Docker Lab 4
Persistent storage with volumes and bind mounts

1 Commands covered

2 Lab setup

This is identical to the previous lab

The root folders for all the various Node projects here can be found in the `Lab 4` subfolder in the main `labcode` folder of your downloaded zip for this workshop.

3 Anonymous volumes

The root folder for this project is: `demo-volumes`

In the root folder, create the Dockerfile for building this project

`Dockerfile`

```
FROM node:14
```

```
WORKDIR /app

COPY package.json /app

#include this if you get certificate related errors
#RUN npm config set strict-ssl false

RUN npm install

COPY . /app

EXPOSE 80

CMD ["node", "server.js"]
```

Build an image from the Dockerfile with a custom tag:

```
docker build -t feedback-node .
```

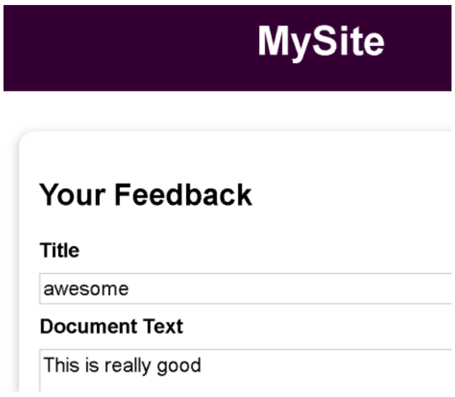
Check that the image is generated with the correct tag:

```
docker images
```

Next create and run a container from this custom image with a custom name in detached mode:

```
docker run -p 3000:80 -d --name feedback-app --rm feedback-node
```

Check that the app is running at `localhost:3000` and enter some random feedback. The title for the feedback will be used as a filename for a file that is saved by the app with the content of the file being the document text.



Your Feedback

Title

awesome

Document Text

This is really good

You can check the contents of the files entered so far at a specific API endpoint designed for this purpose:

`localhost:3000/feedback/`*`nameoftitle`*`.txt`

for e.g.

```
localhost:3000/feedback/awesome.txt
```

Try entering a few different titles with corresponding content, and verify you can retrieve the file content corresponding to these titles from this API endpoint.

The app stores all these text files in the `/feedback` subfolder (check code logic for this). It first stores in the content for theses file in the `/temp` subfolder and if it verifies that there is no existing file with the same name in the `/feedback` subfolder, it renames the file so that it ultimately gets stored in the `/feedback` subfolder.

To verify this, in a second Powershell terminal, open an interactive bash shell to the running container:

```
docker exec -it feedback-app /bin/bash
```

The shell will start in the folder specified as `WORKDIR` in the your Dockerfile instruction (which is `/app`). Then navigate to the `/feedback` subfolder to verify that the specific title files are in there with the correct content.

```
root@ef02dda637bf:/app# pwd
root@ef02dda637bf:/app# ls -la

total 76
drwxr-xr-x  1 root root  4096 Aug  3 08:09 .
drwxr-xr-x  1 root root  4096 Aug  3 08:09 ..
-rwxr-xr-x  1 root root  6148 Oct 30  2020 .DS_Store
-rwxr-xr-x  1 root root   132 Aug  3 08:05 Dockerfile
drwxr-xr-x  1 root root  4096 Aug  3 08:14 feedback
drwxr-xr-x 66 root root  4096 Jul 27 07:43 node_modules
…..

root@ef02dda637bf:/app# cd feedback
root@ef02dda637bf:/app/feedback# ls -la
total 28
drwxr-xr-x 1 root root 4096 Aug  3 08:14 .
drwxr-xr-x 1 root root 4096 Aug  3 08:09 ..
-rw-r--r-- 1 root root   19 Aug  3 08:11 awesome.txt
….
….

root@ef02dda637bf:/app/feedback# cat awesome.txt
this is really good
root@ef02dda637bf:/app/feedback# exit
```

Notice that there are no files contained in the `feedback` subfolder of the current project on the file system of the host machine, which is expected, since the app is running inside the container and there is no mapping yet between the file system of the host machine and the running container.

Let’s stop the container with:

```
docker stop feedback-app
```

Since we started the container with the `--rm` flag, stopping the container will also automatically delete it. Check this with:

```
docker ps -a
```

Let's start the container again, but this time without the `--rm` option:

```
docker run -p 3000:80 -d --name feedback-app feedback-node
```

If you now were to attempt to retrieve the files that you had stored in the previous container, for e.g. with:

```
localhost:3000/feedback/nameoftitle.txt
```

You will see they no longer exist.
You can alternatively open another interactive shell terminal into the latest running container and navigate to the `feedback` folder to verify this.

```
docker exec -it feedback-app /bin/bash
```

This is exactly as expected since this container is started from a fresh image and hence all data stored in the previous container is completely lost.

Now enter some more random titles and content as you did previously, and record down these titles and content into a empty Notepad file so you can double check and verify later on afterwards.
Then, verify that you can retrieve their content with:

```
localhost:3000/feedback/nameoftitle.txt
```

Now stop the container and start it again with:

```
docker stop feedback-app

docker start feedback-app
```

This time after restarting we are able to access the files corresponding to feedback titles that we had entered previously because the container was just stopped, not deleted. Therefore, any temporary data is still resident in the local file system of the container. This data is only lost when the container is permanently deleted (`docker rm container`)

There are several ways in which we can persist data outside of container deletion. One simple way is to use the VOLUME instruction in the Dockerfile to target the folder containing data to be persisted.
https://docs.docker.com/reference/dockerfile/#volume
https://serverfault.com/questions/1157229/does-the-volume-instruction-in-a-dockerfile-create-a-volume-or-only-specify-a-mo
The VOLUME instruction creates a mount point with the specified name and marks it as holding externally mounted volumes from native host. These volumes are anonymous volumes.

We can modify our Dockerfile to include this instruction:

```
FROM node:14
```

```
WORKDIR /app

COPY package.json /app

#include this if you get certificate related errors
#RUN npm config set strict-ssl false

RUN npm install

COPY . /app

VOLUME ["app/feedback"]

EXPOSE 80

CMD ["node", "server.js"]
```

Build a new image with this modified Dockerfile:

```
docker build -t feedback-node:volumes .
```

Remove the previous running container with:

```
docker rm -f feedback-app
```

Start a new container with the same name based on this image:

```
docker run -p 3000:80 -d --name feedback-app feedback-node:volumes
```

In a separate shell terminal, list all existing volumes with:

```
docker volume ls
```

You should see a single volume whose name is a SHA 1 digest of its contents. This is the new anonymous volume generated from the Dockerfile instruction `VOLUME ["app/feedback"]`

```
DRIVER    VOLUME NAME
local     ef06e95851e2698453128b4f2ff8f72678c138e6a9727840c2fbb00232add2d6
```

Go back to the app on localhost:3000, enter some random feedback

Now stop and remove the app with:

```
docker rm -f feedback-app
```

and start it again with:

```
docker run -p 3000:80 -d --name feedback-app feedback-node:volumes
```

Now check for all existing volumes with:

```
docker volume ls
```

Notice now that we have 2 anonymous volumes. This means that every time we recreate a container from the image a new anonymous volume is created, rather than reusing the existing volume.

Verify this one more time by removing the current container and restarting it again:

```
docker rm -f feedback-app
```

```
docker run -p 3000:80 -d --name feedback-app feedback-node:volumes
```

Now check for all existing volumes again with:

```
docker volume ls
```

Notice now that we have 3 anonymous volumes.

Remove the current container with:

```
docker rm -f feedback-app
```

This confirms that every time we recreate a container from the image, a new anonymous volume is created, rather than reusing the previous existing volume matched to the previous container generation.

Since anonymous volumes **do not stay the same between container removal and generation**, there is **no way to actually persist data using anonymous volumes**.

However anonymous volumes do have a specific use case besides persisting data, which we will be seeing later.

One particular feature with anonymous volumes is that if a container is started with `--rm` option (which means it will be automatically removed when it is stopped), the anonymous volume associated with that container will also be removed. This is useful to prevent proliferation of anonymous volumes when we need to use them later on.

Now let's recreate a new container but this time with the `--rm` option:

```
docker run -p 3000:80 -d --name feedback-app --rm feedback-
node:volumes
```

Check for all existing volumes with:

```
docker volume ls
```

You should now see we have 4 anonymous volumes, which is as expected.

However, if you now end the current running container with:

```
docker stop feedback-app
```

and check again with

```
docker volume ls
```

you will see that we are back to 3 anonymous volumes. By default, anonymous volumes are removed if the container was started with the `--rm` option and was stopped thereafter. They **are not removed** if a container was started (and then removed) without that option.

4 Named volumes

To resolve our problem of persisting data permanently after container removals, we will need to use named volumes.

You cannot create named volumes inside a Dockerfile, so remove the VOLUME instruction:

```
FROM node:14

WORKDIR /app

COPY package.json /app

#include this if you get certificate related errors
#RUN npm config set strict-ssl false

RUN npm install

COPY . /app

EXPOSE 80

CMD ["node", "server.js"]
```

Rebuild the image again:

```
docker build -t feedback-node:volumes .
```

Now run the container with a new option `-v` to specify a named volume:

```
docker run -p 3000:80 -d --name feedback-app -v feedback:/app/feedback
--rm feedback-node:volumes
```

After starting, check that the named volume exists with:

```
docker volume ls
```

Return back to the app and verify that it is working properly now and you can enter a few random titles and associated content, and also that you can retrieve their content with:

```
localhost:3000/feedback/nameoftitle.txt
```

Now stop the app and verify that is completely removed

```
docker stop feedback-app
```

```
docker ps -a
```

and verify that the named volume still exists with:

```
docker volume ls
```

Now recreate a totally new container again with the same named volume that we used previously, and again mapping to the same path in the file system of the container:

```
docker run -p 3000:80 -d --name feedback-app -v feedback:/app/feedback
--rm feedback-node:volumes
```

Now if you attempt to access the titles that you had entered previously:

```
localhost:3000/feedback/nameoftitle.txt
```

You will get back the content stored there in the previous run of the container.

Continue to add new titles and associated content. Then repeat what you had just done: shut down the container, verify that it is removed and restart a new container with the same named volume mapping. You will find that you can continue to add new data to the named volume and that this data will continue to be persisted between container removals and generations.

When you are done, remove all containers but leave the named volume as it is.

You can delete all the anonymous volumes (which we will no longer be using) with:

```
docker volume prune
```

Check that you still have your singled name volume remaining with:

```
docker volume ls
```

Sometimes you may wish to inspect the content of the named volume without necessarily starting the container with the app that was originally associated with it.

To do this, you could use a light weight Linux base image (such as `alpine` or `busybox`) to generate a container to which you can mount the named volume to a local path in that container, and then generate an interactive shell to enter the container and inspect that path.

Lets pull the Alpine image first to our local registry (if we don't already have it):

```
docker pull alpine
```

Now run a temporary container (using a `--rm` flag to automatically remove the container when we exit) and mount the named volume at any convenient path in the file system of that container (Docker will create that path if it does not exist):

```
docker run --rm -it -v feedback:/mnt alpine /bin/sh

/ # pwd
/
/ # cd /mnt
/mnt # ls -l
total 8
-rw-r--r--    1 root     root            14 Aug  3 09:14 cats.txt
-rw-r--r--    1 root     root            18 Aug  3 09:14 dogs.txt
/mnt # cat dogs.txt
horrible creatures
/mnt # exit
```

Although named volumes have a name (as opposed to anonymous volumes which are identified by their SHA1 digest), we do not know exactly where on the native file system they are stored. The contents of the named volume is managed by Docker directly: all we do is specify the path in a container file system that we wish to map that named volume to when we start the container.

5 Using Bind mounts with volumes

Bind mounts, like named volumes, provide a way to persist data beyond container removals and restarts. However, bind mounts map an explicit directory on the native host file system to a path within the container. Any changes made to the native host directory will affect the mapped path within the container, and vice versa.

Volumes have several benefits compared to bind mounts:
- Volumes provide better performance than bind mounts.
- Docker automatically manages the creation, deletion, and updating of volumes.

Bind Mounts provide their own benefits as well:
- Bind mounts can be used with non-Docker applications, providing more flexibility.
- They enable you to manage data on the host system directly without relying on Docker.

A classic use case for bind mounts is to be able to see updates in our source code take immediate effect in the container without the need to rebuild the image again. We can bind the folder containing our source code files in our project directory on the native host file system with the equivalent path in the container, so that when we update any of the relevant source code files, these changes are immediately reflected into the same files in the running container. Then with the help of a hot reload monitor (which is common with most web server frameworks such as React, Node, etc) we can then immediately execute these updates live.

To demonstrate this, start up a container using the previous image and named volume mapping again:

```
docker run -p 3000:80 -d --name feedback-app -v feedback:/app/feedback
--rm feedback-node:volumes
```

Make some modification to `pages/feedback.html` on your local project folder which determines what is shown on the main page of this webapp. For e.g. change the main <h1> or <h2> header.

```
…..

  <body>
    <header>
      <h1><a href="/">My Cool Site</a></h1>
    </header>
    <main>
      <section>
        <h2>Your Feedback is required</h2>
        <form action="/create" method="POST">
          <div class="form-control">
……
```

As expected, even after saving, the changes still do not show up immediately on the webpage. This is because we would have to stop / remove the container, rebuild the image using the same Dockerfile in order for this latest source code changes be incorporated into this image, and then start a brand new container with that image.

5.1 Bind mount issues

We will now make a first attempt to introduce a bind mount, which essentially maps a path on the native host file system to a path on the container file system. The bind mount is introduced as a option in the `docker run` command.

When using a bind mount in this way, subtle errors can arise in running the container which are important to be aware of and which we will demonstrate below.

We will map the path for the root folder of the current project: `demo-volumes` to the `WORKDIR` folder in the container (`/app`) as specified in the Dockerfile, as this is the destination that we copy our project root folder to (`COPY .    .`)

Restart the previous container by specifying an additional bind mount volume using an absolute file path to the project root folder (specify double quotes to ensure no problems with spaces), for e.g.

```
docker run -p 3000:80 -d --name feedback-app -v feedback:/app/feedback -v "complete path to root folder:/app" feedback-node:volumes
```

or if you are already on the root path of the current project, then this can be simplified to:

```
docker run -p 3000:80 -d --name feedback-app -v feedback:/app/feedback -v ${pwd}:/app feedback-node:volumes
```

However, as soon as you run this command, you will notice that the container immediately exits instead of running in the background as we expect it to.

```
docker ps -a
```

We can check the logs of the container to figure out the cause for this error:

```
docker logs feedback-app
```

```
Error: Cannot find module 'express'
Require stack:
- /app/server.js
    at Function.Module._resolveFilename (internal/modules/cjs/loader.js:931:15)
    at Function.Module._load (internal/modules/cjs/loader.js:774:27)
……
```

Here the problem is that the import statement in `server.js` for the `express` module fails because this module is not available:

```
……
const express = require('express');
…….
```

All Node applications will store their dependencies (JavaScript libraries / modules imported in the program) in the `node_modules` folder. These dependencies required by the app are specified in `package.json` and are fetched from the npm registry when the `npm install` command is executed. The npm registry is a public collection of open-source code packages for Node.js, front-end web apps, mobile apps, and more

The problem here is that the `node_modules` folder is initially generated in the `/app` folder in the file system of the container used to create the build image after the `npm install` command is executed in Dockerfile. Subsequently after that, when we start a container from the image, the `node_modules` folder should already be present.

However, when we create a bind mount from our current root project folder `demo-volumes` (which does not contain the `node_modules` folder) to the `/app` folder in the running container , this will dynamically copy over the entire contents of this root project folder and overwrite the contents of the `/app` folder. This is what we want to accomplish, since we would want any changes to any source code file in our current project (main `server.js`, the HTML files in `pages` subfolder, etc, etc) to be immediately copied over into the container in the `/app` folder.

Let's verify this by removing the container and running it without the bind mount that we introduced just now:

```
docker rm -f feedback-app

docker run -p 3000:80 -d --name feedback-app -v feedback:/app/feedback
--rm feedback-node:volumes
```

As expected, the container remains up and running, and we can access it at localhost:3000 as we usually do.

Open an interactive Bash shell in the running container and explore the `/app` folder within it to verify the existence of the `node_modules` folder.

```
docker exec -it feedback-app /bin/bash

root@8a283dad5613:/app# ls -l
total 56
-rwxr-xr-x  1 root root   132 Aug  3 09:13 Dockerfile
drwxr-xr-x  2 root root  4096 Aug  3 09:14 feedback
drwxr-xr-x 66 root root  4096 Jul 27 07:43 node_modules
-rw-r--r--  1 root root 20564 Jul 27 07:43 package-lock.json
-rwxr-xr-x  1 root root   260 Oct  7  2020 package.json
....
....

root@8a283dad5613:/app# cd node_modules
root@8a283dad5613:/app/node_modules# ls -l
total 252
drwxr-xr-x 2 root root 4096 Jul 27 07:43 accepts
drwxr-xr-x 2 root root 4096 Jul 27 07:43 array-flatten
drwxr-xr-x 3 root root 4096 Jul 27 07:43 body-parser
.....
root@8a283dad5613:/app/node_modules# exit
```

Remove the container with:

```
docker rm -f feedback-app
```

Now lets restart the container again with the bind mount option specified which will again cause it to crash:

```
docker run -p 3000:80 -d --name feedback-app -v
feedback:/app/feedback -v ${pwd}:/app feedback-node:volumes
```

We would like to inspect the file system of this exited container to see whether the `node_modules` folder is actually present in the `/app` folder, thereby confirming the explanation provided previously. However, there is no way to attach an interactive shell to an exited container (in the same way that we just did for a live running container) and if we try to restart the crashed container with:

```
docker start feedback-app
```

It will immediately crash again

```
docker ps -a
```

This is because when you restart an exited container it will commence with the last command specified in the Dockerfile that generated the image the container was created from, which in this case is `node server.js`, which attempts to run the server.js application and crashes immediately when the express module is not found.

However, we can still copy the contents of a specific path in the file system of the exited container to the host file system so that we can inspect it there instead.
Identify or create a suitable empty directory on the host file system for this purpose, and then copy the `/app` folder within the exited container to this empty directory with this command:

```
docker cp feedback-app:/app /path/to/emptydirectory
```

If you are using Windows, make sure to encode the path with double quotes, i.e.

```
docker cp feedback-app:/app "/path/to/emptydirectory"
```

Once you have completed this copy operation, inspect the destination directory with Windows Explorer and verify that the `node_modules` is **NOT PRESENT** there, which confirms the explanation for the source of this subtle error.

You can then remove the app again with:

```
docker rm -f feedback-app
```

5.2 Solution with targeted bind mounting and anonymous volumes

One obvious way to overcome this problem is to actually create a `node_modules` folder in the project root folder on our host file system by also executing `npm install` there to download all the required dependencies specified in `package.json`. Then, when the bind mount is specified, this `node_modules` folder is also copied over into the `/app` folder in the container which provides dependencies required for the application to run correctly.

There are 2 problems with this obvious approach:

a) The number of dependencies in the `node_modules` folder is extremely large, and copying all of these from our host file system to the container file system is time consuming. It is also essentially redundant since the `npm install` instruction in the Dockerfile that generates the image for the container already will create this `node_modules` folder

b) To execute `npm install` to create a `node_modules` folder in the project root folder on our host file system requires us to install Node.js on our host system. However, the version of Node we install might be a different version from the one used in the container (as specified in the Dockerfile that generates its image: `FROM node:14`). Hence the dependencies in our `node_modules` in our host system may be different from the one required for the application to run correctly. We may also have a different version of Node installed on the host system for local work there, and we may not wish to uninstall this version to simply install the version required by the local container. This also violates the key principle or advantage of Docker: which is to allow us to use tools and languages in a portable and standard manner by isolating / encapsulating the specific version of required tools / languages into the Dockerfile image, so that we can reproduce the exact same container regardless of which platform it runs on.

Two potential solutions that are more practical:

a) Specify the bind mount so that it only affects the directories that we are interested in, rather than the entire project root folder. For e.g. we might only be interested in having live updates to the HTML files in `/pages` propagated to the same folder in `/app` in the container which it is running.

b) Use an anonymous volume to exclude the `node_modules` folder from being affected by the bind mounting. This is a classic use case of the anonymous volume (which as we have seen previously is not useful for persisting data between container deletions and generation).

Let's try the first approach (approach a)

```
docker run -p 3000:80 -d --name feedback-app -v feedback:/app/feedback -v ${pwd}/pages:/app/pages feedback-node:volumes
```

This time the container stays up and running and you are able to interact with it in the usual manner at `localhost:3000`.

Now make a random modification to `pages/feedback.html` on your local project folder which determines what is shown on the main page of this webapp. For e.g. change the main <h1> or <h2> header.

```
.....

  <body>
    <header>
      <h1><a href="/">My Cool Site</a></h1>
    </header>
    <main>
      <section>
        <h2>Your Feedback is required</h2>
        <form action="/create" method="POST">
          <div class="form-control">
......
```

Refresh the webpage at `localhost:3000`, which sends new HTTP GET request to the running server.js app in the container, which responds by serving back the latest modified version of `feedback.html`. You should therefore be able to see your latest changes take effect in real time.

Now create an interactive shell into the container and check that the `node_modules` is in fact now present in the `/app` folder. Also inspect whether the change you have made to `feedback.html` is actually updated live to the same corresponding file in the `/app/pages` folder in the container file system (which is the entire purpose of the bind mount):

```
docker exec -it feedback-app /bin/bash
```

```
root@beec5ece23dc:/app# ls -l
total 12
-rwxrwxrwx  1 root root  132 Aug  3 09:13 Dockerfile
drwxr-xr-x  2 root root 4096 Aug  3 23:05 feedback
drwxr-xr-x 66 root root 4096 Aug  3 23:05 node_modules
-rwxrwxrwx  1 root root  260 Oct  7  2020 package.json
drwxrwxrwx  1 root root  512 Aug  3 08:08 pages
drwxrwxrwx  1 root root  512 Aug  3 08:04 public
-rwxrwxrwx  1 root root 1204 Aug  3 08:57 server.js
drwxrwxrwx  1 root root  512 Aug  3 23:05 temp
root@4e68cc25220c:/app# cd pages
root@4e68cc25220c:/app/pages# ls -l
total 5
-rwxrwxrwx 1 root root 531 Oct  7  2020 exists.html
-rwxrwxrwx 1 root root 857 Aug  3 22:55 feedback.html
root@4e68cc25220c:/app/pages# cat feedback.html

…
…
…
…
```

While you are still in the interactive shell, continue to make more live changes to `pages/feedback.html` on your local project folder, and verify that these changes are immediately updated to this file in the container file system by checking it with: `cat feedback.html`

When you are satisfied, exit from the interactive shell and remove the app again with:

```
docker rm -f feedback-app
```

This first approach is possible if there are only a few folders whose content we want to update live from our host file system to the container file system. However, if you have many folders in your project root folder that you want to update dynamically in this way, you will have to specify many bind mount options and your `docker run` command becomes very long and complicated.

In that case, we can try the second approach (approach b)

Here we specify an anonymous volume to exclude the `node_modules` folder from being affected by the bind mounting

Start the container again with:

```
docker run -p 3000:80 -d --name feedback-app -v feedback:/app/feedback -v "complete path to root folder:/app" -v /app/node_modules feedback-node:volumes
```

or

```
docker run -p 3000:80 -d --name feedback-app -v feedback:/app/feedback -v ${pwd}:/app -v /app/node_modules feedback-node:volumes
```

This time, the container stays up and running and you will be able to access the app at localhost:3000.

Once again, make a random modification to `pages/feedback.html` on your local project folder which determines what is shown on the main page of this webapp. For e.g. change the main <h1> or <h2> header, save it and refresh the web page. The changes you made will be updated in the returned HTML file.

Again, create an interactive shell into the container and check that the `node_modules` is in fact now present in the `/app` folder. Also inspect whether the change you have made to `feedback.html` is actually updated live to the same corresponding file in the `/app/pages` folder in the container file system (which is the entire purpose of the bind mount):

```
docker exec -it feedback-app /bin/bash
```

```
root@beec5ece23dc:/app# ls -l
total 12
-rwxrwxrwx  1 root root  132 Aug  3 09:13 Dockerfile
drwxr-xr-x  2 root root 4096 Aug  3 23:05 feedback
drwxr-xr-x 66 root root 4096 Aug  3 23:05 node_modules
-rwxrwxrwx  1 root root  260 Oct  7  2020 package.json
drwxrwxrwx  1 root root  512 Aug  3 08:08 pages
drwxrwxrwx  1 root root  512 Aug  3 08:04 public
-rwxrwxrwx  1 root root 1204 Aug  3 08:57 server.js
drwxrwxrwx  1 root root  512 Aug  3 23:05 temp
root@4e68cc25220c:/app# cd pages
root@4e68cc25220c:/app/pages# ls -l
total 5
-rwxrwxrwx 1 root root 531 Oct  7  2020 exists.html
-rwxrwxrwx 1 root root 857 Aug  3 22:55 feedback.html
root@4e68cc25220c:/app/pages# cat feedback.html

…
…
…
…
```

While you are still in the interactive shell, continue to make more live changes to `pages/feedback.html` on your local project folder, and verify that these changes are immediately updated to this file in the container file system by checking it with: `cat feedback.html`

A few more things to note related to the bind mount and volumes:

The app is actively saving titles and their content into files in the `/app/feedback` folder. You can navigate here and verify this for yourself

```
root@beec5ece23dc:/app/pages# cd /app/feedback
root@beec5ece23dc:/app/feedback# ls -l
total 24
-rw-r--r-- 1 root root  6 Aug  3 23:08 awesome.txt
..
…
root@beec5ece23dc:/app/feedback# cat awesome.txt
its cool
@beec5ece23dc:/app/feedback#
```

However, on the corresponding `/feedback` folder in the root project folder your host file system, there are no files even though the bind mount maps this folder to the `/app/feedback` folder in the container file system. This is because we have a named volume mapping for this folder in the container file system in the docker run command that we used (`-v feedback:/app/feedback`), which means it is also excluded from the bind mount operation (`-v ${pwd}:/app`)

However, if you navigate back to the main `/app` folder (which is linked by the bind mount to the project root folder in the host file system) and create a file there dynamically, it will automatically be copied over. Try this out for yourself in the interactive shell in the container:

```
root@beec5ece23dc:/app/feedback# cd /app
root@beec5ece23dc:/app# echo "First try" > try1.txt
root@beec5ece23dc:/app# echo "Second try" > try2.txt
root@beec5ece23dc:/app# ls -l
total 12
-rwxrwxrwx  1 root root  132 Aug  3 09:13 Dockerfile
drwxr-xr-x  2 root root 4096 Aug  3 23:09 feedback
drwxr-xr-x 66 root root 4096 Aug  3 23:05 node_modules
-rwxrwxrwx  1 root root  260 Oct  7  2020 package.json
drwxrwxrwx  1 root root  512 Aug  3 08:08 pages
drwxrwxrwx  1 root root  512 Aug  3 08:04 public
-rwxrwxrwx  1 root root 1204 Aug  3 08:57 server.js
drwxrwxrwx  1 root root  512 Aug  3 23:09 temp
-rw-r--r--  1 root root   10 Aug  3 23:22 try1.txt
-rw-r--r--  1 root root   11 Aug  3 23:23 try2.txt
```

Check now that in your host file system in the root project folder, there are now the two files `try1.txt` and `try2.txt` with that same content.

Now delete those two files in your root project folder. There will now also be correspondingly deleted in the container file system.

```
root@beec5ece23dc:/app# ls -l
total 12
-rwxrwxrwx  1 root root  132 Aug  3 09:13 Dockerfile
drwxr-xr-x  2 root root 4096 Aug  3 23:09 feedback
drwxr-xr-x 66 root root 4096 Aug  3 23:05 node_modules
-rwxrwxrwx  1 root root  260 Oct  7  2020 package.json
drwxrwxrwx  1 root root  512 Aug  3 08:08 pages
drwxrwxrwx  1 root root  512 Aug  3 08:04 public
-rwxrwxrwx  1 root root 1204 Aug  3 08:57 server.js
drwxrwxrwx  1 root root  512 Aug  3 23:09 temp
```

The code logic of the application (`server.js`) is as follows:

a) The title of the feedback is used to create a file with the same name and extension .txt and the document text for the title is placed as content of this file.
b) This file is initially placed in the `temp` folder.
c) If there is no existing file with the same name in the `feedback` folder, this file is copied over to the `feedback` folder (where it is persisted to the named volume through the mapping that we provide)
d) If however there is already a file with the same name in the `feedback` folder, an error message is provided on the webpage and the file is maintained in the `temp` folder.

This means that if you attempt to enter a title that already exists (i.e. a file that already exists in the feedback folder), the file that corresponds to this title will be maintained in the `temp` folder. Since this folder is also a subfolder of the `/app` folder which is included in the main bind mounting and it is not part of any other named or anonymous volume, all the files here will be copied over to the `temp` folder in the root project folder on the host file system.

Test this out for yourself.

When you are satisfied, exit from the interactive shell and remove the app again with:

```
docker rm -f feedback-app
```

To summarize:

Named volumes are mapped to a specific path in the container file system and will retain all files created in that path between container removal and regeneration. They are therefore the first choice for persisting data long term. Named volumes can be mapped to different containers when they start, whereby they will transfer all their contents to the mapped path in that container. They are managed directly by Docker via the docker volume command.

Bind mounts dynamically map a specific path on the host file system to another path on the container file system, so that changes made on either file system are propagated to the other dynamically in real time while the container is running. Bind mounts are specified when the container is started and do not exist when the container is deleted (unlike named volumes). They are most frequently use to reflect dynamic source code changes to a running container.

Anonymous volumes are always newly generated when a container is recreated from an image. Since anonymous volumes are not the same between container removal and generation, they are not useful for persisting data (unlike named volumes). By default, anonymous Volumes are removed if the container was started with the --rm option and was stopped thereafter. They are not removed if a container was started (and then removed) without that option.

Container paths mapped to an anonymous volume or named volume are excluded from a bind mount mapping. This provides the classic use case for a anonymous volume - to prevent changes in a host file system from overriding a path in the container file system.

In other words, a bind mount operation will affect all directories and subdirectories THAT ARE NOT matched to either a named or anonymous volume. This is an important principle to keep in mind when you are using bind mounts, named and anonymous volumes together at the same time (as we are doing here).

5.3 Adding a hot reload server monitor

Previously, any live changes that we made to any of the HTML files in `/pages` will show up when we refresh the web page as the `server.js` app that is running simply serves back the latest modified HTML file in response to the HTTP GET request.

However, changes that we make to `server.js` (the main server app) will not get reflected in real time as this app is already part of a process being run by the Node runtime.

To verify this, start the container again with the usual options for anonymous, named and bind mounts.

```
docker run -p 3000:80 -d --name feedback-app -v feedback:/app/feedback -v ${pwd}:/app -v /app/node_modules feedback-node:volumes
```

Add some additional random console output statements into `server.js` at the function that handles page refresh (i.e. HTTP GET request to the root domain /). For e.g.

`server.js`

```
……..

app.get('/', (req, res) => {
  console.log("Loaded main page");

  console.log ("Added some new stuff here");

  const filePath = path.join(__dirname, 'pages', 'feedback.html');
  res.sendFile(filePath);
});

……….
```

Refresh the web page several times. If you check the current log output from the container:

```
docker logs feedback-app
```

you will only see evidence of output from the existing console output statements. Your newly introduced console output statement does not execute.

This is because the Node runtime needs to be restarted again with the main application (`node server.js`) to pick up these latest changes to the source code files. We could now explicitly remove the container and restart it again to accomplish this, but a faster solution is to use a hot reload server monitor.

Remove the container again:

```
docker rm -f feedback-app
```

We can start the application (`server.js`) with a special application (rather than the Node runtime directly), which will monitor the source code of the app and then automatically reload the app whenever it detects a change in the source code. The application we are going to use for this purpose is Nodemon

https://www.digitalocean.com/community/tutorials/workflow-nodemon

There are equivalent tools to this in other programming languages / web frameworks.

To use Nodemon, we add a dependency for it to `package.json` in our root project folder and also have a script to the start the server with nodemon.

`package.json`

```
  "license": "ISC",
  "scripts": {
    "start": "nodemon -L server.js"
  },

  "dependencies": {
    "body-parser": "^1.19.0",
    "express": "^4.17.1"
  },
  "devDependencies": {
    "nodemon": "2.0.4"
  }
```

Then we modify our Dockerfile to use this script instead of starting with the node run time.

```
FROM node:14

WORKDIR /app

COPY package.json /app

#include this if you get certificate related errors
#RUN npm config set strict-ssl false

RUN npm install

COPY . /app

EXPOSE 80

CMD [ "npm", "start" ]
```

Rebuild the image again:

```
docker build -t feedback-node:volumes .
```

One useful command that you can use to inspect the sequence of instructions in the Dockerfile used to generate a particular image is:

```
docker image history feedback-node:volumes
```

Finally, bring up the container again with:

```
docker run -p 3000:80 -d --name feedback-app -v feedback:/app/feedback -v "complete path to root folder:/app" -v /app/node_modules feedback-node:volumes
```

or

```
docker run -p 3000:80 -d --name feedback-app -v feedback:/app/feedback -v ${pwd}:/app -v /app/node_modules feedback-node:volumes
```

Refresh the app and check the log output from the container:

```
docker logs feedback-app
```

You should see the latest console output statement that you had added.
Continue to add more random statements here and save.

```
app.get('/', (req, res) => {
  console.log("Loaded main page");

  console.log ("Added some new stuff here");

  console.log("More new stuff");

  console.log("And more stuff");

  const filePath = path.join(__dirname, 'pages', 'feedback.html');
  res.sendFile(filePath);
});
```

Refresh the page and check the logs again.

```
docker logs feedback-app
```

You will continue to see all the latest console output statement that you had added, as well as messages indicating that the server.js was restarted by nodemon as a result of it picking up changes in that file.

```
[nodemon] 2.0.4
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node server.js`
[nodemon] restarting due to changes...
[nodemon] starting `node server.js`
```

There is an equivalent approach for monitoring of a Python script (which can be done using Nodemon as well as other native Python libraries).

https://stackoverflow.com/questions/49355010/how-do-i-watch-python-source-code-files-and-restart-when-i-save

https://pypi.org/project/py-mon/

Nodemon can also be used to monitor live changes to a Java application:

https://medium.com/@zaynjarvis/power-up-java-with-nodemon-b4a4d50ec507

5.4 Managing multiple volumes with a container app

Check all the volumes that you have at this point of time:

```
docker volume ls
```

You will be able to see a few anonymous volumes (corresponding to the time we started the container without the `--rm` flag) and also the named volume feedback. We will not see any info pertaining to the bind mount (`-v ${pwd}:/app`) appear here since the bind mount is not managed by Docker and only exists while the container is started and does not persist afterwards (unlike named and anonymous volumes).

We can create a named volume before hand which we can subsequently use with our container:

```
docker volume create second-feedback
```

Check that it is created properly with:

```
docker volume ls
```

Now we used this new named volume to replace the one that we specified in our original run command (`-v feedback:/app/feedback`).

Before replacing, verify what files have already been saved in the current `feedback` named volume from the `/app/feedback` folder by checking through an interactive shell in the container.

```
docker exec -it feedback-app /bin/bash
```

```
root@4a6434d1c62c:/app# cd feedback
root@4a6434d1c62c:/app/feedback# ls -l
total 12
…
…
root@4a6434d1c62c:/app/feedback# exit
```

First remove the running container if you have not already done so with:

```
docker rm -f feedback-app
```

Then restart it again with the newly created named volume

```
docker run -p 3000:80 -d --name feedback-app -v second-feedback:/app/feedback -v ${pwd}:/app -v /app/node_modules feedback-node:volumes
```

Since we are mounting a completely new volume which maps to the `/app/feedback` folder, this will be empty at the start. Verify this by again establishing an interactive shell into the running container:

```
docker exec -it feedback-app /bin/bash
```

```
root@4a6434d1c62c:/app# cd feedback
root@4a6434d1c62c:/app/feedback# ls -l
total 12
…
…
root@4a6434d1c62c:/app/feedback# exit
```

We can now continue to interact with the app in the usual way, and all the new files stored in `/app/feedback` will again be transferred to this new volume.

Now stop and remove this container:

```
docker rm -f feedback-app
```

We can now start it back with the mapping to the original volume `feedback`.

```
docker run -p 3000:80 -d --name feedback-app -v feedback:/app/feedback -v ${pwd}:/app -v /app/node_modules feedback-node:volumes
```

And again, we can establish an interactive shell into this container to verify that we have all the original files we saved into this volume from `/app/feedback`

```
docker exec -it feedback-app /bin/bash
```

```
root@4a6434d1c62c:/app# cd feedback
root@4a6434d1c62c:/app/feedback# ls -l
total 12
…
…
root@4a6434d1c62c:/app/feedback# exit
```

Now stop and remove this container:

```
docker rm -f feedback-app
```

As we have seen earlier, you can inspect the contents of both of these two named volumes by using a temporary container based on a light weight Linux base image (such as `alpine` or `busybox`).

Now run a temporary container (using a `--rm` flag to automatically remove the container when we exit) and mount the `feedback` volume at any convenient path in the file system of that container (Docker will create that path if it does not exist):

```
docker run --rm -it -v feedback:/mnt alpine /bin/sh
```

```
/ # pwd
/
/ # cd /mnt
/mnt # ls -l
total 8
…
…
..
…

/mnt # cat ???.txt
…..
/mnt # exit
```

Repeat this operation to inspect the contents of the `second-feedback` volume:

```
docker run --rm -it -v second-feedback:/mnt alpine /bin/sh
```

```
/ # cd /mnt
/mnt # ls -l
….
….
….
```

We may also wish to copy the files in a named volume to the host file system for more permanent backup. To do this, we can just start the Alpine container in a simpler manner to establish the mapping from the volume to an internal path:

```
docker run --name tempcontainer -v feedback:/mnt alpine
```

Identify or create a suitable empty directory on the host file system for this purpose, and then copy the `/mnt` folder within this temporary container to this empty directory with this command:

```
docker cp tempcontainer:/mnt /path/to/emptydirectory
```

If you are using Windows, make sure to encode the path with double quotes, i.e.

```
docker cp tempcontainer:/mnt "/path/to/emptydirectory"
```

To remove all unused anonymous volumes that are not associated with a running container:

```
docker volume prune
```

```
docker volume ls
```

You should now only see the 2 named volumes that we had previously created:

Finally, you can (if you wish), remove a named volume with:

```
docker volume rm second-feedback
```

6 Environment Variables and Build arguments

Docker supports build-time arguments and runtime environment variables which can also be set as options in the Dockerfile and also within the docker run command. This allows you to customize certain variables which are likely to change between different environments that a Docker container is likely to run in or interact with. Examples might be port numbers, database credentials, etc

Build-time arguments and environment variables allow the creation of more flexible images and containers because you don't have to hard-code everything into the Dockerfile for an image. Instead, you can set it dynamically when you build an image or even only when you run a container.

Remove the previous container if it is still running:

```
docker rm -f feedback-app
```

Modify `server.js` so that the port number is provided as an environment variable.

```
app.listen(process.env.PORT);
```

Change the Dockerfile to utilize this environment variable

```
FROM node:14

WORKDIR /app

COPY package.json /app

#include this if you get certificate related errors
#RUN npm config set strict-ssl false

RUN npm install

COPY . /app

ENV PORT 80

EXPOSE $PORT

CMD [ "npm", "start" ]
```

Build a new image from this modified Dockerfile:

```
docker build -t feedback-node:env .
```

Now run the container using this new image using the previous command we had used before just to check that it still works with the mapping using the original port of 80:

```
docker run -p 3000:80 -d --name feedback-app -v feedback:/app/feedback -v ${pwd}:/app -v /app/node_modules feedback-node:env
```

Verify that it is still running as normal by checking the webpage:

Remove it:

```
docker rm -f feedback-app
```

Now we will create the container with an additional option to set the environment variable PORT. This will make the application `server.js` start and listen on a different port, so you will need to change the port mapping to the localhost port as well:

```
docker run -p 3000:8000 -d  --env PORT=8000 --name feedback-app -v feedback:/app/feedback -v ${pwd}:/app -v /app/node_modules feedback-node:env
```

We can establish an interactive shell into the container to verify that `server.js` is in fact listening on the new port 8000 as specified through the environment variable command line option `--env PORT=8000`

```
docker exec -it feedback-app /bin/bash
```

```
root@86c47681613a:/app# ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.2  0.0 686852 44348 ?        Ssl  00:56   0:00 npm
root        18  0.0  0.0   2388   768 ?        S    00:56   0:00 sh -c nodemon -L server.js
root        19  3.1  0.0 881632 41256 ?       Sl   00:56   0:03 node /app/node_modules/.bin/node
root        32  0.1  0.0 587252 41208 ?        Sl   00:56   0:00 /usr/local/bin/node server.js
root        39  0.0  0.0   3864  3272 pts/0    Ss   00:56   0:00 /bin/bash
root        45  0.0  0.0   7636  2748 pts/0    R+   00:58   0:00 ps aux
root@86c47681613a:/app# ss -tupln
Netid     State      Recv-Q     Send-Q         Local Address:Port          Peer Address:Port
tcp       LISTEN      0          511                        *:8000                      *:*
users:(("node",pid=32,fd=19))
root@86c47681613a:/app# exit
```

You can specify multiple environment variable key value pairs as CLI options in the form of

```
--env key1=value1 --env key2=value2 --env key3=value3
```

If you have many environment variables to declare, it might be easier to place them in a separate dedicated file (typically called `.env`) in a suitable folder of your project. Here we can place the file in the root folder.

`.env`

```
PORT=8000
```

Then when running the container, you can reference that file instead of explicitly specifying the environment variable key value pairs:

Stop and remove the container first:

```
docker rm -f feedback-app
```

Now restart it with an option to reference that environment variable file:

```
docker run -p 3000:8000 -d  --env-file ./.env --name feedback-app -v
feedback:/app/feedback -v ${pwd}:/app -v /app/node_modules feedback-
node:env
```

The app should start and work as usual.

Using an `.env` file makes it slightly easier to change environment variable values (which can be done directly in the file) rather than constantly changing the options in the command used to start the container.

Try it out with different values for the internal container ports, and make sure to also change the port mapping in the CLI: `-p 3000:`*`internal port`*

Stop and remove the container first:

```
docker rm -f feedback-app
```

Another typical use case for an environment variable file is to place security-sensitive info such as database authentication credentials or REST API authentication tokens. Hardcoding this info directly into the source code of the app or in the Dockerfile makes it more vulnerable to exploitation: since the source code of the app or the Dockerfile are usually publicly available in a cloud based Git repo account. We can keep this sensitive information separately in a local `.env` file which is confidential and never shared publicly with anyone, and then use this `.env` file to set the environment variables when starting the container.

Build-time arguments are very similar conceptually to environment variables. They extend the flexibility of environment variables by allowing you to customize different images with different default options (which is not possible with environment variables, where the default value has to be hardcoded into the Dockerfile).

Update Dockerfile to use a build time argument to specify the port instead of an environment variable

```
FROM node:14

WORKDIR /app

COPY package.json /app

#include this if you get certificate related errors
#RUN npm config set strict-ssl false

RUN npm install

COPY . /app
```

```
ARG DEFAULT_PORT=80

ENV PORT $DEFAULT_PORT

EXPOSE $PORT

CMD [ "npm", "start" ]
```

We can now construct 2 different images for use in 2 different environments:

We first build a new image for production purposes which uses the default port value of 80 specified in the Dockerfile:

```
docker build -t feedback-node:web-app .
```

Now, we can also build another image for development purposes where a different default port value is specified via the build-time argument:

```
docker build -t feedback-node:dev --build-arg DEFAULT_PORT=8000 .
```

Check that both images have been generated correctly with:

```
docker images
```

You can also verify that the default port in these two images are different by checking their history

```
docker image history feedback-node:web-app

docker image history feedback-node:dev
```

For the first image `feedback-node:web-app`, we can start a container from it with the appropriate port mapping for .e.g.

```
docker run -p 3000:80 -d --name feedback-app -v feedback:/app/feedback -v ${pwd}:/app -v /app/node_modules feedback-node:web-app
```

Verify that we can access the app at port 3000

Stop and remove the container:

```
docker rm -f feedback-app
```

Now start a container from the second image `feedback-node:dev`, again with an appropriate port mapping

```
docker run -p 3000:8000 -d --name feedback-app -v feedback:/app/feedback -v ${pwd}:/app -v /app/node_modules feedback-node:dev
```

Verify that we can access the app at port 3000

Stop and remove the container:

```
docker rm -f feedback-app
```