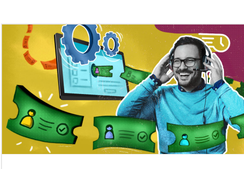
Build a Docker Container from a pnpm monorepo

How to containerize a SvelteKit app


[redacted]
ManageEngine ServiceDesk Plus ManageEngine
Open

Contents

- Containerizing a SvelteKit Mono Repo
- Multi-Stage Dockerfile
  - Build Stage
  - Runtime Stage
- Conclusion

Introduction

Are you developing applications with SvelteKit? Have you modularized your app in a mono-repo using pnpm? Do you deploy a containerized runtime using Docker?

If so, congratulations - you've made great technology choices! But if you're struggling to put all the pieces together or just want to confirm that your Docker Image is as efficient and optimized as it could be, this may be for you.

Example repo with code

Containerizing a SvelteKit Mono Repo

Creating a Docker Image of a SvelteKit mono-repo isn't a case of copying *everything* in our project folder into it. While it may work, and run, it's not going to be very efficient and you'll end up with a Docker image that could easily be Gigabytes larger than it needs to be. The goal is for the image to contain as little as possible - just what is required for runtime execution. A lean minimal container image will not only be quicker to deploy but will also be more secure as it has fewer dependencies installed.

When developing we inevitably use many more packages to compile and build our project than are required to run it. As an example, an out-of-the-box SvelteKit project may contain these `devDependencies` (when set to use the node adapter):

- @sveltejs/adapter-node
- @sveltejs/kit
- @sveltejs/vite-plugin-svelte
- svelte
- svelte-check
- tslib
- typescript
- vite

None of these need to be installed to execute the built version of the app - they either play a part in building it, or the pieces of the packages we use in our app are included in the built output. As we develop larger apps, the dependencies we use often grow and one of the eternal challenges is keeping control on how many packages we pull in. But it's not unusual to end up with packages for UI components, charting, date handling and many others.

There are also runtime `dependencies` that our app will need. These are packages that the server uses and while we *could* build some of them into our app code, it's more efficient and often necessary, to install them on the server. Sometimes they will have their own run-time installed dependencies, such as libraries for image processing or AI, or trying to re-package them into our app is problematic for technical reasons - they simply aren't designed to be used like that.

We need these packages, defined in `dependencies`, to be installed in the docker image, but we don't want the `devDependencies` to be installed.

Multi-Stage Dockerfile

To achieve this we can use a "multi-stage" Dockerfile. This builds the final Docker Image in stages, and the final stage can use some of the outputs produced in earlier stages, but without including all of the dependencies that were needed to create them.

Here's the Dockerfile we're going to use. But before you copy-and-paste this, there are some other pieces needed to make this work, so keep reading ...

```
FROM node:22-bookworm-slim AS build
ENV PNPM_HOME="/pnpm"
ENV PATH="$PNPM_HOME:$PATH"
RUN corepack enable

WORKDIR /app
COPY . .

RUN --mount=type=cache,id=pnpm,target=/pnpm/store \
    pnpm install --frozen-lockfile
RUN pnpm run -r build
RUN pnpm deploy --filter=web --prod out

FROM gcr.io/distroless/nodejs22-debian12
WORKDIR /app
ENV NODE_ENV=production
ENV ORIGIN=http://localhost:8080

COPY --from=build /app/out/ .
EXPOSE 8080
CMD ["server.js"]
```

We'll go through what each line does to explain it.

Build Stage

The "build" part of our multi-stage Dockerfile is going to use the latest slim version of the node image. Unless you use something that needs the extra dependencies (e.g. the full C build toolchain) it only uses a couple of hundred Mb vs more than a Gb:

```
FROM node:22-bookworm-slim AS build
```

Next, we configure and enable `pnpm`. Note that we no longer need to install pnpm using npm - support is built in to recent node versions and activated using `corepack enable`:

```
FROM node:22-bookworm-slim AS build
ENV PNPM_HOME="/pnpm"
ENV PATH="$PNPM_HOME:$PATH"
RUN corepack enable
```

We'll create a workspace to put our project files and copy them in:

```
WORKDIR /app
COPY . .
```

NOTE: this shouldn't copy *everything* that is inside our project folder. We don't want to copy the (hidden) git repository files or the `node_modules` folder that may have versions of packages built for a different architecture, so we use a `.dockerignore` file to exclude these:

```
.git
.gitignore
node_modules
```

At this point, our image will have the basics of a dev environment - a linux distro, with the node runtime, and the project source files. Just as when you first checkout a project from a source repo, we need to install the dependencies. We will use the cache mounts feature to have Docker cache any downloads to speed up repeated runs, otherwise it's a fairly straightforward `pnpm install` making sure to use the dependencies specified in the lockfile:

```
RUN --mount=type=cache,id=pnpm,target=/pnpm/store \
    pnpm install --frozen-lockfile
```

With the dependencies installed, we can build our app! Note that we're using the `-r` (recursive) option that will execute the `build` script in *all* packages inside our mono repo workspace. `pnpm` automatically handles the dependencies to build things in the correct order, and it doesn't matter if we have multiple packages, even multiple SvelteKit instances for apps and component libraries, they are all built:

```
RUN pnpm run -r build
```

And now the "magical" part. The pnpm deploy command transforms the app as if it was optimized for publishing to a node package repository. All dev dependecies are pruned, including other packages in our mono-repo, with just the runtime dependencies included in the `node_modules` folder. We use the `--filter` option to specify the package without our mono-repo (this is the name of the app in its `package.json` file), and the `--prod` and `out` specify the type and location of the output.

```
RUN pnpm deploy --filter=web --prod out
```

As before, there is some extra configuration needed to make this work, this time to make sure it *includes everything our app needs to execute*. SvelteKit, like most frameworks, needs more than the `node_modules` runtime dependencies ... it needs the built app output, normally output to a `build` folder. Unless we configure our app to know which files are needed for runtime, `pnpm deploy` won't know to include them.

We specify the files to include by adding a `files` section to the apps `package.json`, just as we would if we wanted to publish the outputs to npmjs.com:

```
{
    "name": "web",
    "version": "0.0.1",
    "files": [
        "build",
        "server.js"
    ],
```

For this example, I'm also including `server.js`, which I'll explain later.

At this point, we have a built output. To get a better idea of exactly what it's done, you can execute the built and deploy commands in your project folder and examine the `out` folder. Note it has no source code, just the `build` and `node_modules` folders, with the latter containing *just* the runtime dependencies.

```
pnpm run -r build
pnpm deploy --filter=web --prod out
```

Runtime Stage

At this point we have a Docker Image that we *could* deploy and execute. But it would be a waste to do this. The Image it was based off has way more dependencies than we really need, which makes it bigger than it needs to be and can represent a larger attack surface. Also, it has all the build tools and `devDependencies` that simply aren't required (please, don't run `vite` in production ...)

So instead, we will use a distroless image for the runtime base. A distroless image contains only your application and its runtime dependencies. They do not contain package managers, shells or any other programs you would expect to find in a standard Linux distribution. Fewer dependencies make for both a smaller image and a more secure image, with a reduced attack surface. It's a win-win no brainer to use.

```
FROM gcr.io/distroless/nodejs22-debian12
```

As before, we will create an area for our app to live:

```
WORKDIR /app
```

And we'll be sure to set `NODE_ENV=production` so node and any packages know how to behave.

```
ENV NODE_ENV=production
```

Because SvelteKit doesn't know what URL the Docker Image will be serving, we need to configure it. This can be done by setting an `ORIGIN` environment variable, or other variables to allow the host to be determined from the request headers. For this example we'll use the simplest option.

```
ENV ORIGIN=http://localhost:8080
```

Finally, we'll copy just the `out` folder *from* the previous build stage to our app folder. This image now contains the bare minimum to execute our app and for this example comes out at about 200Mb in size.

```
COPY --from=build /app/out/ .
```

We can then expose the port and execute the app. The entrypoint of the node distroless image is set to "node", so it expects the name of a .js file to execute. This is the `server.js` file I mentioned earlier.

```
EXPOSE 8080
CMD ["server.js"]
```

Using a custom server as the entrypoint to a Node Adapter SvelteKit app allows you greater control. You may need to setup a proxy to support Firebase signInWithRedirect or want to add a healthcheck endpoint or http compression to your server.

Here's an example of the latter (note, `http-compression` is an example of something that would be added to the runtime `dependencies` for the app):

```
import http from 'http'
import compression from 'http-compression'
import { handler } from './build/handler.js'

const compress = compression()

console.log('listening on port 8080 ...')

http
  .createServer((req, res) => {
    compress(req, res, () => {
      handler(req, res, err => {
        if (err) {
          res.writeHead(500)
          res.end(err.toString())
        } else {
          res.writeHead(404)
          res.end()
        }
      })
    })
  })
  .listen(8080)
```

Conclusion

I hope you found this useful. It seems more complex than it really is because we've gone through it line-by-line but as long as you remember the `.gitignore` and `files` in `package.json`, it's easy to apply to multiple projects and ensures the Docker Images they output will always be optimized.

There are additional enhancements that can be made such as using environment variables to specify the `PORT` to run on, and if you have multiple applications in your mono-repo project that will each need their own Docker Image (e.g. a client facing site, a tenant console, an admin site etc...) then you can add `ARG`s to the Dockerfile to allow the same one to be used for all of them.

web-development sveltekit docker pnpm

You may also like ...

Project Structure for Using Redux with Polymer 2.0
ES6 imports vs HTML imports ... fight!

Deploy a SvelteKit App to GitHub Pages
The easy way

Re-creating the SvelteKit Session Store
Not identical but close enough

Angular2 Route Security
Restricting access based on auth state or roles

Dependency Injection in Go (Golang)
Do you really need "Spring"?

SvelteKit Hydration Gotcha
Avoiding accidentally large payloads

© 2017 Captain Codeman, Inc. All rights reserved.