```

```
commit cef12becd656b616c22f74efd4e4cadd2881388f
Author: Your Name <you@example.com>
Date:   Wed Feb 22 16:25:00 2017 -0500

    First section complete
```

Cleaning Up

We have a few Docker containers hanging out, both running and stopped. We no longer need them. Let's delete them all!

```
$ docker ps –aq | xargs docker rm -f
```

Containerize the Backend

At this point, you're probably saying to yourself "I integrate with the backend; I use the backend; I don't own the backend!" And we would agree with you. Teamwork is a wonderful thing, someone else can provide that component.

In this case, the lab authors have taken care of the containerized backend for you. It is the image `hstenzel/nodejs-hostname-automatic`, and you can follow its trail from https://hub.docker.com/r/hstenzel/nodejs-hostname-automatic/ to its source in GitHub. To say it is a trivial backend is an understatement, nevertheless it is sufficient for our needs.

Let us go ahead and pull it:

```
$ docker pull hstenzel/nodejs-hostname-automatic
Using default tag: latest
latest: Pulling from hstenzel/nodejs-hostname-automatic
c60055a51d74: Already exists
755da0cdb7d2: Already exists
969d017f67e6: Already exists
37c9a9113595: Already exists
a3d9f8479786: Already exists
3d1b2c224bb0: Pull complete
b7fcb67e411b: Pull complete
Digest:
sha256:10b638f53354f00f026b5c4991efbecc0b7a58f53a2f70bfad09885e27446883
```

```
Status: Downloaded newer image for hstenzel/nodejs-hostname-
automatic:latest
```

That's it. The backend is containerized. There is just one problem – the configuration in or project does not know how to use the containerized backend. We must fix that. But first, let's prepare to run both DataPower and the backend in the same network.

A Network of our Very Own

Same network? If that is your reaction, don't worry, this is just another of the things that Docker abstracts for us. Up to now, every container we have run has been in the default Docker network. But what we want is for DataPower-foo and the backend to run all by themselves in the same dedicated network. This way, Docker will treat the names of the containers as their DNS names so the containers can talk to each other without NAT and without being exposed to other containers. This is the logical equivalent of a home network with only certain containers in the network. Docker fulfills the role of the NAT policy in the home router and the containers themselves are like the hosts that connect to the router.

So, let's create the network called `foo` in Docker:

```
$ docker network create foo
7e44ca1adeb3a46e5c99aba3eeb95a3fbfe1cbc98b71425f6f3a8cb61446fd1a
```

Add the Backend to the Network

Next, start the backend in the `foo` network:

```
$ docker run –d –p 8080:8080 --name backend \
    --network foo \
    hstenzel/nodejs-hostname-automatic
```

Now that the backend is running, see what it does by sending a curl request to it and by checking its logs:

```
$ curl http://localhost:8080
Hello world from 7e44ca1adeb3

$ docker logs backend
Example app listening at http://0.0.0.0:8080
Answered request with 7e44ca1adeb3
```

Reconfigure DataPower to use the Backend

With the backend on the private network and responding to traffic, the next step is for DataPower to use http://backend:8080 instead of http://www.ibm.com/ for the backend URL of `MPGW-foo`.

In order to use the web management to reconfigure DataPower, we'll do the same thing we did before, only this time we'll run DataPower inside our new network. Notice the addition of `--network foo` and `-d` to run detached in the background.

```
localuser@ubuntu-base:~/datapower-devops-lab/datapower$ docker run -itd \
   -v $PWD/config:/drouter/config \
   -v $PWD/local:/drouter/local \
   -e DATAPOWER_ACCEPT_LICENSE=true \
   -e DATAPOWER_INTERACTIVE=true \
   -e DATAPOWER_WORKER_THREADS=2 \
   -p 9090:9090 -p 80:80 -p 443:443 \
   --name datapower \
   --network foo \
   ibmcom/datapower
```

From here, follow these steps, which are similar to when you fixed the backend after the initial import:

1. Point your browser to https://localhost:9090
2. Accept self-signed security warnings
3. Log in as `admin` with the password you chose.
4. Switch to domain `foo`.
5. Click on `MPGW-foo`.
6. Change `Default Backend URL` to http://backend:8080. The name backend comes from `-name backend` when we started the backend container.
7. Click `Apply`.
8. Click `Save Changes`.

Next, unit test the application:

```
$ curl -k -s http://localhost https://localhost/
DataPower Proxied: Hello world from 7e44ca1adeb3
DataPower Proxied: Hello world from 7e44ca1adeb3
```

Fabulous! It works!

Next let's build the foo image. Remember that we will have to fix permissions since we saved configuration changes in DataPower:

```
localuser@ubuntu-base:~/datapower-devops-lab/datapower$ \
  sudo chown --reference=. --recursive  .
```

Then `docker build`:

```
localuser@ubuntu-base:~/datapower-devops-lab/datapower$ \
  docker build -t foo .
Sending build context to Docker daemon 43.01 kB
Step 1/5 : FROM ibmcom/datapower:latest
 ---> 754dcb0115d0
Step 2/5 : ENV DATAPOWER_ACCEPT_LICENSE true DATAPOWER_WORKER_THREADS 2
DATAPOWER_INTERACTIVE true
 ---> Using cache
 ---> 3e573251b5b2
Step 3/5 : COPY config/ /drouter/config/
 ---> 629fa69e0c34
Removing intermediate container 8e2ac8e9fdf7
Step 4/5 : COPY local/ /drouter/local/
 ---> cb97c54e0cdf
Removing intermediate container e3c570ce3914
Step 5/5 : EXPOSE 80 443 9090
 ---> Running in d31fa26ddfbe
 ---> 5e4b267a1daa
Removing intermediate container d31fa26ddfbe
Successfully built 5e4b267a1daa
```

Lastly, run the new container as before, but this time on network `foo`. Also, we can't reuse the ports for the development container, so let Docker choose them for us with `-P`. This means we must look at `docker port` to see how the ports are mapped and that we must change the URLs in `curl` to account for the different ports:

```
$ docker run -itdP --name foo --network foo foo
0200d3c56081334a03243941c4331652096680647c9ff651fac7a7b36ef86105
$ docker port foo
80/tcp -> 0.0.0.0:32780
9090/tcp -> 0.0.0.0:32778
443/tcp -> 0.0.0.0:32779
$ curl -k -s http://localhost:32780 https://localhost:32779
DataPower Proxied: Hello world from 7e44ca1adeb3
DataPower Proxied: Hello world from 7e44ca1adeb3
```

Success! Time to check into version control.

Into Version Control

Let's make sure that the changes we made are present. An easy way to do that is `git diff`. Inspect the changes. Most importantly, we see that the `backend-url` is changed:

```
localuser@ubuntu-base:~/datapower-devops-lab/datapower$ git diff
diff --git a/datapower/config/foo/foo.cfg b/datapower/config/foo/foo.cfg
index 98f512b..f7699b9 100644
--- a/datapower/config/foo/foo.cfg
+++ b/datapower/config/foo/foo.cfg
<snip for brevity>
 %endif%
@@ -529,7 +529,7 @@ mpgw "MPGW-foo"
   ssl-client-type proxy
   default-param-namespace "http://www.datapower.com/param/config"
   query-param-namespace "http://www.datapower.com/param/query"
-  backend-url "http://www.ibm.com/"
+  backend-url "http://backend:8080"
   propagate-uri
   monitor-processing-policy terminate-at-first-throttle
   request-attachments strip
```

DataPower configuration is quite readable! It is easy to see what change was made in the DataPower configuration between different versions. This is one of the critical benefits we were seeking by saving configuration as flat files instead of as exported zip files.

Since the changes are acceptable, it's time to check in:

```
localuser@ubuntu-base:~/datapower-devops-lab/datapower$ git add .
localuser@ubuntu-base:~/datapower-devops-lab/datapower$ \
  git commit -m 'changed backend-url to http://backend:8080'
```

And see that our version history remains correct:

```
localuser@ubuntu-base:~/datapower-devops-lab/datapower$ git log
commit 1349814af7833d7db0fa8298c71f375cff023f11
Author: Your Name <you@example.com>
Date:   Fri Feb 24 13:00:30 2017 -0500

    changed backend-url to http://backend:8080
```

```
commit 115af9282cdaba7c08cbc18d4932bf8405e26ee8
Author: Your Name <you@example.com>
Date:   Thu Feb 23 17:39:57 2017 -0500

    Added Dockerfile

commit cef12becd656b616c22f74efd4e4cadd2881388f
Author: Your Name <you@example.com>
Date:   Wed Feb 22 16:25:00 2017 -0500

    First section complete
```

Congratulations! DataPower and the backend are both containers and they are running in the same Docker network.

And Congratulations again! As a DataPower Developer working on Docker, you've just gone through the basic of making changes to the DataPower configuration:

1. You started DataPower using volumes
2. You used web management to make configuration changes
3. You saved the configuration, which saved it to local directories under version control
4. You unit tested the application while it was running and in development
5. You built a Docker image of the application and tested that
6. You checked the results into version control

It seems like a lot of steps, but hey, it was easy! And we will make it easier still.

Containerize the Client

Now the time has come to containerize the client. First though it is worth thinking about why the client should be containerized at all – in production, clients are connecting from all over the world, why do we want it in the container at all?

The answer is DevOps. It is Continuous Integration. Let me explain. The goal of building a composed application is that all the required pieces are tied together in a way that they can be deployed together. So far, we have that for the backend and for the Foo DataPower application. But we do not have that for `curl`. Is `curl` part of the application? Granted it is not a part that is in line with client data the way DataPower and the backend are. But it **is** test code, and test code is a deliverable.

Also, since continuous integration is our goal, how can we continuously integrate if the test is not absolutely, completely, 100% a part of the delivered application? Clearly CI is dependent on the automated test, so we need to containerize `curl`.

For the curl container to be usable as a test, it should exit and the exit code of the process should indicate the success or failure of the test. Let's get started!

First, let's get a container to start trying things out with. We're eventually going to write a `Dockerfile` to automate this, so we'll start by trying out the steps manually.

```
$ cd ~/datapower-devops-lab/client
localuser@ubuntu-base:~/datapower-devops-lab/client$ \
  docker run -it --rm --network foo --name client ubuntu
```

This will give us a bash prompt in an Ubuntu container on the `foo` network.

Now to get cURL and try it out. We left both the `datapower` and `foo` containers running in the previous step, so we have some built-in choices:

```
root@9e3554655d5e:/# apt-get update
. . .
root@9e3554655d5e:/# apt-get –y install curl
. . .
root@9e3554655d5e:/# curl -k -s http://foo https://foo
DataPower Proxied: Hello world from 7e44ca1adeb3
DataPower Proxied: Hello world from 7e44ca1adeb3
root@9e3554655d5e:/# curl -k -s http://datapower https://datapower
DataPower Proxied: Hello world from 7e44ca1adeb3
DataPower Proxied: Hello world from 7e44ca1adeb3
root@9e3554655d5e:/# curl -k -s http://backend:8080
Hello world from 7e44ca1adeb3
root@9e3554655d5e:/# exit
$
```

Excellent! Now we know what we must put into the `Dockerfile`. So first open `Dockerfile` in an editor:

```
$ atom Dockerfile
```

And add the text:

```
FROM ubuntu:latest
RUN  apt-get update && \
     apt-get –y install curl
CMD  ["curl","-k","-s",”http://datapower”,”https://datapower”]
```

This says to start from Ubuntu, install cURL, and run the specific cURL command by default.

Build the client image:

```
localuser@ubuntu-base:~/datapower-devops-lab/client$ docker build -t
client .
Sending build context to Docker daemon 3.584 kB
Step 1/3 : FROM ubuntu:latest
 ---> f49eec89601e
Step 2/3 : RUN apt-get update &&      apt-get install -y curl
 ---> Using cache
 ---> 4edd1295ea7a
Step 3/3 : CMD /usr/bin/curl -k -s http://datapower https://datapower
 ---> Using cache
 ---> 0c0cbf38822c
Successfully built 0c0cbf38822c
```

And now that it has been built, we can use it:

```
$ docker run -it --rm --network foo client && echo WooHoo
DataPower Proxied: Hello world from 7e44ca1adeb3
DataPower Proxied: Hello world from 7e44ca1adeb3
WooHoo
```

Excellent, it worked, and the exit code was success as proven by the result `WooHoo`.

But we don't yet know if the test succeeds in all circumstances. Let's try it while restarting the `datapower` container:

```
$ docker restart datapower
datapower
$ docker run -it --rm --network foo client && echo WooHoo
$ docker run -it --rm --network foo client && echo WooHoo
$ docker run -it --rm --network foo client && echo WooHoo
DataPower Proxied: Hello world from 7e44ca1adeb3
DataPower Proxied: Hello world from 7e44ca1adeb3
WooHoo
```

Well, that is the definition of a mixed result. The good news is that there are no success messages on failures. The bad news is that the client should retry enough that the test won't incorrectly fail in a startup situation.

This is a job for a script! Let's get started:

```
localuser@ubuntu-base:~/datapower-devops-lab/client$ mkdir -p bin
localuser@ubuntu-base:~/datapower-devops-lab/client$ touch bin/client.sh
localuser@ubuntu-base:~/datapower-devops-lab/client$ \
    chmod a+x bin/client.sh
localuser@ubuntu-base:~/datapower-devops-lab/client$ atom bin/client.sh
```

Then, inside the script, put:

```
#!/bin/bash

# Loop for up to 20 consecutive failures
for (( failures=0; failures<20; ))
do
  # Issue the curl command with urls from script args
  curl -k -s "$@"
  # save the exit code of curl for use in comparison and exit
  rc=$?
  if [ $rc = 0 ]
  then
    failures=0
    if [ ! "$CONTINUOUS" = "true" ]
    then
      echo SUCCESS
      break
    fi
  else
    failures=$((failures + 1))
    echo FAILURE $failures rc=$rc
  fi
  # Otherwise wait a second and try again
  sleep 5
done

exit $rc
```

And test it:

```
localuser@ubuntu-base:~/datapower-devops-lab/client$ \
  docker restart datapower
datapower
localuser@ubuntu-base:~/datapower-devops-lab/client$ \
  docker run -it --rm --network foo \
```

```
  -v $PWD/bin:/usr/local/bin client \
  client.sh http://datapower
FAILURE 1 rc=7
FAILURE 2 rc=7
FAILURE 3 rc=7
FAILURE 4 rc=7
FAILURE 5 rc=7
DataPower Proxied: Hello world from 7e44ca1adeb3
SUCCESS
```

Great, we can see that it retries then exits on first success. Now that it's up, make sure that it exits with success immediately:

```
localuser@ubuntu-base:~/datapower-devops-lab/client$ \
  docker run -it --rm --network foo \
  -v $PWD/bin:/usr/local/bin client client.sh http://datapower
DataPower Proxied: Hello world from 7e44ca1adeb3
SUCCESS
```

Just like we did with the DataPower image, now that we know it works using volumes, we build it in. So back to the `Dockerfile` for `client`, change it to look like this:

```
FROM ubuntu:latest
RUN  apt-get update && \
     apt-get –y install curl
COPY bin/ /usr/local/bin/
CMD  ["/usr/local/bin/client.sh","http://datapower","https://datapower"]
```

Note the addition of the `COPY` line; this gets the script into the image. Also note that the `CMD` line has changed – we now call `client.sh`.

Next `docker build` the `client` image:

```
localuser@ubuntu-base:~/datapower-devops-lab/client$ \
    docker build -t client .
Sending build context to Docker daemon 3.584 kB
. . .
Successfully built aac2b6886deb
```

And test the built image, both when it must wait and when it does not:

```
localuser@ubuntu-base:~/datapower-devops-lab/client$ docker run -it --rm -
-network foo client
FAILURE 1 rc=7
. . .
FAILURE 5 rc=7
DataPower Proxied: Hello world from 7e44ca1adeb3
DataPower Proxied: Hello world from 7e44ca1adeb3
SUCCESS
localuser@ubuntu-base:~/datapower-devops-lab/client$ echo $?
0
```

Next it should not have to wait:

```
localuser@ubuntu-base:~/datapower-devops-lab/client$ docker run -it --rm -
-network foo client
DataPower Proxied: Hello world from 7e44ca1adeb3
DataPower Proxied: Hello world from 7e44ca1adeb3
SUCCESS
localuser@ubuntu-base:~/datapower-devops-lab/client$ echo $?
0
```

There is a third case, one where there is no datapower to talk to, which should cause the test to fail:

```
localuser@ubuntu-base:~/datapower-devops-lab/client$ docker stop datapower
datapower
localuser@ubuntu-base:~/datapower-devops-lab/client$ docker run -it --rm -
-network foo client
FAILURE 1 rc=6
FAILURE 2 rc=6
. . .
FAILURE 20 rc=6
localuser@ubuntu-base:~/datapower-devops-lab/client$ echo $?
6
```

As expected the test failed. Also, the exit code from the Docker image is the same exit code that came from cURL, so we know that anything that causes cURL to fail will cause the test to fail as well.

Into Version Control

You know the routine. We make a little change, we test it, we check it in. Issue the following commands:

1. See what has changed with the `git status` command.

2. Add the new files with the `git add .` command.
3. Check them in with the `git commit –m 'Containerize client'` command.
4. See the changes with the `git log` command.

Cleaning Up

Since we don't need our existing containers any longer, remove them with the `docker ps -aq | xargs docker rm –f` command.

Thoughts on the Client Container

We love cURL, it's simple and easy. But it's not always the best tool. No problem! There are already containerized versions of popular tools like SoapUI and Selenium. Also, there is no rule that says you are limited to a single client container. Have a different image for each of the test harnesses you use if that fits better!

Fixing the First Bug

Let's say that another team member noticed that the client container reports success even when DataPower returns a non-200 HTTP response code. The bug states that to recreate, do the following:

1. Run the `foo` image in a container called `datapower`.
2. Ensure that the `backend` container is **not** running
3. Run the `client` image
4. Observe that the client image returns success even though the content indicates an error occurred.

Of course, you recreate the test:

```
$ docker run -itd --name datapower --network foo foo
$ docker run --rm --network foo client
```

Sure enough, the bug report is correct.

You realize immediately that you forgot the flag on cURL that that instructs cURL to return an error 22 in response to server errors. You check `man curl` and discover that the option is helpfully called `--fail` or `-f`.

A quick verification that this change fixes the problem proves that it does:

```
$ docker run  --rm --network foo client curl -f -k -s  http://datapower
```

```
$ echo $?
22
```

Since the fix is to add the missing `-f` option in `client.sh`, we make that change, test it, then check in the fix:

1. From the client directory, run the `atom bin/client.sh` command.
2. Add the missing `-f` option to `curl` and save the `client.sh` script.
3. Build the client image with the `docker build -t client .` command.
4. Test that it works correctly under failure with the `docker run --rm --network foo client ; echo $?` command.
5. Test that it works correctly under success by starting the backend with the `docker run -d --name backend --network foo hstenzel/nodejs-hostname-automatic ; echo $?` command.
6. Check in with the `git add . && git commit -m 'Add --fail option to curl to fix false success bug'` command.

This is a good time to think about other failure modes. Are we catching them all? What are some of the other interesting cases?

The Composed DataPower Application

Here's the good news: We have three images that we can deploy as containers together, and they work together effectively. The bad news is that it's a rather frustrating and tedious process to get them to work together. Wouldn't it be nice to have a way to stitch together the application so that all the containers that make up the application can be managed together?

In this section, we will use Docker Compose to stitch together our application.

We have several functional goals:

- Have an easy way to build the Foo application.
- Have an easy way to test the Foo application – something appropriate for use as a continuous integration test.
- Have an easy way for a developer to make and verify changes to Foo.

Compose can help us meet these requirements. So can other tools – for instance Kubernetes can do this for us as well. The lab uses Docker Compose because it is easier to work with, but similar principles apply to other ways of assembling containers into applications.

Composed Testing

The `docker-compose.yml` file describes a composed application in a way that Docker Compose can manage it. There are a ton of great resources available about Docker Compose and `docker-compose.yml`, but between your familiarity with the application we're composing and the straightforward description format, it's easiest to just jump right in to create a `docker-compose.yml` suitable for running and testing the Foo application.

```
$ cd ~/datapower-devops-lab
localuser@ubuntu-base:~/datapower-devops-lab$ \
    atom docker-compose.yml
```

Then add the following:

```
version: '3'
services:
  client:
    build: client
    depends_on:
      - datapower
  datapower:
    build: datapower
    depends_on:
      - backend
  backend:
    image: hstenzel/nodejs-hostname-automatic
```

What does this do?

First, notice that we have defined three different services, `client`, `datapower`, and `backend`. These are the three images that we run as containers that make up our composed application.

We also give Compose dependency knowledge. We say that `datapower` depends on `backend` and `client` depends on `datapower` because it does not make sense for this application to have a `datapower` container unless there is a `backend` container to support it.

We also told Compose how it should get the images that make up the composed application. We said that it should build the `client` and `datapower` images for us, and that it should build them out of the directories of the same name. We said that

the `backend` service is provided by an external image. Let us see what happens when we use Compose to `build` and `pull` the images:

```
localuser@ubuntu-base:~/datapower-devops-lab$ docker-compose build
localuser@ubuntu-base:~/datapower-devops-lab$ docker-compose pull
```

Already we have an easy way to meet our build requirement!

Now let us see how we'd just run the test in a continuous integration friendly way:

```
localuser@ubuntu-base:~/datapower-devops-lab$ docker-compose run client
Creating network "datapowerdevopslab_default" with the default driver
Creating datapowerdevopslab_backend_1
Creating datapowerdevopslab_datapower_1
FAILURE 1 rc=7
. . .
DataPower Proxied: Hello world from 46b3bcc14539
DataPower Proxied: Hello world from 46b3bcc14539
SUCCESS
localuser@ubuntu-base:~/datapower-devops-lab$ echo $?
0
localuser@ubuntu-base:~/datapower-devops-lab$ docker-compose down
Stopping datapowerdevopslab_datapower_1 ... done
Stopping datapowerdevopslab_backend_1 ... done
Removing datapowerdevopslab_client_run_1 ... done
Removing datapowerdevopslab_datapower_1 ... done
Removing datapowerdevopslab_backend_1 ... done
Removing network datapowerdevopslab_default
```

So, we `docker-compose run` just the application, then `docker-compose down` to clean everything up.

Pay attention though to what Compose is doing along the way. On `run`, it creates a dedicated network for this run. It runs the dependent services. Then it passes the response code back as returned by the `client` service. On down, it stops and removes all the containers and it removes the networks it created.

This satisfies the requirement to easily runnable.

Into Version Control

```
localuser@ubuntu-base:~/datapower-devops-lab$ git add docker-compose.yml
localuser@ubuntu-base:~/datapower-devops-lab$ git commit -m 'Add docker-
compose.yml for easy testing'
```

Composed Development

While the developer will use the composed testing process, that alone is not enough. There still must exist some way to access web management, some way that DataPower configuration can be saved to version control, and some way that allows easy testing of changes to the service.

In short, the composed development environment is just a bit different than the composed testing environment. One way of handling this is to use a different Compose file for development. This is what we'll do. Let us call it `docker-compose-dev.yml`.

```
localuser@ubuntu-base:~/datapower-devops-lab$ \
    cp docker-compose.yml docker-compose-dev.yml
localuser@ubuntu-base:~/datapower-devops-lab$ atom docker-compose-dev.yml
```

And make it look like this:

```
version: '3'
services:
  client:
    build: client
    depends_on:
      - datapower
    environment:
      - CONTINUOUS=true
  datapower:
    build: datapower
    depends_on:
      - backend
    volumes:
      - ./datapower/config:/drouter/config
      - ./datapower/local:/drouter/local
    ports:
      - "80:80"
      - "443:443"
      - "9090:9090"
  backend:
    image: hstenzel/nodejs-hostname-automatic
```

As you can see, it has the same services, but there are a few additions.

The `client` service is updated to get the environment variable `CONTINUOUS=true`. The DataPower gets the Compose version of the `docker run --publish <port:port> –volume <directory>` args as we used previously. Those are reflected in the `volumes` and `ports` stanzas.

Using the Composed Development Environment

Let's see it in action. Run the command:

```
localuser@ubuntu-base:~/datapower-devops-lab$ \
    docker-compose -f docker-compose-dev.yml up
```

Notice that the logs include the output from all the containers interleaved. Also notice that the test runs continuously, even on success. That is thanks to the `CONTINUOUS` environment variable.

Now let us see what happens with a trivial code change. Edit `hello-too.js`:

```
localuser@ubuntu-base:~/datapower-devops-lab$ \
    atom datapower/local/foo/hello-too.js
```

and change "DataPower Proxied" to "DataPower Foo Proxied" to indicate that it is part of application Foo.

The terminal with Compose **immediately** reflects the change!

Now log in to DataPower web management on https://localhost:9090. This too is as expected.

Validating with the Composed Test

To validate we did as we did before:

```
localuser@ubuntu-base:~/datapower-devops-lab$ \
    docker-compose -p unit-test build
localuser@ubuntu-base:~/datapower-devops-lab$ \
    docker-compose -p unit-test run client
localuser@ubuntu-base:~/datapower-devops-lab$ \
    docker-compose -p unit-test down
```

In three commands, we have built both the `datapower` and client `containers`, run the unit test completely isolated from the other containers, then cleaned everything up. This is literally as easy as 1, 2, 3!

Stop the Development Environment

The development environment is still running. Did you realize that? On this virtual machine you just ran multiple copies of DataPower simultaneously for different reasons and you barely noticed.

```
localuser@ubuntu-base:~/datapower-devops-lab$ \
    docker-compose -f docker-compose-dev.yml down
```

Into Version Control

We've been a little lax in checking in, we now have two distinct changes so we will check them in separately:

```
localuser@ubuntu-base:~/datapower-devops-lab$ \
  git add docker-compose-dev.yml
localuser@ubuntu-base:~/datapower-devops-lab$ \
  git commit -m 'Composed development environment'
```

And

```
localuser@ubuntu-base:~/datapower-devops-lab$ \
  git add .
localuser@ubuntu-base:~/datapower-devops-lab$ \
  git commit -m 'Clarify which service in DataPower proxied'
```

Continuous Integration

The Foo application is now enabled for Continuous Integration. We don't yet **have** CI for Foo, but Foo could run in any CI system that can run Docker Compose.

Since Jenkins will be driving our continuous integration, we need to create an appropriate `Jenkinsfile` that calls Docker Compose. We know what that we have to ask Jenkins to do because we can already run our tests outside Jenkins.

Add Jenkins Integration

The Jenkins integration will be checked into version control at the root of the repository, and the file is called `Jenkinsfile`. Let's create our `Jenkinsfile`:

```
localuser@ubuntu-base:~/datapower-devops-lab$ atom Jenkinsfile
```

Place the following content into the file:

```
#!groovy
node {
  stage('Prepare') {
    checkout scm
    sh 'docker ps'
    sh 'docker images'
  }
  stage('Build') {
    sh 'docker-compose build'
  }
  stage('Test') {
    try {
      sh 'docker-compose run client'
    } catch (Exception e) {
      throw e;
    } finally {
      sh 'docker-compose down --remove-orphans || true'
    }
  }
}
```

The `Prepare` stage checks the project out of source control. It also executes a few Docker commands that are helpful for you to understand the initial state. The `Build` stage builds the application. The `Test` stage is a little more complicated. It runs `client`, and that result is the result of the test in Jenkins. If `client` failed, then the test should also fail so we re-throw the exception. But success or failure, we always want to clean up with `docker-compose down`. This is enough to tell Jenkins how to test Foo!

Now check in.

```
localuser@ubuntu-base:~/datapower-devops-lab$ \
  git add .
localuser@ubuntu-base:~/datapower-devops-lab$ \
  git commit -m ‘Add Jenkinsfile'
```

Into GitLab

The tricky part is having a place to run. Fortunately, we have that covered for you. The `interconnect-labs` project has a Jenkins that can be run inside your lab VM.

The GitLab image will be our source repository. We will integrate it with Jenkins so that every time we check in to GitLab our tests are run in Jenkins.

The first step is to start both GitLab and Jenkins. For this, use a new window:

```
localuser@ubuntu-base:~$ \
  cd ~/interconnect-labs/IC17-6381-DataPower-DevOps/ci-demo
localuser@ubuntu-base:~/interconnect-labs/IC17-6381-DataPower-DevOps/ci-
demo$ docker-compose up -d
Starting cidemo_gitlab_1
Starting cidemo_jenkins_1
```

Configure GitLab

Next, create a project in GitLab:

1. Use the browser in the VM to browse to http://localhost/
2. Change the GitLab password.
3. Log in with `Username` of `root` and the password from the previous step.
4. Choose `New project`.
    a. Enter `datapower-devops-lab` as the project name.
    b. Choose `Public` for `Visibility Level`.
    c. Click `Create project`.
5. GitLab will say `The repository for this project is empty`.

The time has come to upload the local Git repository to GitLab. In `~/datapower-devops-lab`, do the following:

```
localuser@ubuntu-base:~/datapower-devops-lab$ \
  git remote add origin http://localhost/root/datapower-devops-lab.git
localuser@ubuntu-base:~/datapower-devops-lab$ \
  git push -u origin –all
Username for 'http://localhost': root
Password for 'http://root@localhost':
Counting objects: 49, done.
```

```
Delta compression using up to 8 threads.
Compressing objects: 100% (40/40), done.
Writing objects: 100% (49/49), 16.14 KiB | 0 bytes/s, done.
Total 49 (delta 10), reused 0 (delta 0)
To http://localhost/root/datapower-devops-lab.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

Notice that Git will prompt you GitLab credentials. The username is `root` and the password is the one you chose upon first accessing GitLab. If you like, you can reload the GitLab project page and see your code there.

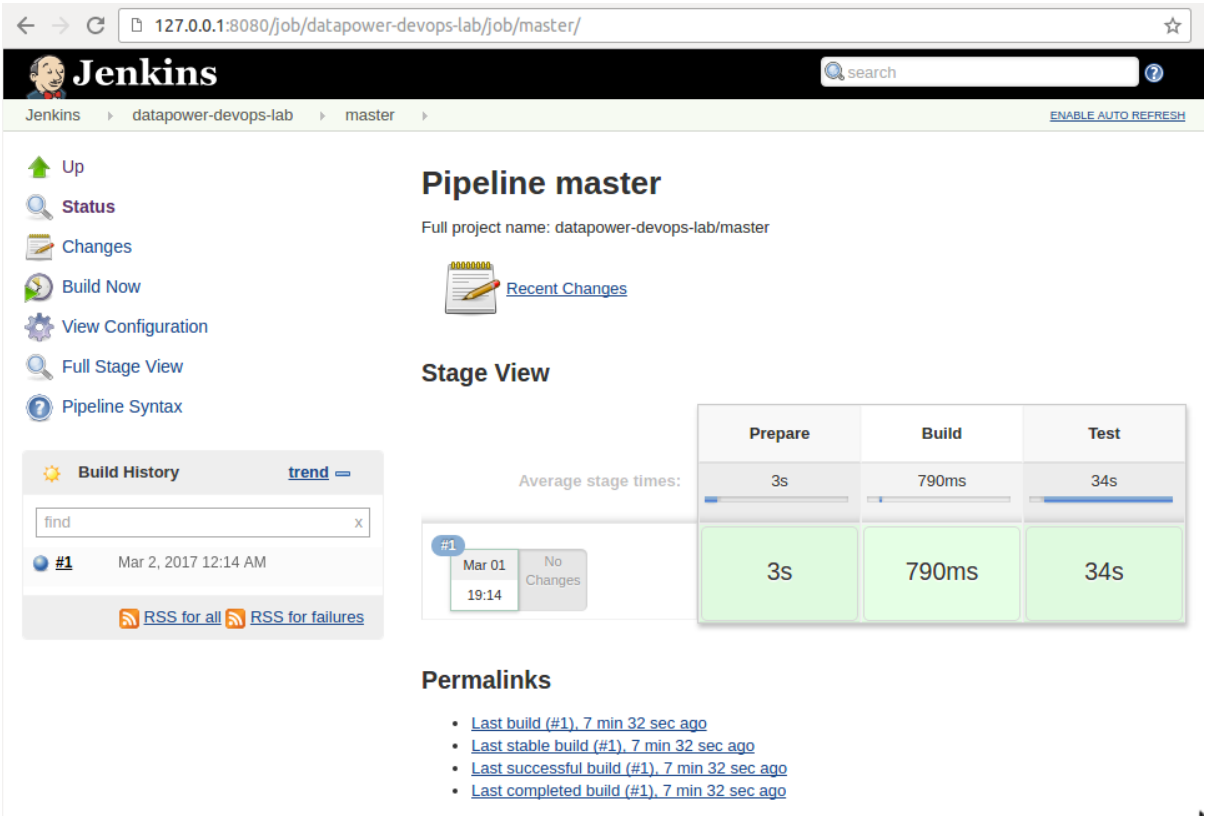
Creating the Jenkins Job

Now that the source is in a Jenkins-accessible location, we can ask Jenkins to run our project via its `Jenkinsfile`.

1. Use the browser in the VM to access Jenkins at http://127.0.0.1:8080/ .
2. Click `New Item` or `create new jobs`.
    a. For `item name`, enter `datapower-devops-lab`.
    b. Select `Multibranch Pipeline`.
    c. Click `OK`. You will then be presented with the `configure` screen.
3. Click `Add source` under `Branch Sources`
    a. Choose `Git`.
    b. Enter http://gitlab/root/datapower-devops-lab/ for `Project Repository`.
    c. Click `Save`.

Now Jenkins will “scan” GitLab looking for `Jenkinsfile` in each branch of project `datapower-devops-lab`. It will find one on branch `master` and begin to process that branch – you will see indication of the processing on the lower left hand side of the screen.

Click `Status` on the left navigation pane, then drill into `master`. You will see:



Congratulations! Your test passed!

While this is a great start, so far we have only told Jenkins about GitLab. We now need to tell our GitLab repository about Jenkins so that the tests are run automatically upon checkin.

1. Navigate to http://localhost/root/datapower-devops-lab/settings/integrations . Another way to get here is to go to the project page, choose the gear icon, then choose integrations.
2. For the URL, use `http://jenkins:8080/project/datapower-devops-lab`.
3. Click `Add Webhook`.
4. Click `Test`. If it worked, you will see a banner `Hook executed successfully: HTTP 200`. Additionally, every time `Test` is pressed, Jenkins runs its jobs – so you will see another test running in Jenkins.

That will do it – every time there is a `git push` to this repository, GitLab will tell Jenkins and Jenkins will run the CI described in `Jenkinsfile` in the Gitlab repository.

Continue Developing DataPower with CI

Now it's time to practice: Quickly iterate over the cycle:

1. Make the change.
2. Test the change.
3. Check it in.
4. Repeat.

Here are a few suggested simple (and one very ambitious) changes.

Fix the bug where port 80 is mapped for two different purposes – for both GitLab and for DataPower when run from the development Compose file.

Add timestamps to the Jenkins output with the `timestamps { … }` directive in `Jenkinsfile`.

Fix a bug where `client` will return success if only one of the URLs checked passes. Do this by looping on the URLs and separately invoking `curl` for each URL passed to `client.sh`.

Access the DataPower CLI with the `docker attach` command. Spoiler alert: it won't work because the `interactive` and `TTY` options, which are required to use the CLI natively from Docker, were not supplied. The good news is that they can be added as directives to the Compose YML.

Add color and remove the control characters in the Jenkins job log with the `ansiColor { … }` directive in `Jenkinsfile`.

Move keys and certs to a volume. Call it `/secret` inside the DataPower container and use symbolic links so that DataPower uses the files on the volume. This is a good first step toward integration with other techniques of handling secrets with Docker.

Following on to moving keys and certs to a volume, you can use Docker Compose's new Secrets feature.

If you are more ambitions, perhaps you will deploy next to an integration test environment made up of physical DataPower Appliances. How could you go from this CI system to the physical deployment? One approach would be to enhance `client` so that it gathers an application domain export after a successful test. It could do that using cURL to fetch the export from the DataPower REST management or XML management services and saving the result to a file that happens to be in a Docker volume. The whole process could then be automated in Jenkins.

Saving Your Work

Before you leave, you may want to save your work to GitHub or your favorite hosted Git repository.

First, log in to GitHub. Then create a new repository by clicking on the "+" drop-down and selecting `New Repository`. After creating the repository, add it as a new remote in your virtual machine's `datapower-devops-lab` directory, then push to that remote as shown below.

```
localuser@ubuntu-base:~/datapower-devops-lab$ git remote add github
https://github.com/hstenzel/datapower-devops-lab.git
localuser@ubuntu-base:~/datapower-devops-lab$ git push -u github master
```

Be sure to use the URL for your repository, for the new `git remote` – you won't be able to use mine!

Conclusion

Despite the variety of tools and approaches in the CI and Docker ecosystems, IBM DataPower Gateways can integrate fully and naturally into your DevOps practice. But it only works if you adopt appropriate techniques, some of which we've done in this lab.

Thank you for your attention and participation.