Reusing Apps Between Teams and Environments Through Containers

Adrian Kosmaczewski

2022-08-26

This was my speech for the WeAreDevelopers Container Day[1] on February 3rd, 2021. The talk will feature a live demo showing how to build, optimize, and distribute containers to be reused in as many environments as possible, 100% based on the experience of the VSHN team.

Introduction

Thanks a lot everyone for attending this presentation, and thanks to the WeAreDevelopers team for having me today on this stage. My name is Adrian Kosmaczewski, I am in charge of Developer Relations at VSHN, the DevOps Company located in Zürich, Switzerland.

When organizations start using containers, they usually see them primarily as a lightweight virtual machine. Developers find it easy to learn how to write Dockerfiles, and Operation teams appreciate being able to start and stop them quickly and with low requirements. As time passes, the usual path for those organizations consists in assembling larger applications in Docker Compose, and finally to migrate those apps to Kubernetes.

But this is not the focus of our meeting today. In this talk I will show you how containers are used inside VSHN to not only run cloud native applications, but also to share internal tools in companies

The Challenge of Internal Tools

In the latest "State of Internal Tools" report[2] by Retool,

> More than 80% of respondents said that internal tools are critical to their company's operational success.

However,

> Despite how important internal tooling is, teams still struggle to get the buy in they need: more than half of respondents indicated that one of their biggest problems building internal tools is that they don't have enough time or resources to get the job done.

[1] https://www.wearedevelopers.com/event/container-day
[2] https://retool.com/blog/state-of-internal-tools-2020/

We at VSHN have lots of internal tools[3] and our teams have to juggle between the maintenance of our own customer's systems, which is our core business, and maintaining and keeping updated the myriad tools that we use every and each day. Most of those custom tools are command-line based.

To make things a bit more complex, each VSHNeer can choose their preferred operating system[4], and this means that our technical teams feature users running various Linux distributions (the vast majority, using Ubuntu, Arch, Fedora, and others), Macs, and even Windows PCs.

We also use the best possible programming language for the job; we have internal tools written in Go, C++ , Java, Python, and JavaScript; each has its own requirement, libraries, workflows, and ecosystem.

Go, C and C++ are great languages for multi-platform command-line tools, but to build them, they require a certain degree of knowledge that not all teams might have.

On the other hand, languages like Ruby, Python, and JavaScript have a great library ecosystem, which make them a fantastic choice for prototyping and releasing new features faster. But they require quite a bit of knowledge in terms of libraries, runtimes, and even language versions.

To make matters more complicated, some tools require custom configuration and resources: fonts, styles, images, default configuration files. All of this increases the chances for something to go wrong during installation or runtime.

And to top it off, we need to reuse that knowledge in CI/CD pipelines ; after all, that is one of the core pillars of DevOps, and our slogan is, quite literally, "The DevOps Company". We have to eat our own dogfood.

Our challenge, then, was to find a way to share tools and knowledge with one another and with other systems, with the least possible amount of friction. And for that, we have chosen containers.

Sharing with users in other platforms

The first challenge is then to reuse tools with users in other platforms. To illustrate how containers help us do that, I have created a small TypeScript application. I have chosen to show an example in the Node.js / JavaScript ecosystem, because this is arguably the one with the highest headache per minute ratio in the industry.

TypeScript is a language that compiles to JavaScript, which means that we need quite a bit of infrastructure to first compile the application, and then to run it. Lots of dependencies, and lots of "downloading the internet"; we live in the times of npm , which has taken the crown from mvn in the past decade.

The project I'm going to show is called "Greeter"[5]; it is a very simple command-line tool that greets the user on the command line using various characters:

[3] https://handbook.vshn.ch/hb/terminology.html#_tool_naming
[4] https://handbook.vshn.ch/hb/your_first_day.html#_laptop
[5] https://gitlab.com/vshn/applications/greeter

Snoopy, R2-D2, a cat, and many more. Characters can "talk" or "think", that is, their text bubbles change depending on the context.

All of this has a certain number of dependencies, and if someone would like to run this app, they would need to install Node.js and perform quite a few operations before being able to run the app.

Our objective today will be reusing this tool in as many environments as possible, with the least amount of stress.

To make things even more interesting, we are going to use Podman[6] instead of Docker[7], and as you can see, I am not using an alias in my command line. This is actually Podman running. And because Docker Hub[8] now imposes certain download restrictions, we are going to use two different container repositories: GitLab and Quay.

As the Joker said, "Gentlemen, let's broaden our minds"[9].

The Dockerfile is using that great technique called "Multi-Stage Builds"[10]. This is probably one of the greatest features Docker introduced in the past 5 years and it has saved countless Petabytes of storage all over the planet.

This technique consists of using a separate image for the build process, and another for runtime. The advantage being that the resulting image that is shared and used is well, as small as possible, and does not contain libraries that are only required during the build.

On the first stage of our Dockerfile[11] we install all the requirements for building our app; that includes the TypeScript compiler, Gulp, and some Gulp plugins. The package.json[12] file of the project shows that there are quite a few dependencies required.

Then we use the pkg[13] project to create a single executable file that contains all the required libraries.

Finally, on the second step of the Dockerfile[14], we use the most lightweight possible distribution (Alpine, of course), we install glibc (sadly, pkg does not yet work very well with musl alone) and finally we copy our executable.

To share our container with others, let's use a public container repository. In this case I will use Red Hat's own Quay.io service, where I create a public project[15] for our "Greeter" image.

We can now push our final container image to Quay.io. To reuse it , I will switch to a VMWare virtual machine where I am running Windows and where I have

[6] https://podman.io/
[7] https://www.docker.com/
[8] https://hub.docker.com/
[9] https://www.youtube.com/watch?v=T8EzcQ0p1Tg
[10] https://docs.docker.com/develop/develop-images/multistage-build/
[11] https://gitlab.com/vshn/applications/greeter/-/blob/master/Dockerfile
[12] https://gitlab.com/vshn/applications/greeter/-/blob/master/package.json
[13] https://github.com/vercel/pkg
[14] https://gitlab.com/vshn/applications/greeter/-/blob/master/Dockerfile#L28
[15] https://quay.io/repository/akosma/greeter

installed Docker. This is all very "Inception" like, or as Edgar Allan Poe would say[16], "A dream within a dream".

If the Quay.io project was private, I would need to login into my Quay.io account using Docker, of course. But this one is public, so I can pull my newly created image with Docker Desktop for Windows, and boom, I can run it. Very simple. Podman produces images using the same standard as Docker, which means that the interoperability is perfect.

Reusing in GitLab CI/CD

But as you can see I'm hosting my application in GitLab, and GitLab has a lovely CI/CD system and a fantastic container registry, all built in. Let's add a simple .gitlab-ci.yml[17] file in our "Greeter" project, and this way we will be able to store the product of our Dockerfile directly into our project's own image registry[18].

This is even better! Now I can login to `registry.gitlab.com` from Podman or Docker, in any platform, and pull and run this image as I need it.

But this is not all; I would like to reuse my greeter image in other contexts. Take, for example, the case of CI/CD pipelines. We have all faced the hurdles of properly configuring pipelines, be it in Jenkins, TeamCity, GitHub Actions or GitLab. It ain't easy, it ain't pretty, but OMG how cool it is when it works, right?

When it comes to CI/CD pipelines, we at VSHN are great fans of not adding much complexity in the pipelines themselves; we like to be able to reproduce locally the steps performed by the pipelines on our own machines. This makes it much simpler to diagnose and troubleshoot problems, and this is where containers shine.

To show an example of this at work, here's a little project[19] called "Fortune"[20]; it is a Flask application that returns a random "fortune cookie of the day" per HTTP. This app is built with Python, which has nothing to do with JavaScript (and thankfully so!)

We are going to reuse our beloved "multi-stage build" procedure, this time introducing a new step in the Dockerfile[21] . Let us commit and push this change, and now our CI/CD pipeline run[22] displays an image of Snoopy greeting us directly from the pipeline. Isn't this lovely?

In a few steps, we have taken an application with lots of dependencies, and have bundled it in a format that is suitable for collaboration and reuse in as many systems as we could think of.

[16] https://www.poetryfoundation.org/poems/52829/a-dream-within-a-dream
[17] https://gitlab.com/vshn/applications/greeter/-/blob/master/.gitlab-ci.yml
[18] https://gitlab.com/vshn/applications/greeter/container_registry/1669520
[19] Unfortunately the project does not exist anymore at the original location. Will restore it as soon as I find it.
[20] https://gitlab.com/akosma/fortune
[21] https://gitlab.com/akosma/fortune/-/blob/master/Dockerfile
[22] https://gitlab.com/akosma/fortune/-/jobs/995577647#L131

Task Examples

What are the kind of things you can do with this approach? At VSHN we have created container images that allow us to do the following things:

- Run tasks on code:
  - Linter
  - Black-box testing
- Run tasks on documentation:
  - HTML dead link verification
    - https://github.com/wjdp/htmltest
  - Style checks using vale
    - https://github.com/vshn/vale
  - Spell checking
    - https://github.com/vshn/hunspell
  - Create PDF or EPUB formats
    - https://github.com/vshn/asciidoctor-pdf
    - https://github.com/vshn/asciidoctor-epub3
  - Creating the index for a search engine
    - https://github.com/vshn/antora-indexer-cli/
  - Live preview of documentation
    - https://github.com/vshn/antora-preview

Another nice benefit is that users can run multiple versions of the same tool, without conflicts! The trick for that consists in *always* tagging your images properly, and *never* using `latest` as a tag.

Gotchas & Drawbacks

As with any technique or approach, there are some things to keep in mind; remember that there is No Silver Bullet[23].

Regarding GitLab

- If you are using GitLab 2FA (and you should), you need to go to GitLab's "Settings / Access Tokens" page and create one of those (with read / write access to the registry, as needed) as the password.

Regarding creating command-line tools

- Follow the Command Line Interface Guidelines[24] and The Art of Unix Programming[25] for your tools; make sure to add help, clear parameters, and to report errors to stderr and to return proper error codes.
  - GitLab CI/CD pipelines AND `docker build` stop executing when the return of any step in the process returns anything else than zero!
    - Use this to your advantage.
    - Make build stop when linter / tests / etc don't pass.

[23] https://en.wikipedia.org/wiki/No_Silver_Bullet
[24] https://clig.dev/
[25] https://en.wikipedia.org/wiki/The_Art_of_Unix_Programming

  - If the tool is complex, consider creating a `man` page for it; you can make one very simply using Asciidoctor[26].
- To have some sort of standardization of your internal tools, think about using `cookiecutter` to help you get started.
  - `cookiecutter` https://gitlab.com/akosma/typescript-cli-cookiecutter
- Using `docker run` is a bit more cumbersome than using an executable.
  - Use aliases and Makefiles to make your life easier!
  - Standardize Makefile tasks across projects
    * `make build`
    * `make release`
    * `make clean`
    * `make lint`
    * `make check`
    * `make preview`
- Pay attention to the path of the tool inside the image.
  - This needs to be documented. Inside the container, the application lives in a different filesystem location than what you'd expect.
  - Document this clearly in the README of the tool: provide usage examples!
  - It can get more complicated when using the `--volume` parameter, but it is also very flexible. Remember to add the `:ro` parameter at the end of the volume definition if you just want to read on a directory.
- If your tool generates files, pay attention to the `USER` that is running your code. In general, remember to `docker run --user "$(id -u)"` so that the files that you generate "belong to you" instead of `root`.
  - Podman does not require it, as it runs not under root , but as the current user.
- If you need to pass secrets to your applications, use environment variables; the easiest way to pass them is using `--env-file secrets.txt`
  - Remember to add the name `secrets.txt` to `.gitignore` ! Don't commit secrets to Git.
  - In GitLab, use the built-in secrets functionality, and create environment variables that you reference in the YAML file
- Use tools like Release It![27] to control your tags and version numbers during build time

Regarding building containers

- Beware of base images that **do not** contain `/bin/bash` and only have `/bin/sh`
  - Either make your shell scripts work with `/bin/sh` if at all possible, or install `bash` in your target image.
  - Similarly, beware of images based in `musl` instead of `glibc` like Alpine! Check out the differences[28] between both.
- Use dive[29] to inspect the internal state of your containers.
- Run `podman ps -a -q` to find out stopped containers.

[26] https://docs.asciidoctor.org/asciidoctor/latest/manpage-backend/
[27] https://github.com/release-it/release-it
[28] https://wiki.musl-libc.org/functional-differences-from-glibc.html
[29] https://github.com/wagoodman/dive

- Run `podman image prune` to get rid of intermediate images (and save a few GBs of disk space every so often!).
- Use `podman` and `docker` to make sure your containers run in both.
  - At the moment, Podman does not really support Windows and Mac, even though there are binaries for those systems in the website
- Docker has introduced image pull limits: *"Docker Hub is slowly starting to enforce a pull rate limit of 100 pulls per 6 hours for anonymous (unauthenticated) IP-Addresses, and 200 pulls per 6 hours for authenticated non-paying users."*
  - Use other container repositories: there's a life after Docker Hub! There are self-hosted options, such as: kraken[30], Harbor[31], Docker distribution[32], and Sonatype Nexus[33]. And here's a UI[34] for that private registry. There are other SaaS registries than Docker Hub: Docker Pro or Team (paid) plans[35], Quay[36], AWS ECR[37] (Future free public registry announcement[38]), GitHub packages[39] (ghcr.io), and Google Container Registry[40] (gcr.io). And finally, there are registries embedded in OpenShift[41] and the one we saw in GitLab[42].

Tips for Python command-line tools

- Use virtual environments.
- Add the `--no-cache-dir` option to `pip install` in Dockerfiles:

```
$ python3 -m venv .venv
$ source .venv/bin/activate
$ pip install PyYAML
$ pip install PyGithub
$ pip freeze > requirements.txt
$ cat requirements.txt

# later on...
$ pip install -r requirements.txt
```

Tips for tools built with Go

- Since the Go compiler produces self-contained binaries, ready to be used, you can use distroless[43] as the base image, which will reduce the size of

[30] https://github.com/uber/kraken
[31] https://goharbor.io/
[32] https://github.com/docker/distribution
[33] https://blog.sonatype.com/nexus-as-a-container-registry
[34] https://joxit.dev/docker-registry-ui/
[35] https://www.docker.com/pricing
[36] https://quay.io/
[37] https://aws.amazon.com/ecr/
[38] https://aws.amazon.com/blogs/containers/advice-for-customers-dealing-with-docker-hub-rate-limits-and-a-coming-soon-announcement/
[39] https://github.com/features/packages
[40] https://cloud.google.com/container-registry/
[41] https://docs.openshift.com/container-platform/4.5/registry/architecture-component-imageregistry.html
[42] https://docs.gitlab.com/ee/user/packages/container_registry/
[43] https://github.com/GoogleContainerTools/distroless

your final image drastically.
  - Remember to set the `ENTRYPOINT` properly in your Dockerfile!

Tips for command-line tools built with JavaScript

- Use multi-step builds!
- Run `npm install --prod` to only install the libraries required at runtime.
- Use pkg[44] to create standalone binaries.
- Use the scratch-node[45] base image for minimalistic, small final container images (GitHub[46]).
- Use TypeScript[47] to make your code stronger and easier to manage in your team.

Regarding `CMD` vs `ENTRYPOINT`

Pay attention to the differences between `CMD` vs `ENTRYPOINT`:

> In short, CMD defines default commands and/or parameters for a container. CMD is an instruction that is best to use if you need a default command which users can easily override. If a Dockerfile has multiple CMDs, it only applies the instructions from the last one.
>
> On the other hand, ENTRYPOINT is preferred when you want to define a container with a specific executable. You cannot override an ENTRYPOINT when starting a container unless you add the –entrypoint flag.
>
> Combine ENTRYPOINT with CMD if you need a container with a specified executable and a default parameter that can be modified easily. For example, when containerizing an application use ENTRYPOINT and CMD to set environment-specific variables.

(Source)[48]

Regarding `ADD` vs `COPY`

> …in a nutshell, the major difference is that ADD can do more than `COPY`:
>
> - `ADD` allows `<src>` to be a URL
> - Referring to comments below, the `ADD` documentation[49] states that:
>
> If is a local tar archive in a recognized compression format (identity, gzip, bzip2 or xz) then it is unpacked as a directory. Resources from remote URLs are not decompressed.

[44] https://github.com/vercel/pkg
[45] https://hub.docker.com/r/astefanutti/scratch-node
[46] https://github.com/astefanutti/scratch-node
[47] https://www.typescriptlang.org/
[48] https://phoenixnap.com/kb/docker-cmd-vs-entrypoint
[49] https://docs.docker.com/engine/reference/builder/#add

> Note that the Best practices for writing Dockerfiles[50] suggests using `COPY` where the magic of `ADD` is not required. Otherwise, you (since you had to look up this answer) are likely to get surprised someday when you mean to copy `keep_this_archive_intact.tar.gz` into your container, but instead, you spray the contents onto your filesystem.

(Source)[51]

Conclusion

Thanks to containers, at VSHN we have reached a very high degree of code reuse. We have reduced the requirements to just one: Docker (or, as you have seen, Podman).

Most of the tools shown in this presentation are open source and free to use. Feel free to build upon them, reuse them, and if you want, to submit your pull requests. We will be very happy to consider your ideas for future versions!

I hope that this presentation has been useful to you, and keep building containers!

[50] https://docs.docker.com/develop/develop-images/dockerfile_best-practices/#add-or-copy

[51] https://stackoverflow.com/a/24958548