Building Kubernetes

Building Kubernetes is easy if you take advantage of the containerized build environment. This document will help guide you through understanding this build process.

Requirements

1. Docker, using one of the following configurations:

- **macOS** Install Docker for Mac. See installation instructions here. **Note**: You will want to set the Docker VM to have at least 8GB of initial memory or building will likely fail. (See: #11852).
- **Linux with local Docker** Install Docker according to the instructions for your OS.
- **Windows with Docker Desktop WSL2 backend** Install Docker according to the instructions. Be sure to store your sources in the local Linux file system, not the Windows remote mount at `/mnt/c`.

Note: You will need to check if Docker CLI plugin buildx is properly installed (`docker-buildx` file should be present in `~/.docker/cli-plugins`). You can install buildx according to the instructions.

2. **Optional** Google Cloud SDK

You must install and configure Google Cloud SDK if you want to upload your release to Google Cloud Storage and may safely omit this otherwise.

Overview

While it is possible to build Kubernetes using a local golang installation, we have a build process that runs in a Docker container. This simplifies initial set up and provides for a very consistent build and test environment.

Key scripts

The following scripts are found in the `build/` directory. Note that all scripts must be run from the Kubernetes root directory.

- `build/run.sh`: Run a command in a build docker container. Common invocations:
  - `build/run.sh make`: Build just linux binaries in the container. Pass options and packages as necessary.
  - `build/run.sh make cross`: Build all binaries for all platforms. To build only a specific platform, add `KUBE_BUILD_PLATFORMS=<os>/<arch>`
  - `build/run.sh make kubectl KUBE_BUILD_PLATFORMS=darwin/amd64`: Build the specific binary for the specific platform (`kubectl` and `darwin/amd64` respectively in this example)
  - `build/run.sh make test`: Run all unit tests

  - `build/run.sh make test-integration`: Run integration test
  - `build/run.sh make test-cmd`: Run CLI tests
- `build/copy-output.sh`: This will copy the contents of `_output/dockerized/bin` from the Docker container to the local `_output/dockerized/bin`. It will also copy out specific file patterns that are generated as part of the build process. This is run automatically as part of `build/run.sh`.
- `build/make-clean.sh`: Clean out the contents of `_output`, remove any locally built container images and remove the data container.
- `build/shell.sh`: Drop into a `bash` shell in a build container with a snapshot of the current repo code.

Basic Flow

The scripts directly under `build/` are used to build and test. They will ensure that the `kube-build` Docker image is built (based on `build/build-image/Dockerfile` and after base image's `KUBE_BUILD_IMAGE_CROSS_TAG` from Dockerfile is replaced with one of those actual tags of the base image, like `v1.13.9-2`) and then execute the appropriate command in that container. These scripts will both ensure that the right data is cached from run to run for incremental builds and will copy the results back out of the container. You can specify a different registry/name and version for `kube-cross` by setting `KUBE_CROSS_IMAGE` and `KUBE_CROSS_VERSION`, see `common.sh` for more details.

The `kube-build` container image is built by first creating a "context" directory in `_output/images/build-image`. It is done there instead of at the root of the Kubernetes repo to minimize the amount of data we need to package up when building the image.

There are 3 different containers instances that are run from this image. The first is a "data" container to store all data that needs to persist across to support incremental builds. Next there is an "rsync" container that is used to transfer data in and out to the data container. Lastly there is a "build" container that is used for actually doing build actions. The data container persists across runs while the rsync and build containers are deleted after each use.

`rsync` is used transparently behind the scenes to efficiently move data in and out of the container. This will use an ephemeral port picked by Docker. You can modify this by setting the `KUBE_RSYNC_PORT` env variable.

All Docker names are suffixed with a hash derived from the file path (to allow concurrent usage on things like CI machines) and a version number. When the version number changes all state is cleared and clean build is started. This allows the build infrastructure to be changed and signal to CI systems that old artifacts need to be deleted.

Build artifacts

The build system output all its products to a top level directory in the source repository named `_output`. These include the binary compiled packages (e.g. kubectl, kube-scheduler etc.) and archived Docker images. If you intend to run a component with a docker image you will need to import it from this directory with the appropriate command (e.g. `docker import _output/release-images/amd64/kube-scheduler.tar k8s.io/kube-scheduler:$(git describe)`).

Releasing

The `build/release.sh` script will build a release. It will build binaries, run tests, (optionally) build runtime Docker images.

The main output is a tar file: `kubernetes.tar.gz`. This includes: * Cross compiled client utilities. * Script (`kubectl`) for picking and running the right client binary based on platform. * Examples * Cluster deployment scripts for various clouds * Tar file containing all server binaries

In addition, there are some other tar files that are created: * `kubernetes-client-*.tar.gz` Client binaries for a specific platform. * `kubernetes-server-*.tar.gz` Server binaries for a specific platform.

When building final release tars, they are first staged into `_output/release-stage` before being tar'd up and put into `_output/release-tars`.

Reproducibility

`make release` and its variant `make quick-release` provide a hermetic build environment which should provide some level of reproducibility for builds. `make` itself is **not** hermetic.

The Kubernetes build environment supports the `SOURCE_DATE_EPOCH` environment variable specified by the Reproducible Builds project, which can be set to a UNIX epoch timestamp. This will be used for the build timestamps embedded in compiled Go binaries, and maybe someday also Docker images.

One reasonable setting for this variable is to use the commit timestamp from the tip of the tree being built; this is what the Kubernetes CI system uses. For example, you could use the following one-liner:

```
SOURCE_DATE_EPOCH=$(git show -s --format=format:%ct HEAD)
```