Networks / Requests

In many application, you'll need more than one container - for **two main reasons:**

1. It's considered a **good practice** to focus each container on **one main task** (e.g. run a web server, run a database, ...)
2. It's **very hard** to configure a Container that **does more than one "main thing"** (e.g. run a web server AND a database)

Multi-Container apps are quite common, especially if you're working on "real applications".

Often, some of these Containers need to **communicate** though:

- either **with each other**
- or with the **host machine**
- or with the **world wide web**

Communicating with the World Wide Web (WWW)

Communicating with the WWW (i.e. sending Http request or other kinds of requests to other servers) is thankfully very easy.

Consider this JavaScript example - though it'll always work, no matter which technology you're using:

```
fetch('https://some-api.com/my-data').then(...)
```

This very basic code snippet tries to send a `GET` request to `some-api.com/my-data`.

This will **work out of the box**, no extra configuration is required! The application, running in a Container, will have no problems sending this request.

Communicating with the Host Machine

Communicating with the Host Machine (e.g. because you have a database running on the Host Machine) is also quite simple, though it **doesn't work without any changes**.

One important note: *If you deploy a Container onto a server (i.e. another machine), it's very unlikely that you'll need to communicate with that machine. Communicating to the Host Machine typically is a requirement during development - for example because you're running some development database on your machine.*

Again, consider this JS example:

```
fetch('localhost:3000/demo').then(...)
```

This code snippet tries to send a `GET` request to some web server running on the local host machine (i.e. **outside** of the Container but **not** the WWW).

On your local machine, this would work - inside of a Container, it **will fail**. Because `localhost` inside of the Container refers to the Container environment, **not to your local host machine which is running the Container / Docker**!

But Docker has got you covered!

You just need to change this snippet like this:

```
fetch('host.docker.internal:3000/demo').then(...)
```

`host.docker.internal` is a special address / identifier which is translated to the IP address of the machine hosting the Container by Docker.

Important: *"Translated" does **not** mean that Docker goes ahead and changes the source code. Instead, it simply detects the outgoing request and is able to resolve the IP address for that request.*

Communicating with Other Containers

Communicating with other Containers is also quite straightforward. You have two main options:

1. **Manually find out the IP** of the other Container (it may change though)
2. Use **Docker Networks** and put the communicating Containers into the same Network

Option 1 is not great since you need to search for the IP on your own and it might change over time.

Option 2 is perfect though. With Docker, you can create Networks via `docker network create SOME_NAME` and you can then attach multiple Containers to one and the same Network.

Like this:

```
docker run -network my-network --name cont1 my-image
docker run -network my-network --name cont2 my-other-image
```

Both `cont1` and `cont2` will be in the same Network.

Now, you can simply **use the Container names** to let them communicate with each other - again, Docker will resolve the IP for you (see above).

```
fetch('cont1/my-data').then(...)
```