Running Express in Docker

This documentation is based on the Xojo Guest Blog Post: Running Xojo Web Applications in Docker and has been adjusted for Express. An extended Version including Post Build Scripts can be found on GitHub: jo-tools.

What is Docker?

If you haven't heard of Docker, go ahead and read their excellent Documentation: Docker - Overview.

I'm just going to quote some basics from the Overview to get a brief introduction of Docker, Docker image and Docker container:

Docker *is an open platform for developing, shipping, and running applications. It provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allow you to run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so you do not need to rely on what is currently installed on the host. You can easily share containers while you work, and be sure that everyone you share with gets the same container that works in the same way.*

*A **Docker image** is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it.*

*A **Docker container** is a runnable instance of an image. You can create, start, stop, move, or delete a container. By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine. A container is defined by its image as well as any configuration options you provide to it when you create or start it.*

*A **Docker Registry** is a stateless, highly scalable server side application that stores and lets you distribute Docker images.*

Why use Docker with Express?

You can package your Express Console Apps in a Docker Image. That allows you to run an instance of your applications easily in a Docker Container. Docker provides the infrastructure to host the apps, start/stop them, switch between image versions - and much more.

The Docker Images containing your Xojo Web Apps can be copied to other machines. This might be some Linux or macOS machine *(Windows seems to be a bit more tricky)*, a Cloud Service - or even a Synology NAS. You then can import the Images on the Docker environment of those machines, and serve your apps from there.

If you're pushing the Docker Images to a Registry (e.g. Docker Hub), it gets even more convenient. Docker Hub is a service provided by Docker for finding and sharing container images with your team or the public. So you'll build the Docker Image with your Xojo Web App, push it to the Registry. After that, you (or your team, or everyone) can just pull the Images to their Docker infrastructure and run them in a Container.

How to build a Docker Image containing an Express Console App?

One could answer obviously: it depends. But let me try to explain the basic required steps. For this example we're going to use a macOS machine to build the Express Console App and Docker Image. Then we'll deploy and run a Docker Container on the local developer machine and on a Synology NAS.

Requirements

- Xojo with a licence allowing you to build a Console App for Target Linux (Intel, 64Bit).

- Docker Desktop installed on the macOS developer machine. Download and install it from here: Docker for macOS.
  After that you should have the Docker GUI up and running. And in Terminal.app you should have the command line tools available. You can double check this by entering this command: `docker version`

- We're going to build a Docker Image based on Ubuntu (Linux). To run a Docker Container with this image, we obviously need a machine supporting this. Both macOS and the Synology NAS will work just fine.

Are you ready and set up? So let's go and try this.

Build an Express Console App

I'm going to use the provided example: `Express-Demo-Console.xojo_project`.

First Debug-Run it locally to make sure the Console App works as expected.

Once we're ready to build we should think of what we're going to need for the Docker Image. Check the following properties in your Xojo Project:

- The most obvious one: Tick the "**Build Target: Linux, x86-64Bit".** Since our Docker Image will run Ubuntu / Linux-64Bit, we need the Express Console App to run in such an environment.
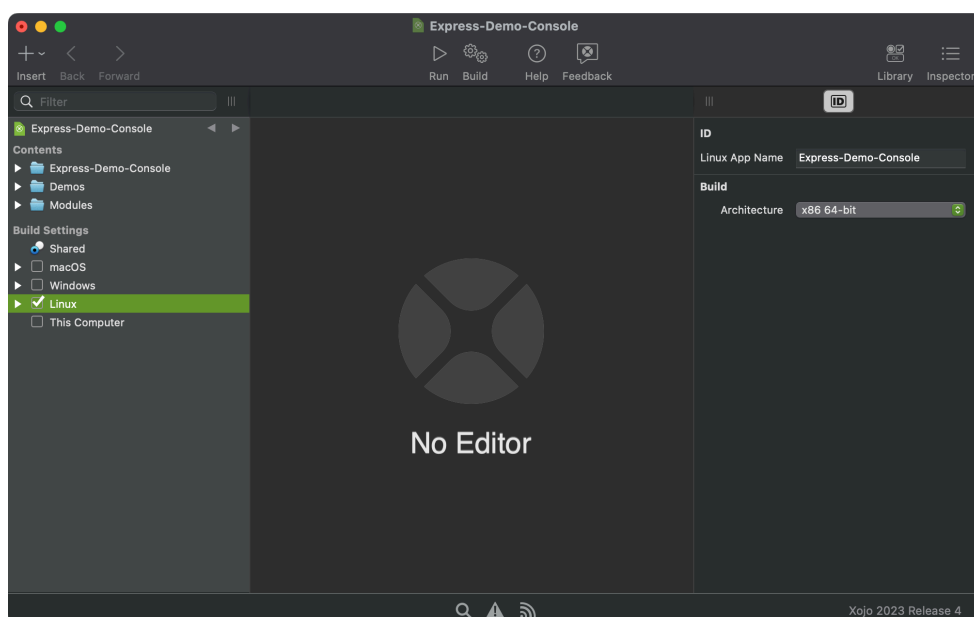
- That's about all we need to do within the Xojo IDE - so let's hit **"Build"**.

We now have the built Express Console App ready. Next step is to package it in a Docker Image.

Dockerfile

Open the Express Console App's Build Folder in Finder. Next we need to write the instructions to get it in a Docker Image. Open your favourite Text Editor. Create a Text-File with Filename: `Dockerfile` (no file extension!). Or copy the one provided in this documentations folder. Save it in the Build folder (next to the application's executable file). The Content we're saving in Dockerfile is:

```
#BASE IMAGE
FROM ubuntu:22.04

#INSTALL REQUIRED LIBRARIES
ENV DEBIAN_FRONTEND noninteractive
RUN apt-get update && apt-get install -y libunwind8 libglib2.0
libsoup2.4-1

#ADD EXPRESS CONSOLE APP TO DOCKER IMAGE
ADD / /app/

#EXPOSE PORT 80 AND RUN EXPRESS CONSOLE APP ON PORT 80
EXPOSE 80
WORKDIR /app
CMD /app/Express-Demo-Console --Port=80
```

This describes how Docker is going to create the Docker Image. Let's briefly explain the steps of this Dockerfile:

- We're going to use Ubuntu 22.04 as a base image. That about the same as saying: "Let's set up an environment running Ubuntu 22.04 first."

- We then set an environment variable allowing to install updates without user interaction. Then the system gets updated, and required Libraries installed.

- The whole Content of the Build Folder (or in other words: everything next to the Dockerfile - that's basically our Express Console App) is being copied into the image to the location `/app`

- Expose 80: This tells that when running this Image as a Container Instance later we can attach to this "virtual port".
  It doesn't matter if several applications in different Images/Containers use the same port. That's because when running an Instance, we can later configure the running Instance to map the Ports, e.g.: "8088 (external) -> 80 (internal)". So a second Web App Instance or App could use e.g."8089 (external) -> 80 (internal)".
  That's why for our own convenience we can use Port 80 for every of our Express Console Apps's Docker Image.

- Finally, the instructions tell where the working directory is, and with the last command the Express Console App is being launched.

Alright - everything is ready so that we can now create the Docker Image.

Docker Image

Open `Terminal.app` and change directory to the Build Folder of the Web App:

```
cd /path/to/your/xojo-express-project/Builds\ -\ Express-Demo-Console/
Linux\ 64\ bit/Express-Demo-Console
```

You're now in the folder with the Dockerfile and your Express Console App.

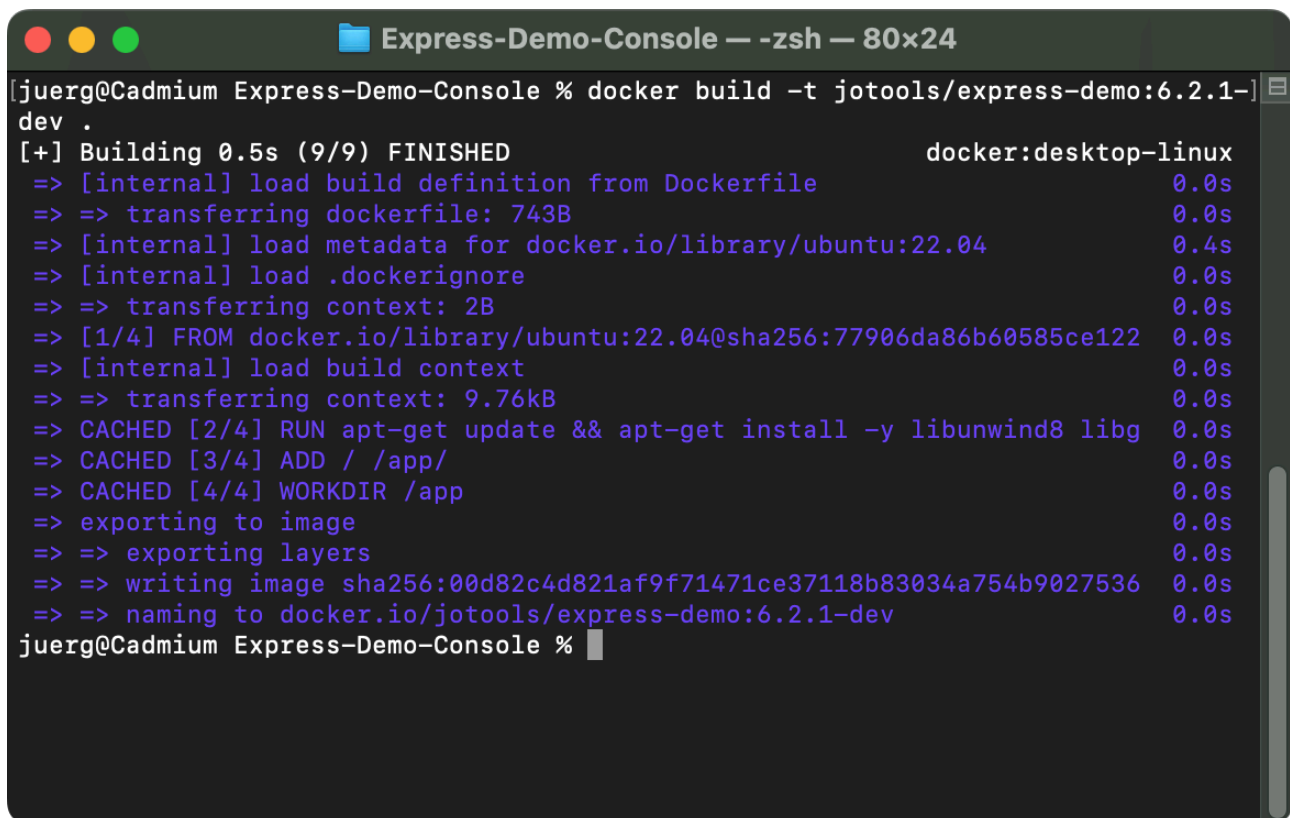A Docker Image always needs a **Tag**. Such a Tag looks like this:

```
jotools/express-demo:6.2.1-dev
```

The parts are: (company-name)/(product-name):(version-label-tag)

So... let's build the Docker Image - enter this in Terminal:

```
docker build -t jotools/express-demo:6.2.1-dev .
```

What this does ist: Build a Docker Image with Tag ___ from "this current directory" (the dot at the end).

```
[juerg@Cadmium Express-Demo-Console % docker build -t jotools/express-demo:6.2.1-]
dev .
[+] Building 0.5s (9/9) FINISHED                        docker:desktop-linux
 => [internal] load build definition from Dockerfile               0.0s
 => => transferring dockerfile: 743B                               0.0s
 => [internal] load metadata for docker.io/library/ubuntu:22.04    0.4s
 => [internal] load .dockerignore                                  0.0s
 => => transferring context: 2B                                    0.0s
 => [1/4] FROM docker.io/library/ubuntu:22.04@sha256:77906da86b60585ce122  0.0s
 => [internal] load build context                                  0.0s
 => => transferring context: 9.76kB                                0.0s
 => CACHED [2/4] RUN apt-get update && apt-get install -y libunwind8 libg  0.0s
 => CACHED [3/4] ADD / /app/                                       0.0s
 => CACHED [4/4] WORKDIR /app                                      0.0s
 => exporting to image                                             0.0s
 => => exporting layers                                            0.0s
 => => writing image sha256:00d82c4d821af9f71471ce37118b83034a754b9027536  0.0s
 => => naming to docker.io/jotools/express-demo:6.2.1-dev          0.0s
juerg@Cadmium Express-Demo-Console %
```

Side-Note: If you're re-building an Image with the same Tag of an already existing Image, it gets overwritten. You could also first remove an existing image like this:
```
docker image rm jotools/express-demo:6.2.1-dev
```

You can now launch the Docker Desktop .app and find your Image there:
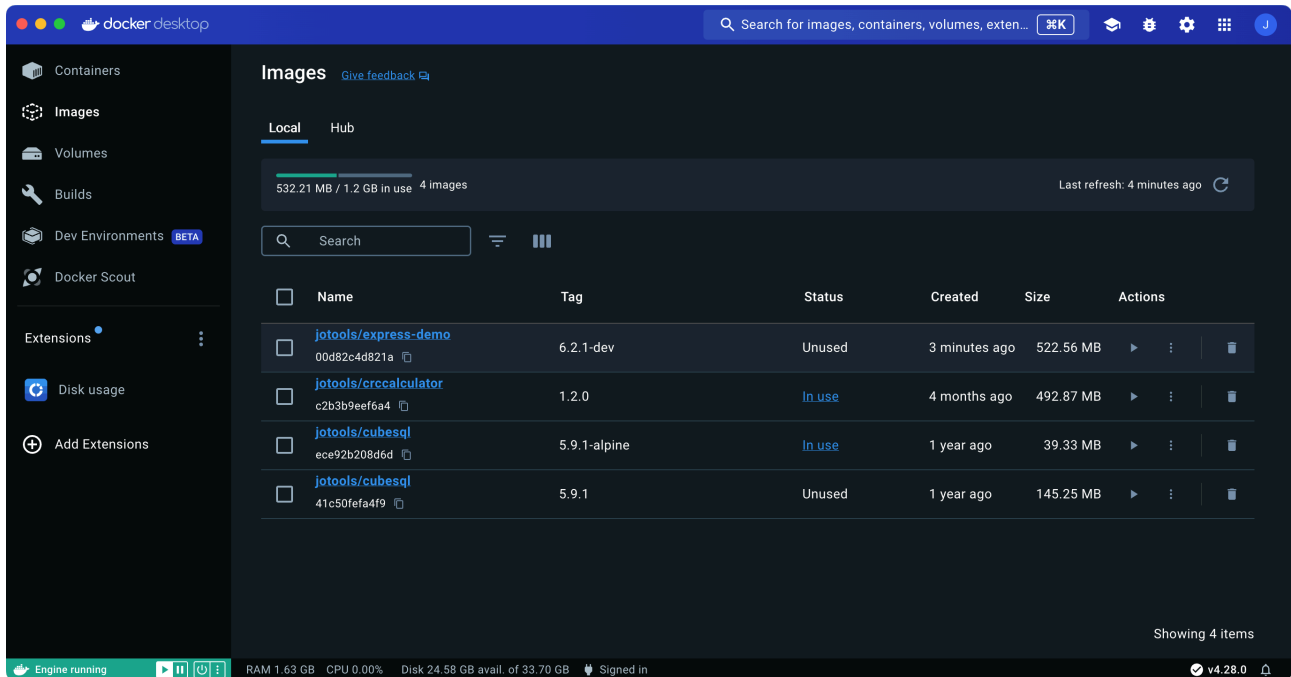
If you prefer to see the image using the command line:

```
docker image ls
```

Run Docker Container

Let's now take the Image and run a new Docker Container using that Image. We will configure the Docker Container Instance to use the local Port 8089. In Docker Desktop GUI:

• Select the Image, click on the Button "Run"

• Expand the optional settings

• Enter a Container Name of your choice

• Map the "Host Port: `8089`" to "Container Port: 80/tcp"

• Add Environment Variable:
  `EXPRESS_DEMO` with Value: 1
  • The Express Console Demo is looking for a launch argument or an environment variable to select the demo to be run. This Environment Variable will launch demo 1. If no Launch Argument and no Environment Variable are set, the Express Console Demo would ask for user input - which obviously won't happen in the Docker Container.
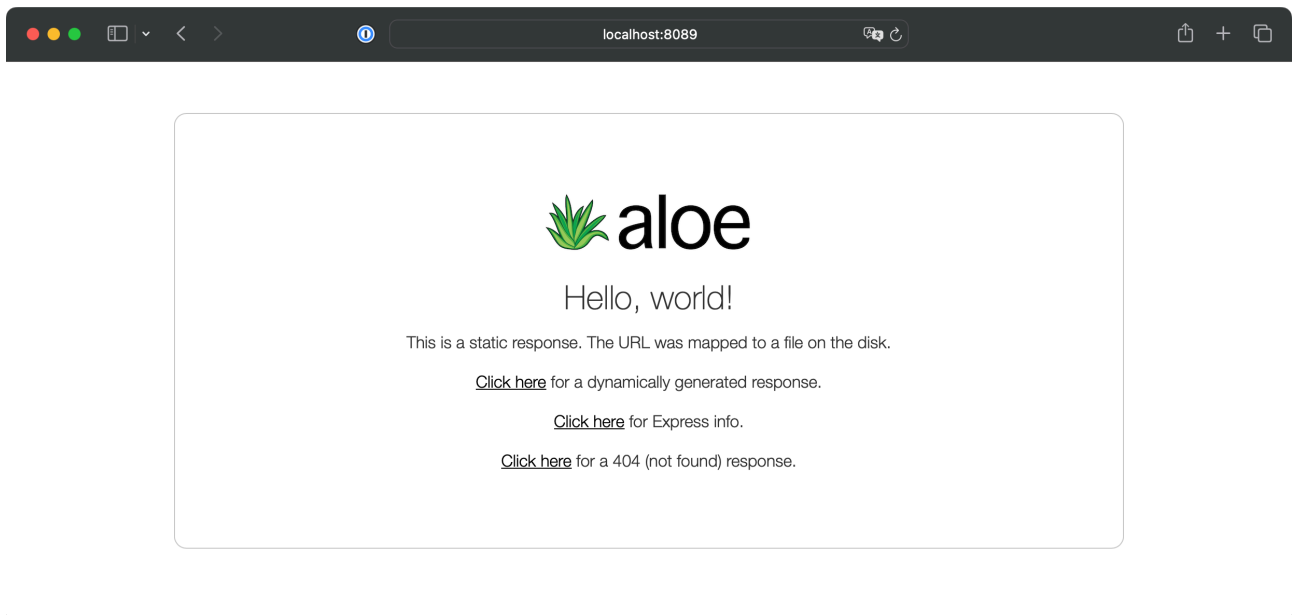
• Then... click on "Run"
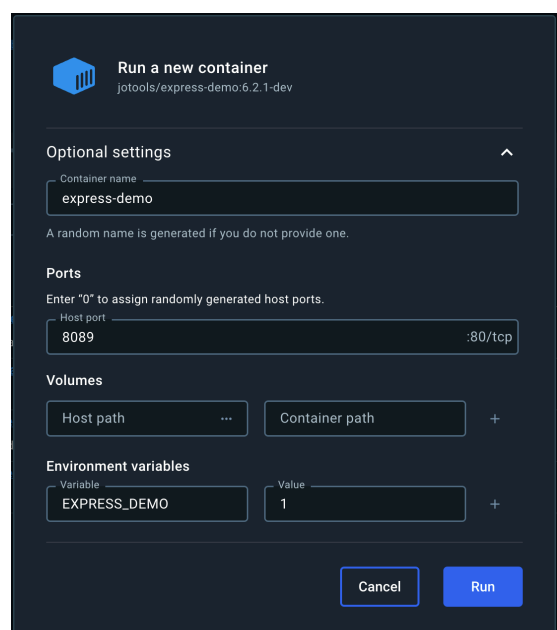
And that's it - we have the Container up and running:
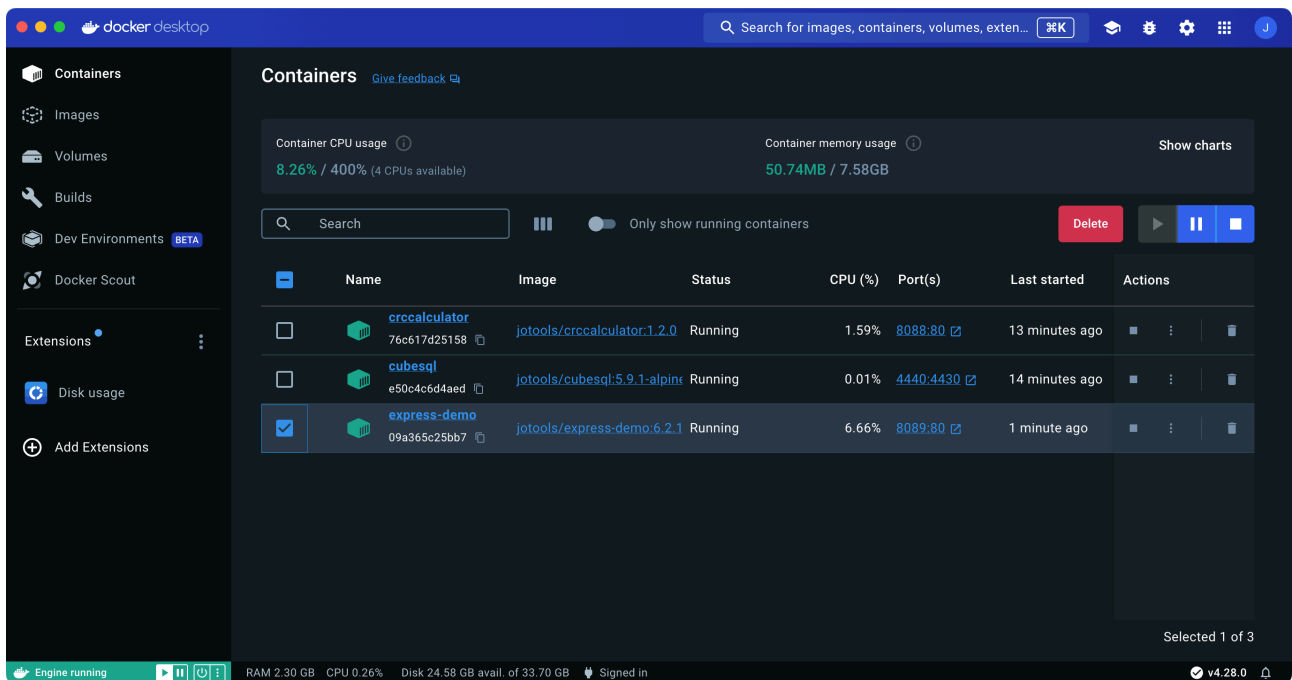
If you prefer the command line:

```
docker run -d --publish=8089:80 --name express-demo jotools/express-demo:6.2.1-dev
```

Let's access our Express Console App running in the local Docker Container using Safari:

Side-Note: If you just want to try this example project then you can pull the image like this in Terminal.app:

```
docker pull jotools/express-demo:6.2.1
```

Export Docker Image

The easiest way is to push the Image to a Registry such as Docker Hub. That obviously requires a registration - so we won't cover this here and now.

Let's export the Image using Terminal.app.

You should still be in the Build - directory of the Xojo Web App. If not, change to that folder.

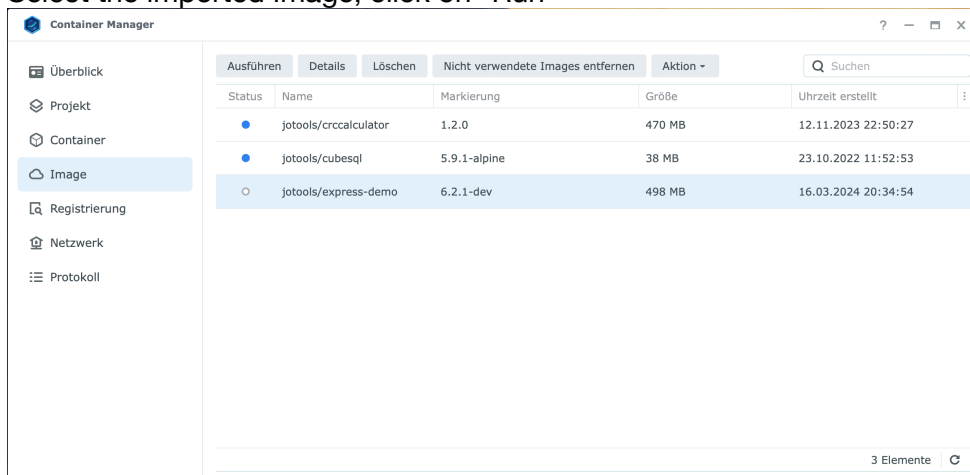Then export the Docker Image to a file like this:

```
docker save jotools/express-demo:6.2.1-dev | gzip > ./Express-Demo.dockerimage.tgz
```

This command is saving the Image/Tag to a .tgz located in the current directory.


Run Docker Image on a Synology NAS

If you have a Synology NAS supporting Docker - this is how you can run the exported Image on your NAS:

- First of course: Install the Synology Container Manager Package

- Copy the exported `Express-Demo.dockerimage.tgz` to a Share on the NAS

- Launch the Container Manager .app on the NAS

- Under "Images": Actions -> Import -> From File -> (choose your `CRCCalculatorWeb.dockerimage.tgz`)

- Select the imported Image, click on "Run"

- Enter a Containername of your choice:

- Port Settings: Same story here - Map the Local<->Container Port.
  In this example: Local 8089 <-> Container 80

- Environment Settings: Same story here:
  Add Environment Variable: `EXPRESS_DEMO=1`

- Finish the assistant. The Container should now be up and running:

Let's access our Xojo built Web App running in the Docker Container of the Synology NAS using Safari:
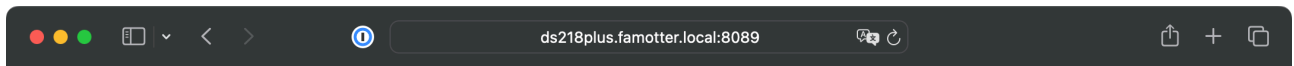
What next?

There are a lot of features and possibilities which we won't cover here.
Let's just pick a couple and briefly explain without going into detail. You'll find excellent documentation on Docker's Website if you want or need to dig in further.

Docker and Persisted Storage

Note that this example has no persisted storage. Which means that you can save files within a Container - but once you stop or remove the Container (e.g. to run a new Container with a newer Image version) the created data is gone. That's a good thing - as a Container will always behave the same when deployed, not being cluttered with data that has come from somewhere.

If your Express Console App requires to store data on the filesystem you should add a Volume. In the Dockerfile add a line like this:

```
VOLUME /data
```

You then can have your Express Console App save data to the folder `/data`.

If you run a Container from such an Image - have a look at the Advanced Settings. Since the Image exposes a Volume, you can choose a local folder (on the machine where Docker is serving the Container) to be attached to /data. Same story on the Synology NAS.

Once you remove the Docker Container, the Data will still be there. So you can run another Container (e.g. with a newer version of your app), attach the same/existing data-folder. That way you can continue and keep existing data.

Environment Variables

Especially if you're going to Deploy the Docker Image on various environments, you might want some "settings". One approach is to use Environment Variables in the Dockerfile:

```
ENV MY_CUSTOM_SETTING=4430
```

Again in the Advanced options when running a Container you can override these Environment Variables to fit your needs.

Build Automation

If you're building regularly and don't want to enter commands in the command line after every build you could think about a Post Build Script which builds the Docker Image and optionally even pushes it to a Registry (e.g. Docker Hub).

An example is available here: Example Project - Xojo Web App 2 Docker

Docker Registry

The easiest way is to distribute a local Docker Image is to use a Registry such as Docker Hub. That obviously requires a registration and your Docker Client to be signed in. You can then push the Docker Image to the Registry, and pull it from there on the Docker Clients.

```
docker push jotools/express-demo:6.2.1-dev
```

```
docker pull jotools/crccalculator:6.2.1-dev
```

Restart policy

In order to automatically launch a Docker Container after a reboot (of the host machine; of the Docker daemon): see Docker's <u>Restart Policy Documentation</u>.

This command changes the restart policy for an already running container named `express-demo`:

```
docker update --restart unless-stopped express-demo
```

And this command will ensure all currently running containers will be restarted unless stopped:

```
docker update --restart unless-stopped $(docker ps -q)
```