4 - Containerize Video Processing Service

01:43

Containerize Video Processing Service (Docker)

It can be difficult to deploy apps and ensure they have all the dependencies they need. For example, we need to install `Node.js` as well as `ffmpeg` on our machine to run our Video Processing Service.

But what if we want to deploy our app to a server that doesn't have `ffmpeg` installed?

This is where containers come in. Containers allow us to package our app *and* all its dependencies into a single image. We can then run this image on any machine that has Docker installed.

For now, we will do so locally. But later on we will deploy our image to Google Cloud Run, which is specifically designed for running containerized apps.

1. Create a Dockerfile

```
# Use an official Node runtime as a parent image
FROM node:18

# Set the working directory in the container to /app
WORKDIR /app

# Install ffmpeg in the container
RUN apt-get update && apt-get install -y ffmpeg

# Copy package.json and package-lock.json into the working
directory
COPY package*.json ./

# Install any needed packages specified in package.json
RUN npm install

# Copy app source inside the docker image
COPY . .

# Make port 3000 available outside this container
EXPOSE 3000

# Define the command to run your app using CMD (only one CMD
allowed)
CMD [ "npm", "start" ]
```

> Note: This Dockerfile is building an image for development purposes. For production, you would want to use a multi-stage build to reduce the size of the image.

The `Dockerfile` defines what goes on in the environment inside your container.

We use the base Node 18 image, which is Debian-based. Docker images are recursive, so we are building our own Docker image using other images, which allows for reusability.

The reason we copy the `package.json` and `package-lock.json` before copying the rest of the code is due to Docker's layer caching mechanism. Docker builds images in layers and each step in your Dockerfile creates a new layer. Docker can reuse layers from previous builds for new builds, which speeds up the build process.

If we copied our code before running `npm install`, every code change would invalidate Docker's cache for the `npm install` step, and Docker would have to reinstall our node modules on every build. By copying just the `package.json` and `package-lock.json` first, Docker can use the cached node modules as long as those files haven't changed.

2. Create a `.dockerignore` File

Similar to `.gitignore`, the `.dockerignore` file tells Docker which files and folders to ignore when building the image.

```
node_modules
```

In this case we are ignoring the `node_modules` folder.

3. (optional) Build a production Dockerfile

The reason this step is optional, is because if you are new to Docker, this may not be easy to understand. But for completeness, I will include it here.

As you become more familiar with Docker, you may come back and revisit this section.

```
# Stage 1: Build stage
FROM node:18 AS builder

# Set the working directory in the container to /app
WORKDIR /app

# Copy package.json and package-lock.json into the working directory
COPY package*.json ./

# Install any needed packages specified in package.json
RUN npm install

# Bundle app source inside the docker image
```

Copy

```
COPY . .

# Build the app
RUN npm run build

# Stage 2: Production stage
FROM node:18

# Install ffmpeg in the container
RUN apt-get update && apt-get install -y ffmpeg

# Set the working directory
WORKDIR /app

# Copy package.json and package-lock.json
COPY package*.json ./

# Install only production dependencies
RUN npm install --only=production

# Copy built app from the builder stage
COPY --from=builder /app/dist ./dist

# Make port 3000 available to the world outside this container
EXPOSE 3000

# Define the command to run your app using CMD which defines
your runtime
CMD [ "npm", "run", "serve" ]
```

Mark Lesson Complete

`production`, is used to run the app.

Notice how we are using `npm run serve` instead of `npm start`. This will serve the compiled files from out `/dist` directory.

The reason the size is smaller, is because in this image we are not including any of the source files, only the compiled files.

Yes, Stage 1 uses the source files to build the app, but those files are not included in the final image. Because in Stage 2, we are copying the compiled files from Stage 1, but *not* the source files.

The final stage is the one that is used to run the app.

It's also worth mentioning that's it's generally preferable to specify a Node version, e.g. `FROM NODE:18.17`. Because by default with `FROM NODE:18` Docker will use the latest version of Node. So it could be possible, though unlikely, that a new Node subversion breaks our app.

4. Build and Run the Docker Image

1. To build your image, navigate to the directory that has your `Dockerfile` and run the following command:

```
docker build -t video-processing-service .
```

The `-t` flag lets you tag your image so it's easier to find later.

> Important: If you are using mac, add `--platform linux/amd64` to the above command.

2. List Docker images

```
docker images
```

3. Now, to run the image, use the `docker run` command:

```
docker run -p 3000:3000 -d video-processing-service
```

The `-p` flag redirects a public port to a private port inside the container.
The `-d` flag runs the container in detached mode, leaving the container running in the background.

> Important: If you are using an ARM based processor (like an M1 Macbook) you may need to build the image using another service like Cloud Build.
> The error:
> **https://cloud.google.com/run/docs/troubleshooting#deployment**
> The fix:
> **https://cloud.google.com/run/docs/building/containers#builder**

5. Test Converting a Local Video

> Note: When we generate the image, ensure that there is still a raw video file in the root directory of the project.

Again, we can send a POST request to our `/process-video` endpoint using Thunder Client.

The request URL:

```
POST http://localhost:3000/process-video
```

The request body:

```
{
  "inputFilePath": "./nc-intro.mp4",
  "outputFilePath": "./processed-nc-intro.mp4"
}
```

To copy a file out of a running Docker container into your host, you can use the `docker cp` command. Here's how to do it:

```
docker cp <container-id-or-name>:/app/processed-nc-intro.mov
./
```

This command will copy the file from the container to the host system.

6. Clean Up

To list the running containers run:

```
docker ps
```

To stop a running container run:

```
docker stop <container-id-or-name>
```

To list all containers, even those that have been stopped, run:

```
docker ps -a
```

To remove a container run:

```
docker rm <container-id-or-name>
```