Docker Compose

Docker Compose is a tool for defining and running multi-container applications. It is the key to unlocking a streamlined and efficient development and deployment experience.

Compose simplifies the control of your entire application stack, making it easy to manage services, networks, and volumes in a single YAML configuration file. Then, with a single command, you create and start all the services from your configuration file.

Compose works in all environments; production, staging, development, testing, as well as CI workflows. It also has commands for managing the whole lifecycle of your application:

- Start, stop, and rebuild services
- View the status of running services
- Stream the log output of running services
- Run a one-off command on a service

Why use Compose?

Key benefits of Docker Compose

Using Docker Compose offers several benefits that streamline the development, deployment, and management of containerized applications:

- Simplified control: Docker Compose allows you to define and manage multi-container applications in a single YAML file. This simplifies the complex task of orchestrating and coordinating various services, making it easier to manage and replicate your application environment.
- Efficient collaboration: Docker Compose configuration files are easy to share, facilitating collaboration among developers, operations teams, and other stakeholders. This collaborative approach leads to smoother workflows, faster issue resolution, and increased overall efficiency.

- Rapid application development: Compose caches the configuration used to create a container. When you restart a service that has not changed, Compose re-uses the existing containers. Re-using containers means that you can make changes to your environment very quickly.
- Portability across environments: Compose supports variables in the Compose file. You can use these variables to customize your composition for different environments, or different users.
- Extensive community and support: Docker Compose benefits from a vibrant and active community, which means abundant resources, tutorials, and support. This community-driven ecosystem contributes to the continuous improvement of Docker Compose and helps users troubleshoot issues effectively.

Common use cases of Docker Compose

Compose can be used in many different ways. Some common use cases are outlined below.

Development environments

When you're developing software, the ability to run an application in an isolated environment and interact with it is crucial. The Compose command line tool can be used to create the environment and interact with it.

The Compose file provides a way to document and configure all of the application's service dependencies (databases, queues, caches, web service APIs, etc). Using the Compose command line tool you can create and start one or more containers for each dependency with a single command (`docker compose up`).

Together, these features provide a convenient way for you to get started on a project. Compose can reduce a multi-page "developer getting started guide" to a single machine-readable Compose file and a few commands.

Automated testing environments

An important part of any Continuous Deployment or Continuous Integration process is the automated test suite. Automated end-to-end testing requires an environment in which to run tests. Compose provides a convenient way to create and destroy isolated testing environments for your test suite. By defining

the full environment in a Compose file, you can create and destroy these environments in just a few commands:

```
$ docker compose up -d
$ ./run_tests
$ docker compose down
```

Single host deployments

Compose has traditionally been focused on development and testing workflows, but with each release we're making progress on more production-oriented features.

For details on using production-oriented features, see Compose in production.

How Compose works

With Docker Compose you use a YAML configuration file, known as the Compose file, to configure your application's services, and then you create and start all the services from your configuration with the Compose CLI.

The Compose file, or `compose.yaml` file, follows the rules provided by the Compose Specification in how to define multi-container applications. This is the Docker Compose implementation of the formal Compose Specification.

The Compose application model

Computing components of an application are defined as services. A service is an abstract concept implemented on platforms by running the same container image, and configuration, one or more times.

Services communicate with each other through networks. In the Compose Specification, a network is a platform capability abstraction to establish an IP route between containers within services connected together.

Services store and share persistent data into volumes. The Specification describes such a persistent data as a high-level filesystem mount with global options.

Some services require configuration data that is dependent on the runtime or platform. For this, the Specification defines a dedicated configs concept. From a service container point of view, configs are comparable to volumes, in that they are files mounted into the container. But the actual definition involves distinct platform resources and services, which are abstracted by this type.

A secret is a specific flavor of configuration data for sensitive data that should not be exposed without security considerations. Secrets are made available to services as files mounted into their containers, but the platform-specific resources to provide sensitive data are specific enough to deserve a distinct concept and definition within the Compose Specification.

> Note
>
> With volumes, configs and secrets you can have a simple declaration at the top-level and then add more platform-specific information at the service level.

A project is an individual deployment of an application specification on a platform. A project's name, set with the top-level `name` attribute, is used to group resources together and isolate them from other applications or other installation of the same Compose-specified application with distinct parameters. If you are creating resources on a platform, you must prefix resource names by project and set the label `com.docker.compose.project`.

Compose offers a way for you to set a custom project name and override this name, so that the same `compose.yaml` file can be deployed twice on the same infrastructure, without changes, by just passing a distinct name.

The Compose file

The default path for a Compose file is `compose.yaml` (preferred) or `compose.yml` that is placed in the working directory. Compose also supports `docker-compose.yaml` and `docker-compose.yml` for backwards compatibility of earlier versions. If both files exist, Compose prefers the canonical `compose.yaml`.

You can use fragments and extensions to keep your Compose file efficient and easy to maintain.

Multiple Compose files can be merged together to define the application model. The combination of YAML files is implemented by appending or overriding YAML elements based on the Compose file order you set. Simple attributes and maps get overridden by the highest order Compose file, lists get merged by appending. Relative paths are resolved based on the first Compose file's parent folder, whenever complimentary files being merged are hosted in other folders. As some Compose file elements can both be expressed as single strings or

complex objects, merges apply to the expanded form. For more information, see Working with multiple Compose files.

If you want to reuse other Compose files, or factor out parts of your application model into separate Compose files, you can also use `include`. This is useful if your Compose application is dependent on another application which is managed by a different team, or needs to be shared with others.

CLI

The Docker CLI lets you interact with your Docker Compose applications through the `docker compose` command, and its subcommands. Using the CLI, you can manage the lifecycle of your multi-container applications defined in the `compose.yaml` file. The CLI commands enable you to start, stop, and configure your applications effortlessly.

Key commands

To start all the services defined in your `compose.yaml` file:

```
$ docker compose up
```

To stop and remove the running services:

```
$ docker compose down
```

If you want to monitor the output of your running containers and debug issues, you can view the logs with:

```
$ docker compose logs
```

To list all the services along with their current status:

```
$ docker compose ps
```

For a full list of all the Compose CLI commands, see the reference documentation.

Illustrative example

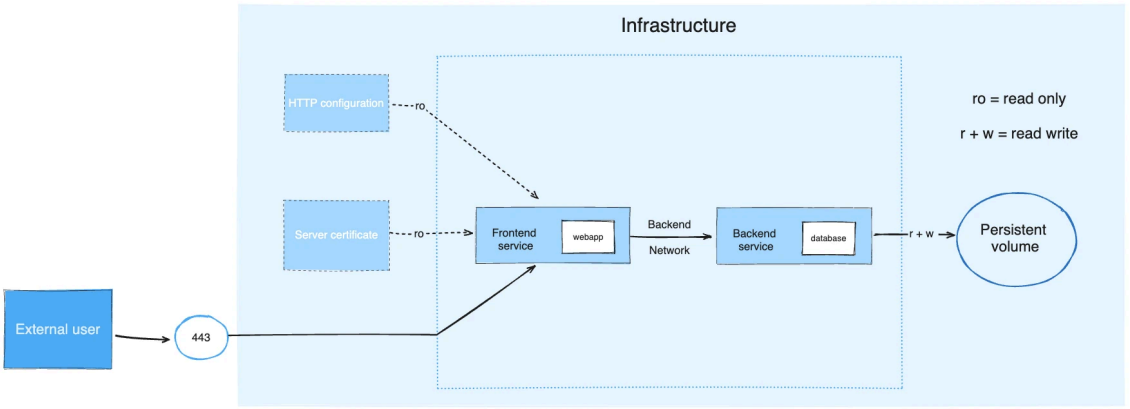
The following example illustrates the Compose concepts outlined above. The example is non-normative.

Consider an application split into a frontend web application and a backend service.

The frontend is configured at runtime with an HTTP configuration file managed by infrastructure, providing an external domain name, and an HTTPS server certificate injected by the platform's secured secret store.

The backend stores data in a persistent volume.

Both services communicate with each other on an isolated back-tier network, while the frontend is also connected to a front-tier network and exposes port 443 for external usage.



The example application is composed of the following parts:

- 2 services, backed by Docker images: `webapp` and `database`
- 1 secret (HTTPS certificate), injected into the frontend
- 1 configuration (HTTP), injected into the frontend
- 1 persistent volume, attached to the backend
- 2 networks

```
services:
 frontend:
   image: example/webapp
   ports:
     - "443:8043"
   networks:
     - front-tier
     - back-tier
   configs:
     - httpd-config
   secrets:
     - server-certificate

 backend:
   image: example/database
   volumes:
     - db-data:/etc/data
   networks:
     - back-tier
```

```
volumes:
  db-data:
    driver: flocker
    driver_opts:
      size: "10GiB"

configs:
  httpd-config:
    external: true

secrets:
  server-certificate:
    external: true

networks:
  # The presence of these objects is sufficient to define them
  front-tier: {}
  back-tier: {}
```

The `docker compose up` command starts the `frontend` and `backend` services, creates the necessary networks and volumes, and injects the configuration and secret into the frontend service.

`docker compose ps` provides a snapshot of the current state of your services, making it easy to see which containers are running, their status, and the ports they are using:

```
$ docker compose ps
```

```
NAME                IMAGE               COMMAND                  SERVICE             CREATED             STATUS              PORTS
example-frontend-1  example/webapp      "nginx -g 'daemon of..."  frontend            2 minutes ago       Up 2 minutes        0.0.0.0:443->8043/tcp
example-backend-1   example/database    "docker-entrypoint.s..."  backend             2 minutes ago
```

Overview of installing Docker Compose

This page summarizes the different ways you can install Docker Compose, depending on your platform and needs.

Installation scenarios

Scenario one: Install Docker Desktop (Recommended)

The easiest and recommended way to get Docker Compose is to install Docker Desktop.

Docker Desktop includes Docker Compose along with Docker Engine and Docker CLI which are Compose prerequisites.

Docker Desktop is available for:

- Linux
- Mac
- Windows

> Tip
>
> **About Docker Desktop**
>
> 

Scenario two: Install the Docker Compose plugin (Linux only)

> Important
>
> This method is only available on Linux.

If you already have Docker Engine and Docker CLI installed, you can install the Docker Compose plugin from the command line, by either:

- Using Docker's repository
- Downloading and installing manually

Scenario three: Install the Docker Compose standalone (Legacy)

> Warning
>
> This install scenario is not recommended and is only supported for backward compatibility purposes.

You can install the Docker Compose standalone on Linux or on Windows Server.

Docker Compose Quickstart

This tutorial aims to introduce fundamental concepts of Docker Compose by guiding you through the development of a basic Python web application.

Using the Flask framework, the application features a hit counter in Redis, providing a practical example of how Docker Compose can be applied in web development scenarios.

The concepts demonstrated here should be understandable even if you're not familiar with Python.

This is a non-normative example that demonstrates core Compose functionality.

Prerequisites

Make sure you have:

- Installed the latest version of Docker Compose
- A basic understanding of Docker concepts and how Docker works

Step 1: Set up

1. Create a directory for the project:

   ```
   $ mkdir composetest
   $ cd composetest
   ```

2. Create a file called `app.py` in your project directory and paste the following code in:

   ```
   import time

   import redis
   from flask import Flask

   app = Flask(__name__)
   cache = redis.Redis(host='redis', port=6379)

   def get_hit_count():
       retries = 5
       while True:
           try:
               return cache.incr('hits')
           except redis.exceptions.ConnectionError as exc:
               if retries == 0:
                   raise exc
               retries -= 1
               time.sleep(0.5)

   @app.route('/')
   ```

```
def hello():
    count = get_hit_count()
    return f'Hello World! I have been seen {count} times.\n'
```

In this example, `redis` is the hostname of the redis container on the application's network and the default port, `6379` is used.

> Note
>
> Note the way the `get_hit_count` function is written. This basic retry loop attempts the request multiple times if the Redis service is not available. This is useful at startup while the application comes online, but also makes the application more resilient if the Redis service needs to be restarted anytime during the app's lifetime. In a cluster, this also helps handling momentary connection drops between nodes.

3. Create another file called `requirements.txt` in your project directory and paste the following code in:

```
flask
redis
```

4. Create a `Dockerfile` and paste the following code in:

```
# syntax=docker/dockerfile:1

FROM python:3.10-alpine

WORKDIR /code

ENV FLASK_APP=app.py

ENV FLASK_RUN_HOST=0.0.0.0

RUN apk add --no-cache gcc musl-dev linux-headers

COPY requirements.txt requirements.txt

RUN pip install -r requirements.txt

EXPOSE 5000

COPY . .

CMD ["flask", "run", "--debug"]
```

Understand the Dockerfile

> Important
>
> Check that the `Dockerfile` has no file extension like `.txt`. Some editors may append this file extension automatically which results in an error when you run the application.

For more information on how to write Dockerfiles, see the Dockerfile reference.

Step 2: Define services in a Compose file

Compose simplifies the control of your entire application stack, making it easy to manage services, networks, and volumes in a single, comprehensible YAML configuration file.

Create a file called `compose.yaml` in your project directory and paste the following:

```
services:
 web:
   build: .
   ports:
     - "8000:5000"
 redis:
   image: "redis:alpine"
```

This Compose file defines two services: `web` and `redis`.

The `web` service uses an image that's built from the `Dockerfile` in the current directory. It then binds the container and the host machine to the exposed port, `8000`. This example service uses the default port for the Flask web server, `5000`.

The `redis` service uses a public Redis image pulled from the Docker Hub registry.

For more information on the `compose.yaml` file, see How Compose works.

Step 3: Build and run your app with Compose

With a single command, you create and start all the services from your configuration file.

1. From your project directory, start up your application by running `docker compose up`.

   ```
   $ docker compose up
   ```

```
Creating network "composetest_default" with the default driver
Creating composetest_web_1 ...
Creating composetest_redis_1 ...
Creating composetest_web_1
Creating composetest_redis_1 ... done
Attaching to composetest_web_1, composetest_redis_1
web_1   |  * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
redis_1 | 1:C 17 Aug 22:11:10.480 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
redis_1 | 1:C 17 Aug 22:11:10.480 # Redis version=4.0.1, bits=64, commit=00000000, modified=0, pid=1, just started
redis_1 | 1:C 17 Aug 22:11:10.480 # Warning: no config file specified, using the default config. In order to specify a config file use redis-server /path/to/redis.conf
web_1   |  * Restarting with stat
redis_1 | 1:M 17 Aug 22:11:10.483 * Running mode=standalone, port=6379.
redis_1 | 1:M 17 Aug 22:11:10.483 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower value of 128.
web_1   |  * Debugger is active!
redis_1 | 1:M 17 Aug 22:11:10.483 # Server initialized
redis_1 | 1:M 17 Aug 22:11:10.483 # WARNING you have Transparent Huge Pages (THP) support enabled in your kernel. This will create latency and memory usage issues with Redis. To fix this issue run the command 'echo never > /sys/kernel/mm/transparent_hugepage/enabled' as root, and add it to your /etc/rc.local in order to retain the setting after a reboot. Redis must be restarted after THP is disabled.
web_1   |  * Debugger PIN: 330-787-903
redis_1 | 1:M 17 Aug 22:11:10.483 * Ready to accept connections
```

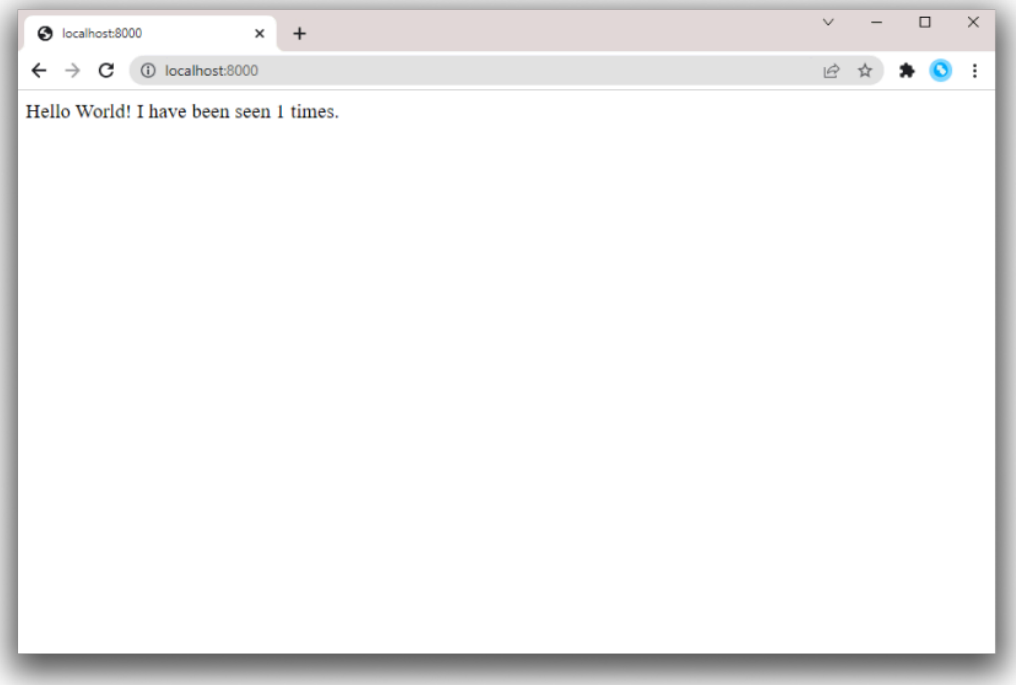
Compose pulls a Redis image, builds an image for your code, and starts the services you defined. In this case, the code is statically copied into the image at build time.

2. Enter `http://localhost:8000/` in a browser to see the application running.

   If this doesn't resolve, you can also try `http://127.0.0.1:8000`.

   You should see a message in your browser saying:

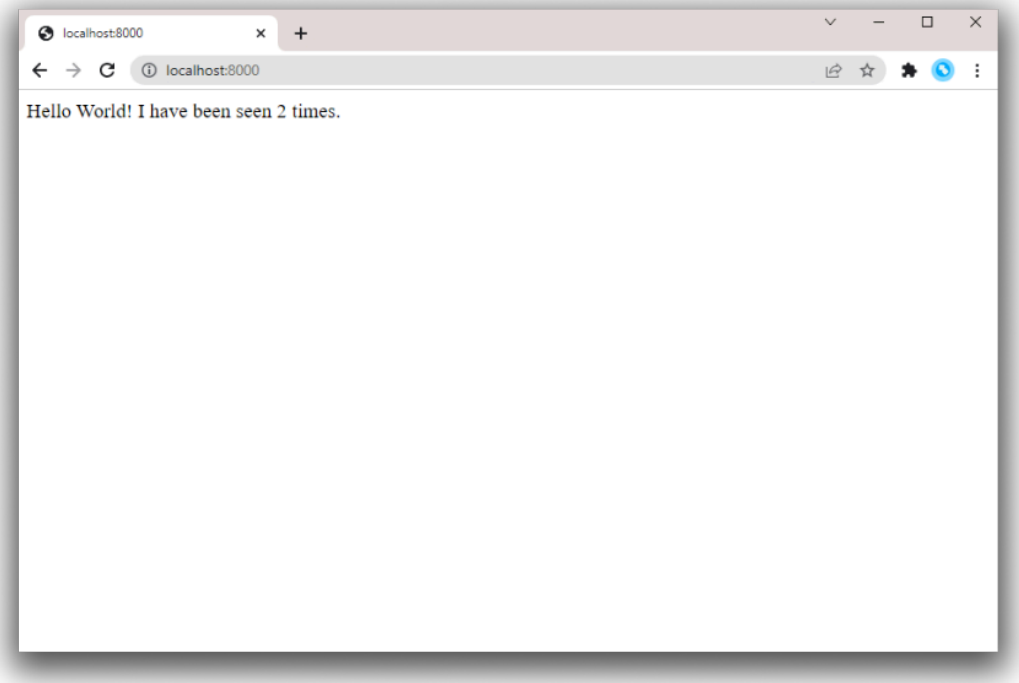
   `Hello World! I have been seen 1 times.`



3. Refresh the page.

   The number should increment.

   Hello World! I have been seen 2 times.



4. Switch to another terminal window, and type `docker image ls` to list local images.

   Listing images at this point should return `redis` and `web`.

   ```
   $ docker image ls
   ```

   ```
   REPOSITORY        TAG          IMAGE ID      CREATED         SIZE
   composetest_web   latest       e2c21aa48cc1  4 minutes ago   93.8MB
   python            3.4-alpine   84e6077c7ab6  7 days ago      82.5MB
   redis             alpine       9d8fa9aa0e5b  3 weeks ago     27.5MB
   ```

   You can inspect images with `docker inspect <tag or id>`.

5. Stop the application, either by running `docker compose down` from within your project directory in the second terminal, or by hitting `CTRL+C` in the original terminal where you started the app.

Step 4: Edit the Compose file to use Compose Watch

Edit the `compose.yaml` file in your project directory to use `watch` so you can preview your running Compose services which are automatically updated as you edit and save your code:

```
services:
 web:
   build: .
   ports:
     - "8000:5000"
   develop:
     watch:
       - action: sync
         path: .
         target: /code
 redis:
   image: "redis:alpine"
```

Whenever a file is changed, Compose syncs the file to the corresponding location under `/code` inside the container. Once copied, the bundler updates the running application without a restart.

For more information on how Compose Watch works, see Use Compose Watch. Alternatively, see Manage data in containers for other options.

> Note
>
> For this example to work, the `--debug` option is added to the `Dockerfile`. The `--debug` option in Flask enables automatic code reload, making it possible to work on the backend API

> without the need to restart or rebuild the container. After changing the `.py` file, subsequent API calls will use the new code, but the browser UI will not automatically refresh in this small example. Most frontend development servers include native live reload support that works with Compose.

Step 5: Re-build and run the app with Compose

From your project directory, type `docker compose watch` or `docker compose up --watch` to build and launch the app and start the file watch mode.

```
$ docker compose watch
[+] Running 2/2
✔ Container docs-redis-1 Created
0.0s
✔ Container docs-web-1  Recreated
0.1s
Attaching to redis-1, web-1
      ⦿ watch enabled
...
```

Check the `Hello World` message in a web browser again, and refresh to see the count increment.

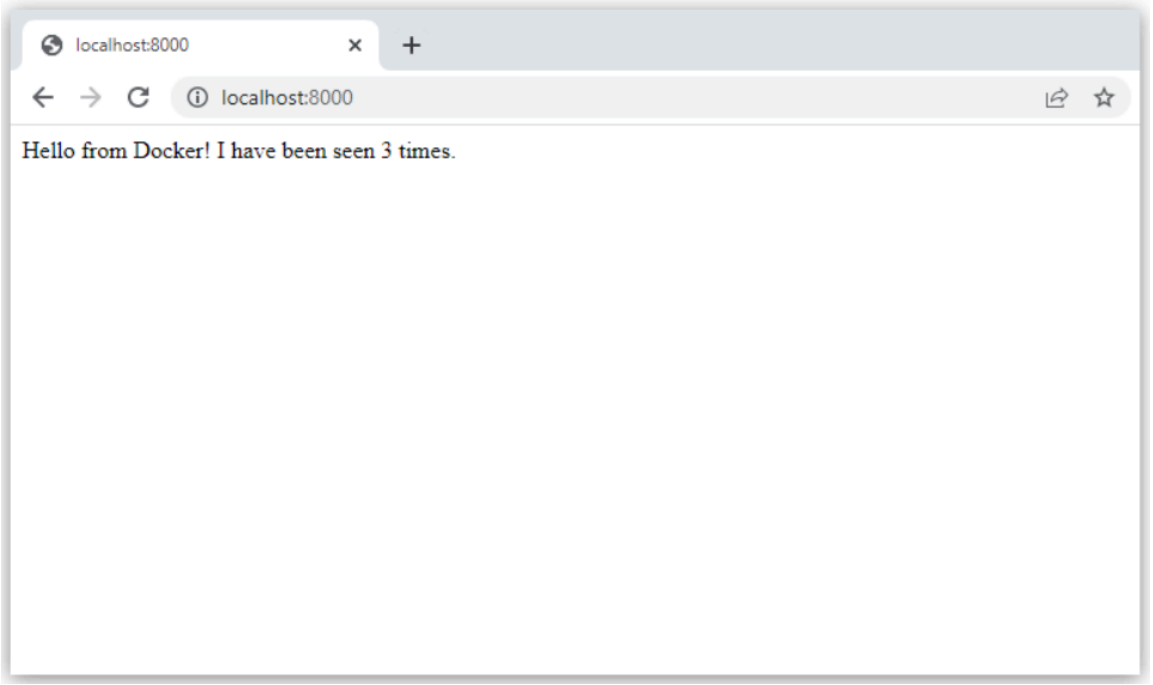
Step 6: Update the application

To see Compose Watch in action:

1. Change the greeting in `app.py` and save it. For example, change the `Hello World!` message to `Hello from Docker!`:

   ```
   return f'Hello from Docker! I have been seen {count} times.\n'
   ```

2. Refresh the app in your browser. The greeting should be updated, and the counter should still be incrementing.



3. Once you're done, run `docker compose down`.

Step 7: Split up your services

Using multiple Compose files lets you customize a Compose application for different environments or workflows. This is useful for large applications that may use dozens of containers, with ownership distributed across multiple teams.

1. In your project folder, create a new Compose file called `infra.yaml`.
2. Cut the Redis service from your `compose.yaml` file and paste it into your new `infra.yaml` file. Make sure you add the `services` top-level attribute at the top of your file. Your `infra.yaml` file should now look like this:

```
services:
 redis:
   image: "redis:alpine"
```

3. In your `compose.yaml` file, add the `include` top-level attribute along with the path to the `infra.yaml` file.

```
include:
  - infra.yaml
services:
 web:
   build: .
   ports:
     - "8000:5000"
   develop:
     watch:
```

```
      - action: sync
        path: .
        target: /code
```

4. Run `docker compose up` to build the app with the updated Compose files, and run it. You should see the `Hello world` message in your browser.

This is a simplified example, but it demonstrates the basic principle of `include` and how it can make it easier to modularize complex applications into sub-Compose files. For more information on `include` and working with multiple Compose files, see Working with multiple Compose files.

Step 8: Experiment with some other commands

- If you want to run your services in the background, you can pass the `d` flag (for "detached" mode) to `docker compose up` and use `docker compose ps` to see what is currently running:

  ```
  $ docker compose up -d

  Starting composetest_redis_1...
  Starting composetest_web_1...

  $ docker compose ps

       Name                     Command               State           Ports
  -------------------------------------------------------------------------------------
  composetest_redis_1   docker-entrypoint.sh redis ...   Up      6379/tcp
  composetest_web_1     flask run                        Up      0.0.0.0:8000->5000/tcp
  ```

- Run `docker compose --help` to see other available commands.
- If you started Compose with `docker compose up -d`, stop your services once you've finished with them:

  ```
  $ docker compose stop
  ```

- You can bring everything down, removing the containers entirely, with the `docker compose down` command.