Utilizing tools from the Registry

As we've already seen it should be possible to containerize almost any project. Since we are in between Dev and Ops let's pretend that some developer teammates of ours did an application with a README that instructs what to install and how to run the application. Now we as the container experts can containerize it in seconds.

Open this https://github.com/docker-hy/material-applications/tree/main/rails-example-project project, and read through the README and think about how to transform it into a Dockerfile. Thanks to the README we should be able to decipher what we will need to do even if we have no clue about the language or technology!

We will need to clone the repository, which you may have already done. After that is done, let's start with a Dockerfile. We know that we need to install Ruby and whatever dependencies it has. Let's place the Dockerfile in the project root.

Dockerfile

```
# We need ruby 3.1.0. I found this from Docker Hub
FROM ruby:3.1.0

EXPOSE 3000

WORKDIR /usr/src/app
```

Ok these are the basics, we have FROM a Ruby version, EXPOSE 3000 was told at the bottom of the README and WORKDIR /usr/src/app is the convention.

The next are told to us by the README. We won't need to copy anything from outside of the container to run these:

```
# Install the correct bundler version
RUN gem install bundler:2.3.3

# Copy the files required for dependencies to be installed
COPY Gemfile* ./

# Install all dependencies
RUN bundle install
```

Here we did a quick trick to separate installing dependencies from the part where we copy the source code in. The COPY will copy both files Gemfile and Gemfile.lock to the current directory. This will help us by caching the dependency layers if we ever need to make changes to the source code. The same kind of caching trick works in many other languages or frameworks, such as Node.js.

And finally, we copy the project and follow the instructions in the README:

```
# Copy all of the source code
COPY . .

# We pick the production mode since we have no intention of developing the software inside the containe
# Run database migrations by following instructions from README
RUN rails db:migrate RAILS_ENV=production

# Precompile assets by following instructions from README
RUN rake assets:precompile

# And finally the command to run the application
CMD ["rails", "s", "-e", "production"]
```

Ok. Let's see how well monkeying the README worked for us and run the following oneliner that builds the image and then runs it with the port 3000 published:

```
docker build . -t rails-project && docker run -p 3000:3000 rails-project
```

After a while of waiting, the application starts in port 3000 in production mode... unless you have a Mac with M1 or M2 processor.

BUILDING THE IMAGE WITH A MORE RECENT MAC

If you have a more recent Mac that has the M1 or M2 processor, building the image fails:

```
 => ERROR [7/8] RUN rails db:migrate RAILS_ENV=production
------
 > [7/8] RUN rails db:migrate RAILS_ENV=production:
#11 1.142 rails aborted!
#11 1.142 LoadError: cannot load such file -- nokogiri
```

This can be fixed by changing the following line in the file *Gemfile.lock*

```
nokogiri (1.13.1-x86_64-darwin)
```

to the form:

```
nokogiri (1.14.2-arm64-darwin)
```

The reason for the problem is that the file Gemfile.lock that defines the *exact* versions of the installed libraries (or Gems in Ruby lingo) is generated with a Linux that has an Intel processor. The Gem Nokogiri has different versions for Intel and Apple M1/M2 processors and to get the right version of the Gem to a more recent Mac, it is now just easiest to make a change in the file Gemfile.lock.

Exercises 1.11-1.14

EXERCISE 1.11: SPRING

Create a Dockerfile for an old Java Spring project that can be found from the course repository.

The setup should be straightforward with the README instructions. Tips to get you started:

There are many options for running Java, you may use eg. amazoncorretto `FROM amazoncorretto:_tag_` to get Java instead of installing it manually. Pick the tag by using the README and Docker Hub page.

You've completed the exercise when you see a 'Success' message in your browser.

Submit the Dockerfile you used to run the container.

The following three exercises will start a larger project that we will configure in parts 2 and 3. They will require you to use everything you've learned up until now. If you need to modify a Dockerfile in some later exercises, feel free to do it on top of the Dockerfiles you create here.

MANDATORY EXERCISES

The next exercises are the first mandatory ones. Mandatory exercises can not be skipped.

MANDATORY EXERCISE 1.12: HELLO, FRONTEND!

A good developer creates well-written READMEs. Such that they can be used to create Dockerfiles with ease.

Clone, fork or download the project from https://github.com/docker-hy/material-applications/tree/main/example-frontend.

Create a Dockerfile for the project (example-frontend) and give a command so that the project runs in a Docker container with port 5000 exposed and published so when you start the container and navigate to http://localhost:5000 you will see message if you're successful.

- note that the port 5000 is reserved in the more recent OSX versions (Monterey, Big Sur), so you have to use some other host port

Submit the Dockerfile.

As in other exercises, do not alter the code of the project

TIPS:

- The project has install instructions in README.
- Note that the app starts to accept connections when "Accepting connections at http://localhost:5000" has been printed to the screen, this takes a few seconds
- You do not have to install anything new outside containers.
- The project might not work with too new Node.js versions

MANDATORY EXERCISE 1.13: HELLO, BACKEND!

Clone, fork or download a project from https://github.com/docker-hy/material-applications/tree/main/example-backend.

Create a Dockerfile for the project (example-backend). Start the container with port 8080 published.

When you start the container and navigate to http://localhost:8080/ping you should get a "pong" as a response.

Submit the Dockerfile and the command used.

Do not alter the code of the project

TIPS:

- you might need this
- If you have M1/M2 Mac, you might need to build the image with an extra option `docker build --platform linux/amd64 -t imagename .`

MANDATORY EXERCISE 1.14: ENVIRONMENT

Start both the frontend and the backend with the correct ports exposed and add ENV to Dockerfile with the necessary information from both READMEs (front, back).

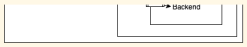
Ignore the backend configurations until the frontend sends requests to `_backend_url_/ping` when you press the button.

You know that the configuration is ready when the button for 1.14 of frontend responds and turns green.

Do not alter the code of either project

Submit the edited Dockerfiles and commands used to run.

The frontend will first talk to your browser. Then the code will be executed from your browser and that will send a message to the backend.



TIPS:

- When configuring web applications keep the browser developer console ALWAYS open, F12 or cmd+shift+I when the browser window is open. Information about configuring cross-origin requests is in the README of the backend project.
- The developer console has multiple views, the most important ones are Console and Network. Exploring the Network tab can give you a lot of information on where messages are being sent and what is received as a response!

Publishing projects

Go to https://hub.docker.com/ to create an account. You can configure Docker hub to build your images for you, but using `push` works as well.

Let's publish the youtube-dl image. Log in and navigate to your dashboard and press Create Repository. The namespace can be either your personal account or an organization account. For now, let's stick to personal accounts and write something descriptive such as youtube-dl to repository name. We will need to remember it in part 2.

Set visibility to *public*.

And the last thing we need is to authenticate our push by logging in:

```
$ docker login
```

Next, you will need to rename the image to include your username, and then you can push it:

```
$ docker tag youtube-dl <username>/<repository>
  ...

$ docker push <username>/<repository>
  ...
```

Exercises 1.15-1.16

EXERCISE 1.15: HOMEWORK

Create Dockerfile for an application or any other dockerised project in any of your own repositories and publish it to Docker Hub. This can be any project, except the clones or forks of backend-example or frontend-example.

For this exercise to be complete you have to provide the link to the project in Docker Hub, make sure you at least have a basic description and instructions for how to run the application in a README that's available through your submission.

EXERCISE 1.16: CLOUD DEPLOYMENT

It is time to wrap up this part and run a containerized app in the cloud.

You can take any web-app, eg. an example or exercise from this part, your own app, or even the course material (see devopsdockeruh/coursepage) and deploy it to some cloud provider.

There are plenty of alternatives, and most provide a free tier. Here are some alternatives that are quite simple to use:

- fly.io (easy to use but needs a credit card even in the free tier)
- render.com (bad documentation, you most likely need google)
- heroku.com (has a free student plan through GitHub Student Developer Pack)

If you know a good cloud service for the purposes of this exercise, please tell us (yes, we know about Amazon AWS, Google Cloud and Azure already...).

Submit the Dockerfile, a brief description of what you did, and a link to the running app.

Edit this page

Previous
« Interacting with the container via volumes and ports

Next
Summary »