

```
{() => fs}
```



## c Basics of Orchestration

### React in container

Let's create and containerize a React application next.

```
$ npx create-react-app hello-front  
...
```

[copy](#)

Happy hacking!

The create-react-app already installed all dependencies for us, so we did not need to run npm install here.

The next step is to turn the JavaScript code and CSS, into production-ready static files. The create-react-app 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.  
...
```

[copy](#)

Great! The final step is figuring 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:16
```

[copy](#)

```
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  
[+] Building 172.4s (10/10) FINISHED
```

[copy](#)

```
$ docker run -it hello-front bash
```

```
root@98fa9483ee85:/usr/src/app# ls  
Dockerfile README.md build node_modules package-lock.json package.json public src
```

```
root@98fa9483ee85:/usr/src/app# ls build/  
asset-manifest.json favicon.ico index.html logo192.png logo512.png manifest.json  
robots.txt static
```

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
```

[copy](#)

```
added 88 packages, and audited 89 packages in 6s
```

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

```
|  
| 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 build directory will become the command to start the container:

```
FROM node:16

WORKDIR /usr/src/app

COPY . .

RUN npm ci

RUN npm run build

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

[copy](#)

Our CMD now includes square brackets and as a result we now used 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 for splitting 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:16 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:1.20-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/build /usr/share/nginx/html
```

[copy](#)

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

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 `REACT_APP_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 `REACT_APP_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.

Run the tests with `CI=true npm test`. Without the env `CI=true` set, the create-react-app will start watching for changes and your pipeline will get stuck.

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 to the production Dockerfile let's create a new one called *dev.Dockerfile*.

Starting the create-react-app in development mode should be easy. Let's start with the following:

```
FROM node:16
```

```
WORKDIR /usr/src/app
```

```
COPY . .
```

A small, light gray button with rounded corners and a subtle shadow, containing the word "copy" in a lowercase, sans-serif font.

```
# 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", "start"]
```

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 create-react-app will be served in port 3000, 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 done yet. 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 3000:3000 -v "$(pwd):/usr/src/app/" hello-front-dev
```

[copy](#)

Compiled successfully!

You can now view hello-front in the browser.

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

Note that it takes some time (for me it took 50 seconds!) for the frontend to get started with `npm start` in the development mode. The frontend has started when the following appears in the container log:

You can now view hello-frontend in the browser.

[copy](#)

**Editor's note:** *hot reload might work in your computer, but it is currently known to have some issues. So if it does not work for you, just continue without the hot reload support, and reload*

*the browser when you change the frontend code. You may also use [The Visual Studio Code Containers extension](#).*

Next, let's move the config to a *docker-compose.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:
      - 3000:3000
    container_name: hello-front-dev # This will name the container hello-front-dev
```

[copy](#)

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

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.yml* to:

```
services:
  app:
    image: hello-front-dev
    build:
      context: .
      dockerfile: dev.Dockerfile
    volumes:
      - ./:/usr/src/app
    ports:
      - 3000:3000
    container_name: hello-front-dev
  debug-helper:
    image: busybox
```

[copy](#)

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 up
Pulling debug-helper (busybox:)...
latest: Pulling from library/busybox
8ec32b265e94: Pull complete
Digest: sha256:b37dd066f59a4961024cf4bed74cae5e68ac26b48807292bd12198afa3ecb778
Status: Downloaded newer image for busybox:latest
Starting hello-front-dev          ... done
Creating react-app_debug-helper_1 ... done
Attaching to react-app_debug-helper_1, hello-front-dev
react-app_debug-helper_1 exited with code 0

hello-front-dev |
hello-front-dev | > react-app@0.1.0 start
hello-front-dev | > react-scripts start
```

[copy](#)

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 run debug-helper wget -O - http://app:3000
```

copy

```
Creating react-app_debug-helper_run ... done
Connecting to hello-front-dev:3000 (172.26.0.2:3000)
writing to stdout
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    ...
```

The URL is the interesting part here. We simply said to connect to the port 3000 of the service *app*. The *app* is the name of the service specified in the *docker-compose.yml*:

```
services:
  app:
    image: hello-front-dev
    build:
      context: .
      dockerfile: dev.Dockerfile
    volumes:
      - ./usr/src/app
    ports:
      - 3000:3000
    container_name: hello-front-dev
```

copy

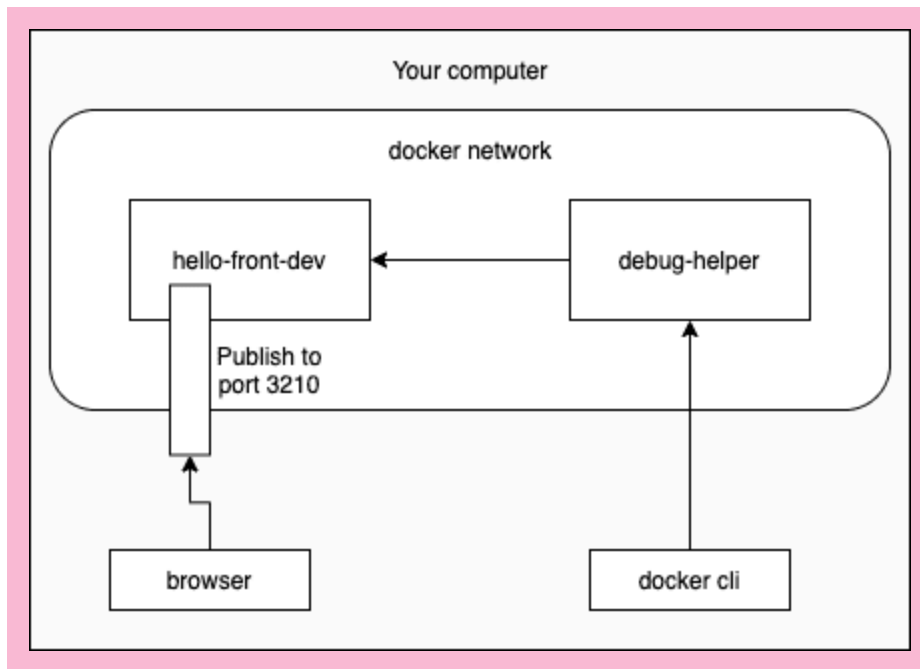
And the port used is the port from which the application is available in that container, also specified in the *docker-compose.yml*. The port does not need to be published for other services in the same network to be able to connect to it. The "ports" in the docker-compose file are only for external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

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

```
services:
  app:
    image: hello-front-dev
    build:
      context: .
      dockerfile: dev.Dockerfile
    volumes:
      - ./usr/src/app
    ports:
      - 3210:3000
    container_name: hello-front-dev
  debug-helper:
    image: busybox
```

copy

With `docker compose up` the application is available in <http://localhost:3210> at the *host machine*, but still `docker compose run debug-helper wget -O - http://app:3000` works since the port is still 3000 within the docker network.



As the above image illustrates, `docker compose run` asks `debug-helper` to send the request within the network. While the browser in 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 3000:3000 in our `docker-compose.yml`.

## 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:
```

[copy](#)

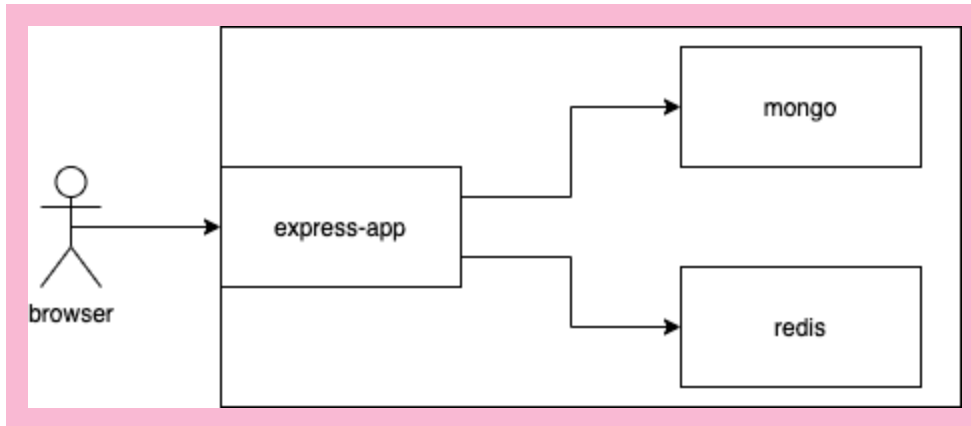
```

- ...
ports:
- ...
environment:
- REDIS_URL=redisurl_here
- MONGO_URL=mongourl_here

```

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 add a reverse proxy to our docker-compose.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.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 { }

```

copy

```
# 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:3000
            proxy_pass http://localhost:3000;
        }
    }
}
```

Next, create an Nginx service in the *docker-compose.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.conf:/etc/nginx/nginx.conf:ro
    ports:
      - 8080:80
    container_name: reverse-proxy
    depends_on:
      - app # wait for the frontend container to be started
```

copy

with that added we can run `docker compose up` and see what happens.

```
$ docker container ls
```

CONTAINER ID	IMAGE	COMMAND	NAMES	CREATED	STATUS
a02ae58f3e8d	nginx:1.20.1	"/docker-entrypoint..."	reverse-proxy	4 minutes ago	Up 4 minutes
0.0.0.0:8080->80/tcp		:::8080->80/tcp			
5ee0284566b4	hello-front-dev	"docker-entrypoint.s..."	hello-front-dev	4 minutes ago	Up 4 minutes
0.0.0.0:3000->3000/tcp		:::3000->3000/tcp			

copy

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

This is because directing requests to <http://localhost:3000> leads to nowhere as the Nginx container does not have an application running in port 3000. By definition, localhost refers to the current computer used to access it. With containers localhost is unique for each container, leading 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
```

[copy](#)

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

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

Since we are inside the container, we can also test the DNS! Let's curl the service name (app) in port 3000

```
root@374f9e62bfa8:/# curl http://app:3000
<!DOCTYPE html>
<html lang="en">
  <head>
    ...
  <meta
    name="description"
    content="Web site created using create-react-app"
  />
  ...
```

[copy](#)

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

If you are still encountering 502, make sure that the create-react-app has been built first. You can read the logs output from the `docker compose up`.

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
```

[copy](#)

```
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:3000;
    }
  }
}
```

[copy](#)

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.

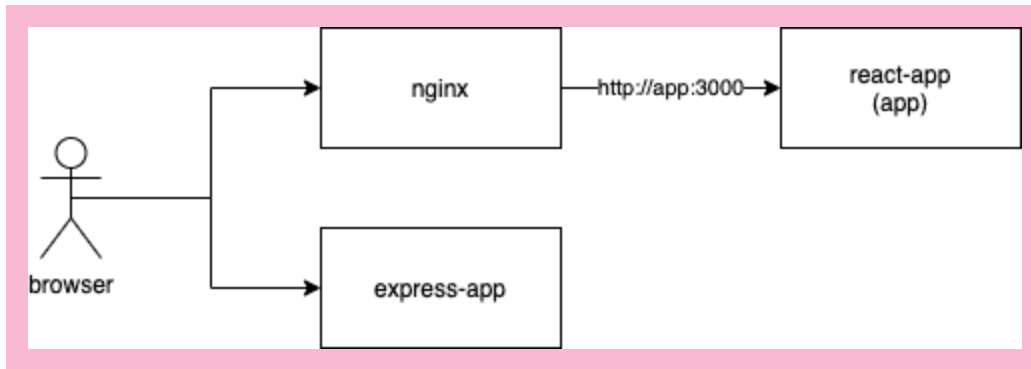
### Exercise 12.17: Set up an Nginx reverse proxy server in front of todo-frontend

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
```

[copy](#)

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
```

[copy](#)

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.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 / {
        # 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:3000
        proxy_pass http://localhost:3000;
    }

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

[copy](#)

```

    }
  }
}

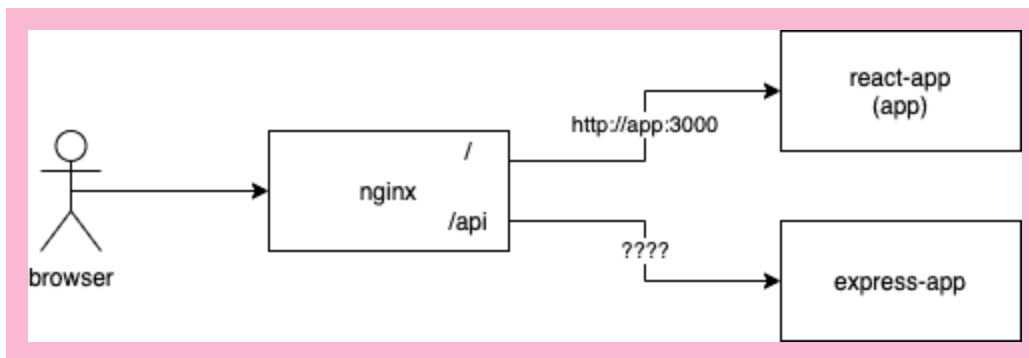
```

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

- 4 Ah, I was missing the trailing slash :( – Vanuan May 2 '16 at 23:54
- 8 AAARGH! TRAILING SLASH ! – barrymac May 26 '17 at 16:27
- 6 3 hours of searching, and yeah... It was the trailing slash. Thanks mate! – Lucas P. Oct 15 '18 at 16:09

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



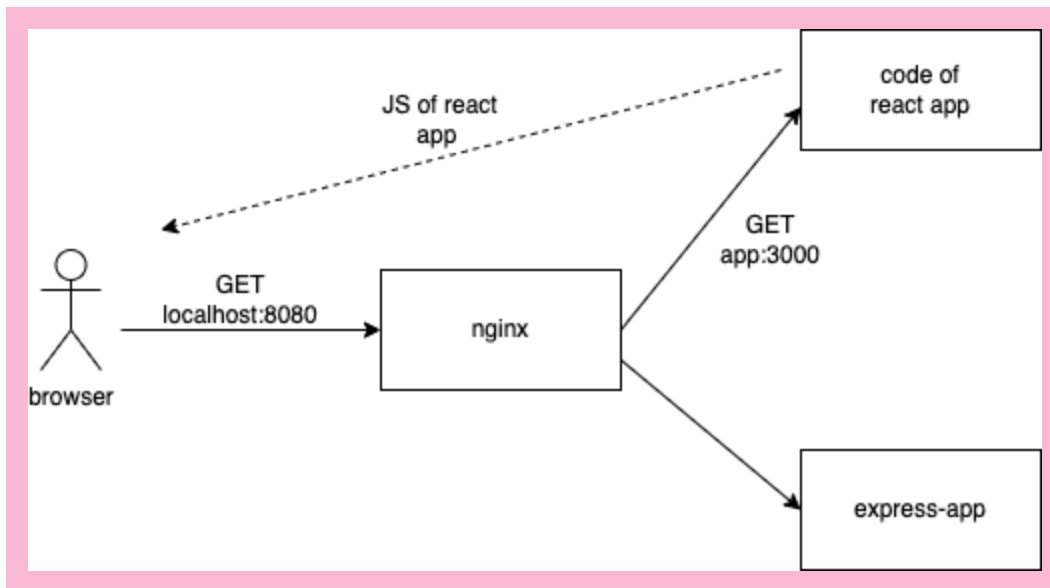
### Exercise 12.19: Connect the services, todo-frontend with todo-backend

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

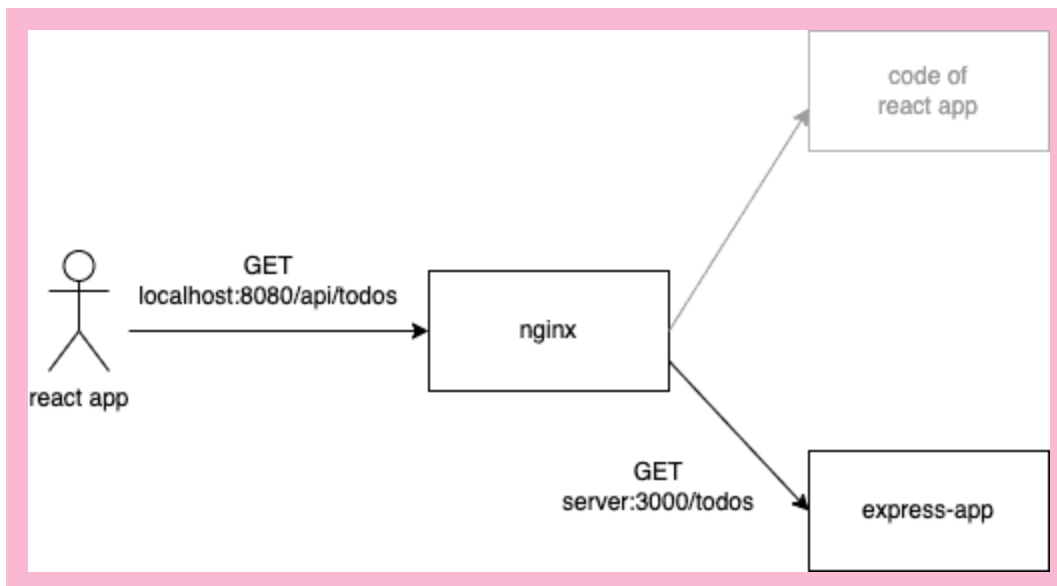
Finally, it is time to put all the pieces together. Before starting, it is essential to understand *where* the React app is actually run. The above figure might give the impression that React app is run in the container but it is totally wrong.

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:



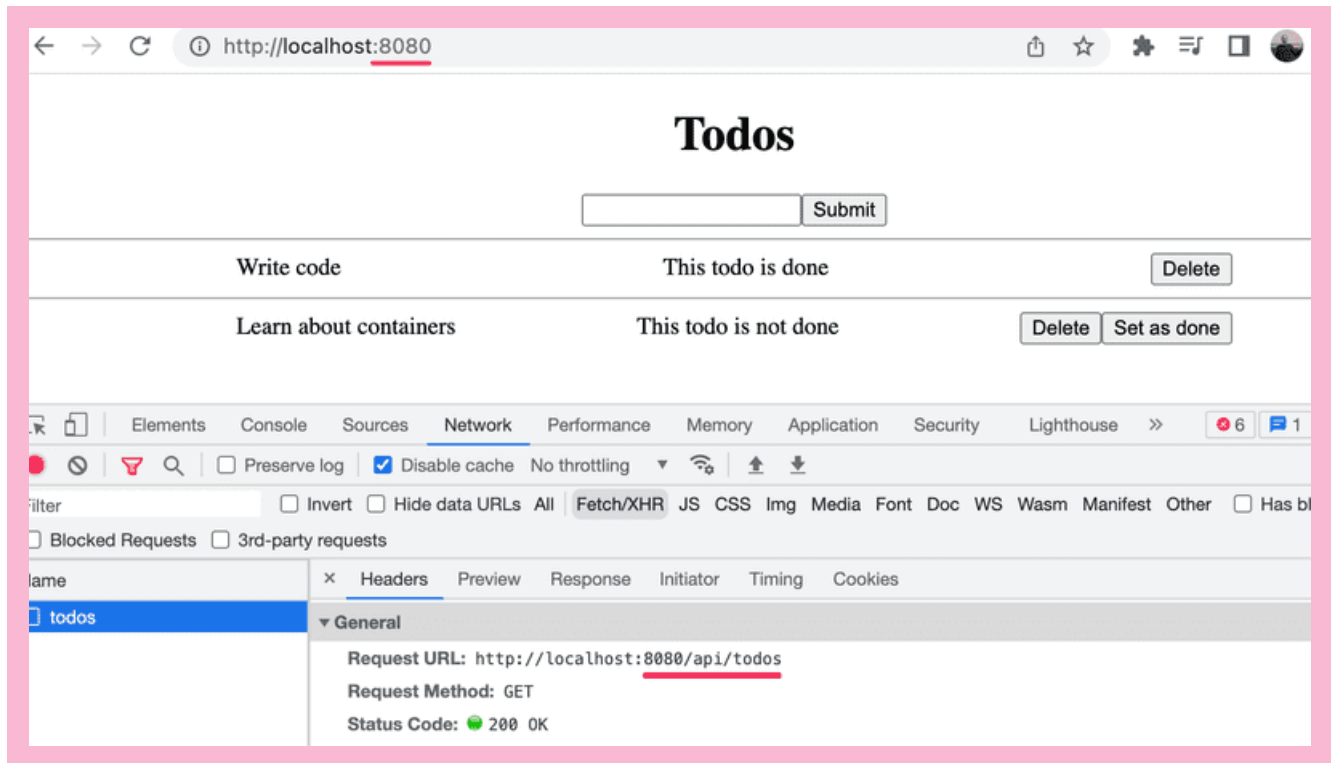
The frontend container is actually no more accessed beyond the first request that gets the React app source code to the browser.

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 `REACT_APP_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 (the hot reloading may or may not work...)

- 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
```

copy

`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
  └─ docker-compose.dev.yml
```

[copy](#)

### 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
  ├── docker-compose.dev.yml
  └─ docker-compose.yml
```

[copy](#)

## 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:

**My submissions**

part	exercises	hours	github	comment	solution
1	22	29	<a href="https://github.com/Kaltsoon/fs-cicd">https://github.com/Kaltsoon/fs-cicd</a>		<a href="#">show</a>
total	22	29			

credits 1 based on exercises

Certificate  

I have completed the course (exam done in Moodle and will not do more exercises) and want to get university credits registered.

**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

