Living in containers — on TEX Live (and ConTEXt) in a Docker setting

Island of TEX

Abstract

Over the course of the last year(s), the Island of TEX has received quite some interest in its Docker containers. This article gives a brief overview about our container infrastructure for TEX Live and ConTEXt, including some examples on using our containers in production environments. Last but not least, we will elaborate on some interesting (mostly still open) problems connected to containerizing TEX Live.

1 Overview of the Island's Docker images

Since 2019, the Island of TEX provides multiple Docker images for TEX Live and ConTEXt. Our first publication on this topic was in *TUGboat*, Volume 40 (2019), No. 3 and stays mostly relevant. Therefore, let us keep this introductory section short.

We still conceive Docker as an easy way to ship a portable setup of software to users by providing them with an operating system layer, operating system packages (like Python) and the software layer (in our case TEX Live or ConTEXt LMTX) bundled into one single compressed file (the Docker image; simplified but sufficient for visualization). By pulling a Docker image the user has a reproducible setup at hand without caring about his own host operating system or software dependencies.

To this end, we provide a fairly minimal ConTEXt LMTX image (`contextgarden/context:lmtx`) with the LMTX standalone distribution and the modules that ship with this ConTEXt distribution. The image is based on Debian (testing branch) and is about 250 MB in size.

On the other end of the spectrum we provide Docker images for TEX Live (`texlive/texlive`) in different flavors. For all releases from 2013 on up until the latest historic release (currently 2022), we provide the historic images following the naming scheme `TL{YEAR}-historic`. They always contain a `scheme-full` TEX Live installation without documentation and source tree unless you explicitly request one or the other of these removed trees by appending `-doc` or `-src` respectively (`doc` first if you want to have both, e.g. `TL2022-historic-doc-src`).

Like the ConTEXt images, the historic images are based on Debian's testing branch. Additionally, they ship with required software to run most of the tools included in TEX Live, e.g. Java for tools like `arara`, Python for `pygments` (needed for the popular `minted` package), and so on. This comes at a cost: the `TL2022-historic` image without documentation and source files tips the scales at 2.16 GB.

A note on the word `historic`: The images are structured into multiple layers, one of them being the historic TEX Live tree that does not change. However, the other layers which contain the operating system and OS software packages are updated monthly. This does not necessarily reflect the operating system software situation that has been present in the historic TEX Live release's year but is beneficial from multiple other points of view, e.g. when you want to run your own scripts for pre- or postprocessing.

Apart from the historic images, we provide images for the latest release of TEX Live (currently 2023). The basic setup concerning the Debian base and software packages is identical to the historic images. However, in addition to splitting off the documentation and source tree you may request any of the TEX Live schemes of the latest release by appending a hyphen and the scheme's name to the `latest` tag, e.g. `latest-small` or `latest-medium-doc`, to name just two variations. The default `latest` tag will pull a `scheme-full` installation: handle with care. All these various images are updated weekly, with both Debian and TEX Live package updates.

You can find our images on both Docker Hub:

`hub.docker.com/r/texlive/texlive`
`hub.docker.com/r/contextgarden/context`

and our GitLab registry; see the projects' code repositories at:

`gitlab.com/islandoftex/images/texlive`
`gitlab.com/islandoftex/images/context`

2 Using the images in a local setup

One of the two primary use cases we focus on when developing the images is use in a local environment, i.e. replacing your local TEX Live installation. To start off, let us emphasize that usually, especially when using the latest TEX Live release locally, you do not want to use our Docker images here for various reasons including imperfect updating strategies and space overhead.

However, in many settings there are a number of benefits using Docker images locally, especially when using the historic images. For the sake of this discussion, let's suppose you are coordinating a team who have used LATEX for their various documents on various operating systems for years.

. . . enter story telling mode . . .

Now let's imagine a new, not so tech-savvy, contributor joining; meet Bob. You have to explain to him how to install a TEX installation on his machine which runs an operating system you are not comfortable interacting with. You use last year's

doi.org/10.47397/tb/44-2/tb137island-docker

```
image: registry.gitlab.com/islandoftex/images/texlive:TL2022-historic

build:
  script:
    - find -name "*.tex" -exec arara -v "{}" \;
  artifacts:
    paths:
      - ./**/*.pdf
```

Figure 1: Preliminary `.gitlab-ci.yml` file to rebuild all `.tex` sources with `arara`.

TEX Live release so you have to refer Bob to one of the guides on how to install a historic TEX Live release there. And you use `arara` for build management and `minted` in your documents ... the thought of guiding Bob through Java and Python installations and debugging a setup on another operating system is not that appealing. But wait a minute, at this point you could also refer him to one of the various setup guides for Docker and let him `docker pull texlive/texlive:TL2022-historic` and be done.

No sooner said than done. Bob now has his Docker-based TEX Live up and running, including all the dependencies needed for his daily typesetting. He creates his first document and runs it in the Docker container using:

```
docker run -i \
  -v "$PWD":/opt/doc:z -w /opt/doc \
  texlive/texlive:TL2022-historic \
  arara -v document.tex
```

To avoid typing all this every time, he configures his editor to run this command on compilation. Bob is happy, you are happy, onboarding done.

As a short interlude for the interested reader: the longish command above pulls and runs the historic image of TEX Live 2022 from Docker Hub (remember that it is the image without documentation or sources). When it starts the image, it will mount the current working directory in the Docker image, ensuring that all documents are accessible and the build results will appear there. It then starts the `arara` call at the end of the command, in interactive mode so you can interact with the output of the command transparently through the Docker boundary.

A week later, Bob is tasked to typeset an updated version of one of the older documents which does not compile on TEX Live 2022 any more. To see how it looked back then, an older TEX Live release is needed. Luckily, you have avoided needing to remind him of the installation instructions for a historic TEX Live release just to find yourself in the situation to explain to him how to set the `PATH` variable of his operating system to the older release and back to the newer one. You just let him duplicate his editor configuration for TEX Live 2022 and use the older release.

```
docker run -i \
  -v "$PWD":/opt/doc:z -w /opt/doc \
  texlive/texlive:TL2018-historic \
  arara -v document.tex
```

and everything works. Docker even pulled the image on first use without needing a separate pull command.

Now that you just happened to have finished onboarding a new contributor you decide to write a short setup guide for future new contributors in your team. Interestingly enough, the whole setup guide fits onto one A4 page. Happy that you have a concise guide covering everything from installing to running multiple TEX Live releases with a reproducible environment and dependencies, you close that onboarding chapter.

3 Using the images in a CI setting

Bob got hooked, all this Docker business was easy enough to be well hidden behind his editor for now. All this technical stuff being a bit magical to him he would still like to verify that documents he makes available to you will always compile. As you are using GitLab anyway, you introduce him to `git` (a lot more work than the one A4 page for the local setup) and set up a continuous integration pipeline on your GitLab instance that compiles all the documents when a new commit is pushed.

The setup is simple: you add a `.gitlab-ci.yml` file to your repository which has the content shown in figure 1.

Done. The pipeline runs and finishes ... after quite some time. Waiting 10 minutes for the feedback that the documents Bob just touched compile seems a bit subpar to you. Your inner Don Knuth starts yelling at you about something with premature optimization but you are convinced: the repository grows, it cannot be a good idea to always run all documents when we are only interested in potential compilation errors of a few of them.

So your preferred setup would instead look something like this:

```
...
  script:
    - bash compile-only-needed.sh
...
```

with some bash magic taking care of compiling only what is needed. A deep dive into GitLab's documentation later you see this is not as hard as you had imagined. So your bash script is surprisingly short:

```
#!/usr/bin/env bash

gitsha="$(curl \
  --header "PRIVATE-TOKEN: \
            $GL_API_ACCESS_TOKEN" \
  "https://gitlab.com/api/v4/projects/\
$CI_PROJECT_ID/pipelines?ref=$CI_DEFAULT_BRANCH\
&sort=desc&status=success" \
  | grep -o -E -m1 '"sha":"([^"]*)"' \
  | head -1 | cut -c 8-47)"

changed_files=$(git diff-tree \
  --no-commit-id --name-only -r \
  "$gitsha".."$CI_COMMIT_SHA")
compile_all=false
for file in $changed_files; do
  if [[ $file == texmf/* ]]; then
    compile_all=true
    break
  fi
done

if [ "$compile_all" = true ]; then
  latex_files=$(find . -name "*.tex")
  for file in $latex_files; do
    if grep -Fq "arara" "$file"; then
      base_dir="$(dirname "$file")"
      base_name="$(basename "$file")"
      cd "$base_dir"
      arara -v "$base_name"
      cd "$(git rev-parse --show-toplevel)"
    fi
  done
else
  for file in $changed_files; do
    base_dir="$(dirname "$file")"
    base_name="$(basename "$file")"
    if [[ ! -f "$file" ]] \
    || [[ "$file" != *.tex ]] \
    || ! grep -Fq "arara" "$file"; then
      continue
    fi
    cd "$base_dir"
    arara -v "$base_name"
    cd "$(git rev-parse --show-toplevel)"
  done
fi
```

You know that you ignored most of the sanity checking you should have done. But as the old engineering adage says: "it works". It even takes into account that it needs to recompile everything if one of your "global" files changes — the ones you have in your `texmf` folder in your repository like your logo, custom packages, etc.

The basic structure is even easy to explain: first, the GitLab API is queried for the last commit on your default branch a pipeline has successfully run. Then, `git` is run to determine all files that have changed since that commit. If one of the changed files affects all documents, the CI runs basically the `find` call of your first CI example[1] with some directory changes. If no such file has been changed, `arara` is run on all relevant changed LaTeX sources.

With this simple bash script, you successfully turned your 10 minute pipeline into a 3 minute on average pipeline leaving you quite satisfied but wondering why it takes so long to typeset one or two documents.

A short investigation of the CI log later you have identified the culprit: you use the full TeX Live image, i.e. `scheme-full` which pulls more than 2 GB each time your pipeline runs, making up more than 2 minutes of that 3 minutes. Unfortunately, you are using a historic image which does not provide a split by schemes (and you use too many packages anyway) so you cannot slim down on that one. But you notice that your team has spare server capacity and set up a GitLab runner for your project that caches the historic TeX Live images.

Now that you have successfully reduced your average pipeline to less than one minute running time you are confident that it is future-proof enough. And after a few more minutes than originally intended you have successfully implemented Bob's request.

. . . story telling mode off . . .

4 Maintenance challenges of the TeX Live images

The Island of TeX manages and builds all its Docker images using GitLab and the GitLab CI. Unfortunately, we reached some limits on the main instance at `gitlab.com` quite early in our image build processes. Thanks to Marei Peischl and Vít Starý Novotný we have access to custom CI runners (read: servers that build our images) which have massively improved the stability of our build process.

[1] At this point we should add that that `find` call in the first example would not work properly due to executing `arara` in the wrong base directory, which is a bad idea in general. But as it was a motivating example and is fixed in the bash script, we consider this error a case of "no harm done".

However, there is still something calling for manual intervention every other week. So we are interested in improving our build setups to avoid all this intervention. Part of that will include switching to new infrastructure, and part of that in turn will include optimizing the build process and caching.

To further build optimizations, we would like to provide the historic images split by scheme as we do for the latest images. This will require more substantial changes than we would like to admit but also bring the benefits of smaller images to many, especially as we acknowledge the importance of the historic images.

Revisiting the topic of automation, there is a minor annoyance also caused by requiring manual intervention: each year, when a new TEX Live is released we have to add the now-historic TEX Live to our build matrix for historic images. We would like to fix this but have not found the way to go yet. Ideas are welcome.

Another topic that looks for helping hands is the layering of the TEX Live images. This is especially important with the split by schemes which could potentially be layered on top of each other but also to improve the update situation for local uses of the latest images. Experimentation, ideas, and fruitful discussions on our issue tracker at `gitlab.com/islandoftex/images/texlive/-/issues` would be greatly appreciated.

Last but not least, there is one topic that has been a challenge so far but is on the short-term roadmap of actually being resolved: providing multi-architecture images. Currently, our images only provide binaries for the `x86_64` architecture but a few platforms, most notably smartphones and Raspberry Pis, run on a different architecture, namely ARM. TEX Live ships with binaries for these architectures and by the next time you hear from us we hope our images do too.

◇ Island of TEX
`https://gitlab.com/islandoftex`
`https://islandoftex.gitlab.io`