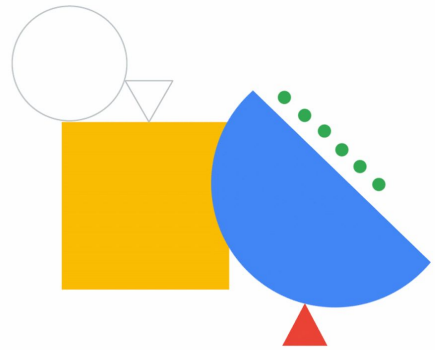


Introduction to Containers



In this module, we introduce Containers and container images, discuss some of their features, and how to build and run them.

You also complete a lab in which you build containers with Docker in Google Cloud.



Agenda



- 01 Containers and container images
- 02 Building container images with Docker
- 03 Lab: Creating and running Docker containers
- 04 Building container images with buildpacks
- 05 Continuous integration and delivery tools
- 06 Best practices
- 07 Quiz

Here is the list of topics in this module.

We'll first discuss what are containers and container images, how you can build container images with Docker, and complete a lab.

You'll learn about buildpacks and CI/CD tools for building and deploying containers, and some best practices to use in this process.

We'll end the module with a short quiz on the topics that were discussed.

01 Containers and container images

Agenda

Let's first discuss what are containers and container images.

What is a container?

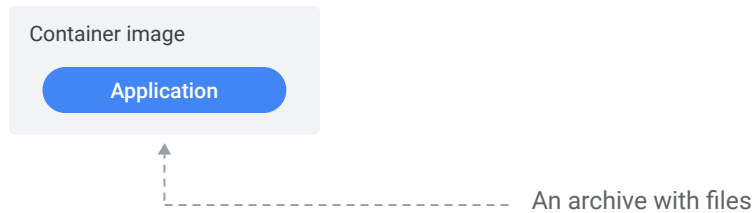


A container is a package of:

- Application code
- Dependencies
 - Programming language runtimes
 - Software libraries

A container is a package of your application code together with dependencies such as specific versions of programming language runtimes and libraries that are required to run your software services.

Container image

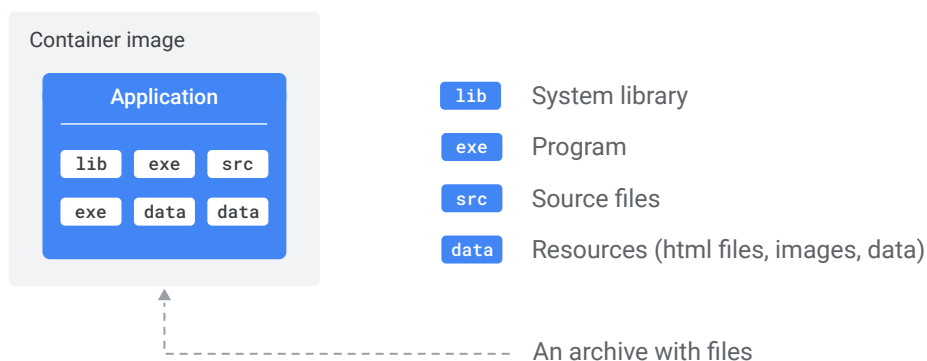


A container image represents a template that defines how a container instance will be realized at runtime.

A container image packages your application along with everything your application needs to run.

For example, in a container image for a Java application, your application is packaged together with the appropriate Java Virtual Machine.

Container image - Archive of files



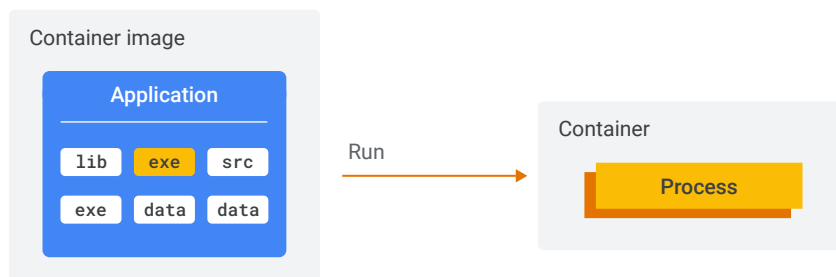
Practically, a container image is an archive with files that contains everything your application needs to run.

A container image includes system libraries, executable programs, and resources, such as (but not limited to) html files, images, binary blobs, and your application source files.

A container image can contain programs written in any programming language, for example Java, Python, JavaScript, PHP, or Go. It can include any binary dependency that you need.

This packaging turns your code and resources into something that you can store, download, and send somewhere else.

Container - Runtime instance

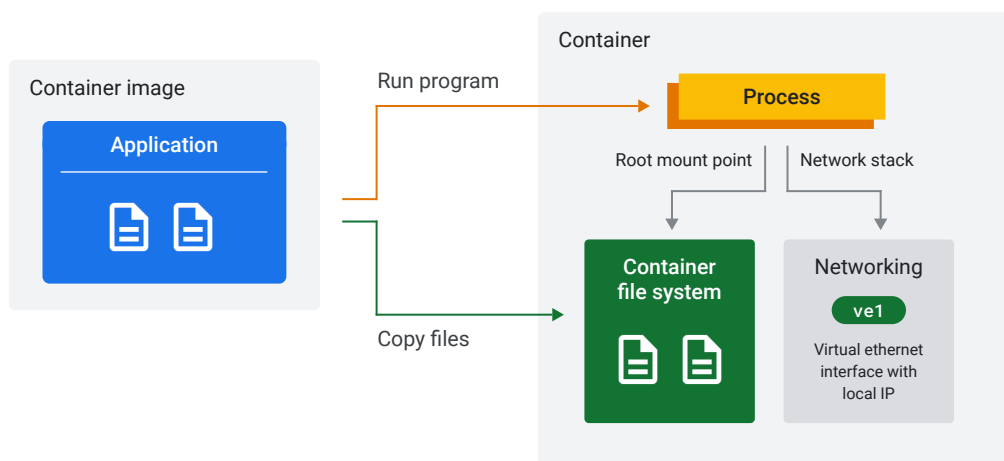


When you run a container image, you execute one of the programs inside the container image.

For example, if your container image contains a Java application, the program you execute is the Java Virtual Machine.

A container represents the running processes of your application and only exists at runtime. If there are no running processes, there is no container.

Container - File system and network stack



Two other relevant things happen when a container runs:

- The contents of the container image are used to seed a private file system for the container. (All files that the processes of your application see)
- The processes in the container have access to a *virtual network interface* with a local IP, so your application can bind to this interface and start listening on a port for incoming traffic.

Remember

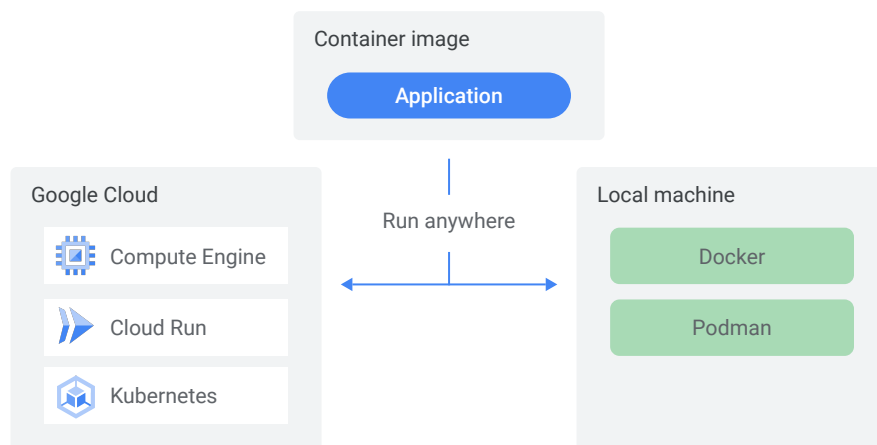
- 1 A container image is an archive with files.
- 2 A container image includes your application and all dependencies that the application needs to run.
- 3 A container is a runtime instance of a container image.



Here are a few things to remember:

- A **container image** is an archive with files: it includes executables, system libraries, data files, and more.
- A container image is **self-contained**. Everything that your application needs to run is inside the container image. If your application is a Node.js application, for example, your source files are in the image along with the Node.js runtime.
- A **container** is a runtime instance of a container image and represents the running processes of your container.

Running containers



After your application is packaged into a container image, you can run it anywhere. On Google Cloud, you can run containers on a Compute Engine in a virtual machine, or on a Kubernetes cluster, or on Cloud Run. On your local machine, you can use the Docker or Podman container runtimes.

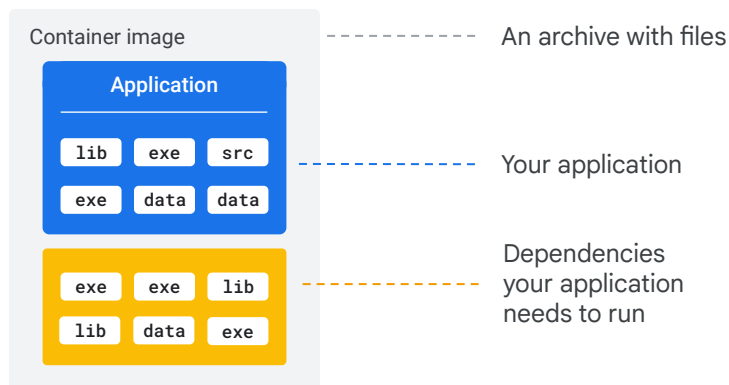
[Docker](#) is an open platform for developing, and running container-based applications.

[Podman](#) is an open source Linux tool that is used to build, and run applications using containers that are based on the Open Containers Initiative ([OCI](#)).

We discuss Cloud Run in more detail later in this course.

Container images are archives

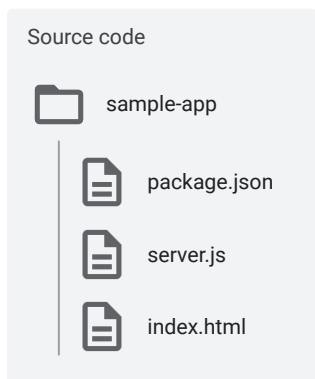
Container images hold applications and everything that they need to run.



As mentioned previously, a container image is an archive with files that includes your application and everything it needs to run.

To discover the different types of files that you might find in a container image, let's look at a very simple example application.

Sample web application



What does this web application need to run?

Here's a minimal Node.js sample web application.

The application is made up of three files: `server.js`, `package.json`, and `index.html`.

To learn what you need to run the application, let's review the content in these files.

server.js file



server.js

```
const express = require('express');  
const app = express();  
app.get('/', (req, res) => {  
  res.sendFile(_dirname + '/index.html')  
});  
app.listen(process.env.PORT || 8080);
```

Library dependency

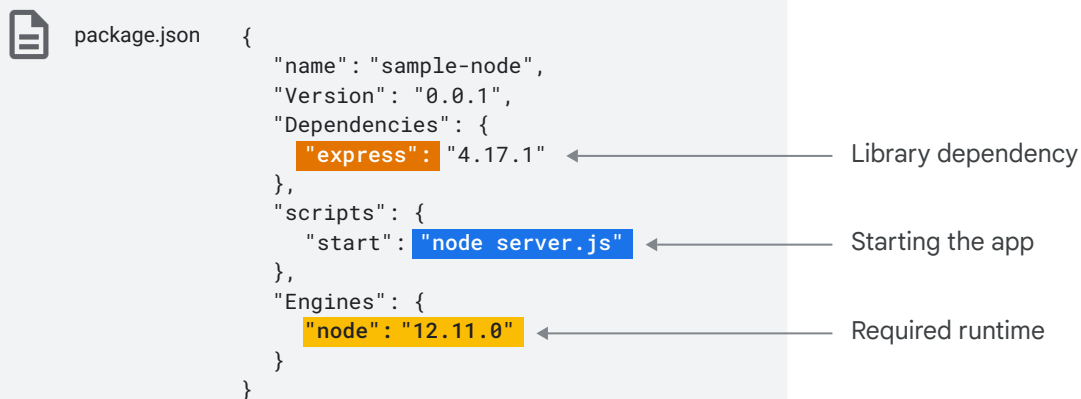
Responding to an
HTTP requestStarting an
HTTP server

The server.js file is the main file of the web application.

It refers to a library or module dependency (express), which is a web application framework for Node.js.

The express module is used to create an endpoint that returns the contents of the index.html file. The app starts to listen for requests on port 8080.

package.json file



```
package.json {
  "name": "sample-node",
  "Version": "0.0.1",
  "Dependencies": {
    "express": "4.17.1"
  },
  "scripts": {
    "start": "node server.js"
  },
  "Engines": {
    "node": "12.11.0"
  }
}
```

Library dependency

Starting the app

Required runtime

Google Cloud

The package.json file is read by the tool **npm** (the Node.js package manager) to install dependencies.

In addition to the application name and version, the package.json specifies the:

- *Library dependency*: which is the express module, with the exact version number, so that npm knows which version to download and install.
- *Start command*: that instructs how to run the application with the node executable.
- *Required Node.js runtime*: that specifies the version of Node.js to use.

index.html file

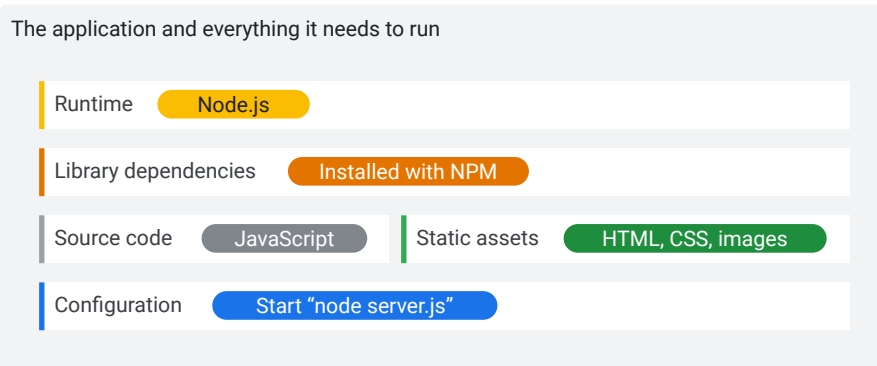


index.html

```
<html>
<head>
  <title>HTML Test</title>
</head>
<body>
  <p><strong>Hello World!</strong></p>
</body>
</html>
```

The index.html is a static HTML file that is returned by the web application on an HTTP request.

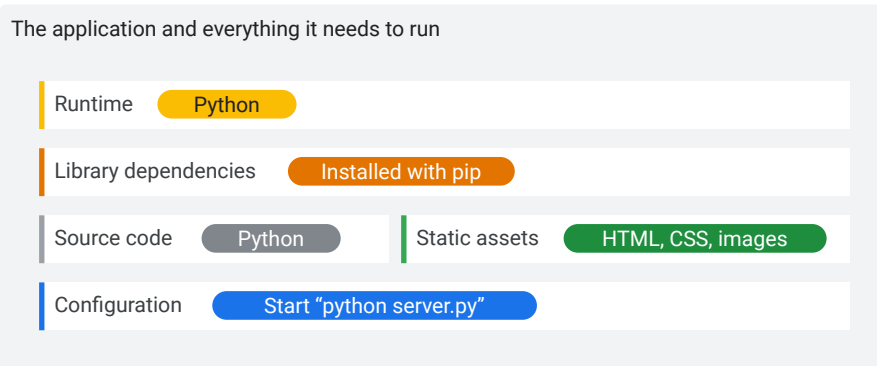
Node.js web application



To summarize, this sample application requires these five components to run the application:

- The runtime
- Dependencies, which you need to install before you can run the app
- The source code (javascript files)
- The index.html file, or static assets in general (in reality, you can have images, CSS files, and more HTML files)
- Configuration, which includes a way to start the application (in this example, it's "node server.js", but it can also be more complicated)

Python web application



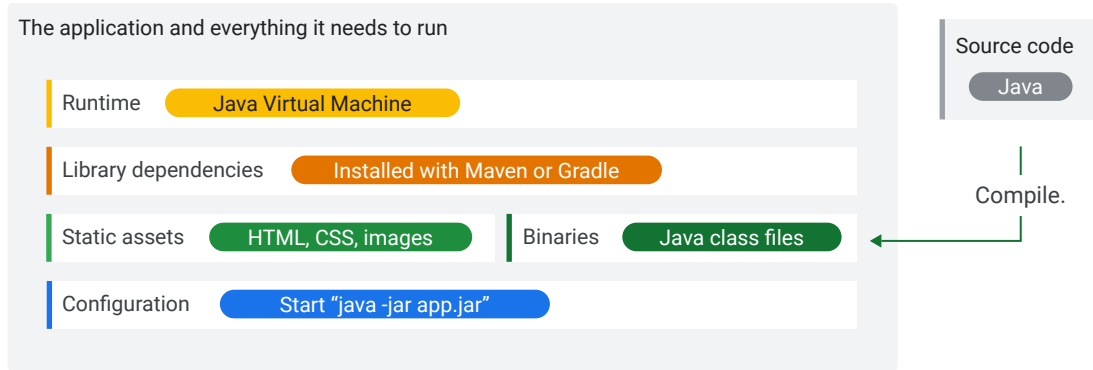
Most web applications written in Python have similar requirements.

Python is an interpreted language, so you need a *runtime* (Python) to run it.

In Python applications, you specify dependencies with a `requirements.txt` file, and you usually install dependencies by using the package manager **pip**.

Your source code will be Python files, and the command to start your application is `"python server.py"` or something similar.

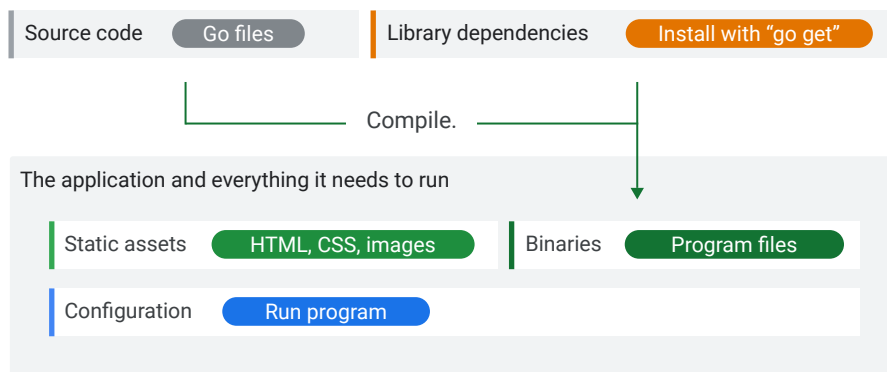
Java application



An application written in Java needs to be compiled first.

The source code is no longer required to run the application, but the compiled binaries are. You'll need to compile the sources first.

Go application



When building an application in Go, you install dependencies and compile them together with the source code to a binary.

You can also choose to embed the assets into the binary.

Applications with system dependencies

Examples:

- A headless browser to turn html into a pdf
- Tools (curl, tar, zip)
- Additional system fonts
- ImageMagick to process images
- OpenOffice to convert document formats



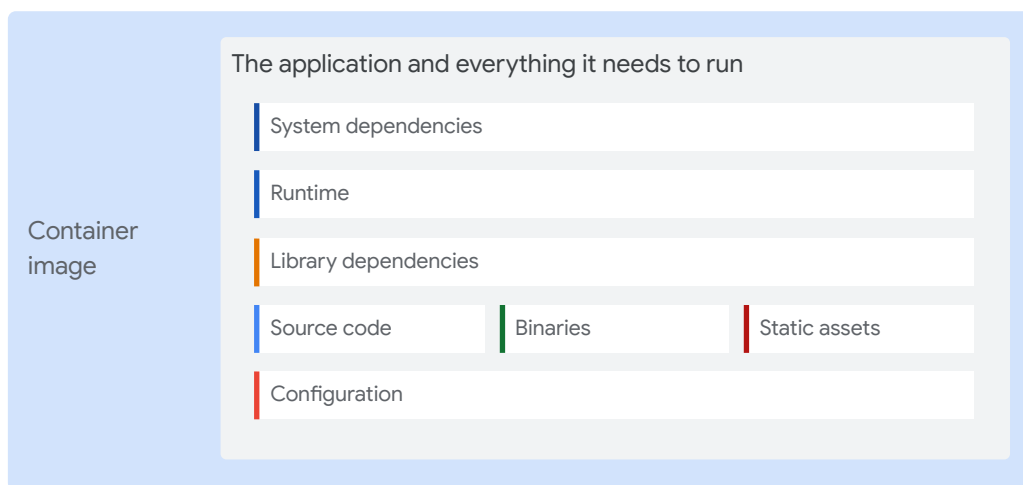
Some applications have dependencies on system tools that can't be expressed as an application library dependency.

Some application examples include:

- A headless browser
- Tools that you might want to use to download or process files
- Additional system fonts
- ImageMagick (a suite of open source programs that can be used to convert images from one format to another, and process them)

In development, you might also need additional tools to aid in debugging.

Container image



Google Cloud

To summarize, the types of files you might need to run your application and be included in a container image are:

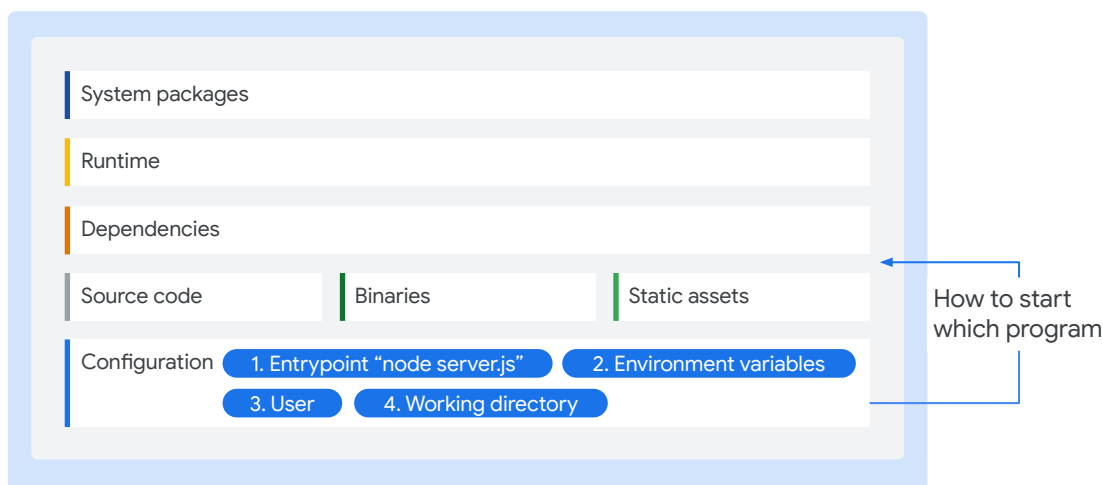
- System packages
- A runtime
- Library dependencies
- Source code
- Binaries
- Static assets
- Configuration

Your application might not need everything on this list. Sometimes you might just have a single binary.

The last item is the container configuration. The container configuration details how to turn a container image into a container—a running process.

In the previous examples, the command needed to run the application was specified as “configuration,” which is usually an important component. But there’s more!

Container configuration



The command to be run when a container is started is called the entrypoint. Some other important settings are:

- Environment variables, which are used to pass configuration settings to your application.
- A working directory
- The user to run the program with.

It's important to set the user. If not set, the root user (or system administrator) is used as the default, which is not a best practice for security reasons.

When you start the container, you can override the values of application arguments and environment variables.

Remember

- 1 A container image contains your application and the dependencies that your application needs to run.
- 2 A **minimal container image** has only one program file and a command to run it.
- 3 Some programming languages need a **runtime** (Java, Python, Node.js).
- 4 Your application might need additional **system dependencies** to work.



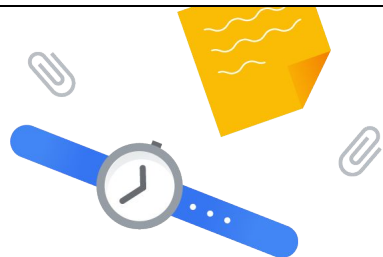
Here's what's important to remember about container images.

A container image is a package which can include an entire runtime, source code, binaries, library dependencies, assets, and container configuration that specifies what program to start and how.

A minimal container image has only one program file and a command to run it.

Your application might need additional system dependencies to work that are also included in the image.

The container image is a self-contained package with your application and everything that it needs to run.



01 Containers and container images

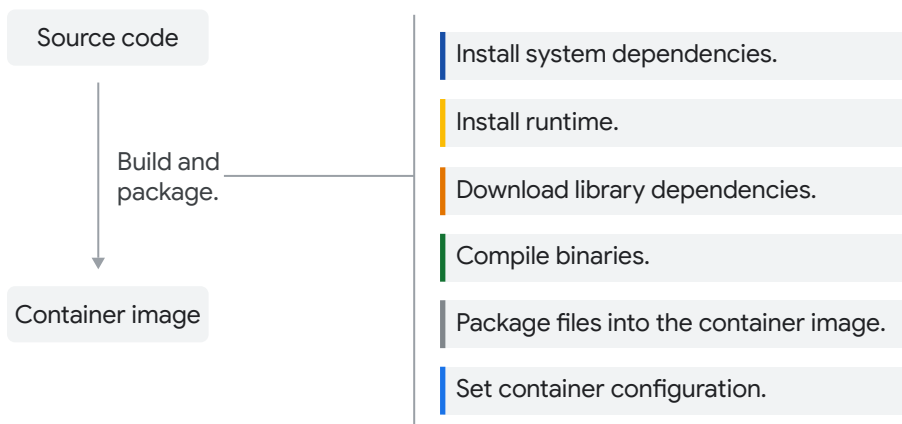
02 Building container images with Docker

Agenda



Now that you have an understanding of what a container image is, let's discuss how to build and package an application into a container image.

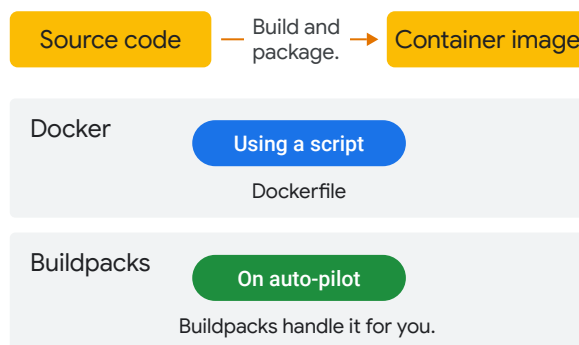
Build and package your application



To build and package your application into a container image, perform these steps:

- Install any system dependencies (if your application depends on them).
- Install a runtime (for example Node.js or Python).
- Download your application's dependencies (npm install, go get, pip install, or invoking your package manager of choice).
- Compile the binaries (or process / bundle the source code).
- Package the files into the image.
- Set the container configuration.

Docker and buildpacks



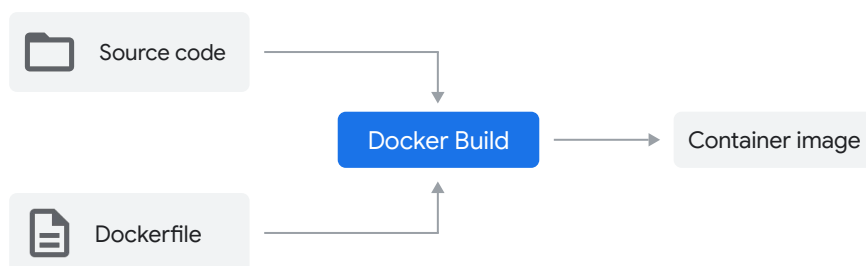
Docker is an open platform that enables you to package and run applications in containers. It provides the tools to manage the lifecycle of your containers, from development and packaging to deployment.

Docker lets you express the application build process using a script, called a Dockerfile. Dockerfiles provide a low-level approach that offers flexibility at the cost of complexity.

The Dockerfile is a manifest that details how to turn your source code into a container image.

Buildpacks are another approach for building container images. They are different from Docker, and provide a convenient approach to building container images by using heuristics to build and package the source code. We also discuss how to use buildpacks in this module.

Docker Build

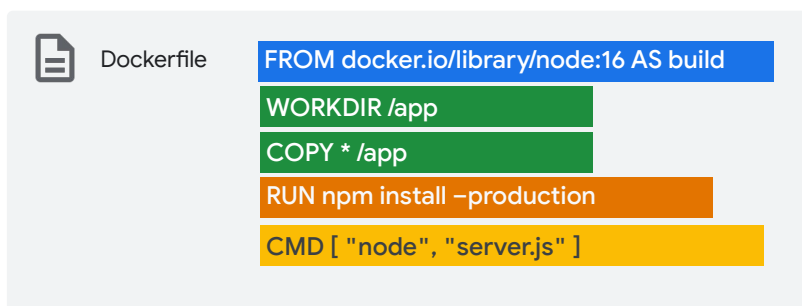


Let's first dive into Docker. Docker is a container engine: you can use it to run containers on your local machine. You can also use it to build container images.

Docker Build is a set of features and tools in Docker that allow you to build and package your applications into container images.

Docker Build takes your source code and a Dockerfile. You express the building and packaging of your source code by using a set of instructions in the Dockerfile.

Sample Dockerfile



Here's an example of a Dockerfile that builds a sample Node.js application into a container image.

The instructions in the Dockerfile:

- Starts from a Node.js base image.
- Creates the application directory in the container file system.
- Copies the source code and other files to the container image.
- Installs the application dependencies excluding any devDependencies listed in the package.json file.
- Sets configuration to run the application when it starts. (node server.js)

Dockerfile instructions



To understand how this works, it's important to realize that with Docker you build your application *inside* the container image.

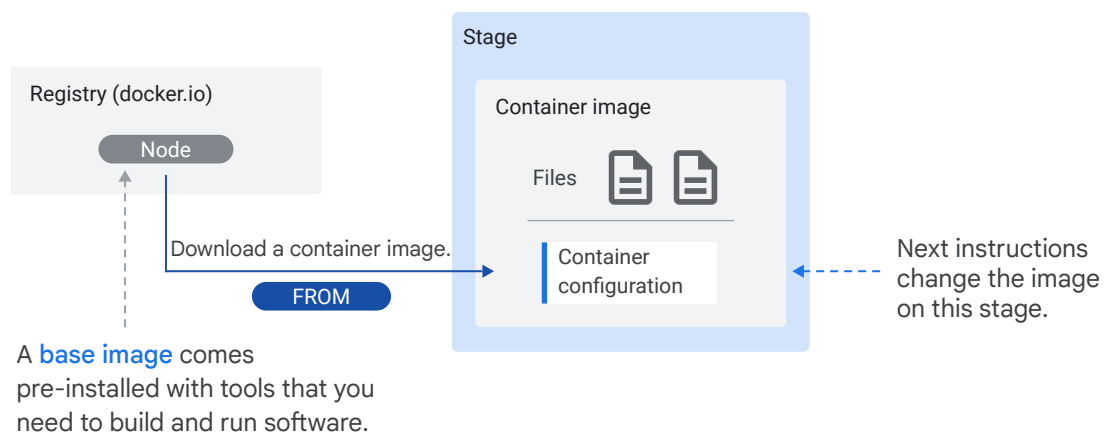
You start with putting a container image on a stage, and every Dockerfile instruction changes that staged container image.

The general process is:

- Start with a base image, which contains tooling to build your application.
- Pull your source code and other required files into the container image.
- To build your application, run a program to update files in the image.
- Configure the image to start your application.

Dockerfiles combine the building and packaging of a container image into a single process.

The FROM instruction

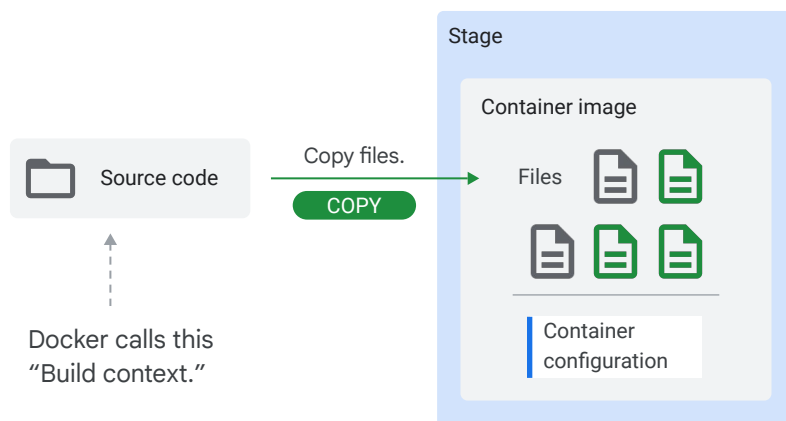


The `FROM` instruction downloads a base image from a registry and puts that on the stage, so that it can be modified by subsequent instructions.

Examples of base images are:

- `golang` (it has tools to build *go* programs)
- `nodejs` (it has tools to install and build node programs)

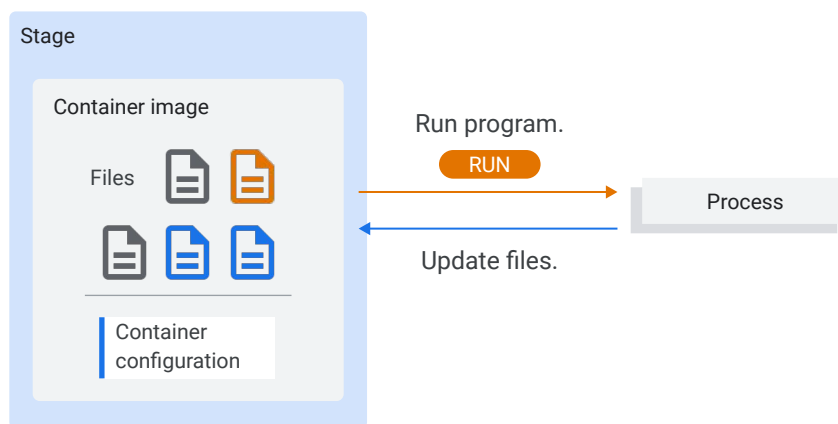
The COPY instruction



The COPY instruction pulls in source code. Docker has the concept of a “build context,” which is the set of files in the source code directory.

Use it to bring source code into the staged image that you’ve just downloaded with the FROM instruction.

The RUN instruction



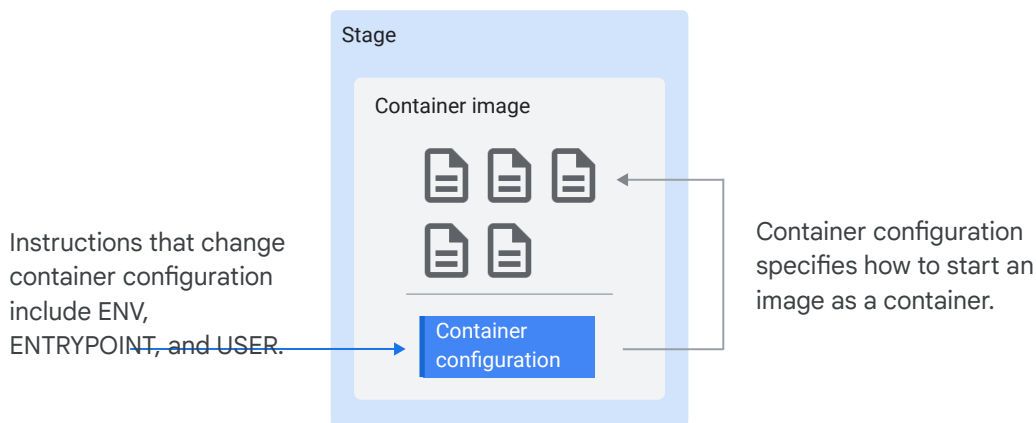
The RUN instruction lets you run a program **from** the image, **on** the image. This means:

- The program file that you execute needs to be present in the container image.
- The only files accessible to the program are those that exist on the container image.

Examples of tasks that RUN is used for, include:

- Installing another system package that you need to build your application.
- Downloading library dependencies.
- Compiling your source code into binaries.

Other instructions



Finally, *container configuration* tells the container runtime (such as Docker, or Cloud Run), what program file to start from the container image, and with what parameters.

There are several instructions that can change the container configuration. Examples include:

ENTRYPOINT: points to the program file to start and run the container as an executable.

CMD: provides defaults for an executing container, which includes the command to run when the container is started. If the executable command is not specified, then the ENTRYPOINT instruction is required.

ENV: is used to set environment variables.

WORKDIR: sets the working directory of the program.

USER: sets the user to use when starting the program.

A full reference of all Dockerfile instructions can be found here:

<https://docs.docker.com/engine/reference/builder/>

Remember

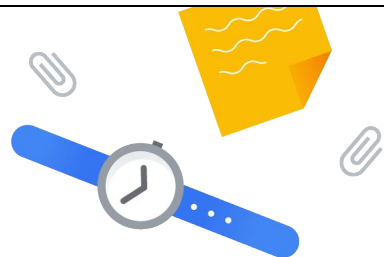
- 1 The **FROM** instruction downloads a base image to start from.
- 2 The **COPY** instruction pulls in files from the build context.
- 3 The **RUN** instruction lets you run a program from the container image to update files.
- 4 Other instructions like **CMD** change the container configuration, like setting the start program.



Here's what's important to remember about Dockerfile instructions:

You start with putting a container image on a stage, and every Dockerfile instruction changes that staged container image.

- The **FROM** instruction downloads a base image to start from. A base image can be “golang” for example, and it includes tools you need to build and run your software.
- The **COPY** instruction pulls in files from the build context, which is usually the directory that contains the Dockerfile.
- The **RUN** instruction lets you run a program **from** the container image to change files **in the** image.
- Other instructions like **CMD** change the container configuration, which points out which program file to start and how.



- 01 Containers and container images
- 02 Building container images with Docker
- 03 Lab: Creating and running Docker containers

Agenda



Let's now complete a lab on creating and running Docker containers.

Lab



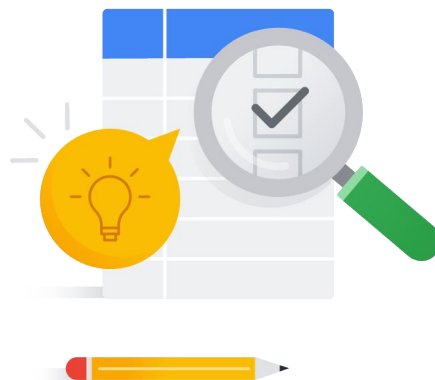
45 min



Individual

Creating and running Docker containers

Create Docker containers and run them locally in Cloud Shell. Publish a container image to Artifact Registry.



In this lab, you create Docker containers and run them locally in Cloud Shell. You also publish a container image to Artifact Registry.

Lab instructions



45 min



Individual



Tasks

- Create a Docker container image by using a Dockerfile.
- Create another version of the containerized application and test both versions.
- Troubleshoot containerized applications.
- Push container images to Artifact Registry.

1

Build a container image.

2

Modify the container image.

3

Troubleshoot containers.

4

Publish a container image.



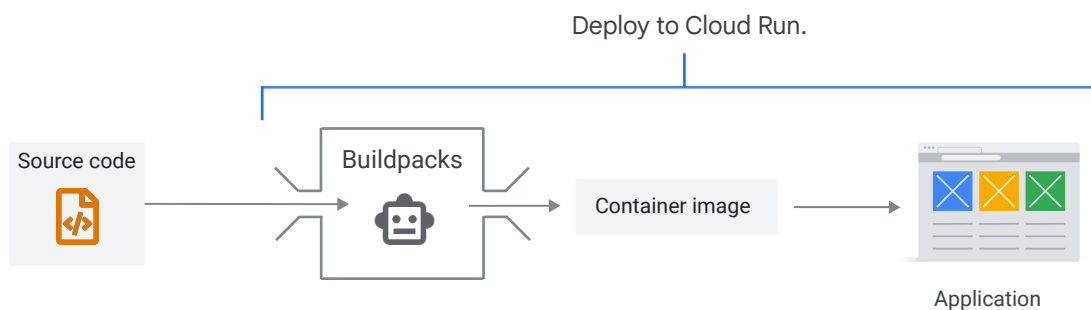
Agenda



- 01 Containers and container images
- 02 Building container images with Docker
- 03 Lab: Creating and running Docker containers
- 04** Building container images with buildpacks

Let's discuss how you can create container images with buildpacks.

Buildpacks



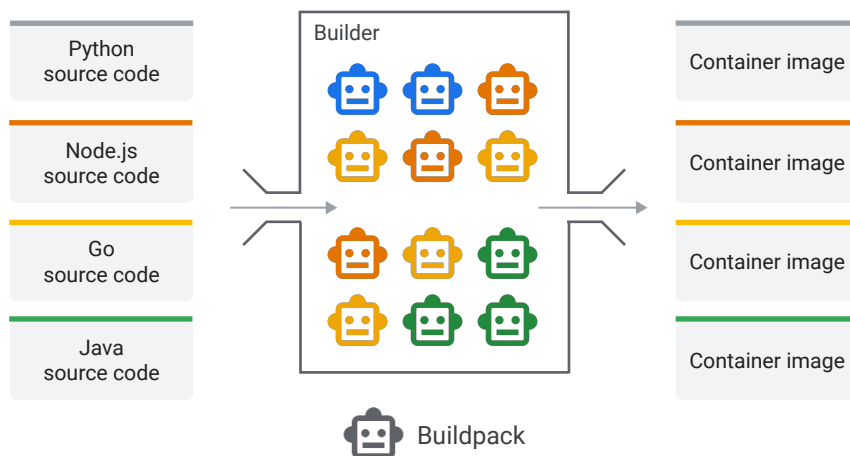
Buildpacks are a way to turn source code into a container image without writing a Dockerfile.

Buildpacks provide developers with a convenient way to work with container images, without thinking about the complexities that come with building them.

You can create your own buildpacks, or use those provided by multiple vendors.

Buildpacks are built into Cloud Run to enable a source-based deployment workflow.

Builders



Buildpacks are distributed and executed in OCI images called builders. Each builder can have one or more buildpacks.

OCI stands for the **Open Container Initiative**, a Linux Foundation project that was started in 2015 to design open standards for operating-system-level virtualization of Linux containers.

A builder turns your source code into a container image. The buildpacks do the actual work to build and package the container image.

Builders can support source code written in multiple languages. In this example, the builder can build and package a Python, Node.js, Go, and a Java application into a container image.

If a builder starts to process a source directory, it executes two phases of a buildpack:

1. The **detect** phase:

Runs against your source code to determine if a buildpack is applicable or not. When a buildpack is detected to be applicable, the builder proceeds to the build phase. If detection fails, the build phase for a specific buildpack is skipped.

For example, to pass the detect phase:

- a. A Python buildpack might look for a `requirements.txt` or a `setup.py` file.

- a. A Node buildpack might look for a `package-lock.json` file.

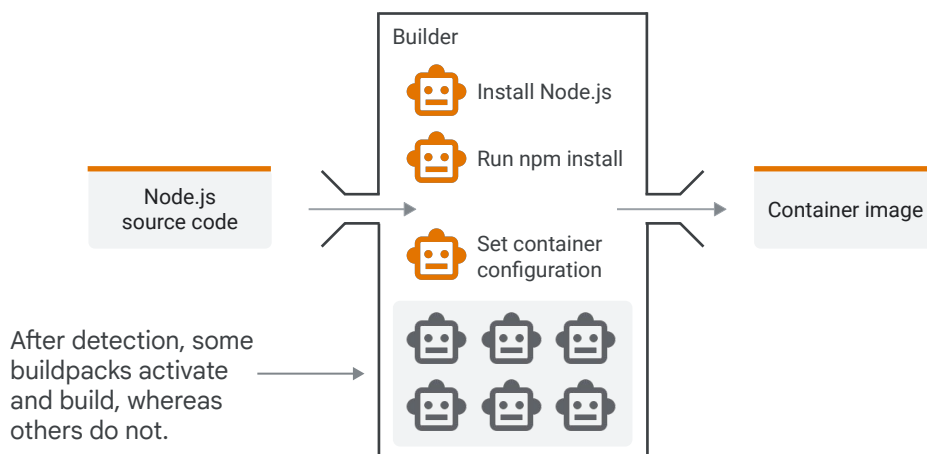
2. The **build** phase:

Runs against your source code to set up the build-time and run-time environment, download dependencies and compile your source code (if needed), and set appropriate entry point and startup scripts.

For example:

- a. A Python buildpack might run `pip install -r requirements.txt` if it detected a `requirements.txt` file.
- b. A Node buildpack might run `npm install` if it detected a `package-lock.json` file.

Builders



Let's look at an example.

After the builder runs the detect phase, suitable buildpacks activate and perform a build, while others do not.

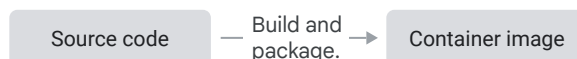
In this example, three buildpacks activate to build a directory with a Node.js project.

1. One buildpack installs the Node.js runtime.
2. Another buildpack runs npm install.
3. The final buildpack configures the resulting image to start the Node.js runtime.

The result is a container image that you can deploy to Cloud Run or run with Docker locally.

Pack

Pack is a command line tool. It needs a **builder** to turn a **source directory** into a **container image**.



Shell

```
$: pack build  
--builder gcr.io/buildpacks/builder:v1  
--path ./source-dir  
sample-app
```

As a developer, it's easy to use buildpacks.

With the command line tool “pack”, you can use a builder to turn source code into a container image.

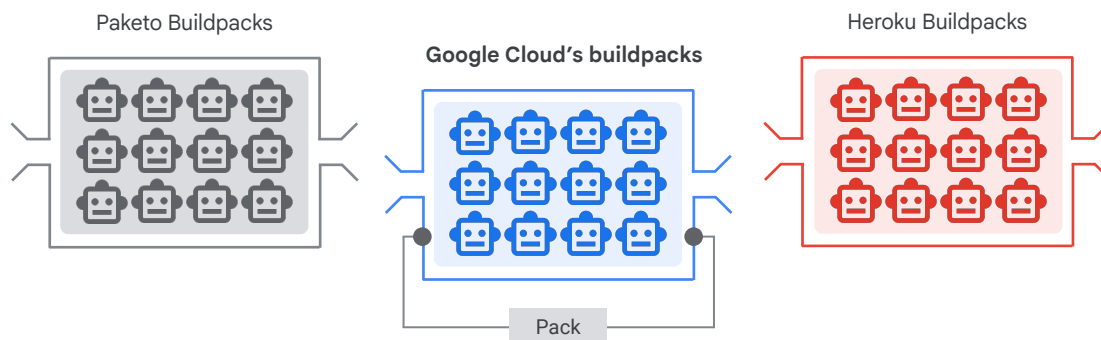
Pack is a tool that is maintained by the Cloud Native Buildpacks project to support the use of buildpacks.

The example shows how to use *pack* to build a source directory using the Google Cloud's buildpacks builder that's built into Cloud Run.

By running this command on your local machine, you can reproduce the container image in a similar manner as Cloud Run does in Google Cloud.

For more information on using Google Cloud's buildpacks, view the [documentation](#).

Choice of builders



The command-line tool pack can work with all of them.

There are many projects that use the buildpacks standard to create their own builders. You can choose to use a builder from any of these projects.

Some examples are:

Paketo Buildpacks: This is a Cloud Foundry Foundation project, which is dedicated to maintaining vendor-neutral governance.

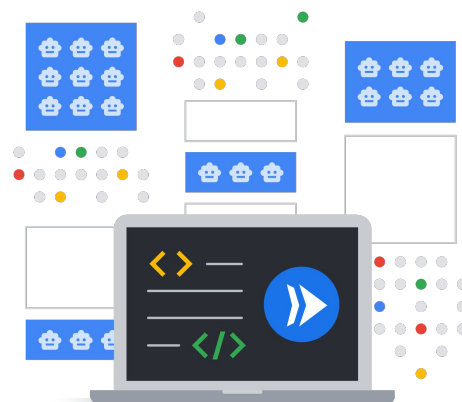
Heroku Buildpacks: Heroku is a well known platform as a service product that makes it easy to build cloud native applications.

Regardless of what buildpack you use to build your container image, Cloud Run can run it.

Google Cloud's buildpacks are built into Cloud Run.

Google Cloud's buildpacks

- The builder is used internally by App Engine, Cloud Functions, and Cloud Run.
- The builder supports Go, Java, Node.js, Python, and .NET Core.
- The project is open source.
- Source code deployed to Cloud Run is built with Google Cloud's buildpacks.



Google Cloud's buildpacks are used internally by App Engine, Cloud Functions, and Cloud Run.

The Google Cloud's buildpacks builder supports applications written in Go, Java, Node.js, Python, and .NET Core.

You can deploy source code, and container images to Cloud Run. Cloud Run builds source code with Google Cloud's buildpacks.

Google Cloud's buildpacks are optimized for security, speed, and reusability.

github.com/GoogleCloudPlatform/buildpacks

Remember

- 1 Buildpacks are a way to create a container image from source code without writing a Dockerfile.
- 2 Buildpacks are distributed and executed in OCI images called builders. Each builder can have one or more buildpacks.
- 3 Builders can support source code written in multiple languages.
- 4 Use the command line tool *pack* with a builder to turn source code into a container image.



Here's what's important to remember about building container images with buildpacks:

- Buildpacks are a way to turn source code into a container image without writing a Dockerfile.
- Buildpacks are distributed and executed in OCI images called builders. Each builder can have one or more buildpacks.
- Builders can support source code written in multiple languages.
- With the command line tool “pack”, you can use a builder to turn source code into a container image.



Agenda



- 01 Containers and container images
- 02 Building container images with Docker
- 03 Lab: Creating and running Docker containers
- 04 Building container images with buildpacks
- 05** Continuous integration and delivery tools

In this lesson, we review some of the tools to implement continuous integration and delivery (CI/CD) for your container images.

Scaffold



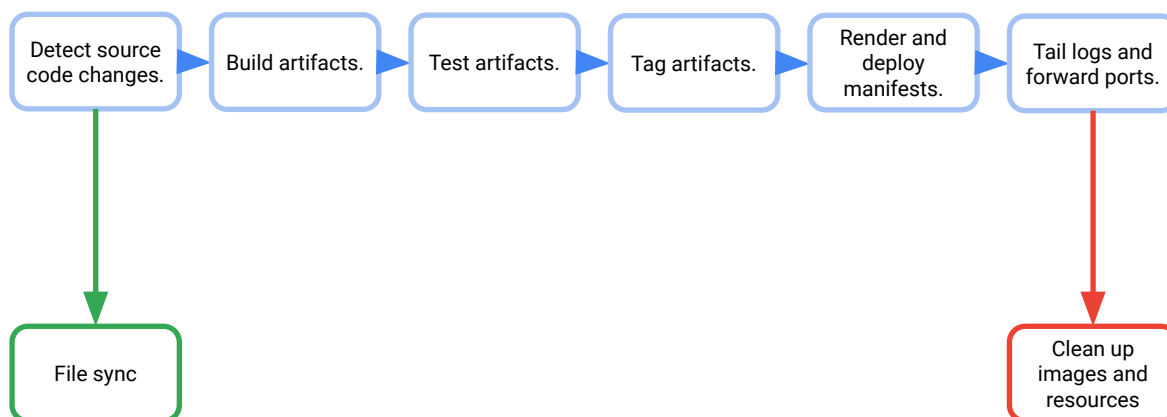
- It is a tool to orchestrate continuous development, continuous integration (CI), and continuous delivery (CD) of your application code.
- It is a Google open source project.
- It provides building blocks for creating CI/CD pipelines.
- It uses portable, declarative configuration.

Scaffold is a command-line tool that orchestrates continuous development, continuous integration (CI), and continuous delivery (CD) of container-based and Kubernetes applications.

It's a Google open source project that provides declarative, portable configuration with a pluggable architecture.

Scaffold handles the workflow for building and deploying your application, and provides building blocks for creating CI/CD pipelines. Scaffold can be used to continuously deploy containers to your local or remote Kubernetes cluster, Docker environment, or Cloud Run project.

Skaffold workflow



Skaffold uses a multi-stage workflow.

When you start Skaffold, it collects source code changes in your project and builds artifacts with the tool of your choice. Skaffold supports various tools like Dockerfiles, Cloud Native Buildpacks and others for building your containers locally, and remotely with Google Cloud Build. For files that don't need to be built, Skaffold supports syncing or copying changed files to a deployed container.

When the artifacts are successfully built, they are tested, tagged, and pushed to the repository you specify. Skaffold lets you skip stages. For example, if you run Kubernetes locally with [Minikube](#), Skaffold will not push artifacts to a remote repository.

In the test phase, you can run custom commands for unit and integration tests, validation, security scans, and structure tests on the container image. You can also tag your built image with different tag policies like `gitCommit`, `sha256`, and others that are supported by Skaffold.

Skaffold also helps you deploy the artifacts to your Kubernetes cluster by using the tools you prefer.

To deploy your containers to Kubernetes, Skaffold supports the rendering of manifests using raw YAML, or rendering tools *helm*, *kpt*, and *kustomize*. In this phase, Skaffold replaces untagged image names in the manifests with the final tagged image names, and might expand templates for *helm*, or calculate overlays for *kustomize*.

docker to deploy to a Docker runtime. Skaffold also supports deployment to Cloud Run with a valid Cloud Run yaml manifest.

Skaffold has built-in support for tailing logs for containers built and deployed by Skaffold when running in either dev, debug, or run mode. There is also support for forwarding ports from exposed Kubernetes resources on your cluster to your local machine.

Skaffold also has cleanup functionality to delete Kubernetes resources and Docker images.

<https://skaffold.dev/docs/pipeline-stages/>

Skaffold configuration - skaffold.yaml

```
apiVersion: skaffold/v1beta13
kind: Config
build:
  artifacts:
    - image: skaffold-app
      context: app
      docker:
        dockerfile: Dockerfile
deploy:
  kustomize:
    paths:
      - overlays/dev
```

← The build section defines how to build the image.

← The deploy section defines how to deploy the image.

Google Cloud

Skaffold requires a YAML configuration file (skaffold.yaml) that defines how your project is built and deployed.

By running the *skaffold init* command, you can get started with Skaffold using a wizard that generates the required skaffold.yaml file in the root of your project directory.

The skaffold.yaml file defines your build and deploy configuration for your container and resources.

Here's a partial sample skaffold.yaml file. Notice the Kubernetes-like YAML format specification.

The build section contains configuration that defines how the container image should be built. It has an artifacts section that defines the container images that comprise the application, and specifies the Dockerfile to be used to build the container image.

The deploy section contains configuration that defines how the container should be deployed. In this sample, the default deployment uses the Kustomize tool. Kustomize is a tool that is used to generate Kubernetes manifests by combining a set of common YAML files in a base directory with one of more overlays. An overlay typically corresponds to a deployment environment, for example, dev, staging, or production.

Skaffold also supports a *profiles* section in the configuration file (not shown), that

defines build, test, and deployment configurations for different contexts or environments in your deployment pipeline, like staging or production. This allows you to easily manage different manifests for each environment, without repeating common configuration.

For more information on Skaffold and its commands, please refer to the documentation on the [Skaffold website](#).

Artifact Registry



Artifact Registry

- A Google Cloud service used to store and manage software artifacts like container images.
- Integrates with Cloud Build.

Artifact Registry is a service that is used to store and manage software artifacts in private repositories, including container images, and software packages.

It's the recommended container registry for Google Cloud.

Artifact Registry integrates with Cloud Build to store the packages and container images from your builds.

Cloud Build

- Is a service that executes builds on Google Cloud.
- It imports source code from various repositories or cloud storage spaces.
- It produces artifacts such as Docker containers or Java archives.
- It works with a build configuration to:
 - Fetch dependencies.
 - Run unit and integration tests.
 - Perform static analyses.
 - Create artifacts with build tools.

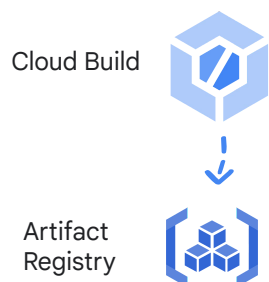


Cloud Build is a service that executes your builds on Google Cloud. With Cloud Build, you can continuously build, test, and deploy your application by using a CI/CD pipeline.

Cloud Build can import source code from various repositories or cloud storage spaces, execute a build to your specifications, and produce artifacts such as Docker containers or Java archives.

To provide instructions to Cloud Build, you create a build configuration file that contains a set of tasks. These instructions can configure builds to fetch dependencies, run unit and integration tests, perform static analyses, and create artifacts with build tools (builders) like docker, gradle, maven, and others.

Build process



Cloud Build:

- Automatically builds your code into a container image.
- Pushes the image to Artifact Registry.

Build Process:

- Executes in your project.
- The Cloud Build API must be enabled.
- Provides access to all build logs through Cloud Logging.

The process of building a container image is entirely automatic and requires no direct input from you.

The Cloud Build API must be enabled for your project.

All the resources used in the build process execute in your own user project, and you have access to all the build logs through Cloud Logging.

Features of Cloud Build

- Write code in any programming language.
- Use a build configuration file.
- Integrate with different source code repositories.
- Automatically build, test, and deploy your code.
- Store build artifacts in different registries and storage systems.
- Deploy to popular platforms.

With Cloud Build, you can build your applications written in Java, Go, Python, Node.js, or any programming language of your choice.

Cloud Build integrates with different source code repositories such as GitHub, Bitbucket, and GitLab. You can store your application source code in any of these repositories and use Cloud Build to automate building, testing, and deploying your code.

You can use Artifact Registry with Cloud Build to store build artifacts. You can also store artifacts in other storage systems such as Cloud Storage.

Cloud Build supports the deployment of your application code to popular deployment platforms such as Cloud Run, Google Kubernetes Engine, Cloud Functions, Anthos, and Firebase.

Cloud Build steps

```
steps:
- name: 'gcr.io/cloud-builders/docker'
  args: ['build',
        'us-central1-docker.pkg.dev/${PROJECT_ID}/my-repo/my-image', '.']
- name: 'gcr.io/cloud-builders/docker'
  args: ['push',
        'us-central1-docker.pkg.dev/${PROJECT_ID}/my-repo/my-image']
- name: 'gcr.io/google.com/cloudsdktool/cloud-sdk'
  entrypoint: 'gcloud'
  args: ['compute', 'instances', 'create-with-container', 'my-vm-name',
        '--container-image',
        'us-central1-docker.pkg.dev/${PROJECT_ID}/my-repo/my-image']
  env:
  - 'CLOUDSDK_COMPUTE_REGION=us-central1'
  - 'CLOUDSDK_COMPUTE_ZONE=us-central1-a'
```

Google Cloud

Here's a basic build configuration file. The build configuration file is named `cloudbuild.yaml` and can be written in YAML or JSON format.

Instructions are written as a set of steps. Each step must contain a *name* field that specifies a cloud builder, which is a container image that runs common tools. In this sample, we have a build step with a docker builder, which is an image running Docker.

The *args* field of a step takes a list of arguments and passes them to the builder. The values in the *args* list are used to access the builder's entrypoint. If the builder does not have an entrypoint, the first element in the *args* list is used as the entrypoint.

In the example shown:

1. The first build step builds a container image using the docker builder from source code located in the current directory.
2. The second build step uses the docker push command to push the image that was built in the previous step to Artifact Registry.
3. The third step uses the Cloud SDK with the `gcloud` entrypoint to create a Compute Engine instance from the container image.

There are additional fields that you can use to configure the build. For a complete list of fields, view the [Cloud Build documentation](#).

Running builds with Cloud Build

`gcloud builds submit` command:

1. Uploads your application code, and other files in the current directory to Cloud Storage.
2. Initiates a build in the specified region.
3. Tags the image with the specified name.
4. Pushes the built image to Artifact Registry.

Build with Dockerfile

```
gcloud builds submit \  
--region=us-central1 --tag \  
$REPO/my-image .
```

Build with configuration file

```
gcloud builds submit \  
--region=us-central1 \  
--config=cloudbuild.yaml .
```

Google Cloud

With Cloud Build, you can run builds manually or use build triggers.

To start a build manually, you use the Google Cloud `gcloud` CLI or Cloud Build API.

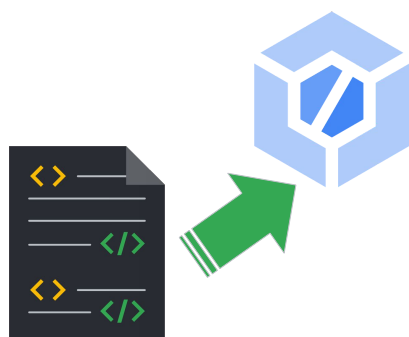
You can use a Dockerfile or a Cloud Build configuration file with the *gcloud builds submit* command.

The command first uploads your application source code and other files from the specified directory to Cloud Storage. It then builds the container image containing your application from the instructions specified in the Dockerfile or build configuration file, tags the image with the specified image name and pushes the image to the appropriate registry.

You can also use this command with Cloud Native Buildpacks to build a container image without a Dockerfile or build configuration file.

To use the Cloud Build API, you create a JSON request file containing the source code location, build steps, and other information. Then, submit a POST request to the `/builds` endpoint by using Curl or other HTTP client and with the appropriate authentication credentials.

Cloud Build triggers



- Automatically run builds with Cloud Build triggers.
- Changes can be made to source code in:
 - Cloud Source Repositories
 - GitHub
 - Bitbucket
- A Dockerfile or Cloud Build configuration file is required.

You can run builds automatically with Cloud Build triggers.

A Cloud Build **trigger** automatically starts a build whenever you make any changes to your source code in a Google Cloud Source repository, GitHub, or Bitbucket repository. Build instructions must be supplied in a Dockerfile or Cloud Build configuration file.

You must first connect Cloud Build to your source repository before building the code in that repository. Repositories in Cloud Source Repositories are connected to Cloud Build by default. To connect an external repository hosted in GitHub or Bitbucket, follow the steps that are outlined in the [documentation](#).

Creating a Cloud Build trigger

Create a trigger for GitHub

```
gcloud beta builds triggers
create github \
--name=my-trigger \
--region=us-central1 \
--repo-name=repos \
--branch-pattern=".*" \
--build-config=cloudbuild.yaml \
--service-account=sa_email
```

To create a Cloud Build trigger, provide:

- The trigger name, description, and cloud region
- The trigger event
- Source repository and branch
- Build configuration
- Service account email

Google Cloud

Cloud Build triggers are created in the Google Cloud console or with the gcloud CLI.

To create a build trigger, you provide:

- The name, description, and cloud region for the trigger
- The trigger event in the remote origin of your source code repository, which can be:
 - A commit or push to a particular branch
 - A commit or push that contains a specific tag
 - A commit to a pull request in GitHub
- The source repository that contains your source code
- A regular expression that identifies the source branch or tag
- The location and type of your build configuration, which can be:
 - A Cloud Build configuration file in YAML or JSON format
 - A Dockerfile
 - Buildpacks
- The email associated with your service account, or the default Cloud Build service account

For a complete list of build trigger configuration items, refer to the [documentation](#).

The example shows how you can use the gcloud CLI to create a build trigger to

automatically build your source code that resides in a GitHub repository, when a push event occurs on any branch in the repository.

When using GitHub or Bitbucket repositories, you must first connect Cloud Build to your repository before you build code in that repository. For more information on how to connect Cloud Build to source repositories, refer to the [documentation](#).

Other types of Cloud Build triggers



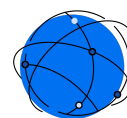
Manual triggers

Invoke builds manually and deploy code to other environments.



Pub/Sub triggers

Execute builds in response to Pub/Sub events.



Webhook triggers

Connect external source code management systems to Cloud Build.

You can also create triggers to invoke builds manually. Manual triggers enable you to create a pipeline to automatically fetch code from a hosted repository with a specific branch or tag, and then manually build and deploy that code to other environments.

Cloud Build Pub/Sub triggers enable you to execute builds in response to Pub/Sub events. For example, you can create a Pub/Sub trigger to automate builds in response to events on Artifact Registry, and Cloud Storage. Use cases for these types of events include pushing, tagging, or deleting images in the registry.

Cloud Build enables you to define webhook triggers which authenticate and process incoming webhook events to a custom URL. With this capability, you can connect external source code management systems (for example GitLab, Bitbucket.com) to Cloud Build and use a build configuration file to automate your build.

View the documentation for more information on using Cloud Build [Pub/Sub triggers](#) and [webhook triggers](#).



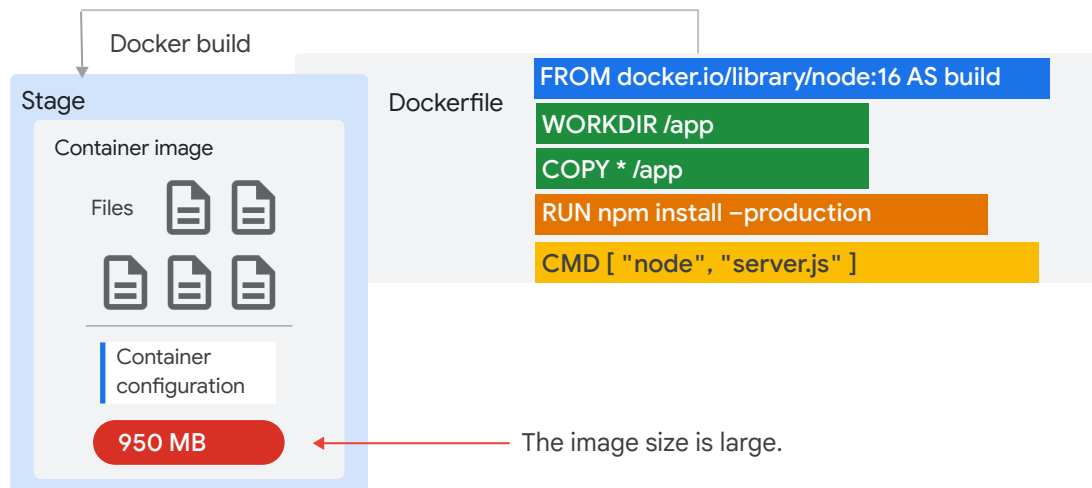
Agenda



- 01 Containers and container images
- 02 Building container images with Docker
- 03 Lab: Creating and running Docker containers
- 04 Building container images with Buildpacks
- 05 Continuous integration and delivery tools
- 06** Best practices

Let's now discuss some of the best practices when building and securing containers.

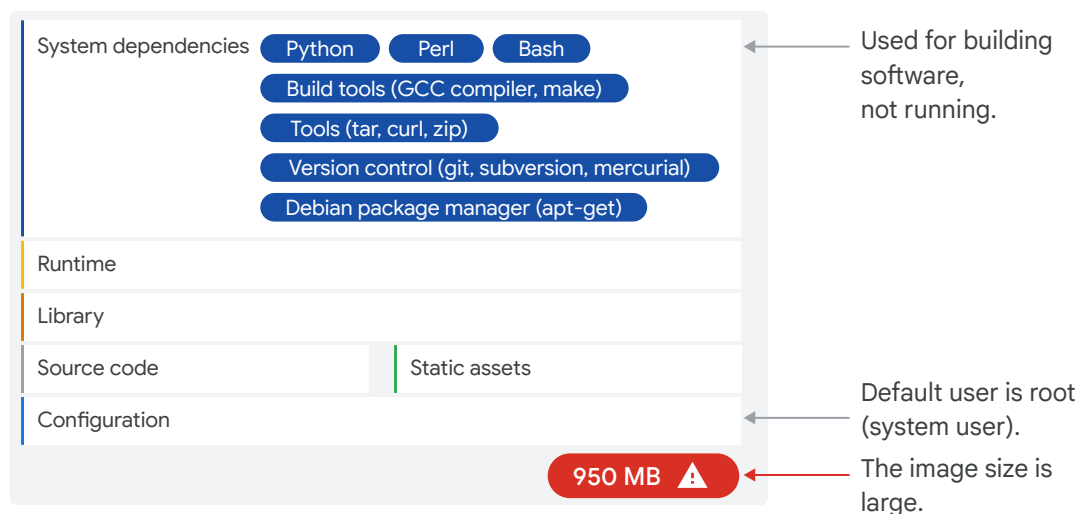
Container image size



In a previous lesson, we containerized a Node.js application with Docker.

The image that was created is almost 1 GB in size. What is the reason for this?

Base image size

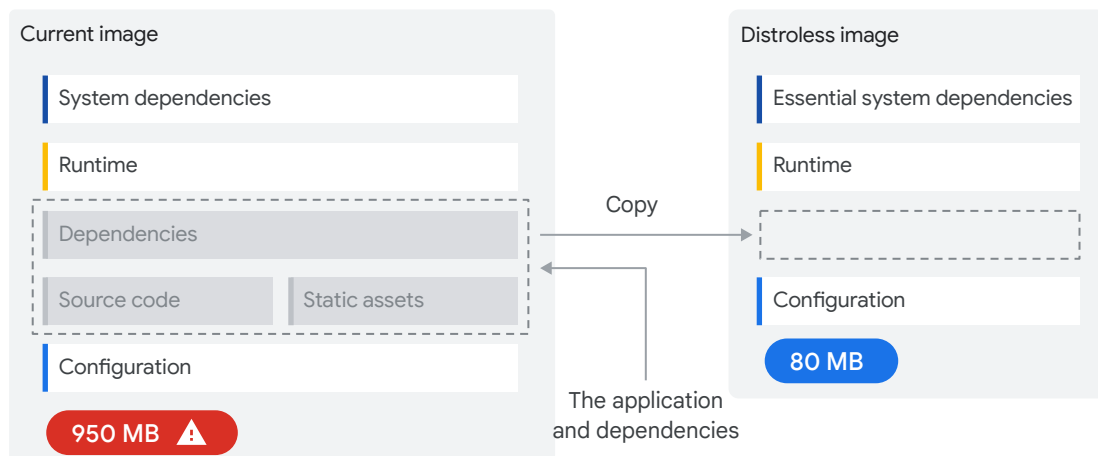


The Node.js runtime is approximately 80 MB, and the app and library dependencies are not more than 1 MB. What's in the other 869 MB?

This is because the image was based on the Docker Hub *node* image, which comes packed with the system packages you need to build software. Even the Debian package manager `apt-get` is included so you can install even more system packages.

This *base image* is more suitable for building than for running software. In production, smaller images are better for security reasons. There might be security vulnerabilities in the additional system packages.

Improve security and image size with distroless

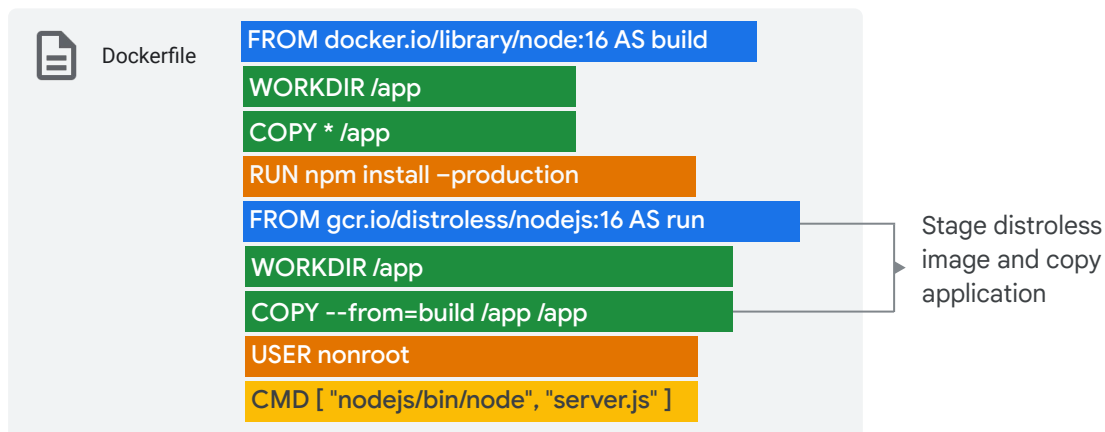


The solution that Docker has for this problem is a multi-stage image build.

It involves downloading a new image after you've completed your initial build, and copying all application files, assets, and library dependencies.

We review this process with an image from the distroless project that is optimized for running applications, because it contains only the application and its runtime (not build time) dependencies.

An improved Dockerfile



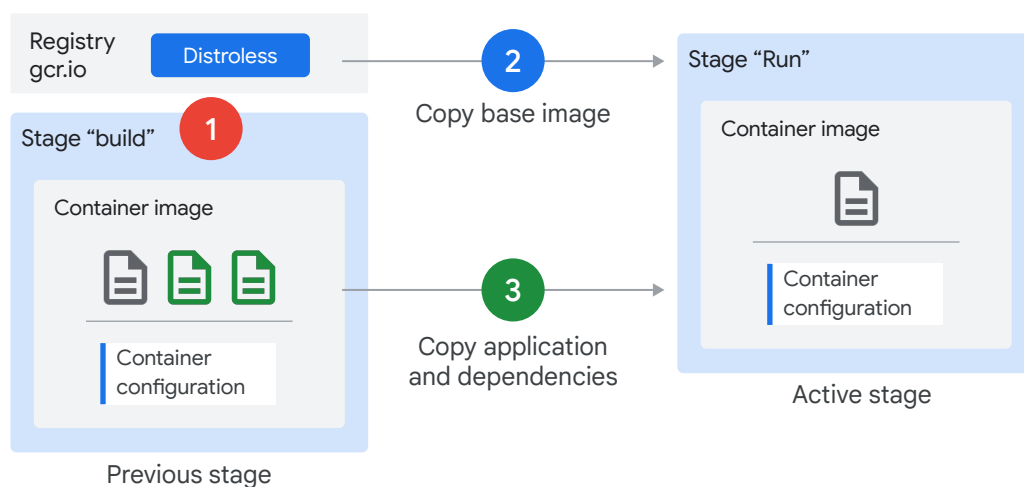
Here's an improved version of the same example that you saw earlier.

After running `npm install`, it has a repeated `FROM` instruction, this time from the `distroless` project.

The `FROM` instruction is followed by a `COPY` instruction, that pulls the application from the previously staged image onto the active stage.

The Dockerfile includes additional container configuration, which sets the *nonroot* user, which is a user without system administrator permissions.

Multi-stage build



Use multi-stage builds to create a minimal container image.

In a multi-stage build:

1. The first stage that is built contains the Node.js image along with the application code using the first set of FROM and COPY instructions.
2. The second FROM instruction stages a new container image from the Distroless project.
3. The second COPY command copies the application and dependencies from the previous stage onto the active stage.

Remember

- 1 Base images are bloated with packages that you don't need for running your application.
- 2 Distroless is a project that provides minimal runtime container images.
- 3 If you repeat the FROM instruction, you create a multi-stage build.
- 4 To finish a build, you copy the application and its dependencies into the final stage.



Here's what's important to remember about building secure and small container images with Dockerfiles.

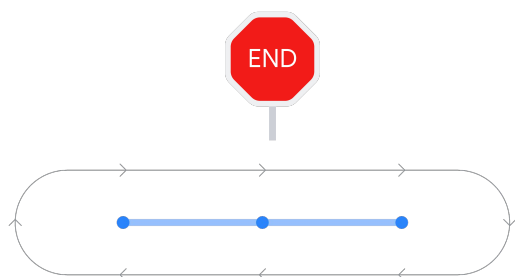
Base images are bloated with packages that you don't need for running your application, but do need to build your application. For example, they contain package managers and version control software. It's a risk to add these binaries to the container image you deploy, because they might contain security vulnerabilities.

Distroless is a project that provides minimal runtime container images. These images can't be used to *build* your application, but do contain only a minimal set of system dependencies to *run* your application.

If you repeat the FROM instruction, you create a multi-stage build.

To finish a build, you copy the application and its dependencies into the final stage.

Process and signal handling



- Launch your container application process with the `CMD` or `ENTRYPOINT` instruction in your Dockerfile.
 - This allows the process to receive signals.
 - It enables graceful shutdown of the app when terminated.
- Register and implement signal handlers in your application code.

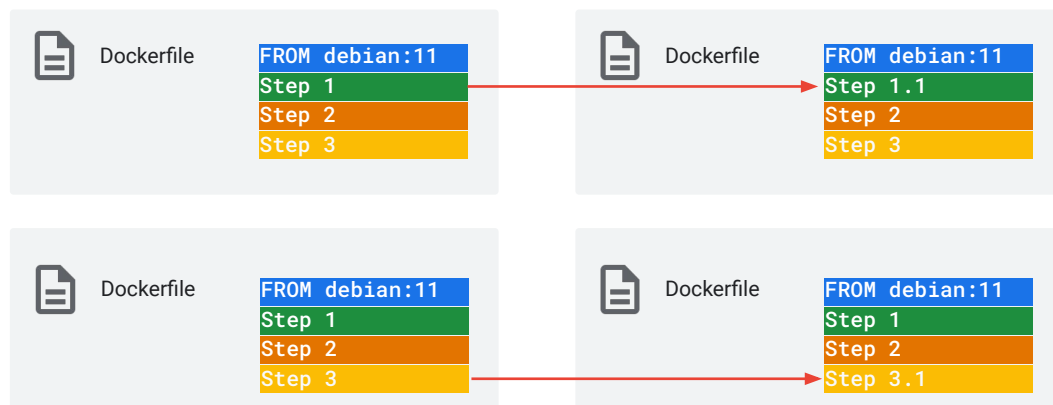
Process identifiers (PIDs) are unique identifiers that the Linux kernel gives to each process. The first process launched in a container gets PID 1.

Container platforms such as Docker, Kubernetes, and Cloud Run use signals to communicate with the processes inside containers, most notably to terminate them.

Because these platforms can only send signals to the process that has PID 1 inside a container, you must launch your process with the `CMD` or `ENTRYPOINT` instruction in your Dockerfile. This allows the process to receive signals and gracefully shut down the app when it's terminated.

Also, because signal handlers aren't automatically registered for the process with PID 1, you must implement and register these signal handlers in your application code.

Docker build cache



When building a container image, Docker steps through the instructions in your Dockerfile executing them in the specified order. Each instruction creates a layer in the resulting image. For each instruction, Docker looks for existing image layers in its cache that can be reused.

Docker can use its build cache for an image only if all previous build steps used it. By positioning build steps that involve frequent changes at the bottom of the Dockerfile, you can enable faster builds by utilizing the Docker build cache.

Because a new Docker image is usually built for each new version of your source code, add the source code to the image as late as possible in the Dockerfile.

In the example, if STEP 1 involves changes, Docker can reuse only the layers from the FROM debian:11 step. If STEP 3 involves changes, Docker can also reuse the layers for STEP 1 *and* STEP 2.

More best practices



Remove unnecessary tools

To reduce the attack surface of your app, remove unnecessary tools and utilities from your container image.



Build the smallest image possible

A smaller image reduces image upload and download times.



Run the app as a non-root user

Avoid running the app as root in the container.



Create images with common layers

Reduce container image build time by using common, standard, base images.

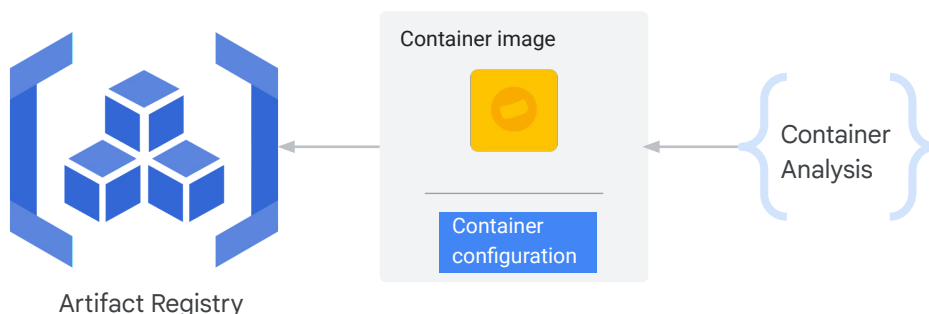
Here are some additional best practices to follow when building container images:

- You should keep as few things as possible in your container image, ideally only your app, and remove any unnecessary tools or utilities. This reduces the attack surface of your app and protects it from potential harm from attackers.
- To prevent attackers from modifying root-owned files in your container image using a package-manager such as *apt-get*, avoid running the app as the *root* user inside the container. You must also disable or uninstall the *sudo* command. Also, consider launching the container in read-only mode.
- Build a smaller image to reduce image upload and download times. If the image is smaller, a node can download it faster. By referencing a smaller base image in the FROM instruction of your Dockerfile, you can control the overall size of the resulting container image.
- Provide developers within your organization with a set of common, standard, base images by downloading each base image only once. After the initial download, only the layers that make each container image unique are needed, thus reducing the amount of time needed to build the developer's container image.

When building container images with Buildpacks your source code is built into a

secure, efficient, production ready container image.

Vulnerability scanning



It's a best practice to scan your container images for software vulnerabilities, and if found, rebuild the image to include any patches that fix the vulnerabilities, and then redeploy your container.

Container Analysis is a service that provides vulnerability scanning and metadata storage for containers on Google Cloud. The scanning service performs vulnerability scans on images in Artifact Registry, then stores the resulting metadata and makes it available for consumption through an API.

When enabled, this service can automatically scan your container image and is triggered when a new image is pushed to Artifact Registry. With the on-demand scanning API, you can also enable manual scans of container images that are stored in these registries or stored locally.

For more information on the Container Analysis service, view the [documentation](#).

Automated patching



Store your images in Artifact Registry and enable vulnerability scanning.



Configure a job or use Pub/Sub to get notified of vulnerabilities.



Trigger a rebuild of your image with the available fixes.



Using your continuous deployment process, deploy the rebuilt image to a staging environment.



Test the image, and trigger the deployment of the image to production.

To fix vulnerabilities that are discovered in your container image, it's recommended to patch the image with an automated process by using your continuous integration pipeline that was originally used to build the image.

To achieve this, here are a set of high-level steps:

1. Store your images in Artifact Registry and enable vulnerability scanning.
2. Configure a job to fetch vulnerability metadata from the Container Analysis service, and if a vulnerability is detected with available fixes, trigger a rebuild of your image. You can also use Pub/Sub integration to get notified of vulnerabilities and trigger a rebuild of your image.
3. Deploy the rebuilt image to a staging environment by using your continuous deployment process.
4. Test the image in staging and check that the fixes are applied.
5. Trigger the deployment of the image to your production environment.

On-demand scanning

- Grant the default Cloud Build service account the *On-Demand Scanning Admin* IAM role.
- To scan your image for vulnerabilities, run the `gcloud artifacts docker images scan` command.
- Exit the build if vulnerabilities are found with a specified severity level.

```
gcloud artifacts docker images  
scan <image> \  
--format='value(response.scan)' >  
scan.txt
```

As part of your Cloud Build pipeline, you can use on-demand scanning to scan a container image for vulnerabilities after it's built. If the scan detects vulnerabilities at a specified severity level, you can then block the upload of the image to Artifact Registry.

If the default Cloud Build service account is used for the build, grant it the On-Demand Scanning Admin IAM role: (roles/ondemandscanning.admin).

To scan your built container image for vulnerabilities, use the `gcloud artifacts docker images scan` command after the build step in your Cloud Build configuration file, and save the command output to a text file.

Exit the build if the scan command output indicates the desired severity level of any detected vulnerabilities. Otherwise continue with the build to push the built image to Artifact Registry.

For more information, view this [tutorial for on-demand scanning](#).



Agenda



- 01 Containers and container images
- 02 Building container images with Docker
- 03 Lab: Creating and running Docker containers
- 04 Building container images with buildpacks
- 05 Continuous integration and delivery tools
- 06 Best practices
- 07 Quiz**

Let's do a short quiz on this module.

Quiz

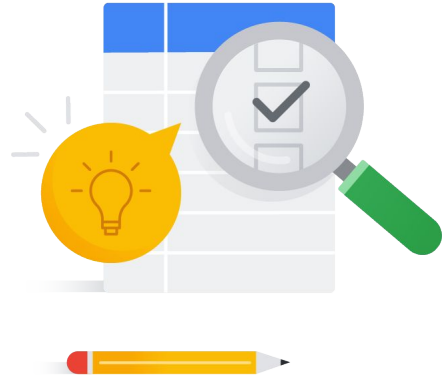


5 min



Group

Introduction to containers



Let's do a short quiz on this module.

Quiz | Question 1

Question

A container image is:

- A. A package with your application and everything that the application needs to run.
- B. A manifest that is used to build a virtual machine.
- C. A base runtime system that you can use to bootstrap your application.
- D. A script that is used to deploy source code to a container platform.

Quiz | Question 2

Question

Which of the following statements about using Docker are correct? Select three.

- A. A Dockerfile contains a set of instructions that is used to build a container image.
- B. A multi-stage build is used to create optimized and secure container images.
- C. You must prebuild your application from source files before using Docker.
- D. Dockerfiles provide a powerful, flexible mechanism to create container images.
- E. When you build a container image, if you try to use a base image that is not available in the local image repository, the Docker build fails.

Quiz | Question 3

Question

Which of the following statements about buildpacks are correct? Select three.

- A. When processing a source directory, a builder executes two phases of a buildpack: the *detect* phase, and the *build* phase.
- B. Buildpacks are distributed and executed in OCI images called builders, where each builder can support a single source code language.
- C. Buildpacks are a way to turn your application source code into a container image without writing a Dockerfile.
- D. To create a container image with buildpacks, you must first develop your own builder.
- E. Google Cloud's buildpacks are built into Cloud Run.

Quiz | Question 4

Question

Which of the following statements about Cloud Build are correct? Select two.

- A. You cannot use a Dockerfile when building container images with Cloud Build.
- B. With Cloud Build, you can continuously build, test, and deploy your application on Google Cloud.
- C. You can only run builds manually when using Cloud Build.
- D. In addition to Cloud Source Repositories, Cloud Build integrates with different source code repositories such as GitHub, Bitbucket, and GitLab.

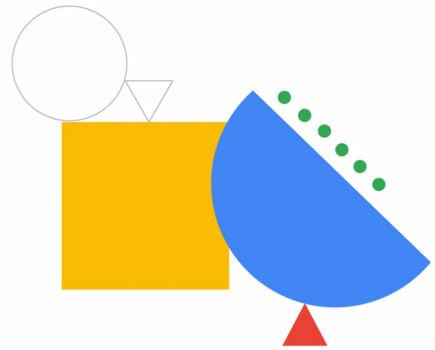
Quiz | Question 5

Question

What are three best practices to use when working with containers or container images?

- A. When you build a Docker container image, the instruction to add your source code should be placed as early as possible near the top of the Dockerfile.
- B. Run your application as the *root* user in the container.
- C. Remove unnecessary files and tools from the container image.
- D. Launch the container process with the *CMD* or *ENTRYPOINT* instruction in your Dockerfile.
- E. Scan your container images for software vulnerabilities.

Review: Introduction to Containers



Let's briefly review the topics that were covered in this module.

In this module, you learned about:

- | | |
|----|---|
| 01 | Containers and container images |
| 02 | Building container images with Docker |
| 03 | Building container images with buildpacks |
| 04 | Continuous integration and delivery tools |
| 05 | Best practices |



You learned that a **container image** is an archive with files that includes executables, system libraries, data files, and more, and a **container** is a runtime instance of a container image that represents the running processes of your container.

We discussed how you can build a container image with Docker by using the different instructions such as FROM, COPY, and RUN. These instructions download base images, copy your source files, build your application with dependencies, and set configuration to run your application.

You also learned about buildpacks, and how to use the pack command line tool to build container images with Google Cloud's buildpacks.

We discussed tools like Skaffold, Cloud Build, and Artifact Registry, which can be used to implement your CI/CD process to build, test, and deploy containers.

And, finally we introduced some best practices when building and running containers.