{() => fs}

Fullstack > Part 12 > Introduction to Containers

a Introduction to Containers

The part was updated 21th Mar 2024: Create react app was replaced with Vite in the todo-frontend.

If you started the part before the update, you can see here the old material. There are some changes in the frontend configurations.

Software development includes the whole lifecycle from envisioning the software to programming and to releasing it to the end-users and even maintaining it. This part will introduce containers, a modern tool utilized in the latter parts of the software lifecycle.

Containers encapsulate your application into a single package. This package will then include all of the dependencies with the application. As a result, each container can run isolated from the other containers.

Containers prevent the application inside from accessing files and resources of the device. Developers can give the contained applications permission to access files and specify available resources. More accurately, containers are OS-level virtualization. The easiest-to-compare technology is a virtual machine (VM). VMs are used to run multiple operating systems on a single physical machine. They have to run the whole operating system, whereas a container runs the software using the host operating system. The resulting difference between VMs and containers is that there is hardly any overhead when running containers; they only need to run a single process.

As containers are relatively lightweight, at least compared to virtual machines, they can be quick to scale. And as they isolate the software running inside, it enables the software to run identically almost anywhere. As such, they are the go-to option in any cloud environment or application with more than a handful of users.

Cloud services like AWS, Google Cloud, and Microsoft Azure all support containers in multiple different forms. These include AWS Fargate and Google Cloud Run, both of which run containers as serverless - where the application container does not even need to be running if it is not used. You can also install container runtime on most machines and run containers there yourself - including your own machine.

So containers are used in cloud environment and even during development. What are the benefits of using containers? Here are two common scenarios:

Scenario 1: You are developing a new application that needs to run on the same machine as a legacy application. Both require different versions of Node installed.

You can probably use nvm, virtual machines, or dark magic to get them running at the same time. However, containers are an excellent solution as you can run both applications in their respective containers. They are isolated from each other and do not interfere.

Scenario 2: Your application runs on your machine. You need to move the application to a server.

It is not uncommon that the application just does not run on the server despite it works just fine on your machine. It may be due to some missing dependency or other differences in the environments. Here containers are an excellent solution since you can run the application in the same execution environment both on your machine and on the server. It is not perfect: different hardware can be an issue, but you can limit the differences between environments.

Sometimes you may hear about the *"Works in my container"* issue. The phrase describes a situation in which the application works fine in a container running on your machine but breaks when the container is started on a server. The phrase is a play on the infamous *"Works on my machine"* issue, which containers are often promised to solve. The situation also is most likely a usage error.

About this part

In this part, the focus of our attention will not be on the JavaScript code. Instead, we are interested in the configuration of the environment in which the software is executed. As a result, the exercises may not contain any coding, the applications are available to you through GitHub and your tasks will include configuring them. The exercises are to be submitted to *a single GitHub repository* which will include all of the source code and configuration you do during this part.

You will need basic knowledge of Node, Express, and React. Only the core parts, 1 through 5, are required to be completed before this part.

Exercise 12.1

Warning

Since we are stepping right outside of our comfort zone as JavaScript developers, this part may require you to take a detour and familiarize yourself with shell / command line / command prompt / terminal before getting started.

If you have only ever used a graphical user interface and never touched e.g. Linux or terminal on Mac, or if you get stuck in the first exercises we recommend doing the Part 1 of "Computing tools for CS studies" first: https://tkt-lapio.github.io/en/ . Skip the section for "SSH connection" and Exercise 11. Otherwise, it includes everything you are going to need to get started here!

Exercise 12.1: Using a computer (without graphical user interface)

Step 1: Read the text below the Warning header.

Step 2: Download this repository and make it your submission repository for this part.

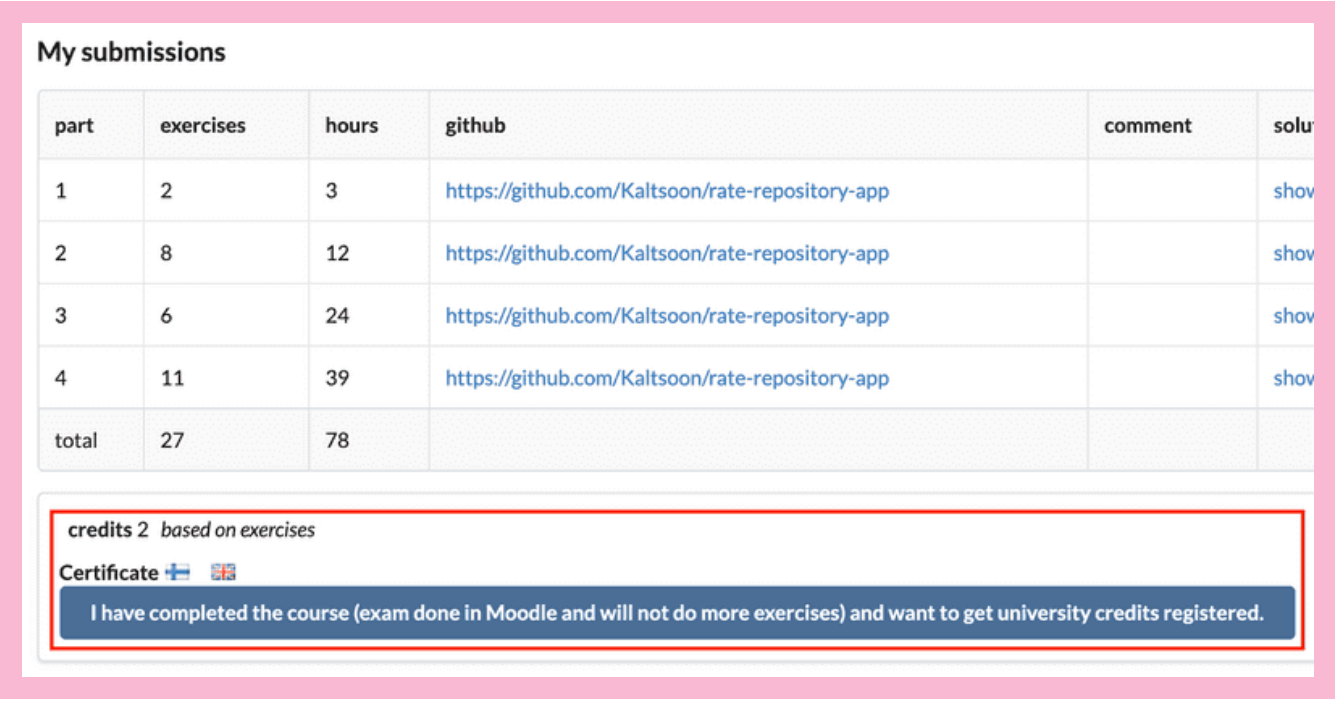
Step 3: Run *curl http://helsinki.fi* and save the output into a file. Save that file into your repository as file *script-answers/exercise12_1.txt*. The directory *script-answers* was created in the previous step.

Submitting exercises and earning credits

Submit the exercises via the submissions system just like in the previous parts. Exercises in this part are submitted *to its own course instance*.

Completing this part on containers will get you 1 credit. Note that you need to do all the exercises for earning the credit or the certificate.

Once you have completed the exercises and want to get the credits, let us know through the exercise submission system that you have completed the course:



You can download the certificate for completing this part by clicking one of the flag icons. The flag icon corresponds to the language of the certificate.

Tools of the trade

The basic tools you are going to need vary between operating systems:

- WSL 2 terminal on Windows
- Terminal on Mac
- Command Line on a Linux

Installing everything required for this part

We will begin by installing the required software. The installation step will be one of the possible obstacles. As we are dealing with OS-level virtualization, the tools will require superuser access on the computer. They will have access to your operating systems kernel.

The material is built around Docker, a set of products that we will use for containerization and the management of containers. Unfortunately, if you can not install Docker you probably can not complete this part.

As the install instructions depend on your operating system, you will have to find the correct install instructions from the link below. Note that they may have multiple different options for your operating system.

- Get Docker

Now that that headache is hopefully over, let's make sure that our versions match. Yours may have a bit higher numbers than here:

```
$ docker -v
Docker version 25.0.3, build 4debf41
```

copy

Containers and images

There are two core concepts in this part: *container* and *image*. They are easy to confuse with one another.

A *container* is a runtime instance of an *image*.

Both of the following statements are true:

- Images include all of the code, dependencies and instructions on how to run the application
- Containers package software into standardized units

It is no wonder they are easily mixed up.

To help with the confusion, almost everyone uses the word container to describe both. But you can never actually build a container or download one since containers only exist during runtime. Images, on the other hand, are **immutable** files. As a result of the immutability, you can not edit an image after you have created one. However, you can use existing images to create *a new image* by adding new layers on top of the existing ones.

Cooking metaphor:

- Image is pre-cooked, frozen treat.
- Container is the delicious treat.

Docker is the most popular containerization technology and pioneered the standards most containerization technologies use today. In practice, Docker is a set of products that help us to manage images and containers. This set of products will enable us to leverage all of the benefits of containers. For example, the Docker engine will take care of turning the immutable files called images into containers.

For managing the Docker containers, there is also a tool called Docker Compose that allows one to **orchestrate** (control) multiple containers at the same time. In this part we shall use Docker Compose to set up a complex local development environment. In the final version of the development environment that we set up, even installing Node to our machine is not a requirement anymore.

There are several concepts we need to go over. But we will skip those for now and learn about Docker first!

Let us start with the command *docker container run* that is used to run images within a container. The command structure is the following: `container run IMAGE-NAME` that we will tell Docker to create a container from an image. A particularly nice feature of the command is that it can run a container even if the image to run is not downloaded on our device yet.

Let us run the command

```
$ docker container run hello-world
```

copy

There will be a lot of output, but let's split it into multiple sections, which we can decipher together. The lines are numbered by me so that it is easier to follow the explanation. Your output will not have the numbers.

```
1. Unable to find image 'hello-world:latest' locally
2. latest: Pulling from library/hello-world
3. b8dfde127a29: Pull complete
4. Digest: sha256:5122f6204b6a3596e048758cabba3c46b1c937a46b5be6225b835d091b90e46c
5. Status: Downloaded newer image for hello-world:latest
```

copy

Because the image *hello-world* was not found on our machine, the command first downloaded it from a free registry called Docker Hub. You can see the Docker Hub page of the image with your browser here: https://hub.docker.com/_/hello-world

The first part of the message states that we did not have the image "hello-world:latest" yet. This reveals a bit of detail about images themselves; image names consist of multiple parts, kind of like an URL. An image name is in the following format:

- `registry/organisation/image:tag`

In this case the 3 missing fields defaulted to:

- `index.docker.io/library/hello-world:latest`

The second row shows the organisation name, "library" where it will get the image. In the Docker Hub url, the "library" is shortened to _.

The 3rd and 5th rows only show the status. But the 4th row may be interesting: each image has a unique digest based on the *layers* from which the image is built. In practice, each step or command that was used in building the image creates a unique layer. The digest is used by Docker to identify that an image is the same. This is done when you try to pull the same image again.

So the result of using the command was a pull and then output information about the **image**. After that, the status told us that a new version of *hello-world:latest* was indeed downloaded. You can try pulling the image with `docker image pull hello-world` and see what happens.

The following output was from the container itself. It also explains what happened when we ran `docker container run hello-world`.

```
Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker container run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

The output contains a few new things for us to learn. *Docker daemon* is a background service that makes sure the containers are running, and we use the *Docker client* to interact with the daemon. We now have interacted with the first image and created a container from the image. During the execution of that container, we received the output.

Exercise 12.2

Some of these exercises do not require you to write any code or configurations to a file. In these exercises you should use script command to record the commands you have used; try it yourself with `script` to start recording, `echo "hello"` to generate some output, and `exit` to stop recording. It saves your actions into a file named "typescript" (that has nothing to do with the TypeScript programming language, the name is just a coincidence).

If `script` does not work, you can just copy-paste all commands you used into a text file.

Exercise 12.2: Running your second container

> Use `script` to record what you do, save the file as script-answers/exercise12_2.txt

The hello-world output gave us an ambitious task to do. Do the following

Step 1. Run an Ubuntu container with the command given by hello-world

The step 1 will connect you straight into the container with bash. You will have access to all of the files and tools inside of the container. The following steps are run within the container:

Step 2. Create directory */usr/src/app*

Step 3. Create a file */usr/src/app/index.js*

Step 4. Run *exit* to quit from the container

Google should be able to help you with creating directories and files.

Ubuntu image

The command you just used to run the Ubuntu container, `docker container run -it ubuntu bash` , contains a few additions to the previously run hello-world. Let's see the --help to get a better understanding. I'll cut some of the output so we can focus on the relevant parts.

```
$ docker container run --help

Usage:  docker container run [OPTIONS] IMAGE [COMMAND] [ARG...]
Run a command in a new container
```

```
Options:
  ...
  -i, --interactive                 Keep STDIN open even if not attached
  -t, --tty                         Allocate a pseudo-TTY
  ...
```

The two options, or flags, `-it` make sure we can interact with the container. After the options, we defined that image to run is `ubuntu`. Then we have the command `bash` to be executed inside the container when we start it.

You can try other commands that the ubuntu image might be able to execute. As an example try `docker container run --rm ubuntu ls`. The `ls` command will list all of the files in the directory and `--rm` flag will remove the container after execution. Normally containers are not deleted automatically.

Let's continue with our first Ubuntu container with the **index.js** file inside of it. The container has stopped running since we exited it. We can list all of the containers with `container ls -a`, the `-a` (or --all) will list containers that have already been exited.

```
$ docker container ls -a                                                 copy
CONTAINER ID   IMAGE     COMMAND   CREATED          STATUS
NAMES
b8548b9faec3   ubuntu    "bash"    3 minutes ago    Exited (0) 6 seconds ago
hopeful_clarke
```

> *Editor's note: that the command* `docker container ls` *has also a shorter form* `docker ps`, I prefer the shorter one.

We have two options when addressing a container. The identifier in the first column can be used to interact with the container almost always. Plus, most commands accept the container name as a more human-friendly method of working with them. The name of the container was automatically generated to be **"hopeful_clarke"** in my case.

The container has already exited, yet we can start it again with the start command that will accept the id or name of the container as a parameter: `start` *`CONTAINER-ID-OR-CONTAINER-NAME`*.

```
$ docker start hopeful_clarke                                            copy
hopeful_clarke
```

The start command will start the same container we had previously. Unfortunately, we forgot to start it with the flag `--interactive` (that can also be written `-i`) so we can not interact with it.

The container is actually up and running as the command `container ls -a` shows, but we just can not communicate with it:

```
$ docker container ls -a
CONTAINER ID   IMAGE     COMMAND   CREATED          STATUS
NAMES
b8548b9faec3   ubuntu    "bash"    7 minutes ago    Up (0) 15 seconds ago
hopeful_clarke
```

Note that we can also execute the command without the flag `-a` to see just those containers that are running:

```
$ docker container ls
CONTAINER ID   IMAGE     COMMAND   CREATED          STATUS             NAMES
8f5abc55242a   ubuntu    "bash"    8 minutes ago    Up 1 minutes       hopeful_clarke
```

Let's kill it with the `kill` *`CONTAINER-ID-OR-CONTAINER-NAME`* command and try again.

```
$ docker kill hopeful_clarke
hopeful_clarke
```

`docker kill` sends a signal SIGKILL to the process forcing it to exit, and that causes the container to stop. We can check it's status with `container ls -a` :

```
$ docker container ls -a
CONTAINER ID   IMAGE     COMMAND   CREATED             STATUS                      NAMES
b8548b9faec3   ubuntu     "bash"   26 minutes ago      Exited 2 seconds ago
hopeful_clarke
```

Now let us start the container again, but this time in interactive mode:

```
$ docker start -i hopeful_clarke
root@b8548b9faec3:/#
```

Let's edit the file *index.js* and add in some JavaScript code to execute. We are just missing the tools to edit the file. Nano will be a good text editor for now. The install instructions were found from the first result of Google. We will omit using sudo since we are already root.

```
root@b8548b9faec3:/# apt-get update
root@b8548b9faec3:/# apt-get -y install nano
root@b8548b9faec3:/# nano /usr/src/app/index.js
```

Now we have Nano installed and can start editing files!

Exercise 12.3 - 12.4

Exercise 12.3: Ubuntu 101

> Use `script` to record what you do, save the file as script-answers/exercise12_3.txt

Edit the `/usr/src/app/index.js` file inside the container with the now installed Nano and add the following line

```
console.log('Hello World')
```

copy

If you are not familiar with Nano you can ask for help in the chat or Google.

Exercise 12.4: Ubuntu 102

> Use `script` to record what you do, save the file as script-answers/exercise12_4.txt

Install Node while inside the container and run the index file with `node /usr/src/app/index.js` in the container.

The instructions for installing Node are sometimes hard to find, so here is something you can copy-paste:

```
curl -sL https://deb.nodesource.com/setup_20.x | bash
apt install -y nodejs
```

copy

You will need to install the `curl` into the container. It is installed in the same way as you did with `nano`.

After the installation, ensure that you can run your code inside the container with the command

```
root@b8548b9faec3:/# node /usr/src/app/index.js
Hello World
```

copy

Other Docker commands

Now that we have Node installed in the container, we can execute JavaScript in the container! Let's create a new image from the container. The command

```
commit CONTAINER-ID-OR-CONTAINER-NAME NEW-IMAGE-NAME
```

copy

will create a new image that includes the changes we have made. You can use `container diff` to check for the changes between the original image and container before doing so.

```
$ docker commit hopeful_clarke hello-node-world
```

copy

You can list your images with `image ls` :

```
$ docker image ls
REPOSITORY                                  TAG         IMAGE ID       CREATED
SIZE
hello-node-world                            latest      eef776183732   9 minutes ago
252MB
ubuntu                                      latest      1318b700e415   2 weeks ago
72.8MB
hello-world                                 latest      d1165f221234   5 months ago
13.3kB
```

copy

You can now run the new image as follows:

```
docker run -it hello-node-world bash
root@4d1b322e1aff:/# node /usr/src/app/index.js
```

copy

There are multiple ways to achieve the same conclusion. Let's go through a better solution. We will clean the slate with `container rm` to remove the old container.

```
$ docker container ls -a
CONTAINER ID   IMAGE     COMMAND   CREATED          STATUS                     NAMES
b8548b9faec3   ubuntu    "bash"    31 minutes ago   Exited (0) 9 seconds ago
hopeful_clarke

$ docker container rm hopeful_clarke
hopeful_clarke
```

copy

Create a file *index.js* to your current directory and write `console.log('Hello, World')` inside it. No need for containers yet.

Next, let's skip installing Node altogether. There are plenty of useful Docker images in Docker Hub ready for our use. Let's use the image https://hub.docker.com/_/node, which has Node already installed. We only need to pick a version.

By the way, the `container run` accepts `--name` flag that we can use to give a name for the container.

```
$ docker container run -it --name hello-node node:20 bash
```

copy

Let us create a directory for the code inside the container:

```
root@77d1023af893:/# mkdir /usr/src/app
```

copy

While we are inside the container on this terminal, open another terminal and use the `container cp` command to copy file from your own machine to the container.

```
$ docker container cp ./index.js hello-node:/usr/src/app/index.js
```

copy

And now we can run `node /usr/src/app/index.js` in the container. We can commit this as another new image, but there is an even better solution. The next section will be all about building your images like a pro.

Propose changes to material

Part 11
Previous part

Part 12b
Next part

About course

Course contents

FAQ

Partners

Challenge


HOUSTON

{() => fs}

Fullstack > Part 12 > Building and configuring environments

b Building and configuring environments

The part was updated 21th Mar 2024: Create react app was replaced with Vite in the todo-frontend.

If you started the part before the update, you can see here the old material. There are some changes in the frontend configurations.

In the previous section, we used two different base images: ubuntu and node and did some manual work to get a simple "Hello, World!" running. The tools and commands we learned during that process will be helpful. In this section, we will learn how to build images and configure environments for our applications. We will start with a regular Express/Node.js backend and build on top of that with other services, including a MongoDB database.

Dockerfile

Instead of modifying a container by copying files inside, we can create a new image that contains the "Hello, World!" application. The tool for this is the Dockerfile. Dockerfile is a simple text file that contains all of the instructions for creating an image. Let's create an example Dockerfile from the "Hello, World!" application.

If you did not already, create a directory on your machine and create a file called *Dockerfile* inside that directory. Let's also put an *index.js* containing `console.log('Hello, World!')` next to the Dockerfile. Your directory structure should look like this:

```
├── index.js
└── Dockerfile
```

inside that Dockerfile we will tell the image three things:

- Use the `node:20` as the base for our image
- Include the index.js inside the image, so we don't need to manually copy it into the container
- When we run a container from the image, use Node to execute the index.js file.

The wishes above will translate into a basic Dockerfile. The best location to place this file is usually at the root of the project.

The resulting *Dockerfile* looks like this:

```
FROM node:20

WORKDIR /usr/src/app

COPY ./index.js ./index.js

CMD node index.js
```

FROM instruction will tell Docker that the base for the image should be node:20. COPY instruction will copy the file *index.js* from the host machine to the file with the same name in the image. CMD instruction tells what happens when `docker run` is used. CMD is the default command that can then be overwritten with the parameter given after the image name. See `docker run --help` if you forgot.

The WORKDIR instruction was slipped in to ensure we don't interfere with the contents of the image. It will guarantee all of the following commands will have */usr/src/app* set as the working directory. If the directory doesn't exist in the base image, it will be automatically created.

If we do not specify a WORKDIR, we risk overwriting important files by accident. If you check the root (`/`) of the node:20 image with `docker run node:20 ls` , you can notice all of the directories and files that are already included in the image.

Now we can use the command `docker build` to build an image based on the Dockerfile. Let's spice up the command with one additional flag: `-t` , this will help us name the image:

```
$ docker build -t fs-hello-world .
[+] Building 3.9s (8/8) FINISHED
...
```

So the result is "Docker please build with tag (you may think the tag to be the name of the resulting image) *fs-hello-world* the Dockerfile in this directory". You can point to any Dockerfile, but in our case, a simple dot will mean the Dockerfile in *this* directory. That is why the command ends with a period. After the build is finished, you can run it with `docker run fs-hello-world` :

```
$ docker run fs-hello-world
Hello, World
```

copy

As images are just files, they can be moved around, downloaded and deleted. You can list the images you have locally with `docker image ls` , delete them with `docker image rm` . See what other command you have available with `docker image --help` .

One more thing: in above it was mentioned that the default command, defined by the CMD in the Dockerfile, can be overridden if needed. We could e.g. open a bash session to the container and observe it's content:

```
$ docker run -it fs-hello-world bash
root@2932e32dbc09:/usr/src/app# ls
index.js
root@2932e32dbc09:/usr/src/app#
```

copy

More meaningful image

Moving an Express server to a container should be as simple as moving the "Hello, World!" application inside a container. The only difference is that there are more files. Thankfully `COPY` instruction can handle all that. Let's delete the index.js and create a new Express server. Lets use express-generator to create a basic Express application skeleton.

```
$ npx express-generator
  ...

  install dependencies:
    $ npm install

  run the app:
    $ DEBUG=playground:* npm start
```

copy

First, let's run the application to get an idea of what we just created. Note that the command to run the application may be different from you, my directory was called playground.

```
$ npm install
$ DEBUG=playground:* npm start
```

copy

```
  playground:server Listening on port 3000 +0ms
```

Great, so now we can navigate to http://localhost:3000 and the app is running there.

Containerizing that should be relatively easy based on the previous example.

- Use node as base
- Set working directory so we don't interfere with the contents of the base image
- Copy ALL of the files in this directory to the image
- Start with DEBUG=playground:* npm start

Let's place the following Dockerfile at the root of the project:

```
FROM node:20

WORKDIR /usr/src/app

COPY . .

CMD DEBUG=playground:* npm start
```

Let's build the image from the Dockerfile and then run it:

```
docker build -t express-server .
docker run -p 3123:3000 express-server
```

The `-p` flag in the run command will inform Docker that a port from the host machine should be opened and directed to a port in the container. The format for is `-p host-port:application-port`.

The application is now running! Let's test it by sending a GET request to http://localhost:3123/.

> If yours doesn't work, skip to the next section. There is an explanation why it may not work even if you followed the steps correctly.

Shutting the app down is a headache at the moment. Use another terminal and `docker kill` command to kill the application. The `docker kill` will send a kill signal (SIGKILL) to the application to force it to shut down. It needs the name or the id of the container as an argument.

By the way, when using the id as the argument, the beginning of the ID is enough for Docker to know which container we mean.

```
$ docker container ls
  CONTAINER ID   IMAGE            COMMAND                  CREATED         STATUS
```

```
PORTS                                       NAMES
  48096ca3ffec   express-server   "docker-entrypoint.s…"   9 seconds ago   Up 6 seconds
0.0.0.0:3123->3000/tcp, :::3123->3000/tcp   infallible_booth

$ docker kill 48
  48
```

In the future, let's use the same port on both sides of `-p` . Just so we don't have to remember which one we happened to choose.

Fixing potential issues we created by copy-pasting

There are a few steps we need to change to create a more comprehensive Dockerfile. It may even be that the above example doesn't work in all cases because we skipped an important step.

When we ran npm install on our machine, in some cases the **Node package manager** may install operating system specific dependencies during the install step. We may accidentally move non-functional parts to the image with the COPY instruction. This can easily happen if we copy the *node_modules* directory into the image.

This is a critical thing to keep in mind when we build our images. It's best to do most things, such as to run `npm install` during the build process *inside the container* rather than doing those prior to building. The easy rule of thumb is to only copy files that you would push to GitHub. Build artefacts or dependencies should not be copied since those can be installed during the build process.

We can use *.dockerignore* to solve the problem. The file .dockerignore is very similar to .gitignore, you can use that to prevent unwanted files from being copied to your image. The file should be placed next to the Dockerfile. Here is a possible content of a *.dockerignore*

```
.dockerignore
.gitignore
node_modules
Dockerfile
```

copy

However, in our case, the .dockerignore isn't the only thing required. We will need to install the dependencies during the build step. The `Dockerfile` changes to:

```
FROM node:20

WORKDIR /usr/src/app

COPY . .

RUN npm install
CMD DEBUG=playground:* npm start
```

copy

The npm install can be risky. Instead of using npm install, npm offers a much better tool for installing dependencies, the `ci` command.

Differences between ci and install:

- install may update the package-lock.json
- install may install a different version of a dependency if you have ^ or ~ in the version of the dependency.
- ci will delete the node_modules folder before installing anything
- ci will follow the package-lock.json and does not alter any files

So in short: `ci` creates reliable builds, while `install` is the one to use when you want to install new dependencies.

As we are not installing anything new during the build step, and we don't want the versions to suddenly change, we will use `ci` :

```
FROM node:20

WORKDIR /usr/src/app

COPY . .

RUN npm ci
CMD DEBUG=playground:* npm start
```

Even better, we can use `npm ci --omit=dev` to not waste time installing development dependencies.

> As you noticed in the comparison list; npm ci will delete the node_modules folder so creating the .dockerignore did not matter. However, .dockerignore is an amazing tool when you want to optimize your build process. We will talk briefly about these optimizations later.

Now the Dockerfile should work again, try it with `docker build -t express-server . && docker run -p 3123:3000 express-server`

> Note that we are here chaining two bash commands with &&. We could get (nearly) the same effect by running both commands separately. When chaining commands with && if one command fails, the next ones in the chain will not be executed.

We set an environment variable `DEBUG=playground:*` during CMD for the npm start. However, with Dockerfiles we could also use the instruction ENV to set environment variables. Let's do that:

```
FROM node:20
```

```
WORKDIR /usr/src/app

COPY . .

RUN npm ci

ENV DEBUG=playground:*
CMD npm start
```

> *If you're wondering what the DEBUG environment variable does, read here.*

Dockerfile best practices

There are 2 rules of thumb you should follow when creating images:

- Try to create as **secure** of an image as possible
- Try to create as **small** of an image as possible

Smaller images are more secure by having less attack surface area, and smaller images also move faster in deployment pipelines.

Snyk has a great list of 10 best practices for Node/Express containerization. Read those here.

One big carelessness we have left is running the application as root instead of using a user with lower privileges. Let's do a final fix to the Dockerfile:

```
FROM node:20

WORKDIR /usr/src/app

COPY --chown=node:node . .
RUN npm ci

ENV DEBUG=playground:*

USER node
CMD npm start
```

Exercise 12.5.

Exercise 12.5: Containerizing a Node application

The repository you cloned or copied in the first exercise contains a todo-app. See the todo-app/todo-backend and read through the README. We will not touch the todo-frontend yet.

- Step 1. Containerize the todo-backend by creating a *todo-app/todo-backend/Dockerfile* and building an image.
- Step 2. Run the todo-backend image with the correct ports open. Make sure the visit counter increases when used through a browser in http://localhost:3000/ (or some other port if you configure so)

Tip: Run the application outside of a container to examine it before starting to containerize.

Using Docker compose

In the previous section, we created an Express server and knew that it runs in port 3000, and ran it with `docker build -t express-server . && docker run -p 3000:3000 express-server`. This already looks like something you would need to put into a script to remember. Fortunately, Docker offers us a better solution.

Docker compose is another fantastic tool, which can help us to manage containers. Let's start using compose as we learn more about containers as it will help us save some time with the configuration.

Now we can turn the previous spell into a yaml file. The best part about yaml files is that you can save these to a Git repository!

Create the file **docker-compose.yml** and place it at the root of the project, next to the Dockerfile. The file content is

```
version: '3.8'            # Version 3.8 is quite new and should work

services:
  app:                    # The name of the service, can be anything
    image: express-server # Declares which image to use
    build: .              # Declares where to build if image is not found
    ports:                # Declares the ports to publish
      - 3000:3000
```

The meaning of each line is explained as a comment. If you want to see the full specification see the documentation.

Now we can use `docker compose up` to build and run the application. If we want to rebuild the images we can use `docker compose up --build`.

You can also run the application in the background with `docker compose up -d` (`-d` for detached) and close it with `docker compose down`.

> *Note that some older Docker versions (especially in Windows) do not support the command* `docker compose`*. One way to circumvent this problem is to install the stand alone command*

> `docker-compose` *that works mostly similarly to* `docker compose` *. However, the preferable fix is to update the Docker to a more recent version.*

Creating files like `docker-compose.yml` that *declare* what you want instead of script files that you need to run in a specific order / a specific number of times is often a great practice.

Exercise 12.6.

Exercise 12.6: Docker compose

Create a *todo-app/todo-backend/docker-compose.yml* file that works with the Node application from the previous exercise.

The visit counter is the only feature that is required to be working.

Utilizing containers in development

When you are developing software, containerization can be used in various ways to improve your quality of life. One of the most useful cases is by bypassing the need to install and configure tools twice.

It may not be the best option to move your entire development environment into a container, but if that's what you want it's certainly possible. We will revisit this idea at the end of this part. But until then, *run the Node application itself outside of containers*.

The application we met in the previous exercises uses MongoDB. Let's explore Docker Hub to find a MongoDB image. Docker Hub is the default place where Docker pulls the images from, you can use other registries as well, but since we are already knee-deep in Docker it's a good choice. With a quick search, we can find https://hub.docker.com/_/mongo

Create a new yaml called *todo-app/todo-backend/docker-compose.dev.yml* that looks like following:

```
version: '3.8'

services:
  mongo:
    image: mongo
    ports:
      - 3456:27017
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: example
      MONGO_INITDB_DATABASE: the_database
```

The meaning of the two first environment variables defined above is explained on the Docker Hub page:

> *These variables, used in conjunction, create a new user and set that user's password. This user is created in the admin authentication database and given the role of root, which is a "superuser" role.*

The last environment variable `MONGO_INITDB_DATABASE` will tell MongoDB to create a database with that name.

You can use `-f` flag to specify a *file* to run the Docker Compose command with e.g.

```
docker compose -f docker-compose.dev.yml up
```

copy

Now that we may have multiple compose files, it's useful.

Now start the MongoDB with `docker compose -f docker-compose.dev.yml up -d`. With `-d` it will run it in the background. You can view the output logs with `docker compose -f docker-compose.dev.yml logs -f`. There the `-f` will ensure we *follow* the logs.

As said previously, currently we **do not** want to run the Node application inside a container. Developing while the application itself is inside a container is a challenge. We will explore that option later in this part.

Run the good old `npm install` first on your machine to set up the Node application. Then start the application with the relevant environment variable. You can modify the code to set them as the defaults or use the .env file. There is no hurt in putting these keys to GitHub since they are only used in your local development environment. I'll just throw them in with the `npm run dev` to help you copy-paste.

```
$ MONGO_URL=mongodb://localhost:3456/the_database npm run dev
```

copy

This won't be enough; we need to create a user to be authorized inside of the container. The url http://localhost:3000/todos leads to an authentication error:

```
[nodemon] 2.0.12
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node ./bin/www`
/Users/mluukkai/dev/fs-ci-lokakuu/repo/todo-app/todo-
backend/node_modules/mongodb/lib/cmap/connection.js:272
          callback(new MongoError(document));
                   ^
MongoError: command find requires authentication
```

copy

```
    at MessageStream.messageHandler (/Users/mluukkai/dev/fs-ci-lokakuu/repo/todo-app/todo-backend/node_modules/mongodb/lib/cmap/connection.js:272:20)
```

Bind mount and initializing the database

In the MongoDB Docker Hub page under "Initializing a fresh instance" is the info on how to execute JavaScript to initialize the database and a user for it.

The exercise project has a file *todo-app/todo-backend/mongo/mongo-init.js* with contents:

```
db.createUser({
  user: 'the_username',
  pwd: 'the_password',
  roles: [
    {
      role: 'dbOwner',
      db: 'the_database',
    },
  ],
});

db.createCollection('todos');

db.todos.insert({ text: 'Write code', done: true });
db.todos.insert({ text: 'Learn about containers', done: false });
```

This file will initialize the database with a user and a few todos. Next, we need to get it inside the container at startup.

We could create a new image FROM mongo and COPY the file inside, or we can use a bind mount to mount the file *mongo-init.js* to the container. Let's do the latter.

Bind mount is the act of binding a file (or directory) on the host machine to a file (or directory) in the container. A bind mount is done by adding a `-v` flag with `container run`. The syntax is `-v FILE-IN-HOST:FILE-IN-CONTAINER`. Since we already learned about Docker Compose let's skip that. The bind mount is declared under key *volumes* in docker-compose-yml. Otherwise the format is the same, first host and then container:

```
  mongo:
    image: mongo
    ports:
     - 3456:27017
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: example
      MONGO_INITDB_DATABASE: the_database
    volumes:
```

```
      - ./mongo/mongo-init.js:/docker-entrypoint-initdb.d/mongo-init.js
```

The result of the bind mount is that the file *mongo-init.js* in the mongo folder of the host machine is the same as the *mongo-init.js* file in the container's /docker-entrypoint-initdb.d directory. Changes to either file will be available in the other. We don't need to make any changes during runtime. But this will be the key to software development in containers.

Run `docker compose -f docker-compose.dev.yml down --volumes` to ensure that nothing is left and start from a clean slate with `docker compose -f docker-compose.dev.yml up` to initialize the database.

If you see an error like this:

```
mongo_database | failed to load: /docker-entrypoint-initdb.d/mongo-init.js
mongo_database | exiting with code -3
```

copy

you may have a read permission problem. They are not uncommon when dealing with volumes. In the above case, you can use `chmod a+r mongo-init.js` , which will give everyone read access to that file. Be careful when using `chmod` since granting more privileges can be a security issue. Use the `chmod` only on the mongo-init.js on your computer.

Now starting the Express application with the correct environment variable should work:

```
MONGO_URL=mongodb://the_username:the_password@localhost:3456/the_database npm run dev
```

copy

Let's check that the http://localhost:3000/todos returns the two todos we inserted in the initialization. We can and *should* use Postman to test the basic functionality of the app, such as adding or deleting a todo.

Still problems?

For some reason, the initialization of Mongo has caused problems for many.

If the app does not work and you still end up with the following error

```
/Users/mluukkai/dev/fs-ci-lokakuu/repo/todo-app/todo-
backend/node_modules/mongodb/lib/cmap/connection.js:272
          callback(new MongoError(document));
                   ^
MongoError: command find requires authentication
    at MessageStream.messageHandler (/Users/mluukkai/dev/fs-ci-lokakuu/repo/todo-
app/todo-backend/node_modules/mongodb/lib/cmap/connection.js:272:20)
```

copy

run these commands:

```
docker compose -f docker-compose.dev.yml down --volumes
docker image rm mongo
```

After these, try to start Mongo again.

If the problem persists, let us drop the idea of a volume altogether and copy the initialization script to a custom image. Create the following *Dockerfile* to the directory *todo-app/todo-backend/mongo*

```
FROM mongo

COPY ./mongo-init.js /docker-entrypoint-initdb.d/
```

Build it to an image with the command

```
docker build -t initialized-mongo .
```

Change now the *docker-compose.dev.yml* to use the new image:

```
  mongo:
    image: initialized-mongo
    ports:
      - 3456:27017
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: example
      MONGO_INITDB_DATABASE: the_database
```

Now the app should finally work.

Persisting data with volumes

By default, database containers are not going to preserve our data. When you close the database container you *may or may not* be able to get the data back.

> Mongo is actually a rare case in which the container indeed does preserve the data. This happens, since the developers who made the Docker image for Mongo have defined a volume to be used. This line in the Dockerfile will instruct Docker to preserve the data in a volume.

There are two distinct methods to store the data:

- Declaring a location in your filesystem (called bind mount)
- Letting Docker decide where to store the data (volume)

The first choice is preferable in most cases whenever one *really* needs to avoid the data being deleted.

Let's see both in action with Docker compose. Let us start with *bind mount:*

```
services:
  mongo:
    image: mongo
    ports:
     - 3456:27017
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: example
      MONGO_INITDB_DATABASE: the_database
    volumes:
      - ./mongo/mongo-init.js:/docker-entrypoint-initdb.d/mongo-init.js
      - ./mongo_data:/data/db
```

The above will create a directory called `mongo_data` to your local filesystem and map it into the container as `/data/db`. This means the data in `/data/db` is stored outside of the container but still accessible by the container! Just remember to add the directory to .gitignore.

A similar outcome can be achieved with a *named volume:*

```
services:
  mongo:
    image: mongo
    ports:
     - 3456:27017
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: example
      MONGO_INITDB_DATABASE: the_database
    volumes:
      - ./mongo/mongo-init.js:/docker-entrypoint-initdb.d/mongo-init.js
      - mongo_data:/data/db

volumes:
  mongo_data:
```

Now the volume is created but managed by Docker. After starting the application (`docker compose -f docker-compose.dev.yml up`) you can list the volumes with `docker volume ls`, inspect one of them with `docker volume inspect` and even delete them with `docker volume rm`:

```
$ docker volume ls
DRIVER    VOLUME NAME
local     todo-backend_mongo_data
$ docker volume inspect todo-backend_mongo_data
[
    {
        "CreatedAt": "2024-19-03T12:52:11Z",
        "Driver": "local",
        "Labels": {
            "com.docker.compose.project": "todo-backend",
            "com.docker.compose.version": "1.29.2",
            "com.docker.compose.volume": "mongo_data"
        },
        "Mountpoint": "/var/lib/docker/volumes/todo-backend_mongo_data/_data",
        "Name": "todo-backend_mongo_data",
        "Options": null,
        "Scope": "local"
    }
]
```

The named volume is still stored in your local filesystem but figuring out *where* may not be as trivial as with the previous option.

Exercise 12.7.

Exercise 12.7: Little bit of MongoDB coding

Note that this exercise assumes that you have done all the configurations made in the material after exercise 12.5. You should still run the todo-app backend *outside a container*, just the MongoDB is containerized for now.

The todo application has no proper implementation of routes for getting one todo (GET */todos/:id*) and updating one todo (PUT */todos/:id*). Fix the code.

Debugging issues in containers

> *When coding, you most likely end up in a situation where everything is broken.*
>
> - Matti Luukkainen

When developing with containers, we need to learn new tools for debugging, since we can not just "console.log" everything. When code has a bug, you may often be in a state where at least something

works, so you can work forward from that. Configuration most often is in either of two states: 1. working or 2. broken. We will go over a few tools that can help when your application is in the latter state.

When developing software, you can safely progress step by step, all the time verifying that what you have coded behaves as expected. Often, this is not the case when doing configurations. The configuration you may be writing can be broken until the moment it is finished. So when you write a long docker-compose.yml or Dockerfile and it does not work, you need to take a moment and think about the various ways you could confirm something is working.

Question Everything is still applicable here. As said in part 3 : The key is to be systematic. Since the problem can exist anywhere, *you must question everything*, and eliminate all possible sources of error one by one.

For myself, the most valuable method of debugging is stopping and thinking about what I'm trying to accomplish instead of just bashing my head at the problem. Often there is a simple, alternate, solution or quick google search that will get me moving forward.

exec

The Docker command `exec` is a heavy hitter. It can be used to jump right into a container when it's running.

Let's start a web server in the background and do a little bit of debugging to get it running and displaying the message "Hello, exec!" in our browser. Let's choose Nginx which is, among other things, a server capable of serving static HTML files. It has a default index.html that we can replace.

```
$ docker container run -d nginx
```

copy

Ok, now the questions are:

- Where should we go with our browser?
- Is it even running?

We know how to answer the latter: by listing the running containers.

```
$ docker container ls
CONTAINER ID   IMAGE   COMMAND  CREATED     STATUS    PORTS     NAMES
3f831a57b7cc   nginx   ...      3 sec ago   Up 2 sec  80/tcp    keen_darwin
```

copy

Yes! We got the first question answered as well. It seems to listen on port 80, as seen on the output above.

Let's shut it down and restart with the `-p` flag to have our browser access it.

```
$ docker container stop keen_darwin
$ docker container rm keen_darwin

$ docker container run -d -p 8080:80 nginx
```

> ***Editor's note_*** *when doing development, it is* ***essential*** *to constantly follow the container logs. I'm usually not running containers in a detached mode (that is with -d) since it requires a bit of an extra effort to open the logs.*
>
> *When I'm 100% sure that everything works... no, when I'm 200% sure, then I might relax a bit and start the containers in detached mode. Until everything again falls apart and it is time to open the logs again.*

Let's look at the app by going to http://localhost:8080. It seems that the app is showing the wrong message! Let's hop right into the container and fix this. Keep your browser open, we won't need to shut down the container for this fix. We will execute bash inside the container, the flags `-it` will ensure that we can interact with the container:

```
$ docker container ls
CONTAINER ID   IMAGE     COMMAND  PORTS                  NAMES
7edcb36aff08   nginx     ...      0.0.0.0:8080->80/tcp   wonderful_ramanujan

$ docker exec -it wonderful_ramanujan bash
root@7edcb36aff08:/#
```

Now that we are in, we need to find the faulty file and replace it. Quick Google tells us that file itself is `/usr/share/nginx/html/index.html`.

Let's move to the directory and delete the file

```
root@7edcb36aff08:/# cd /usr/share/nginx/html/
root@7edcb36aff08:/# rm index.html
```

Now, if we go to http://localhost:8080/ we know that we deleted the correct file. The page shows 404. Let's replace it with one containing the correct contents:

```
root@7edcb36aff08:/# echo "Hello, exec!" > index.html
```

Refresh the page, and our message is displayed! Now we know how exec can be used to interact with the containers. Remember that all of the changes are lost when the container is deleted. To preserve the changes, you must use `commit` just as we did in previous section.

Exercise 12.8.

Exercise 12.8: Mongo command-line interface

> Use `script` to record what you do, save the file as script-answers/exercise12_8.txt

While the MongoDB from the previous exercise is running, access the database with the Mongo command-line interface (CLI). You can do that using docker exec. Then add a new todo using the CLI.

The command to open CLI when inside the container is `mongosh`

The Mongo CLI will require the username and password flags to authenticate correctly. Flags `-u root -p example` should work, the values are from the docker-compose.dev.yml.

- Step 1: Run MongoDB
- Step 2: Use docker exec to get inside the container
- Step 3: Open Mongo cli

When you have connected to the Mongo cli you can ask it to show dbs inside:

```
> show dbs
admin         0.000GB
config        0.000GB
local         0.000GB
the_database  0.000GB
```

copy

To access the correct database:

```
> use the_database
```

copy

And finally to find out the collections:

```
> show collections
todos
```

copy

We can now access the data in those collections:

```
> db.todos.find({})
[
  {
    _id: ObjectId("633c270ba211aa5f7931f078"),
    text: 'Write code',
    done: false
  },
  {
    _id: ObjectId("633c270ba211aa5f7931f079"),
    text: 'Learn about containers',
    done: false
  }
]
```

Insert one new todo with the text: "Increase the number of tools in my toolbelt" with the status done as *false*. Consult the documentation to see how the addition is done.

Ensure that you see the new todo both in the Express app and when querying from Mongo CLI.

Redis

Redis is a key-value database. In contrast to eg. MongoDB, the data stored in key-value storage has a bit less structure, there are eg. no collections or tables, it just contains junks of data that can be fetched based on the *key* that was attached to the data (the *value*).

By default, Redis works *in-memory*, which means that it does not store data persistently.

An excellent use case for Redis is to use it as a cache. Caches are often used to store data that is otherwise slow to fetch and save the data until it's no longer valid. After the cache becomes invalid, you would then fetch the data again and store it in the cache.

Redis has nothing to do with containers. But since we are already able to add *any* 3rd party service to your applications, why not learn about a new one?

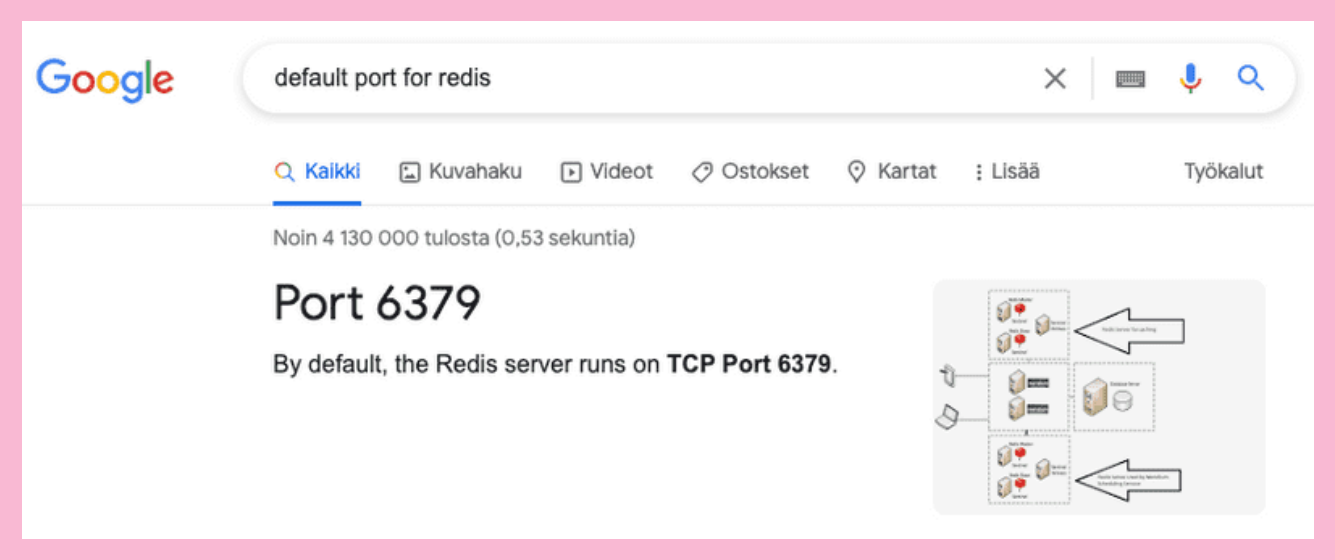
Exercises 12.9. - 12.11.

Exercise 12.9: Set up Redis for the project

The Express server has already been configured to use Redis, and it is only missing the `REDIS_URL` environment variable. The application will use that environment variable to connect to the Redis. Read through the Docker Hub page for Redis, add Redis to the *todo-app/todo-backend/docker-compose.dev.yml* by defining another service after mongo:

```
services:
  mongo:
    ...
  redis:
    ???
```

Since the Docker Hub page doesn't have all the info, we can use Google to aid us. The default port for Redis is found by doing so:



We won't have any idea if the configuration works unless we try it. The application will not start using Redis by itself, that shall happen in the next exercise.

Once Redis is configured and started, restart the backend and give it the *REDIS_URL*, which has the form *redis://host:port*

```
$ REDIS_URL=insert-redis-url-here
MONGO_URL=mongodb://the_username:the_password@localhost:3456/the_database npm run dev
```

You can now test the configuration by adding the line

```
const redis = require('../redis')
```

to the Express server eg. in file *routes/index.js*. If nothing happens, the configuration is done right. If not, the server crashes:

```
events.js:291
      throw er; // Unhandled 'error' event
```

```
      ^

Error: Redis connection to localhost:637 failed - connect ECONNREFUSED 127.0.0.1:6379
    at TCPConnectWrap.afterConnect [as oncomplete] (net.js:1144:16)
Emitted 'error' event on RedisClient instance at:
    at RedisClient.on_error (/Users/mluukkai/opetus/docker-fs/container-app/express-app/node_modules/redis/index.js:342:14)
    at Socket.<anonymous> (/Users/mluukkai/opetus/docker-fs/container-app/express-app/node_modules/redis/index.js:223:14)
    at Socket.emit (events.js:314:20)
    at emitErrorNT (internal/streams/destroy.js:100:8)
    at emitErrorCloseNT (internal/streams/destroy.js:68:3)
    at processTicksAndRejections (internal/process/task_queues.js:80:21) {
  errno: -61,
  code: 'ECONNREFUSED',
  syscall: 'connect',
  address: '127.0.0.1',
  port: 6379
}
[nodemon] app crashed - waiting for file changes before starting...
```

Exercise 12.10:

The project already has https://www.npmjs.com/package/redis installed and two functions "promisified" - getAsync and setAsync.

- setAsync function takes in key and value, using the key to store the value.
- getAsync function takes in a key and returns the value in a promise.

Implement a todo counter that saves the number of created todos to Redis:

- Step 1: Whenever a request is sent to add a todo, increment the counter by one.
- Step 2: Create a GET /statistics endpoint where you can ask for the usage metadata. The format should be the following JSON:

```
{
  "added_todos": 0
}
```

Exercise 12.11:

> Use `script` to record what you do, save the file as script-answers/exercise12_11.txt

If the application does not behave as expected, direct access to the database may be beneficial in pinpointing problems. Let us try out how redis-cli can be used to access the database.

- Go to the Redis container with `docker exec` and open the redis-cli.
- Find the key you used with `KEYS *`

- Check the value of the key with the command GET
- Set the value of the counter to 9001, find the right command from here
- Make sure that the new value works by refreshing the page http://localhost:3000/statistics
- Create a new todo with Postman and ensure from redis-cli that the counter has increased accordingly
- Delete the key from the cli and ensure that the counter works when new todos are added

Persisting data with Redis

In the previous section, it was mentioned that *by default* Redis does not persist the data. However, the persistence is easy to toggle on. We only need to start the Redis with a different command, as instructed by the Docker hub page:

```
services:
  redis:
    # Everything else
    command: ['redis-server', '--appendonly', 'yes'] # Overwrite the CMD
    volumes: # Declare the volume
      - ./redis_data:/data
```

The data will now be persisted to the directory *redis_data* of the host machine. Remember to add the directory to .gitignore!

Other functionality of Redis

In addition to the GET, SET and DEL operations on keys and values, Redis can do also quite a lot more. It can for example automatically expire keys, which is a very useful feature when Redis is used as a cache.

Redis can also be used to implement the so-called publish-subscribe (or PubSub) pattern which is an asynchronous communication mechanism for distributed software. In this scenario, Redis works as a *message broker* between two or more services. Some of the services are *publishing* messages by sending those to Redis, which on arrival of a message, informs the parties that have *subscribed* to those messages.

Exercise 12.12.

Exercise 12.12: Persisting data in Redis

Check that the data is not persisted by default: after running

```
docker compose -f docker-compose.dev.yml down
docker compose -f docker-compose.dev.yml up
```

copy

the counter value is reset to 0.

Then create a volume for Redis data (by modifying *todo-app/todo-backend/docker-compose.dev.yml*) and make sure that the data survives after running

```
docker compose -f docker-compose.dev.yml down
docker compose -f docker-compose.dev.yml up
```

copy

Propose changes to material

Part 12a
Previous part

Part 12c
Next part

About course

Course contents

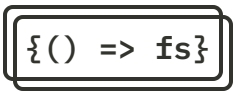
FAQ

Partners

Challenge


UNIVERSITY OF HELSINKI


HOUSTON



c Basics of Orchestration

The part was updated 21th Mar 2024: Create react app was replaced with Vite in the todo-frontend.

If you started the part before the update, you can see here the old material. There are some changes in the frontend configurations.

We have now a basic understanding of Docker and can use it to easily set up eg. a database for our app. Let us now move our focus to the frontend.

React in container

Let's create and containerize a React application next. We start with the usual steps

```
$ npm create vite@latest hello-front -- --template react
$ cd hello-front
$ npm install
```

The next step is to turn the JavaScript code and CSS, into production-ready static files. Vite already has `build` as an npm script so let's use that:

```
$ npm run build
  ...

  Creating an optimized production build...
```

```
  ...
  The build folder is ready to be deployed.
  ...
```

Great! The final step is figuring out a way to use a server to serve the static files. As you may know, we could use our `express.static` with the Express server to serve the static files. I'll leave that as an exercise for you to do at home. Instead, we are going to go ahead and start writing our Dockerfile:

```
FROM node:20

WORKDIR /usr/src/app

COPY . .

RUN npm ci

RUN npm run build
```

That looks about right. Let's build it and see if we are on the right track. Our goal is to have the build succeed without errors. Then we will use bash to check inside of the container to see if the files are there.

```
$ docker build . -t hello-front
 => [4/5] RUN npm ci
 => [5/5] RUN npm run
 ...
 => => naming to docker.io/library/hello-front

$ docker run -it hello-front bash

root@98fa9483ee85:/usr/src/app# ls
  Dockerfile  README.md  dist  index.html  node_modules  package-lock.json  package.json
public  src  vite.config.js

root@98fa9483ee85:/usr/src/app# ls dist
  assets    index.html  vite.svg
```

A valid option for serving static files now that we already have Node in the container is `serve`. Let's try installing serve and serving the static files while we are inside the container.

```
root@98fa9483ee85:/usr/src/app# npm install -g serve

  added 89 packages in 2s

root@98fa9483ee85:/usr/src/app# serve dist
```

```
┌──────────────────────────────────────────┐
│                                          │
│   Serving!                               │
│                                          │
│   - Local:    http://localhost:3000      │
│   - Network:  http://172.17.0.2:3000     │
│                                          │
└──────────────────────────────────────────┘
```

Great! Let's ctrl+c and exit out and then add those to our Dockerfile.

The installation of serve turns into a RUN in the Dockerfile. This way the dependency is installed during the build process. The command to serve the *dist* directory will become the command to start the container:

```
FROM node:20

WORKDIR /usr/src/app

COPY . .

RUN npm ci

RUN npm run build

RUN npm install -g serve
CMD ["serve", "dist"]
```

Our CMD now includes square brackets and as a result, we now use the so-called *exec form* of CMD. There are actually **three** different forms for the CMD out of which the exec form is preferred. Read the documentation for more info.

When we now build the image with `docker build . -t hello-front` and run it with `docker run -p 5001:3000 hello-front` , the app will be available in http://localhost:5001.

Using multiple stages

While serve is a *valid* option we can do better. A good goal is to create Docker images so that they do not contain anything irrelevant. With a minimal number of dependencies, images are less likely to break or become vulnerable over time.

Multi-stage builds are designed to split the build process into many separate stages, where it is possible to limit what parts of the image files are moved between the stages. That opens possibilities for limiting the size of the image since not all by-products of the build are necessary for the resulting image. Smaller images are faster to upload and download and they help reduce the number of vulnerabilities your software may have.

With multi-stage builds, a tried and true solution like Nginx can be used to serve static files without a lot of headaches. The Docker Hub page for Nginx tells us the required info to open the ports and "Hosting

some simple static content".

Let's use the previous Dockerfile but change the FROM to include the name of the stage:

```
# The first FROM is now a stage called build-stage
FROM node:20 AS build-stage
WORKDIR /usr/src/app

COPY . .

RUN npm ci

RUN npm run build

# This is a new stage, everything before this is gone, except the files we want to COPY
FROM nginx:1.25-alpine
# COPY the directory build from build-stage to /usr/share/nginx/html
# The target location here was found from the Docker hub page
COPY --from=build-stage /usr/src/app/dist /usr/share/nginx/html
```

We have declared also *another stage* where only the relevant files of the first stage (the *dist* directory, that contains the static content) are copied.

After we build it again, the image is ready to serve the static content. The default port will be 80 for Nginx, so something like `-p 8000:80` will work, so the parameters of the run command need to be changed a bit.

Multi-stage builds also include some internal optimizations that may affect your builds. As an example, multi-stage builds skip stages that are not used. If we wish to use a stage to replace a part of a build pipeline, like testing or notifications, we must pass **some** data to the following stages. In some cases this is justified: copy the code from the testing stage to the build stage. This ensures that you are building the tested code.

Exercises 12.13 - 12.14.

Exercise 12.13: Todo application frontend

Finally, we get to the todo-frontend. View the todo-app/todo-frontend and read through the README.

Start by running the frontend outside the container and ensure that it works with the backend.

Containerize the application by creating *todo-app/todo-frontend/Dockerfile* and use `ENV` instruction to pass `VITE_BACKEND_URL` to the application and run it with the backend. The backend should still be running outside a container.

Note that you need to set `VITE_BACKEND_URL` before building the frontend, otherwise, it does not get defined in the code!

Exercise 12.14: Testing during the build process

One interesting possibility to utilize multi-stage builds is to use a separate build stage for testing. If the testing stage fails, the whole build process will also fail. Note that it may not be the best idea to move *all testing* to be done during the building of an image, but there may be *some* containerization-related tests where it might be worth considering.

Extract a component *Todo* that represents a single todo. Write a test for the new component and add running tests into the build process.

You can add a new build stage for the test if you wish to do so. If you do so, remember to read the last paragraph before exercise 12.13 again!

Development in containers

Let's move the whole todo application development to a container. There are a few reasons why you would want to do that:

- To keep the environment similar between development and production to avoid bugs that appear only in the production environment
- To avoid differences between developers and their personal environments that lead to difficulties in application development
- To help new team members hop in by having them install container runtime - and requiring nothing else.

These all are great reasons. The tradeoff is that we may encounter some unconventional behavior when we aren't running the applications like we are used to. We will need to do at least two things to move the application to a container:

- Start the application in development mode
- Access the files with VS Code

Let's start with the frontend. Since the Dockerfile will be significantly different from the production Dockerfile let's create a new one called *dev.Dockerfile*.

Note we shall use the name *dev.Dockerfile* for development configurations and *Dockerfile* otherwise.

Starting the Vite in development mode should be easy. Let's start with the following:

```
FROM node:20

WORKDIR /usr/src/app
```

copy

```
COPY . .

# Change npm ci to npm install since we are going to be in development mode
RUN npm install

# npm start is the command to start the application in development mode
CMD ["npm", "run", "dev", "--", "--host"]
```

> Note the extra parameters `-- --host` in the `CMD`. Those are needed to expose the development server to be visible outside the Docker network. By default the development server is exposed only to localhost, and despite we access the frontend still using the localhost address, it is in reality attached to the Docker network.

During build the flag `-f` will be used to tell which file to use, it would otherwise default to Dockerfile, so the following command will build the image:

```
docker build -f ./dev.Dockerfile -t hello-front-dev .
```

copy

The Vite will be served in port 5173, so you can test that it works by running a container with that port published.

The second task, accessing the files with VSCode, is not yet taken care of. There are at least two ways of doing this:

- The Visual Studio Code Remote - Containers extension
- Volumes, the same thing we used to preserve data with the database

Let's go over the latter since that will work with other editors as well. Let's do a trial run with the flag `-v`, and if that works, then we will move the configuration to a docker-compose file. To use the `-v`, we will need to tell it the current directory. The command `pwd` should output the path to the current directory for you. Try this with `echo $(pwd)` in your command line. We can use that as the left side for `-v` to map the current directory to the inside of the container or you can use the full directory path.

```
$ docker run -p 5173:5173 -v "$(pwd):/usr/src/app/" hello-front-dev
> todo-vite@0.0.0 dev
> vite --host

  VITE v5.1.6  ready in 130 ms
```

copy

Now we can edit the file *src/App.js*, and the changes should be hot-loaded to the browser!

If you have a Mac with M1/M2 processor, the above command fails. In the error message, we notice the following:

```
Error: Cannot find module @rollup/rollup-linux-arm64-gnu
```

The problem is the library `rollup` that has its own version for all operating systems and processor architectures. Due to the volume mapping, the container is now using the `node_modules` from the host machine directory where the `@rollup/rollup-darwin-arm64` (the version suitable Mac M1/M2) is installed, so the right version of the library for the container `@rollup/rollup-linux-arm64-gnu` is not found.

There are several ways to fix the problem. Let's use the perhaps simplest one. Start the container with bash as the command, and run the `npm install` inside the container:

```
$ docker run -it -v "$(pwd):/usr/src/app/" front-dev bash
root@b83e9040b91d:/usr/src/app# npm install
```

Now both versions of the library rollup are installed and the container works!

Next, let's move the config to the file *docker-compose.dev.yml*. That file should be at the root of the project as well:

```
services:
  app:
    image: hello-front-dev
    build:
      context: . # The context will pick this directory as the "build context"
      dockerfile: dev.Dockerfile # This will simply tell which dockerfile to read
    volumes:
      - ./:/usr/src/app # The path can be relative, so ./ is enough to say "the same
location as the docker-compose.yml"
    ports:
      - 5173:5173
    container_name: hello-front-dev # This will name the container hello-front-dev
```

With this configuration, `docker compose up` can run the application in development mode. You don't even need Node installed to develop it!

Note we shall use the name *docker-compose.dev.yml* for development environment compose files, and the default name *docker-compose.yml* otherwise.

Installing new dependencies is a headache for a development setup like this. One of the better options is to install the new dependency **inside** the container. So instead of doing e.g. `npm install axios`, you have to do it in the running container e.g. `docker exec hello-front-dev npm install axios`, or add it to the package.json and run `docker build` again.

Exercise 12.15

Exercise 12.15: Set up a frontend development environment

Create *todo-frontend/docker-compose.dev.yml* and use volumes to enable the development of the todo-frontend while it is running *inside* a container.

Communication between containers in a Docker network

The Docker Compose tool sets up a network between the containers and includes a DNS to easily connect two containers. Let's add a new service to the Docker Compose and we shall see how the network and DNS work.

Busybox is a small executable with multiple tools you may need. It is called "The Swiss Army Knife of Embedded Linux", and we definitely can use it to our advantage.

Busybox can help us to debug our configurations. So if you get lost in the later exercises of this section, you should use Busybox to find out what works and what doesn't. Let's use it to explore what was just said. That containers are inside a network and you can easily connect between them. Busybox can be added to the mix by changing *docker-compose.dev.yml* to:

```
services:
  app:
    image: hello-front-dev
    build:
      context: .
      dockerfile: dev.Dockerfile
    volumes:
      - ./:/usr/src/app
    ports:
      - 5173:5173
    container_name: hello-front-dev
  debug-helper:
    image: busybox
```

The Busybox container won't have any process running inside so we can not `exec` in there. Because of that, the output of `docker compose up` will also look like this:

```
$ docker compose -f docker-compose.dev.yml up
0.0s
Attaching to front-dev, debug-helper-1
```

```
debug-helper-1 exited with code 0
front-dev       |
front-dev       | > todo-vite@0.0.0 dev
front-dev       | > vite --host
front-dev       |
front-dev       |
front-dev       |   VITE v5.2.2  ready in 153 ms
```

This is expected as it's just a toolbox. Let's use it to send a request to hello-front-dev and see how the DNS works. While the hello-front-dev is running, we can do the request with `wget` since it's a tool included in Busybox to send a request from the debug-helper to hello-front-dev.

With Docker Compose we can use `docker compose run SERVICE COMMAND` to run a service with a specific command. Command wget requires the flag `-O` with `-` to output the response to the stdout:

```
$ docker compose -f docker-compose.dev.yml run debug-helper wget -O - http://app:5173
```

copy

```
Connecting to app:5173 (192.168.240.3:5173)
writing to stdout
<!doctype html>
<html lang="en">
  <head>
    <script type="module">
      ...
```

The URL is the interesting part here. We simply said to connect to port 5173 of the service *app*. The *app* is the name of the service specified in the *docker-compose.dev.yml*:

```
services:
  app:
    image: hello-front-dev
    build:
      context: .
      dockerfile: dev.Dockerfile
    volumes:
      - ./:/usr/src/app
    ports:
      - 5173:5173
    container_name: hello-front-dev
```

copy

The port used is the port from which the application is available in that container, also specified in the *docker-compose.dev.yml*. The port does not need to be published for other services in the same network to be able to connect to it. The "ports" in the docker-compose file are only for external access.

Let's change the port configuration in the *docker-compose.dev.yml* to emphasize this:

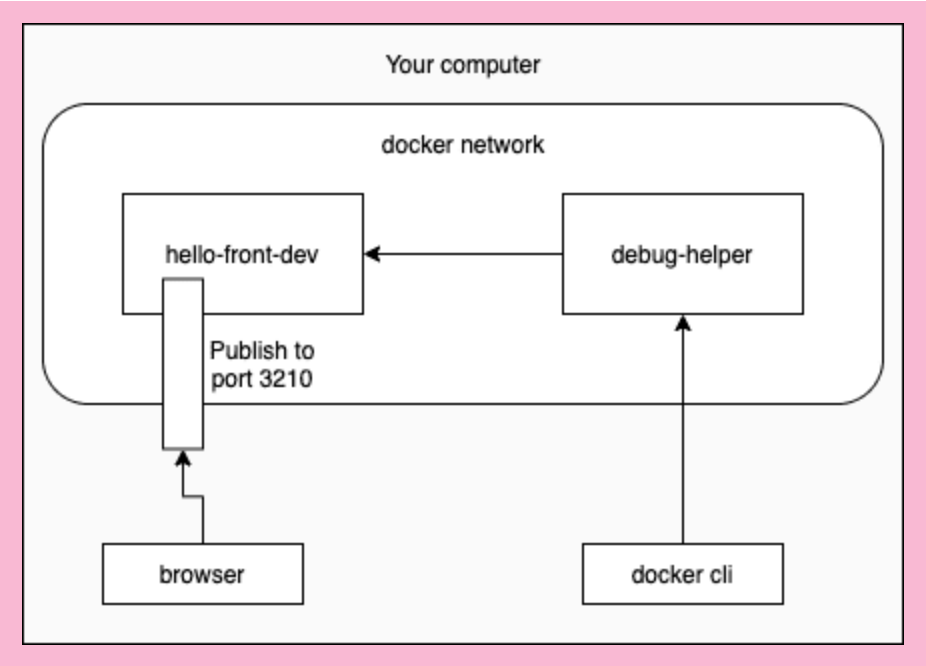
```
services:
  app:
    image: hello-front-dev
    build:
      context: .
      dockerfile: dev.Dockerfile
    volumes:
      - ./:/usr/src/app
    ports:
      - 3210:5173
    container_name: hello-front-dev
  debug-helper:
    image: busybox
```

With `docker compose up` the application is available in http://localhost:3210 at the *host machine*, but the command

```
docker compose  -f docker-compose.dev.yml run debug-helper wget -O - http://app:5173
```

works still since the port is still 5173 within the docker network.

The below image illustrates what happens. The command `docker compose run` asks debug-helper to send the request within the network. While the browser in the host machine sends the request from outside of the network.



Now that you know how easy it is to find other services in the *docker-compose.yml* and we have nothing to debug we can remove the debug-helper and revert the ports to 5173:5173 in our compose file.

Exercise 12.16

Exercise 12.16: Run todo-backend in a development container

Use volumes and Nodemon to enable the development of the todo app backend while it is running *inside* a container. Create a *todo-backend/dev.Dockerfile* and edit the *todo-backend/docker-compose.dev.yml*.

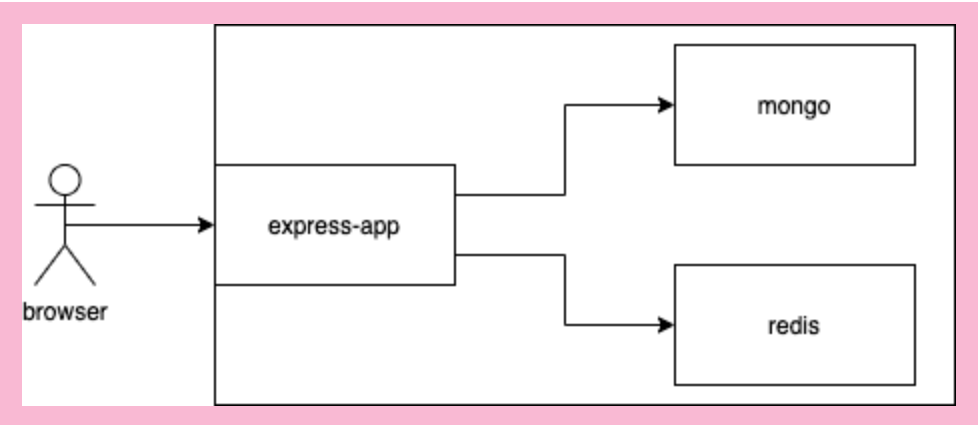
You will also need to rethink the connections between backend and MongoDB / Redis. Thankfully Docker Compose can include environment variables that will be passed to the application:

```
services:
  server:
    image: ...
    volumes:
      - ...
    ports:
      - ...
    environment:
      - REDIS_URL=redisurl_here
      - MONGO_URL=mongourl_here
```

copy

The URLs are purposefully wrong, you will need to set the correct values. Remember to *look all the time what happens in console*. If and when things blow up, the error messages hint at what might be broken.

Here is a possibly helpful image illustrating the connections within the docker network:



Communications between containers in a more ambitious environment

Next, we will configure a reverse proxy to our docker-compose.dev.yml. According to wikipedia

> *A reverse proxy is a type of proxy server that retrieves resources on behalf of a client from one or more servers. These resources are then returned to the client, appearing as if they originated from the reverse proxy server itself.*

So in our case, the reverse proxy will be the single point of entry to our application, and the final goal will be to set both the React frontend and the Express backend behind the reverse proxy.

There are multiple different options for a reverse proxy implementation, such as Traefik, Caddy, Nginx, and Apache (ordered by initial release from newer to older).

Our pick is `Nginx`.

Let us now put the *hello-frontend* behind the reverse proxy.

Create a file *nginx.dev.conf* in the project root and take the following template as a starting point. We will need to do minor edits to have our application running:

```
# events is required, but defaults are ok
events { }

# A http server, listening at port 80
http {
  server {
    listen 80;

    # Requests starting with root (/) are handled
    location / {
      # The following 3 lines are required for the hot loading to work (websocket).
      proxy_http_version 1.1;
      proxy_set_header Upgrade $http_upgrade;
      proxy_set_header Connection 'upgrade';

      # Requests are directed to http://localhost:5173
      proxy_pass http://localhost:5173;
    }
  }
}
```

Note we are using the familiar naming convention also for Nginx, *nginx.dev.conf* for development configurations, and the default name *nginx.conf* otherwise.

Next, create an Nginx service in the *docker-compose.dev.yml* file. Add a volume as instructed in the Docker Hub page where the right side is `:/etc/nginx/nginx.conf:ro`, the final ro declares that the volume will be *read-only*:

```
services:
  app:
    # ...
```

```
  nginx:
    image: nginx:1.20.1
    volumes:
      - ./nginx.dev.conf:/etc/nginx/nginx.conf:ro
    ports:
      - 8080:80
    container_name: reverse-proxy
    depends_on:
      - app # wait for the frontend container to be started
```

with that added, we can run `docker compose -f docker-compose.dev.yml up` and see what happens.

```
$ docker container ls
CONTAINER ID   IMAGE            COMMAND  PORTS                    NAMES
a02ae58f3e8d   nginx:1.20.1     ...      0.0.0.0:8080->80/tcp     reverse-proxy
5ee0284566b4   hello-front-dev  ...      0.0.0.0:5173->5173/tcp   hello-front-dev
```

Connecting to http://localhost:8080 will lead to a familiar-looking page with 502 status.

This is because directing requests to http://localhost:5173 leads to nowhere as the Nginx container does not have an application running in port 5173. By definition, localhost refers to the current computer used to access it. Since the localhost is unique for each container, it always points to the container itself.

Let's test this by going inside the Nginx container and using curl to send a request to the application itself. In our usage curl is similar to wget, but won't need any flags.

```
$ docker exec -it reverse-proxy bash

root@374f9e62bfa8:/# curl http://localhost:80
  <html>
  <head><title>502 Bad Gateway</title></head>
  ...
```

To help us, Docker Compose has set up a network when we ran `docker compose up` . It has also added all of the containers mentioned in the *docker-compose.dev.yml* to the network. A DNS makes sure we can find the other containers in the network. The containers are each given two names: the service name and the container name and both can be used to communicate with a container.

Since we are inside the container, we can also test the DNS! Let's curl the service name (app) in port 5173

```
root@374f9e62bfa8:/# curl http://app:5173
<!doctype html>
<html lang="en">
```

```
  <head>
    <script type="module" src="/@vite/client"></script>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite + React</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.jsx"></script>
  </body>
</html>
```

That is it! Let's replace the proxy_pass address in nginx.dev.conf with that one.

One more thing: we added an option depends_on to the configuration that ensures that the `nginx` container is not started before the frontend container `app` is started:

```
services:
  app:
    # ...
  nginx:
    image: nginx:1.20.1
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
    ports:
      - 8080:80
    container_name: reverse-proxy
    depends_on:
      - app
```

If we do not enforce the starting order with *depends_on* there a risk that Nginx fails on startup since it tries to resolve all DNS names that are referred in the config file:

```
http {
  server {
    listen 80;

    location / {
      proxy_http_version 1.1;
      proxy_set_header Upgrade $http_upgrade;
      proxy_set_header Connection 'upgrade';

      proxy_pass http://app:5173;
    }
  }
}
```

Note that *depends_on* does not guarantee that the service in the depended container is ready for action, it just ensures that the container has been started (and the corresponding entry is added to DNS). If a service needs to wait another service to become ready before the startup, other solutions should be used.

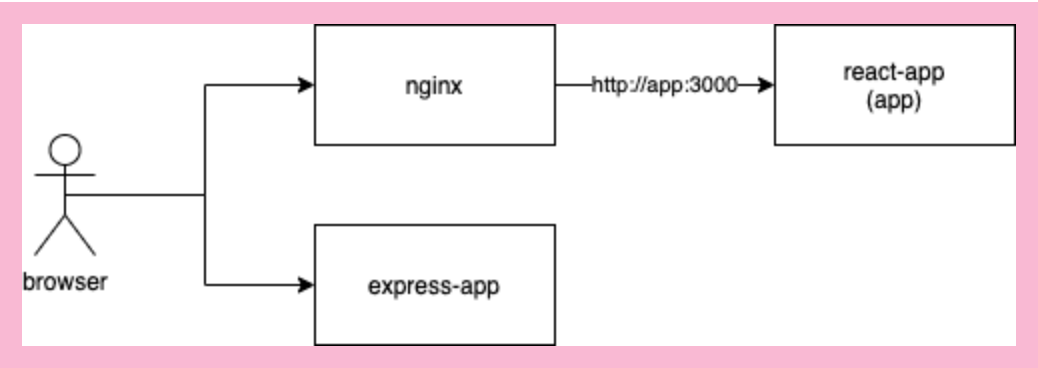
Exercises 12.17. - 12.19.

Exercise 12.17: Set up an Nginx reverse proxy server in front of todo-frontend

We are going to put the Nginx server in front of both todo-frontend and todo-backend. Let's start by creating a new docker-compose file *todo-app/docker-compose.dev.yml* and *todo-app/nginx.dev.conf*.

```
todo-app
├── todo-frontend
├── todo-backend
├── nginx.dev.conf
└── docker-compose.dev.yml
```

Add the services Nginx and todo-frontend built with *todo-app/todo-frontend/dev.Dockerfile* into the *todo-app/docker-compose.dev.yml*.



In this and the following exercises you **do not** need to support the the build option, that is, the command

```
docker compose -f docker-compose.dev.yml up --build
```

It is enough to build the frontend and backend at their own repositories.

Exercise 12.18: Configure the Nginx server to be in front of todo-backend

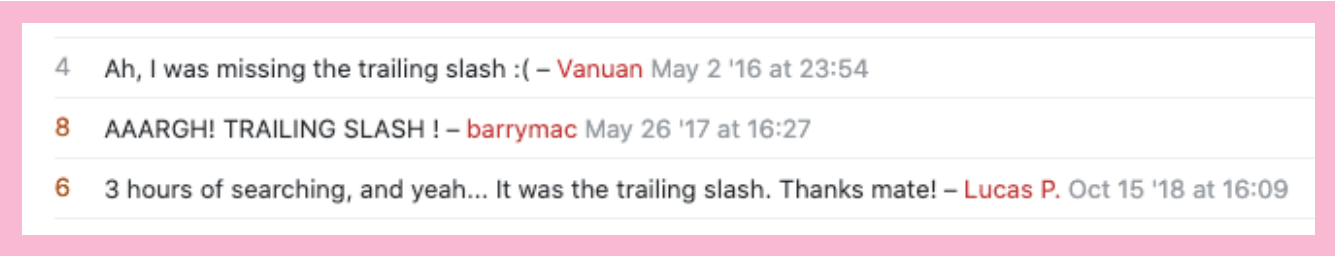
Add the service todo-backend to the docker-compose file *todo-app/docker-compose.dev.yml* in development mode.

Add a new location to the *nginx.dev.conf* so that requests to `/api` are proxied to the backend. Something like this should do the trick:

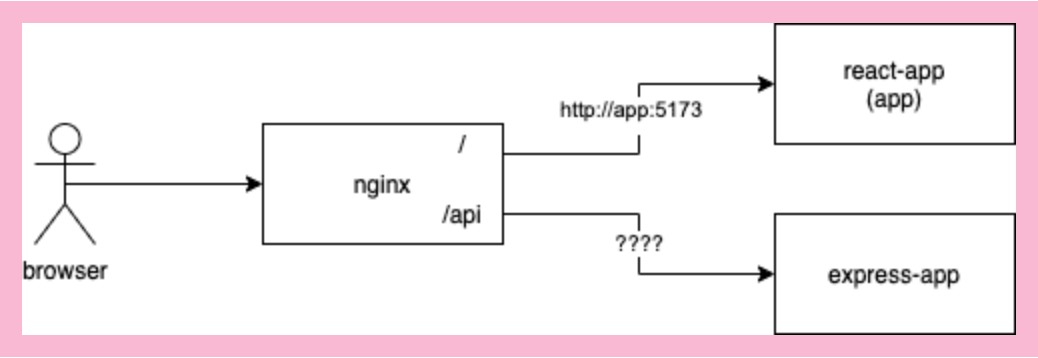
```
server {
  listen 80;

  # Requests starting with root (/) are handled
  location / {
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection 'upgrade';

    proxy_pass ...
  }

  # Requests starting with /api/ are handled
  location /api/ {
    proxy_pass ...
  }
}
```

The `proxy_pass` directive has an interesting feature with a trailing slash. As we are using the path `/api` for location but the backend application only answers in paths `/` or `/todos` we will want the `/api` to be removed from the request. In other words, even though the browser will send a GET request to `/api/todos/1` we want the Nginx to proxy the request to `/todos/1`. Do this by adding a trailing slash `/` to the URL at the end of `proxy_pass`.

This is a common issue



This illustrates what we are looking for and may be helpful if you are having trouble:

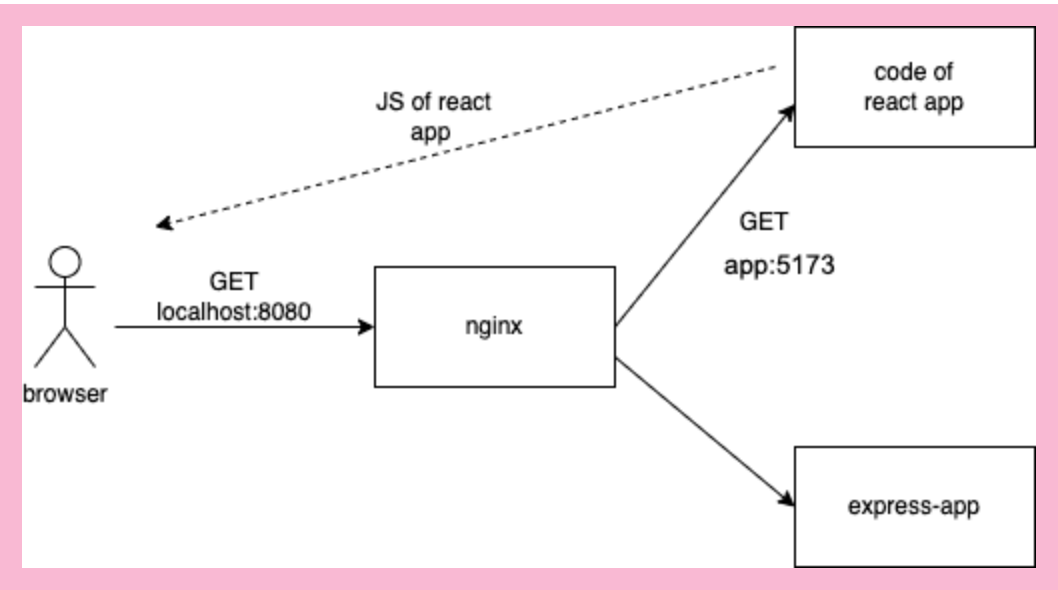


Exercise 12.19: Connect the services, todo-frontend with todo-backend

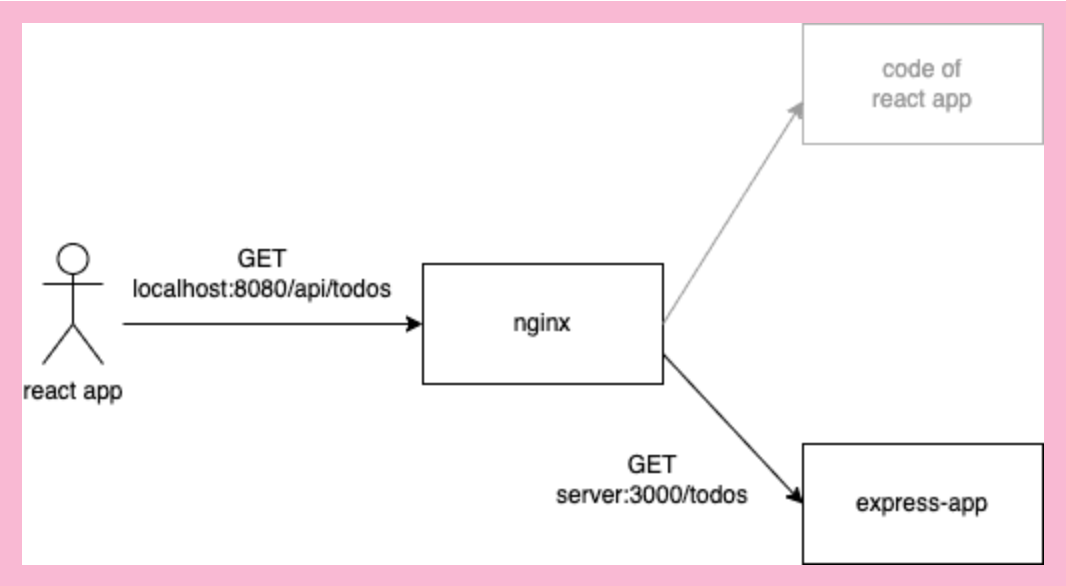
> In this exercise, submit the entire development environment, including both Express and React applications, dev.Dockerfiles and docker-compose.dev.yml.

Finally, it is time to put all the pieces together. Before starting, it is essential to understand *where* the React app is actually run. The above figure might give the impression that React app is run in the container but it is totally wrong.

It is just the *React app source code* that is in the container. When the browser hits the address http://localhost:8080 (assuming that you set up Nginx to be accessed in port 8080), the React source code gets downloaded from the container to the browser:



Next, the browser starts executing the React app, and all the requests it makes to the backend should be done through the Nginx reverse proxy:

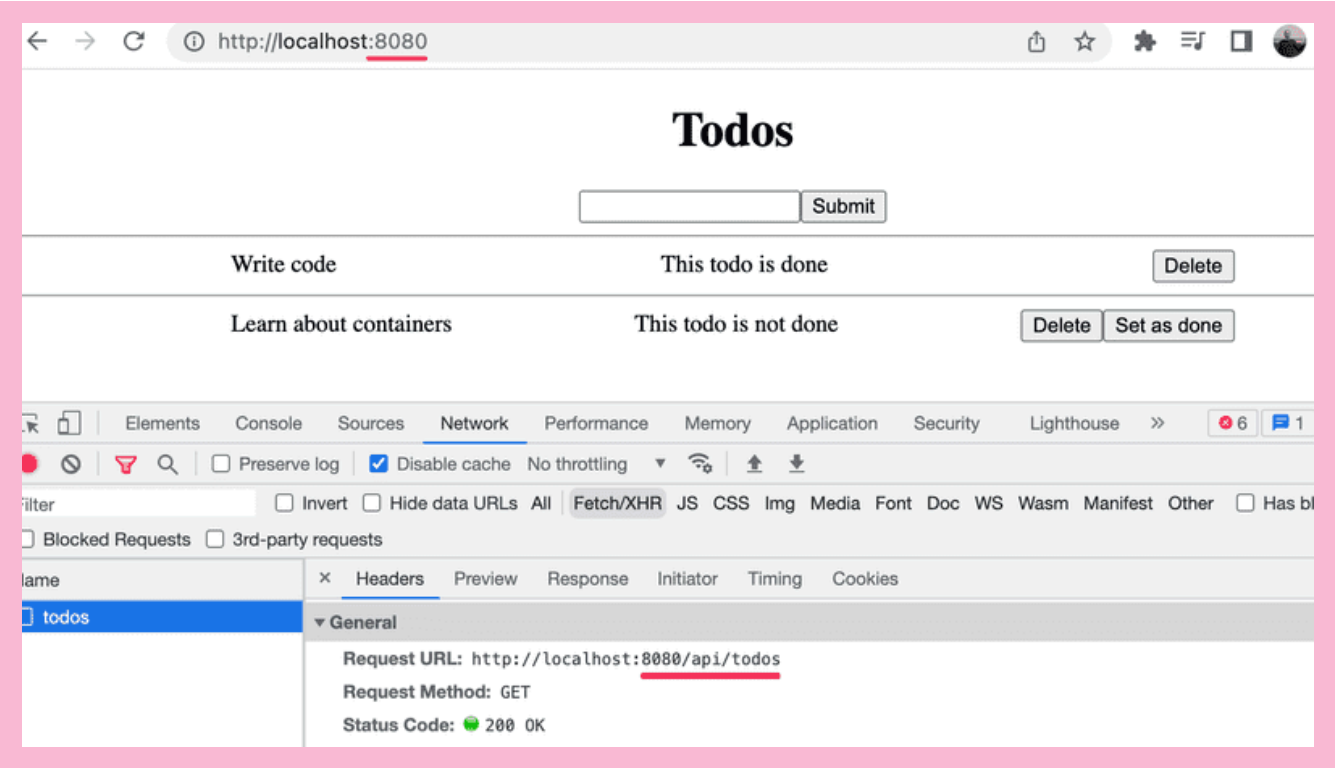


The frontend container is actually no more accessed beyond the first request that gets the React app source code to the browser.

Now set up your app to work as depicted in the above figure. Make sure that the todo-frontend works with todo-backend. It will require changes to the `VITE_BACKEND_URL` environmental variable in the frontend.

Make sure that the development environment is now fully functional, that is:

- all features of the todo app work
- you can edit the source files *and* the changes take effect by reloading the app
- frontend should access the backend through Nginx, so the requests should be done to http://localhost:8080/api/todos :



Note that your app should work even if no exposed port are defined for the backend and frontend in the docker compose file:

```
services:
  app:
    image: todo-front-dev
    volumes:
      - ./todo-frontend/:/usr/src/app
    # no ports here!

  server:
      image: todo-back-dev
      volumes:
        - ./todo-backend/:/usr/src/app
      environment:
        - ...
      # no ports here!

  nginx:
    image: nginx:1.20.1
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
    ports:
      - 8080:80 # this is needed
    container_name: reverse-proxy
    depends_on:
      - app
```

We just need to expose the Nginx port to the host machine since the access to the backend and frontend is proxied to the right container port by Nginx. Because Nginx, frontend and backend are defined in the same Docker compose configuration, Docker puts those to the same Docker network and thanks to that, Nginx has direct access to frontend and backend containers ports.

Tools for Production

Containers are fun tools to use in development, but the best use case for them is in the production environment. There are many more powerful tools than Docker Compose to run containers in production.

Heavyweight container orchestration tools like Kubernetes allow us to manage containers on a completely new level. These tools hide away the physical machines and allow us, the developers, to worry less about the infrastructure.

If you are interested in learning more in-depth about containers come to the DevOps with Docker course and you can find more about Kubernetes in the advanced 5 credit DevOps with Kubernetes course. You should now have the skills to complete both of them!

Exercises 12.20.-12.22.

Exercise 12.20:

Create a production *todo-app/docker-compose.yml* with all of the services, Nginx, todo-backend, todo-frontend, MongoDB and Redis. Use Dockerfiles instead of *dev.Dockerfiles* and make sure to start the applications in production mode.

Please use the following structure for this exercise:

```
todo-app
├── todo-frontend
├── todo-backend
├── nginx.dev.conf
├── docker-compose.dev.yml
├── nginx.conf
└── docker-compose.yml
```

copy

Exercise 12.21:

Create a similar containerized development environment of one of *your own* full stack apps that you have created during the course or in your free time. You should structure the app in your submission

repository as follows:

```
└── my-app
    ├── frontend
    |   └── dev.Dockerfile
    ├── backend
    |   └── dev.Dockerfile
    ├── nginx.dev.conf
    └── docker-compose.dev.yml
```

Exercise 12.22:

Finish this part by creating a containerized *production setup* of your own full stack app. Structure the app in your submission repository as follows:

```
└── my-app
    ├── frontend
    |   ├── dev.Dockerfile
    |   └── Dockerfile
    ├── backend
    |   └── dev.Dockerfile
    |   └── Dockerfile
    ├── nginx.dev.conf
    ├── nginx.conf
    ├── docker-compose.dev.yml
    └── docker-compose.yml
```

Submitting exercises and getting the credits

This was the last exercise in this section. It's time to push your code to GitHub and mark all of your finished exercises to the exercise submission system.

Exercises of this part are submitted just like in the previous parts, but unlike parts 0 to 7, the submission goes to an own course instance. Remember that you have to finish *all the exercises* to pass this part!

Once you have completed the exercises and want to get the credits, let us know through the exercise submission system that you have completed the course:

Note that you need a registration to the corresponding course part for getting the credits registered, see here for more information.

You can download the certificate for completing this part by clicking one of the flag icons. The flag icon corresponds to the certificate's language.

Propose changes to material

Part 12b
Previous part

Part 13
Next part

About course

Course contents

FAQ

Partners

Challenge


UNIVERSITY OF HELSINKI


HOUSTON