


☐

I'm not robot


reCAPTCHA

Continue

Multiple entry points in dockerfile

12.2.2016 at 21:35 | Comments When using Docker tanks on several building blocks of your app, it is recommended that a separate container be executed for each building block so that the components are very separate. Docker best practices suggest driving one process per container. However, you may imagine a scenario where it makes sense to perform multiple services in one container (an example follows). Then the question arises as to how to run these services with a single EntryPOINT command from Dockerfile. Sample scenario Assume that your app communicates with an external service through an SSH tunnel (or VPN or non-direct connection - name it). Then there are several ways to install the tunnel together with the application itself: Install the tunnel on the host, run the tank with host network mode (i.e. --net=host switch) - which means that the tank does not have a separate stack of nets, but it uses the host stack, so it can only enter the tunnel. Install the tunnel on the host, run the container in the default network mode (i.e. on the bridge), and somehow use the host tunnel from the container. For example, this refers to using the --add-host switch (with the host's IP address) when using the store, which adds an /etc/hosts entry to the store (see documentation for more information). Or you can try other hacks as well. Drive the tunnel into the container. The problem with options 1 and 2 is that you lose a run in any part of docker philosophy, because anywhere is limited to any ssh tunnel. In addition, your infrastructure is no longer unchanged because by setting up a tunnel you just made a change to the host. A change you need to remember every time you run a tank somewhere else. Therefore, option 3 seems to be the way to go. But since Docker allows only one ENTRYPOINT (more specifically, only the last ENTRYPOINT of Dockerphile has an effect), you need to find a way to perform multiple processes (tunnel and application) with a single command. Let's see how you can do it. Solution The simplest idea is to create an interface script to run the necessary processes and use the script as an ENTRYPOINT. But I wouldn't write a blog post about writing a shell manuscript, would I? Instead, let's look at the recommended technology that uses the moderator - the process control system. In the big picture, supervisord is a tool that allows you to run multiple programs at once from one place. The advantages of a standard old interface script include numerous configuration and monitoring options. Here I'm only going to cover basic use, which is enough for our scenario - feel free to explore more advanced stuff yourself. To use a moderator, you must first install it, preferably by using package manager. Ubuntu/Debian-based you must add to Dockery the following: 1 RUN apt-get update && apt-get install -y supervisor Since in this example you also need an SSH tunnel, an SSH client is also installed: 1 RUN apt-get update && apt-get install -y supervisor openssh-client Note: always remember to combine the apt-get update and apt-get installation in one command for the latest package versions - see more documents. Now that the moderator and the SSH client are installed, it's time for the configuration. For example, assume that the starting point for your application is /opt/myapp/bin/myapp. The supervised configuration is then something: 1 2 3 4 5 6 7 8 9 10 11 12 [supervisord] nodaemon=true logfile=/var/log/supervisord/supervisord.log childlogdir=/var/log/myapp [program:ssh] command=ssh -N -L8080:localhost:8080 user@example.com [program:myapp] stdout_logfile=/dev/stdout stdout_logfile_maxbytes=0 command=/opt/myapp/bin/myapp --social-configuration Note: If you put the build file in a default location inside the container, i.e. /etc/supervisor/conf.d/supervisord.conf, the moderator will retrieve it automatically. However, it is assumed that you selected a custom location within the container because of this example, such as /etc/myapp/supervisord.conf. The [Supervisord] section defines the parent administrator process. Nodaemon=true means that the process should remain in the foreground (because the tank will otherwise be stopped). In addition, you can specify a log file for controlled logs (logfile parameter) and directory messages taken from the stdout and stderr of subpro processes (here: ssh and myapp) - with the childlogdir parameter. Next comes the process definitions presented in the store. The SSH is configured with the arbitrary port sub-icing -L switch. In addition, you can use the -N ticket, which means that the remote command will not run - which is only useful for forwarding ports according to the SSH male page. In a Myapp configuration, stdout_logfile parameter indicates where the process stdout should go - in this case, it goes to the tank stdout. You can use the log rotation policy to stdout_logfile_maxbytes a field where a zero value indicates that there is no rotation. The command parameter is self-evident - this is the perfect command for the application to run. After you specify a moderator, the final step is to run it when the store starts - with entrypoint: 1 ENTRYPOINT [usr/bin/supervisord, -c, /etc/myapp/supervisord.conf] Here you can see that we are using the -c switch to provide a path to the custom configuration file. This would not be necessary if the assembly were /etc/supervisor/conf.d/supervisord.conf. After running through the tank, you should see a few logs from the moderator, indicating that both the daemon and the processes have been started: 1 2 3 4 5 6 2016-02-09 16:47:13.949 CRIT Supervisor running as root (no user in config file) 2016-02-09 16:47:13.951 INFO supervisord started with pid 1 2016-02-09 16:47:14.953 INFO spawned: 'ssh' with pid 8 2016-02-09 16:47:14.954 INFO spawned: 'myapp' with pid 9 2016-02-09 16:47:16.754 INFO success: ssh entered RUNNING state, process has stayed up for > than 1 seconds (startsecs) 2016-02-09 16:47:16.755 INFO success: myapp entered RUNNING state, process has stayed up for > than 1 seconds (startsecs) Summary You have just learned the recommended way to run multiple processes in a Docker container - if you know what you're doing, i.e. you really need more than one process in your container. View administrator documents if your scenario is anything other than this basic one. By Jacek Kunicki Feb 12th, 2016 9:35 pm docker « SSL client certificates on the JVM Implementing a custom Akka Streams graph stage » Published 05.12.2019 Tags #Docker #Container #Kubernetes There has been a lot of confusion around the entry point and parameters of the container. This message sheds light on this topic and presents a script that serves as a flexible starting point. ENTRYPOINT and COMMAND Dockerfile can contain an ENTRYPOINT and/or COMMAND entry to determine the store's startup behavior. The relationship between the two filters is described by a few basic rules: If Dockerfile contains only the COMMAND command, included commands run when the store starts If Dockerfile contains only an ENTRYPOINT entry, the included commands run when the store starts If Dockerfile contains entrypoint and COMMAND configuration files, entrypoint commands run with command as parameters Note that these rules are also valid when ENTRYPOINT and COMMAND contain multiple commands. ENTRYPOINT can be ignored by specifying an --entrypoint parameter. COMMAND is ignored by specifying the parameters of the docker command line: docker run myimage these are parameters Shell vs Exec format Both ENTRYPOINT and COMMAND can be expressed in either format: Commands in the command interface format are defined as simple strings: ENTRYPOINT /entrypoint.sh COMMAND These are parameters Commands are prefixed to shell command content that is by default /bin/sh -c. In Exec format, commands are expressed as a JSON matrix: ENTRYPOINT [/entrypoint.sh] COMMAND [these, are, parameters] In this case, the SHELL command is ignored and the commands run as is. It's always a good idea to enter an additional heart point in the different ways of the store. The default behavior shall be determined to correspond to the primary use case. For example, if a container image is supposed to run nginx, it is a startup by default. Nevertheless, it is often necessary to: analyse the behavior of the container image without building a separate image. The starting point can be customized to start an interactive interface or run arbitrary commands. The next starting point (/entrypoint.sh) uses a lot of the variable \$@, which contains all the parameters passed to the script. \$1 stands for \$@'s first entry. The contents of the \$@ command are updated --. At the end, the updated parameters are used to execute the commands that are included in replacing the current process (exec). #!/bin/bash set -o errexit case \$1 in sh|bash) set -- \$@ ;; *) set -- nginx \$@ ;; esac exec \$@ The next Dockerfile embeds the starting point above and sets the default parameters to run nginx in the foreground. From nginx:alpine COPY entrypoint.sh / ENTRYPOINT [/entrypoint.sh] CMD [-g, daemon off;] After you build the image above, it can be used in a following way: Allow nginx to start with default behavior: docker run -d myimage Run interactive shell: docker run -it myimage sh Run command in tank: docker run -it myimage sh -c pwd Show nginx version and test configuration: docker run myimage vt Based on this concept, container entry points can determine a more complex startup behavior. Feedback is always welcome! If you would like to contact me about the content of this article, please use Twitter. Twitter.