11falsenonelisttrue

How Docker Relates to the HPC Environment

Why Docker?

In the past, HPC environments were built upon “static operating system images”, which is to say that each execution node of a cluster had the same Operating System Image, with a set of applications that were curated and installed by the managing IT team. Each application was therefore exposed to every application’s “dependency tree”. For example, if Application A required Library Z version 1, but Application B required Library Z version 2, the applications conflicted with each other. Various methods were devised over the years to try to isolate application environments from each other. The development of “modules” that would use environment variables and shared filesystems was an attempt to solve this problem. In the end, the modules and the environments needed to be built by the cluster managers. End users had limited ability, if any at all, to deploy the software they wanted “on the fly”.

Container technologies are not exactly “new”. The “chroot” system call has been around since 1979. FreeBSD has had “jails” since 2000. Solaris Containers were introduced in 2004. There have been many. But with the advent of Docker, the ecosystem of support around the container technology has finally made it relatively easy for end users to build these portable runtime environments.

Docker allows users to build their own software environments independently of anyone else. The cluster management team no longer needs to be the gatekeeper controlling the available software.

What is Docker?

As described in What is a Container?,

> A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

A *container image* is *built* and *pushed* to a *container registry* for later use. In the RIS Compute Service, a user submits a *job* that is managed by the IBM Spectrum LSF *job scheduler* that, when executed, *pulls* the docker container image to an *execution node* where it is executed on behalf of the user.

The Relationship Between Users, LSF, and Docker

> The following diagram represents relationships between the user and portions of the cluster:
>
> The user submits jobs from the *lsf client* using the *bsub* command. The jobs enter the scheduler and traverse the following states:
>
> When the job is dispatched to the *execution node* there is a *wrapper script* that constructs the *docker run command*.
>
> Recall the date command used in the Quick Start:
>
> Here, the user, a member of the Active Directory group named ${group_name}, is submitting a *job* to the general-interactive *queue*. When the job lands on the execution node, the *wrapper script* will “pull” the Docker container image named “alpine”:
>
> The wrapper then constructs a shell script to execute the desired command:
>
> The wrapper then constructs a *docker run* command to execute the script:
>
> The important thing to know here is that *the user is using Docker, but not directly*. There exists both a “job scheduler” and a “wrapper script” between the user and the docker run command.
>
> This is important to know because many of the features RIS offers users are mitigated by this wrapper script, and some are disallowed, usually for security reasons.
>
> Most features of the docker wrapper are exposed through environment variables. See Docker Wrapper Environment Variables for the list of them and their use.

Public vs Private Registries

Sometimes a user does not want, or is forbidden, to put code into a public registry. RIS suggests use of Docker Hub’s private registries as WashU does not currently have a private container registry. See Build Images Using Compute for how to log into and build using private registries on compute.

Locate Preexisting Docker Images

There are many sources of preexisting Docker images, primarily within Docker Hub's public registry. Before building an image from scratch it is recommended you search for an image to meet your requirements.

Build Images Using Compute1

If you are unable to locate an image on a public registry that meets all of your needs, it will need to either be built from scratch or by building upon an existing container and pushing for future use.

RIS recommends building Docker images on a workstation or other computer you have local access to as it makes debugging the build process easier. However, some build processes may require more resources than you have available locally. For these situations, the compute cluster can be used.

The `docker_build` LSF application accepts all the same arguments as the command-line `docker build`, except for Windows-specific options such as `--security-opt`. It works with build contexts that are subdirectories or URLs that are git repos or tarballs.

1. Log Into Public or Private Registry

The docker_build application uses various environment variables to interact with public and private repositories. Registry credentials are stored in the file `$HOME/.docker/config.json`. Rather than passing the password in as an environment variable, it's a better practice to first log into the registry interactively with `LSB_DOCKER_LOGIN_ONLY` enabled. The credentials will automatically be populated into the config file and used when pushing to that registry.

If the `docker login` process would ask for a password, be sure to run the login build with bsub's interactive flag, for example:

Docker Hub:

LSB_DOCKER_LOGIN_ONLY=1 \ bsub -G ${group_name} -q general-interactive -Is -a 'docker_build' -- .]]>

Other repository:

LSB_DOCKER_LOGIN_ONLY=1 \ LSB_DOCKER_LOGIN_SERVER=repo.example.com \ bsub -G ${group_name} -q general-interactive -Is -a 'docker_build' -- .]]>

2. Build and Push Image

Once logged in, repositories accessible with those credentials, both public and private, will be accessible.

The `docker_build` LSF application accepts all the same arguments as the command-line `docker build`, except for Windows-specific options such as `--security-opt`. It works with build contexts that are subdirectories or URLs that are git repos or tarballs.

For example, this command will submit a job to build a container based on files in the 'my_container' subdirectory, tag it with `1.1.4` and `1.1`. All the tags that appear, both as a "docker_build" argument and "–tag" option, must also be valid to push to with "docker push", and all tags are pushed after the build completes.

bsub -G ${group_name} -q general-interactive -Is -a 'docker_build(repo.example.com/repo_username/example_container_name:1.1.4)' -- --tag repo.example.com/repo_username/example_container_name:1.1 example_container_build_directory]]>

Take note of the `--` that separates the arguments to bsub from the arguments to "docker build". Without it, bsub will try to interpret all the arguments for itself and generate an error. Also, the "build context" directory path must be the final argument. Other arguments recognized by "docker build" must appear before the build context argument.

This command will submit a job where the Dockerfile is in a non-standard place and is not named `Dockerfile`

bsub -G ${group_name} -q general-interactive -Is -a 'docker_build(repo.example.com/repo_username/example_container_name:latest)' -- -f /path/to/the/NonStandardDockerfile example_container_build_directory]]>

Docker Images Must Include /bin/sh

By default, the IBM Spectrum LSF software job scheduler requires that the Docker containers launched have a /bin/sh present. Users may observe that Docker uses hello-world as an example in documentation. This container is an example of one that does not include a /bin/sh. Launching a container that does not supply /bin/sh results in a **"no such file or directory"** error:

bsub -G ${group_name} -Is -q general-interactive -a 'docker(hello-world)' /hello Job <5674> is submitted to queue . <> <> Using default tag: latest latest: Pulling from library/hello-world Digest: sha256:c3b4ada4687bbaa170745b3e4dd8ac3f194ca95b2d0518b417fb47e5879d9b5f Status: Image is up to date for hello-world:latest docker.io/library/hello-world:latest standard_init_linux.go:211: exec user process caused "no such file or directory"]]>

This can be circumvented by the use of an LSF variable so that Docker image that do not have supply /bin/sh can be run on the Compute Platform. Please see our LSF env variable documentation for more information.

The Job Execution Wrapper Script

The wrapper process that builds the `docker run` command does several things in order to construct a “safe” docker run command. The following is an example of job submission in order to demonstrate what the wrapper script is doing:

is submitted to queue . <> <> Feb 05 19:43:44 docker_run[337938]: DEBUG: temp dir /tmp/.lsbtmp1416339 is not accessible, NOT being passed to docker Feb 05 19:43:44 docker_run[337938]: DEBUG: Pass --gpus all to docker Feb 05 19:43:44 docker_run[337938]: DEBUG: Execute docker pull Feb 05 19:43:44 docker_run[337938]: DEBUG: ['/usr/bin/docker', 'pull', 'alpine'] Using default tag: latest latest: Pulling from library/alpine Digest: sha256:ab00606a42621fb68f2ed6ad3c88be54397f981a7b70a79db3d1172b11c4367d Status: Image is up to date for alpine:latest docker.io/library/alpine:latest Feb 05 19:43:45 docker_run[337938]: DEBUG: docker pull returns 0 Feb 05 19:43:45 docker_run[337938]: DEBUG: Execute docker run Feb 05 19:43:45 docker_run[337938]: DEBUG: /usr/bin/docker run --cidfile /tmp/lsf.compute1-lsf.job.23870.1580953423 -u 1416339:1000070 -v /opt/ibm/lsfsuite/lsf:/opt/ibm/lsfsuite/lsf -v /home/mcallawa:/home/mcallawa -w /home/mcallawa/ -v /home/mcallawa/.lsbatch:/home/mcallawa/.lsbatch -v /tmp:/tmp --rm -h compute1-exec-203.ris.wustl.edu --name=lsf_23870_337938 -u 1416339:1000070 --mount type=bind,source=/tmp/tmpysrUPG,target=/etc/passwd --mount type=bind,source=/tmp/tmpAV6MiF,target=/etc/group --group-add 1000070 --group-add 1253902 --group-add 1253901 --group-add 1256392 --group-add 1262589 --group-add 1183521 --group-add 1181132 --group-add 1260486 --group-add 1248446 --group-add 1209233 --group-add 1182940 --group-add 1254277 --group-add 1208826 --group-add 1240927 --group-add 1240939 --group-add 1184777 --group-add 1188010 --group-add 1193236 --group-add 1192550 --group-add 1188482 --group-add 1202092 --group-add 1248450 --group-add 1208827 --group-add 1248452 --group-add 1209246 --group-add 1250918 -w /home/mcallawa --mount type=bind,source=/var/lib/sss/pipes,target=/var/lib/sss/pipes --mount type=bind,source=/etc/nsswitch.conf,target=/etc/nsswitch.conf --mount type=bind,source=/dev/log,target=/dev/log --mount type=bind,source=/tmp/23870.tmpdir,target=/tmp/23870.tmpdir --security-opt label=user:user_u --security-opt label=role:user_r --security-opt label=type:user_t --security-opt label=level:s0 --gpus all --sig-proxy=false -i -t -e LSF_NIOS_JOBSTATUS_INTERVAL=1 --cpu-shares 2 -m 4096m --memory-swap -1 --cap-drop=all --cap-add=chown --cap-add=kill --cap-add=setpcap --security-opt=no-new-privileges --cap-add=IPC_LOCK --cap-add=CAP_SYS_NICE --device=/dev/infiniband/issm0 --device=/dev/infiniband/rdma_cm --device=/dev/infiniband/ucm0 --device=/dev/infiniband/umad0 --device=/dev/infiniband/uverbs0 --ulimit memlock=-1 --env-file=/tmp/lsf.compute1-lsf.job.23870.1580953423.env alpine /home/mcallawa/.lsbatch/1580953421.23870 Thu Feb 6 01:43:46 UTC 2020 Feb 05 19:43:46 docker_run[337938]: DEBUG: run_docker returns 0 Feb 05 19:43:46 docker_run[337938]: DEBUG: Cleaning up /tmp/lsf.compute1-lsf.job.23870.1580953423 Feb 05 19:43:46 docker_run[337938]: DEBUG: About to exit 0]]>

Let us zoom in on the `docker run` command that is being built:

What does all this mean?

Important take aways from the above exploration include:

- We take care of users and groups.
- We set SELinux contexts. SELinux is complex. We will return to this topic later.
- We set OS Capabilities, and you do not have access to all of them.
- Jobs run *as you* and *never as root or any other user*, this may matter for pre-built containers that expect specific users.
- You are *never allowed to pass in Volumes that are not GPFS Volumes* ie. storage1, or scratch1. You are not to see the disk of the execution node.
- You are not to see the execution node’s `/tmp`, but rather *your job’s* `/tmp`.
- We automatically pass in the “current working directory” which overrides a container’s default `WORKDIR`. Often this is $HOME, but pay attention to what you need it to be.
- We *require* a Docker `ENTRYPOINT` that is `/bin/sh` or omitted. This is a limitation of the IBM Spectrum LSF Docker integration.
- We set the hostname within the container to match the execution node it runs on