CHAPTER 11

CREATING IMAGES AND CONTAINER REGISTRIES

In the previous chapter, we discussed the basics of containerization and explored different aspects of containerization using Docker. In this chapter, we will create a **custom container image** and **push it to a container registry**. Let us first discuss the OCI specifications.

Open Container Initiative (OCI)↗ is a Linux Foundation↗ project aiming to standardize the container ecosystem. It is supported by prominent companies in the industry, including Docker, Amazon, Google, Microsoft, IBM, and many others. The project↗ is open source and consists of three main sub-specifications: `runtime-spec`, `image-spec`, and `distribution-spec`.

The `runtime-spec` defines the standard format for outlining how a containerized application should run and its container lifecycle. The runc↗, a low-level tool for spawning and running containers, implements the `runtime-spec`. Additionally, *containerd*, **the most popular container runtime**, uses **runc** as its underlying layer to run containers. *containerd* also provides a full container lifecycle management API for managing tasks such as image transfers, snapshots, network configuration, and more.

The `image-spec` defines the container image format and its contents, including the blob objects and manifest. It introduces the layer concept to the container images and specifies how layers are structured together to form a complete image. Still, the most essential part of the `image-spec` is the standardized image format, ensuring portability so that OCI-compatible images can be run in any OCI-compliant environment without any changes.

Lastly, the `distribution-spec` defines how to distribute the container images. It describes the protocols and HTTP API specifications used to transfer the container images over a network. Furthermore, it establishes both the HTTP API **blueprints for pushing and pulling container images** to and from a registry, as well as the authentication methods required to access them. Thanks to this specification, the same container image can be stored and distributed by different container registries.

Docker broadly implements the OCI specifications and extends them with some additional features. This enables us to use the Docker CLI to build, tag, push, and pull container images, which can then be run in any other OCI-compliant environment.

11.01. DOCKER BUILD

The build is the initial step of the containerization process. It is the process of taking the source code of an application, adding the dependencies, and turning it into a reusable artifact. Afterward, this container image artifact can be used to run the application in a containerized environment.

We will use Docker CLI to generate the container images, particularly the `$ docker buildx build` subcommand. We could have installed the standalone BuildKit↗ for the task. However, it is easier to use the Docker CLI since it also provides other tools to interact with a registry to push and pull the images.

BuildKit is the new generation build tool that replaced the legacy Docker builder. It is faster and more efficient than the previous builder, and it has many other features, such as **multi-stage builds**, caching, parallelizing build steps, **support for multi-platform images**, and the ability to detect and skip unused build stages. However, the Windows containers are not yet fully supported↗ with the BuildKit. Therefore, you might want to use the legacy builder↗ to create Windows containers.

Docker uses Dockerfile↗ as a blueprint for the build process for both the legacy builder and the new BuildKit. A Dockerfile is a text file that contains the instructions for the build process to create a container image. The `$ docker buildx build` command uses these instructions to build an OCI-compatible image.

A Dockerfile contains one or more instructions, and each instruction starts with an instruction name followed by its arguments. Docker supports more than 15 different instructions. These instructions are executed in sequence, and the output of each instruction is used as the input for the next instruction. Starting from the very first instruction in the base image, these instructions come together in a strict order to create the final image.

To accelerate the build process, Docker utilizes a build cache to skip instructions that have not been modified and have the same inputs. Taking advantage of this feature makes the build process significantly faster.

According to StackOverflow's 2025 Developer Survey↗, JavaScript (JS) is the most popular programming language, and Next.js↗[1] is the most popular JS framework. Therefore, in this section, we will assume that we are building a Next.js application and packaging it as a container image. However, the same process can be applied to any other application, framework, or language.

For the sake of simplicity, we assumed that you have a recent version of the Node.js runtime installed on your machine, preferably using the nvm↗, which we mentioned in the previous chapters. The journey starts with a sample application we create using the official create-next-app↗ CLI. By default, this CLI will prompt for the options provided below, yet to make things as non-interactive as possible, we will supply them directly as arguments.

```
$ npx create-next-app@15.5.12 \
  --typescript \
  --eslint \
  --tailwind \
  --turbopack \
  --src-dir \
  --app \
  --import-alias '@/*' \
  sample-app
```

The npx↗ command is a package runner that comes with *npm (Node Package Manager)* and *Node.js*. It lets users run the executable binary of a package directly. In this case, `npx` will download the version `15.5.12` of `create-next-app` from the npm registry↗ and run it.

After running the command, a `sample-app` directory will be created, containing the sample Next.js application. We will use this application to create an image using a Dockerfile. Let us also create a simple Dockerfile at the root of the application and then explain the instructions.

11.01.basic.dockerfile

```dockerfile
# syntax=docker/dockerfile:1
FROM node:24.7-trixie

WORKDIR /usr/src/app

COPY . .
RUN npm ci
RUN npm run build

ENV NODE_ENV=production
CMD npm run start
EXPOSE 3000
```

In the **first line of the Dockerfile**, we use the `# syntax` line to specify the Dockerfile syntax version. This is optional, but it is recommended that you use it to get the latest features and optimizations. It also helps text editors provide syntax highlighting for Dockerfiles.

In the **second line**, with the `FROM` instruction, we specify the parent image we want to use. In this case, we are using the official Node.js↗ image with the node version `24.7`, which is the latest version at the time of writing. This image is based on the Debian 13↗ codenamed `Trixie`.

The parent image contains the Node.js runtime, npm package manager, and many other dependencies. Both Dockerfiles for `Node.js 24.7` and `Debian 13` are open source and available on the Docker Hub↗.

At **line 4**, we use the `WORKDIR` instruction to set the working directory. This is equivalent to running `$ cd /usr/src/app` in the shell, but it is more persistent. Until we specify a different directory with another `WORKDIR` instruction, the `/usr/src/app` directory will be our working directory for each following instruction. Furthermore, this instruction lets individual instructions freely use the `$ cd` command.

At **line 6**, we use the `COPY` instruction to copy the whole application code, including the `package.json` and `package-lock.json` files, to the working directory. In this step, the first `.` represents the source, meaning the build context specified as the positional argument of the `$ docker buildx build` command, and the second `.` represents the destination, which is the working directory specified with the `WORKDIR` instruction.

Since now we have the necessary files in the working directory inside the image, we can run the `$ npm ci` command to install the dependencies at **line 7** using the `RUN` instruction. The npm ci↗ command is a specialized version of the standard `$ npm install` command, designed for use in CI/CD pipelines, and specifically searches for the `package-lock.json` file to install dependencies. After running this command, the `node_modules` folder will be cleaned, and all packages, including those needed for development, will be freshly installed from the lock file.

On **line 8**, we run the `$ npm run build` command to build the application. This will create a `.next` directory inside the working directory with the application's production build.

With the instructions in the **last three lines**, we prepared our image to run the application. First, we set the `NODE_ENV` environment variable to `production` to ensure the application runs in production mode. This variable will enable many runtime optimizations in libraries.

Then, we use the `EXPOSE` instruction to expose the port `3000` to the outside world. This is optional and doesn't affect the functionality of the service running inside the container. Still, it is a good practice to do so, as container runtimes might be more aggressive and block unexposed ports for security reasons. Furthermore, the `EXPOSE` instruction is helpful when we need to determine which ports the application uses, as seen when inspecting the image or Dockerfile.

Finally, we use the `CMD` instruction to run the `$ npm run start` command to start the Next.js application. The difference between `CMD` and `RUN` instructions is that `RUN` instructions are executed during the build process, but `CMD` instructions are executed during the runtime of the container. Since the `CMD` instruction sets the command to be executed when running a container from an image, there can only be a single `CMD` instruction in a Dockerfile. If there is more than one `CMD` instruction, only the latest one will take effect.

Let us build the image with the `$ docker buildx build` command which has multiple aliases including the `$ docker build` command.

```
$ docker buildx build \
  --no-cache \
  --progress=plain \
  --platform linux/amd64 \
  --tag chlabs.io/web-app:0.0.1 \
  --file ~/projects/sample-app/11.01.basic.dockerfile \
  ~/projects/sample-app/
```

This command has various options and a single positional argument. The single positional argument is the final argument that specifies the build context directory as `~/projects/sample-app/`.

A build context might be a path, a URL, or the `-` character. The `-` character lets us read the build context from the standard input, which we can use to specify only the Dockerfile to create a contextless build. Contextless builds are useful when we do not want to copy any files from the host machine; instead, we can use the instructions in the Dockerfile to form the image and build it. In the following chapters, we will use this technique to **create a custom PostgreSQL image with extensions**.

In the second and third lines, we use the `--no-cache` to instruct Docker not to use the cache when building the image so that we can see the logs of the build steps. The `--progress=plain` argument is used to get the container build process output. Other progress values can be used to get a more structured output, such as `rawjson`. Both of these arguments are useful while improving or troubleshooting the build process. However, you would certainly want to remove the `--no-cache` argument in a production environment.

The default value for the platform is the one on which the Docker daemon is currently running, which, in my case, is `linux/arm64` since it is running on an M series Mac. Even though we specified another platform, the Docker image will still work on the Mac, since M series Macs also support the `linux/amd64` architecture using the Rosetta 2 translation layer.

Fifth line specifies the tag for the image. The tag is the identifier of the image. An image can have multiple tags, and we can use the `$ docker image tag` command to add more tags to an existing image. Most of the time, the tag keyword is used interchangeably as the repository name, the version tag, and the full identifier. Nevertheless, a descriptive tag should have these properties:

- A prefix that indicates the base project, an organization, or the team that maintains the image. For example, `chlabs.io` refers to the domain from which the image will be served, `dc` represents the digital channels, and `nextjs` refers to the project.
- A subdomain, if possible. If a project has multiple images or sub-projects, we can use the subdomain to make it easier to identify. For instance, even though we are building a single image, we are using the `web-app` subdomain to make it easier to identify.
- A version number. By default, Docker separates the identifier string into two parts using the (`:`) colon character. Avoid using the `latest` version tag in the earliest stages since it is not descriptive and can lead to unexpected behavior.

In this case, our tag is `chlabs.io/web-app:0.0.1`. It describes the image as part of the `chlabs.io` project, a web application, and the first version of the app. In a Docker image tag, the first part before the colon, `chlabs.io/web-app` is the repository name, and the second part after the colon, `0.0.1`, is the actual version tag.

To add multiple tags or platforms, we can either repeat multiple options like `--platform linux/amd64 --platform linux/arm64` which is preferred or use a comma to separate them in a single one like `--platform linux/amd64,linux/arm64`.

In the sixth line, we are providing the path of the Dockerfile. If this option is a relative directory, it will be counted as relative to the build context directory. By default, Docker will use the `Dockerfile` (just "Dockerfile", without any prefix or suffix) in the build context directory if not specified.

At the last line, we specified the build context directory as the positional argument. When building images, I try to use full paths instead of relative paths whenever possible. This helps to avoid any confusion and makes the command more readable.

After running the build command, we should see the build steps in the output ending with an `exporting to image` step. However, this image has three big problems in the order of their importance:

1. **The image contains the application code.** After building the `Next.js` application, we did not delete the application code and the Dockerfile from the image. This might be a significant security risk if the image is leaked or, due to a misconfiguration, the working directory becomes publicly accessible.
2. **The build process is unoptimized.** We installed the dependencies using the `npm ci` command, which installs both the development and

production dependencies. Yet, after building the application, we no longer need the development dependencies, so it is a waste of disk space. Additionally, we should process the less-changing parts of the project, such as `package.json` and `package-lock.json`, first to maximize the benefits of the cache.

3. **It uses the shell form of the CMD instruction.** While defining the `CMD` and `ENTRYPOINT` instructions, we can either use the shell form (e.g. `CMD npm run start`) or the JSON form (e.g. `CMD ["npm", "run", "start"]`). Both of these instructions can be overridden later at the start of the container. However, it is a good practice to use the JSON form to prevent unintended behavior↗. This is because the shell form will be overridden by the build process[2] and the command will be executed as a child process to a shell which doesn't pass OS signals to the command. As a result, the container will not be able to handle signals such as `SIGINT` to exit gracefully, and will be forcefully terminated when it needs to be stopped.

To fix these issues and provide Docker with guidance on optimizing the build process, we will create a multi-stage Dockerfile. Multi-stage builds have numerous benefits over single-stage builds. However, they are also more complex to understand and debug. Let us create a multi-stage Dockerfile.

11.02.multi-stage.dockerfile

```dockerfile
# stage 1
FROM node:current-slim AS builder

WORKDIR /usr/src/app
COPY package*.json ./
RUN npm ci

COPY . .
RUN npm run build
RUN npm prune --omit=dev

# stage 2
FROM node:22-alpine AS runner

WORKDIR /usr/src/app
COPY --from=builder /usr/src/app/node_modules ./node_modules
COPY --from=builder /usr/src/app/package.json ./package.json
COPY --from=builder /usr/src/app/.next ./.next
COPY --from=builder /usr/src/app/public ./public

EXPOSE 3000
ENV NODE_ENV=production
CMD ["npm", "run", "start"]
```

We made lots of improvements in the second Dockerfile. First, we split the build process into two stages: `builder` and `runner`.

By placing the dependency installation step as early as possible and as a separate stage, we instruct Docker to isolate it from the rest. This way, Docker can easily mark the inputs of this stage and use the cached output if no changes are detected in the `package.json` and `package-lock.json` files. Rather than the previous approach, this will make the caching more effective and straightforward to avoid misses. The

`COPY` instruction also supports wildcards; we take advantage of this feature to copy both `package.json` and `package-lock.json` files.

After installing the dependencies, we copied the rest of the application code from the build context. Then, we run the `$ npm run build` command to build the application. After building the application, we run the `$ npm prune --omit=dev` command to remove the development dependencies from the `node_modules` directory, since we do not need them in the final image.

In the second stage, we copied the production dependencies from the `node_modules` directory, the `package.json` file, the `public` directory with static files such as images, and the `.next` directory to the final image. The `.next` directory contains all the code needed for the *Next.js* application to run. These are the only elements we need in the final image to run the application.

We used two different parent images in this Dockerfile. The `build` stage used the current `current-slim` version of the Node.js image, which resolves the latest stable version `24.7.0` based on a Debian 12 image. This parent image will improve the build process by using the cutting-edge features of the Node.js runtime. On the other hand, the `runner` stage used the latest LTS version of the Node.js image based on an extremely minimal Alpine↗ image. In this way, we can expect a more stable image artifact with a smaller size and attack surface while running the application in production.

Being able to use multiple parent images might be even more useful in some codebases, such as a Kotlin Spring Boot application, where the code is compiled with `JDK 21` and run using a `JRE 8`. But remember that the Docker build system will cache the corresponding images of the parent images unless they are explicitly removed or forced to be updated using the `--pull` flag during the build process.

The size of the single-step image (chlabs.io/web-app:0.0.1) was `1.9 GB`, while the multi-stage image (chlabs.io/web-app:0.0.2) was `607 MB`. That is a **68% reduction** in image size before compression. After compression, the single-step image is `733 MB`, and the multi-stage image is `188 MB`. That is an impressive **75% reduction** in the final image size[3].

Even though we significantly improved the image, there is still room for further optimizations and security enhancements, such as:

- Currently, we are copying everything from the build context to the image, including the `.git` and `node_modules` directories. By using a .dockerignore↗ file, we can tell Docker to ignore some files and directories from the context. This is especially useful for removing sensitive data, such as `.env` files. Remember that, unlike the `.gitignore` and many other ignore files, the `.dockerignore` file is a single file located in the root of the build context, and Docker does not consider the `.dockerignore` files inside the subdirectories.
- We do not need the `package.json` file in the final image since we can directly run the equivalent of the `$ npm run start` command. Also, The `package.json` file is a security risk because it contains the version numbers of the dependencies we use. Another security optimization would be to create a custom user, like `1001`, to run the application instead of using the root user.
- We can remove optional dependencies that are relatively large in size, such as swc↗, by adding the `--omit=optional` parameter to the

`$ npm prune` command. Without the optional dependencies, we can save more than 200 MB in the final uncompressed image and achieve a compressed version with only 99 MB.

According to [USENIX↗] paper, on average, pulling images accounts for 76 percent of the container start time, and only 7 percent of the data is read from the image. These findings highlight how the size of a container image can significantly impact the container startup time.

- The `.next` folder contains build artifacts such as the `trace` file or source `.map` files that are only needed for debugging and shouldn't be included in the final image. Most of the time, we need to share these artifacts with error reporting tools such as [Sentry↗].
- In JavaScript applications, the largest portion of the image size will be due to the `node_modules` directory. To check and analyze the size of this directory, we can add a `$ du` command to the Dockerfile. The instruction below will display the heaviest 50 directories and files in `node_modules` at each run, which is a consistent way to easily identify large dependencies.

```
RUN du -h ./node_modules | sort -rh | head -n 50
```

However, as every project has different requirements, it is up to you to decide how much further to optimize the image.

After building the image, we can inspect it in more detail using the `$ docker image inspect` command, providing either the Image ID or the repository name and tag. The inspect command outputs the image metadata in JSON format, including exposed ports, environment variables, architecture, layers, and more.

```
$ docker image inspect chlabs.io/web-app:0.0.2
...
"RootFS": {
  "Type": "layers",
  "Layers": [
    "sha256:418dccb7d85a63a6aa574439840f7a6fa6fd2321b3e2394568a317735e
    "sha256:de2a20a843eb35e20246aac401fda0b9f97a35a228aac890fe2151143a
...
```

A Docker image consists of layers. **Each layer is read-only and represents and contains the changes that are made.** These layers are stacked on top of each other to form the final image. Additionally, they can be shared or reused across images, as we did with the parent images.

Using the `$ docker image history` command, we can see the layers of the image starting from the parent image, their sizes, and the commands that are used to create them. Or, we can use dive↗ to get a more detailed view of the image layers.

```
$ docker image history chlabs.io/web-app:0.0.2
IMAGE          CREATED BY                                       SIZE
dabf70822791   CMD ["npm" "run" "start"]                        0B
<missing> ENV NODE_ENV=production 0B        buildkit.dockerfile.v0
<missing> EXPOSE map[3000/tcp:{}]                           0B        bu
<missing> COPY /usr/src/app/public ./public                 3.31kB    bu
<missing> COPY /usr/src/app/.next ./.next                   6.04MB    bu
<missing> COPY /usr/src/app/package.json ./package.jso…     586B      bu
<missing> COPY /usr/src/app/node_modules ./node_module…     219MB     bu
<missing> WORKDIR /usr/src/app                              0B        bu
<missing> CMD ["node"]                                      0B        bu
<missing> ENTRYPOINT ["docker-entrypoint.sh"]               0B        bu
<missing> COPY docker-entrypoint.sh /usr/local/bin/         388B      bu
<missing> RUN /bin/sh -c apk add --no-cache --virtual …     5.37MB    bu
<missing> ENV YARN_VERSION=1.22.22 0B        buildkit.dockerfile.v0
<missing> RUN /bin/sh -c addgroup -g 1000 node     && …     146MB     bu
<missing> ENV NODE_VERSION=22.19.0 0B        buildkit.dockerfile.v0
<missing> CMD ["/bin/sh"]                                   0B        bu
<missing> ADD alpine-minirootfs-3.22.1-x86_64.tar.gz /…     8.31MB    bu
```

For example, when we build the image on top of a `node:22-alpine` parent image, the parent image layers will always be the same for every image we build from the same parent image. This way, we do not have to download layers of the parent image again or store the same layers for every new version in the image repository. This layer structure saves both disk space and time in the build process. Moreover, the later the changes are in the Dockerfile, the more layers will be reused.

Let us take a look at some of the other image related Docker CLI commands:

- The **`$ docker image remove <tag>`** command removes the image from the local machine. If a container uses an image, we must first remove the container before removing the image. Furthermore, if the image has more than one tag, this command will remove the tag instead of deleting the image since Docker treats tags as references. The image will be deleted only after all the tags are removed, and even then, some layers may remain on the local machine if other images use them.

  We can also use the `$ docker image prune --all` command to remove all unused and dangling images in a single command.

- `$ docker image save <tag>` streams the image in tar format to standard output. Then, we can use the `>` redirection operator to save the tar output to a file. Alternatively, we can pipe it to another command like `$ docker image save chlabs.io/web-app:0.0.2 | tar xf -` to extract the image layers, manifest, and other metadata to a directory.
- The `$ docker image load <path>` command loads the image from the tar file to the local machine. This command can take either the file as a positional argument or from the standard input. Using the standard input, we can redirect the save pipe to the load command using any tool, such as `$ scp`, to copy the image to a remote machine. Nonetheless, there are much better ways to copy the image to a remote machine, which we will explore very soon.

We regularly use two other fundamental commands to download images from and upload images to the container registries. The `$ docker image pull <tag>` and the `$ docker image push <tag>`. However, we first need to learn about Container Registry.

11.02. CONTAINER REGISTRIES

A container registry stores images and their metadata, providing a way to download and upload them. By taking advantage of the layered structure we mentioned earlier, container registries use significantly less disk space than storing full images separately.

In addition to the primary purpose of storing the images, container registries also provide a way for images to be searched for vulnerabilities and other metadata. Identifying security issues in images before they are deployed is crucial for enforcing DevOps security compliance. Furthermore, these enforcement measures are typically required by regulations.

Each prominent cloud provider offers its own container registry service. For example, AWS has Amazon Elastic Container Registry (ECR)↗, Google Cloud has Google Cloud Artifact Registry↗, and Microsoft Azure has Azure Container Registry↗. Additionally, the OCI Distribution Specification↗ helps smaller companies build their own compatible registries, such as DigitalOcean↗, Scaleway↗, and Vultr↗.

Even though nearly any company provides a container registry solution, the most popular one is still the Docker Hub↗, as it is entirely free for public repositories and hosts nearly all parent images. Also, it is the default registry for the Docker CLI.

To make things more understandable, let us create a private repository to deploy the image we built earlier. However, because of the slight variations between different container registries, we will split this section into two paths. In the first path, we will create a private repository inside the AWS ECR, the largest cloud provider's container registry, and push the image to it. In the second path, as a generic example, we will create a private namespace inside the Scaleway Container Registry and perform the same task.

Even if you do not have an account with these cloud providers or are using another container registry, I recommend reading both to understand their differences and similarities.

11.02.01. AWS ELASTIC CONTAINER REGISTRY

The AWS ECR is the container registry service provided by Amazon Web Services. It is a fully integrated part of the AWS ecosystem, and like any other AWS service, it is highly reliable and scalable with a pay-as-you-go pricing model.

The private registry creation process starts with creating an AWS account and logging into the **AWS Management Console**. Besides a few exceptions, the services and resources created inside the AWS Management Console are bound to the region where you created them. For example, if you created a repository in the `us-east-1` region, you will not be able to see that repository in the `us-west-1` region. However, this only applies to the Management Console; you can always access resources from other regions or the internet.

After choosing the region and selecting the **Amazon ECR service**, we should see a "Create a Repository" button. Let us click on it to access the private repository creation page. Unlike a public repository, a private repository can only be read and written by the user who created it.

On the repository creation page, in the General Settings section at the top, we should see a "Repository name" field. This field starts with a URL label and ends with a text field. The URL format is `<account-id>.dkr.ecr.<region>.amazonaws.com/` and the text field format is `<namespace>/<repository-name>`. Let us explain the namespace and repository name fields.

A namespace is similar to a project or folder within the registry. We can use it to group the repositories. A repository, on the other hand, is like a subfolder inside the namespace. Both are different levels within the registry hierarchy and are beneficial for organizing images.

In our example, we will use `chlabs.io` as the namespace and `web-app` as the repository name. Both of these fields are required and cannot be changed later.

After choosing the namespace and the repository name, we should select the `Tag Immutability` setting. This setting is useful for preventing unexpected behavior, such as deploying two different images with the same tag. I strongly encourage you to enable this setting to ensure the repository's integrity is maintained. The last thing you would want is to discover there is another image with the same name and version tag in the system.

There are also two additional security settings to configure when creating a repository within AWS ECR. The first is `Encryption Configuration`, and the second is `Scan on Push.` Using the

`Encryption Configuration` setting, we can choose to encrypt the images in the repository while they are stored and decrypt them when we pull them. The `Scan on Push` setting is used to scan images for vulnerabilities when they are pushed to the repository. Both of these settings are optional and are used to comply with security regulations mandated by certain sectors, such as the financial industry.

Click the `Create` button to create the repository. After a few seconds, we should be redirected to the `Private Repositories` page and see our newly created repository inside the list. Next to the name of the repository, we should see a URI to the repository.

`URI` is an acronym for Uniform Resource Identifier. In the context of the Container Registries, it is used to identify the repository in URL form. We will use this URI to interact with the repository; in my case, the URI is displayed in the first output in the code snippet below. From now on, to push the image to the repository, we should also tag our image with a prefix that includes this URI, as shown in the second output, which uses the version tag `0.0.2`.

```
URI:
154971117931.dkr.ecr.us-east-1.amazonaws.com/chlabs.io/web-app

New Tag:
154971117931.dkr.ecr.us-east-1.amazonaws.com/chlabs.io/web-app:0.0.2

Format:
<registry-host>/<namespace>/<repository>:<tag>
```

Before pushing the images to the repository, we need to authenticate our user using the `$ docker login` command. The most widely available and easiest authentication method is Basic Authentication, also known as the username and password method. In this method, we need to provide three arguments: The repository URI, the username, and the password.

- The repository or namespace URI is the URL we got from the repository creation page. Whenever we attempt to interact (push or pull) an image name that is prefixed with this repository URI, Docker will recognize that it should use these credentials.
- The username is always `AWS` for the AWS ECR.
- For the password, we need a token from the AWS CLI, and to get this short-lived token, we need an access key and a secret key.

Since the password creation process is a bit tedious and complex, let us go over it step by step.

First, let us install the AWS CLI by following the instructions on the official documentation↗. At the end of the installation process, we should be able to see a meaningful output for the `$ aws --version` command.

After installing the CLI, log in to the AWS Management Console and navigate to the Identity and Access Management (IAM) service. In the navigation pane at the left, click on the `Users` section and then click the `Create user` button. I try to create individual users for each project and environment and assign them only the necessary permissions. That way, if a user account is compromised or suspected to be compromised, the damage is limited to a single part of the project and environment.

Since this user will be used to interact with the AWS ECR from the local machine's command line, let us name our user `local.cli`. For the same reason, we can turn off the "Provide user access to the AWS Management Console" option and click the `Next` button.

The next screen is the permissions page. In this step, we have three options to add permissions to the user:

- We can add the user to an existing group to let the user inherit the permissions from the group.
- We can copy permissions from an existing user.
- Lastly, we can attach inline policies to the user directly.

Since attaching policies directly is the easiest and most flexible option, let us select that radio option. At this stage, we should see a list of `Permission policies` with over a thousand policies. In the search bar, search for `ec2` and select the `AmazonEC2ContainerRegistryFullAccess` policy using the checkbox on the left of the row.

In AWS, the policies include the permission list that the user can have. The AmazonEC2ContainerRegistryFullAccess↗ policy grants over 50 permissions, enabling users to access and manage all resources in the Amazon EC2 Container Registry. These permissions include pushing and pulling images, creating repositories, and initiating security scans.

After selecting the policy, click the `Next` button on the bottom right of the page. After that, we should see a `Review and create` page with the user name and the permission policy we just added. If the permission policy list is missing the `AmazonEC2ContainerRegistryFullAccess` policy, please go back and check the previous steps. Click the `Create user` button, and we will be directed to the `Users` page.

Now, we have created our `local.cli` user, let us add a `Security Credential` to access the services. Click on the user we just created to access the user details page. At the `Security credentials` tab, we will see lots of options to create different types of security credentials. Scroll down to the `Access keys` section and click the `Create access key` button.

Let us choose the `Command Line Interface` option from the option list, enable the checkbox that says, "I understand the above recommendation and want to proceed to create an access key." and click the `Next` button. On the next screen, you can set a description for the access key if desired, and then click the `Create access key` button.

After that, we should see a `Success` message displaying the Access key and the Secret access key. The access key is public and can be used to identify the user or the key. However, the Secret access key is private and should be kept in a secure location. We can click the `Show` button to see the Secret access key or use the buttons on the left of the keys to copy them to the clipboard.

In my case, the Access key is `AKIASIFH7TFVSMQ2XRAA`, and the Secret access key starts with `677` and is 40 characters long.

After getting these credentials, let us use the `$ aws configure` command to configure the AWS CLI with the keys we just created. This command will prompt us to provide the Access key ID, Secret access key, region, and output format. We can press `Enter` to use the default values.

```
$ aws configure
AWS Access Key ID [****************3CU5]: AKIASIFH7TFVSMQ2XRAA
AWS Secret Access Key [****************vOXH]: 677................
Default region name [us-east-1]:
Default output format [json]:
```

After setting up the user with the AWS CLI, we should be able to run the `$ aws ecr get-login-password --region <region>` command to get the password token for the repository.

```
$ aws ecr get-login-password --region us-east-1
eyJwYXlsb2FkIjoiTjYwUzBPNFRGdkowcnlCYkRuaVFjMXBFRWd.....
```

Now that we have created the password token, we can use the `$ docker login` command to authenticate our user. Since this password token is pretty long (~1700 characters), it is better to redirect the output to the `login` command like the following code piece:

```
1  $ aws ecr get-login-password \
2    --region us-east-1 \
3    | docker login \
4    --username AWS \
5    --password-stdin \
6    154971117931.dkr.ecr.us-east-1.amazonaws.com/chlabs.io/web-app
7  Login Succeeded
```

As we can see, this command has two parts. The first part is the **`$ aws ecr get-login-password`** command we saw earlier. With the help of the pipe operator **`|`** at the third line, we directed the output to the **`$ docker login`** command.

The second part is the **`$ docker login`** command. The first argument is the username **`AWS`**, and the second is the password on line 5, which we specified that it will be taken from the standard input using the **`--password-stdin`** option. The third and only positional argument is the repository URI, which we got from the private repository list.

After running the command, the Docker CLI sends a request to the repository URI to verify the provided credentials. If they are, the Docker CLI will output a **`Login Succeeded`** message. Additionally, if we have more than one repository inside the same namespace, we can use the namespace URI instead of the repository URI (without the **`/web-app`** part), instead of logging in multiple times.

After the login process, we can retag the image with the repository URI and push it using the **`$ docker tag`** and **`$ docker push`** commands.

```
$ docker tag \
  chlabs.io/web-app:0.0.2 \
  154971117931...amazonaws.com/chlabs.io/web-app:0.0.2
$ docker push \
  154971117931...amazonaws.com/chlabs.io/web-app:0.0.2
The push refers to repository [154971117931.dkr.ecr.us-east-1.amazonaw
59c492476b50: Layer already exists
7f8bd23fb241: Pushed
...
```

The `$ docker tag` command takes two arguments. The first one is the current image name or tag, and the second one is the new tag. In the example above, we retagged the image from `chlabs.io/web-app:0.0.2` to `154971117931...onaws.com/chlabs.io/web-app:0.0.2`. Due to the horizontal character limit in the code block in the book, we had to use `...` instead of the full URI when displaying the command.

The `$ docker push` command takes the image tag and pushes the image to the repository. The push process happens layer by layer, and the Docker CLI will send a request to the repository to check if some layers are already available. If not, the CLI will push the missing layers to the repository.

After pushing the image, we can view it inside the repository in the AWS ECR web interface. We can also view the image details, such as the digest, size, and tags. From now on, we can use the `$ docker pull <tag>` command to pull the image from the repository using the same tag from any other authenticated machine.

As a side note, the password token created by AWS ECR expires after 12 hours for security reasons. These short-lived tokens are common security features by cloud providers, also found in Google Cloud's Artifact Registry and Microsoft Azure's Container Registry. Nevertheless, they all offer a way to create a new token in various environments, such as the CI/CD pipelines, which we will cover in the following chapters.

11.02.02. SCALEWAY CONTAINER REGISTRY

As a generic option, the Scaleway Container Registry↗ is a service provided by Scaleway↗. Scaleway is one of the biggest cloud providers in Europe and offers a wide range of services from simple virtual machines to more complex container orchestration services.

The Scaleway Container Registry is rather similar to the AWS ECR. Yet, it lacks some AWS ECR features, such as tag immutability and image scanning. However, it is reliable and considerably (nearly 10 times) cheaper than alternatives. It is also where I host my private images.

Unlike AWS ECR, the Scaleway Container Registry allows users to create namespaces instead of repositories. This way, we can create multiple repositories within the same namespace **without needing to revisit the web interface**.

Let us log in to the Scaleway Console (Scaleway's web interface) and follow the steps: *Containers* -> *Container Registry* -> *Create namespace*. The namespace creation process is nearly identical, except for the nuance mentioned in the previous paragraph.

The process starts with a name. In our case, we will provide the name of the namespace as `chlabs-io` since the dot character is not allowed in the namespace name in Scaleway, unlike AWS ECR. After the name, we should choose the region. The region is the physical location where the image metadata and layers will be stored. Scaleway offers three regions, all located in Europe: Paris, Amsterdam, and Warsaw. You can choose any of them.

After the region, we should choose the privacy of the namespace. Set the privacy to `Private` to ensure the repositories and images inside the namespace are not accessible publicly, and click the `Create namespace` button at the bottom of the page.

Now that we have created the namespace, we need credentials to let Docker CLI access it. The namespace URL format is `rg.<region>.scw.cloud/<namespace>` in Scaleway, which can be found in the namespace details page as the `Registry endpoint`. In my case, the URL is `rg.fr-par.scw.cloud/chlabs-io`. The username is always `nologin`, and the password is the token obtained from the Scaleway Console. Let us continue with getting the password token from the Scaleway Console.

Log in to the Scaleway Console and navigate to the `Identity and Access Management (IAM)` page.

Although Scaleway supports advanced permission management through projects, applications, and policies, we will not be using them in this example. We will just create an API Key for our user with all the permissions on all projects.

Click on the `API Keys` section and follow the `Generate API Key` steps. We should see a `Generate an API key` dialog. In the dialog, select `API key bearer` as `Myself (IAM user)`. After that, select `1 Month` for the `Expiration` field. If you want to create a key that will never expire, you can select `Never`, or you can use the Scaleway CLI↗ to generate new keys as the old ones expire.

Leave the other fields as default and click the `Generate API key` button. You should now see `Access Key ID` and `Secret Key`. On Scaleway, the Access Key ID is used to identify the keys, similar to AWS Access keys, and the Secret Key is used to authenticate requests as the password. Let us copy the `Secret Key` to the clipboard, which is a `UUIDv4` string.

Next, let us log in to the namespace with the Docker CLI using the `$ docker login` command.

```
$ docker login \
  --username nologin \
  --password XXXXXXXX-XXXX-4XXX-XXXX-XXXXXXXXXXXX \
  rg.fr-par.scw.cloud/chlabs-io
WARNING! Using --password via the CLI is insecure. Use --password-stdi
Login Succeeded
```

Instead of using the `--password-stdin` option, this time we provided the secret key as the password directly to the `$ docker login` command. As the warning message suggests, this is not a secure way to provide the password, since the entire command, including the secret key, will be visible in the process list and saved in the command history.

For a more secure approach, we can retrieve the secret key from a secure source and assign it to an environment variable, such as `$SCW_SECRET_KEY`. Then, we can echo and pipe the secret key to the login command like `$ echo $SCW_SECRET_KEY | docker login ...`. Alternatively, we can use the `<<<` operator to provide the secret key to the login command as `$ docker login ... --password-stdin <<< $SCW_SECRET_KEY`.

We provided the namespace URL that we got from the Scaleway Console as the last parameter to the `$ docker login` command. After executing the command, the Docker CLI sends a request to an endpoint that checks if the credentials are valid. If they are, the CLI will output a `Login Succeeded` message.

From now on, we can retag the image by prefixing it with the namespace URL and pushing it to the registry. The repository will be created automatically upon pushing the image to the registry. As a result, any repository inside the namespace will have the same security and privacy settings as the namespace.

```
$ docker tag \
  chlabs.io/web-app:0.0.2 \
  rg.fr-par.scw.cloud/chlabs-io/web-app:0.0.2
$ docker push \
  rg.fr-par.scw.cloud/chlabs-io/web-app:0.0.2
59c492476b50: Pushing [===========>     ]  104.1MB/144.8MB
7f8bd23fb241: Pushed
...
```

The repository name can also include the `/` character if you need multiple levels of repositories inside the namespace. For example, `rg.fr-par.scw.cloud/chlabs-io/web-app/v1` and `rg.fr-par.scw.cloud/chlabs-io/web-app/v2` are two different repositories inside the same namespace.

The credentials we provided to the Docker CLI in the login command are stored inside the external credential store, such as the native keychain on macOS or the Credential Manager on Windows. Although Docker CLI does not have a built-in way to list the stored credentials, namespace, or repository URLs, we can print the content of the `~/.docker/config.json` file to see the list of URLs.

```
$ cat ~/.docker/config.json
{
  "auths": {
    "154971117931.dkr.ecr.us-east-1.amazonaws.com": {},
    "rg.fr-par.scw.cloud": {}
 },
  "credsStore": "osxkeychain",
  "currentContext": "orbstack"
}
$ docker logout rg.fr-par.scw.cloud
```

As shown above, we logged into two different registries. The first one is AWS ECR, and the second is Scaleway Container Registry. We can use the `$ docker logout <registry-host>` command to log out from the registry and remove the credentials from the external credential store.

In the registry or namespace URL, the first part before the slash character, `/`, is called the `registry hostname` with the format of `<registry-host>/<namespace>/<repository>:<tag>`. If none are provided, the Docker CLI will assume that we want to use Docker's public registry, which is `registry-1.docker.io`. The foundational tools for these registries are developed by Docker, Inc. and donated to the Cloud Native Computing Foundation (CNCF)↗. You can find more information about registries, including other authentication methods, API specifications, and storage drivers, in the official CNCF Distribution documentation↗.

As a security measure, when interacting with the images, we can use digest references instead of version tags to identify the image as a security measure. The digest is a unique identifier (a SHA-256 hash) of the image that is created when the image is pushed to the registry. To learn the digest of the image, we can either use the `$ docker image list --digests` or the `$ docker image inspect <tag>`.

```
$ docker image list --digests
$ docker image inspect \
  --format='{{range .RepoDigests}}{{println .}}{{end}}' \
  rg.fr-par.scw.cloud/chlabs-io/web-app:0.0.2
154971117931.dkr.ecr.us-east-1.amazonaws.com/chlabs.io/web-app@sha256:
rg.fr-par.scw.cloud/chlabs-io/web-app@sha256:92db2ba42857ebc313f6e8513
sha256:92db2ba42857ebc313f6e85134dded3a7f3911b0e305a6e2755c7372b0b8144
$ docker container run \
  rg.fr-par.scw.cloud/chlabs-io/web-app@sha256:92db2ba42857ebc313f6e85
```

Digests ensure the image is immutable and verifiable, independent of the registry or version tags. Still, their use is uncommon unless security is the top priority.

In this section, we learned how to build our own images and push them to the container registries. In the next section, we will discuss container orchestration solutions for deploying our images to production environments.

FOOTNOTES

1. Next.js has suffered from several security vulnerabilities in the near past like CVE-2025-55182↗ and CVE-2025-66478↗. They were **by far** the most impactful vulnerabilities of 2025. It is recommended to use

an SBOM (Software Bill of Materials) tool to track vulnerabilities in your application's dependencies. ↩

2. At the start of container launch, the container runtime will decide the main process command by combining the `ENTRYPOINT` and `CMD` instructions. If any of these instructions are set in the shell form, the build process will override it with `/bin/sh -c` to execute the command as a child process to a shell, and the shell will not pass OS signals properly to the command.

   To resolve this issue, many containers rely on a script such as `docker-entrypoint.sh` to parse the commands and its arguments, replacing the main shell command with the specified command and arguments using the `$ exec` directive of the shell. However, the easiest way to fix this issue is to use the JSON form of the `CMD` and `ENTRYPOINT` instructions. ↩

3. We can learn the raw image size by using the `$ docker image inspect` command. When pushing to or pulling from the container registries, the images will be compressed using the `gzip` to reduce the size of the layers. To get the compressed image size, we can use the `$ docker image save` command, pipe it to the `gzip` command, and save the output to a file.

```
$ docker image inspect <tag> --format='{{.Size}}'
$ docker image save <tag> | gzip > <tag>.tar.gz
$ ls -lh <tag>.tar.gz
```

↩