`ENTRYPOINT`s for Development

Jacob Howard

Docker Community All Hands

11 March 2021

Container Workflows

Container tooling was primarily designed to run static, long-lived processes in production.

Development is typically more *dynamic*:

- Recompiling applications
- Reloading applications
- Rebuilding static dependencies
- Interactive usage (via a shell or REPL)
- Interactive debugging

Aligning perfectly with production can be limiting.

What workflows and patterns do we have for development?

Automated
Automatic Hot Reload
External Watch-Based Reload
Rebuild/Re-Run on Start
Manual Sidecar **`run`**
Full Image Rebuilds

Interactive
Container **`exec`**
Interactive **`ENTRYPOINT`**
Run Outside Container

Automation Patterns

Automation Pattern 1

Automatic Hot Reload

Many build tools and frameworks support fast automatic rebuilds and reloads on changes

```
# Example: Use Hugo's server mode to monitor the bind
# mount at /site for changes and automatically reload
ENTRYPOINT ["hugo", "server", \
  "--bind", "0.0.0.0", \
  "--source", "/site" \
]
```

Automation Pattern 2

External Watch-Based Reload

If your framework doesn't support hot reloads, you can use an external tool to watch for filesystem changes and restart your code

```
# Example: Use an external tool (nodemon) to watch code
# for changes and restart a Python application
ENTRYPOINT ["nodemon", \
    "--watch", "/server", \
    "/server/main.py" \
]
```

Automation Pattern 3

Rebuild/Re-Run on Start

For more direct control over the rebuild or restart boundaries of your application, you can tie your build to container start / restart

```
# Example: Recompile and run Go code on container start
ENTRYPOINT ["go", "run", "/code/server"]
```

```
# Restart the service to rebuild code
$ docker-compose restart web
```

Automation Pattern 4

Manual Sidecar run (pt.1)

You can also use Compose (+profiles) to define reusable commands as services that run on demand

```
# Example: Regenerate a static site to a shared volume
ENTRYPOINT ["hugo", "-s", "/site", "-d", "/public"]

# Example: Run a benchmark of another service
ENTRYPOINT ["ab", "-n", "100", "-c", "10", "http://api/"]

# Example: Cross-compile code for Linux
ENTRYPOINT ["make"]
```

Automation Pattern 4

Manual Sidecar run (pt. 2)

You'll use Compose profiles to stop these commands from running by default with `docker-compose up`:

```
services:
  <name>:
    build:
      context: <path/to/Dockerfile/directory>
    profiles:
      - adhoc
    ...
```

You can run these commands with `docker-compose run`:

```
$ docker-compose run --rm <name>
```

Automation Pattern 5

Full Image Rebuilds

You can tell Docker Compose to rebuild images on every run (in case you build code in images):

```
$ docker-compose up --build
```

Ideally use only when absolutely necessary (e.g. multi-stage builds); try to target more precisely:

```
$ docker-compose build <service>
$ docker-compose up
```

Interactive Patterns

Interactive Pattern 1

Container exec

Dropping into a shell in an existing container can be a useful technique for debugging:

```
$ docker-compose exec <service> /bin/sh
```

You can also do this with the Docker CLI:

```
$ docker exec -it <container> /bin/sh
```

And use other shells:

```
$ docker-compose exec database psql
```

Interactive Pattern 2

Interactive ENTRYPOINT

For less *ad hoc* usage, you can define an interactive `ENTRYPOINT`

```
# Example: Create a container that runs an IPython shell
ENTRYPOINT ["ipython3"]

# Example: Create a container that runs a psql shell
ENTRYPOINT ["psql", "-h", "database", "appdb"]
```

Useful for:

- **Shipping reproducible interactive environments**
- **Defining interactive sidecar services in Compose**

Interactive Pattern 3

Run Outside Containers

With Docker Desktop, you can communicate with network services running on the host system using `host.docker.internal`

Useful for:

- Initial or partial containerization
- Using host-side tools (IDE, profiler, etc.)
- Interfacing with external infrastructure

One more thing...

Signals

There's one important gotcha when using development tools for container `ENTRYPOINT`s: signals

Different shells, language runtimes, and build tools handle (or *don't* handle) signals differently.

The solution is to use an init process.

Handling Signals

Docker Compose makes this trivial:

```
# Use an init process to wrap a service ENTRYPOINT
services:
  my_build_service:
    init: true
```

For the CLI, use the `--init` flag when creating containers (via `create` or `run`).

Takeaways

We need to push containers further to fully extract their value for development.

The best practices, ideas, and idioms are almost certainly still to come.

Keep an eye on the Compose Spec and Docker Desktop for new features and ideas.

Thanks for your time!

Slides available at

`havoc.io/talks/entrypoints-for-development`

Examples available at

`github.com/havoc-io/entrypoints-for-development`

Ping me with questions or feedback

@havoc_io