

Docker Workshop

Lab 4

Persistent storage with volumes and bind mounts

1	COMMANDS COVERED	1
2	LAB SETUP	1
3	TESTING THE APP LOCALLY	2
4	CREATING CUSTOM IMAGE FROM DOCKERFILE.....	3
5	NAMED VOLUMES	6
5.1	INSPECTING CONTENTS OF NAMED VOLUMES	7
5.2	CHANGING BETWEEN DIFFERENT VOLUMES WITH A CONTAINER	8
6	USING BIND MOUNTS WITH VOLUMES	10
6.1	SPECIFIC BIND MOUNT.....	11
6.2	ADDING A HOT RELOAD SERVER MONITOR	14
6.3	ISSUES WITH GENERIC BIND MOUNTS.....	17
6.4	BIND MOUNT EXCLUSION WITH ANONYMOUS VOLUMES.....	21
6.5	COMPARING NAMED VOLUMES, ANONYMOUS VOLUMES AND BIND MOUNTS	24
7	USING BIND MOUNTS WITH UTILITY CONTAINERS	25

1 Commands covered

2 Lab setup

This is identical to the previous lab

We will use Visual Studio Code to edit the source code for the various applications.

Ensure that the Docker extension for VS Code from Microsoft is installed as it facilitates the creation of Dockerfiles as well as managing your various Docker containers and images.

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.

Copy these folders to the folder that you initially created on your local machine for this workshop and work from them there.

3 Testing the app locally

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

If you have Node installed, you can run this Express.js application directly on your local machine.

Open a command prompt in the folder and type the following to download the appropriate JavaScript dependencies:

```
npm install
```

Notice that this creates the `node_modules` folder that contains all the dependencies required by this Express application, as well as `package-lock.json`

Then you can run the application with

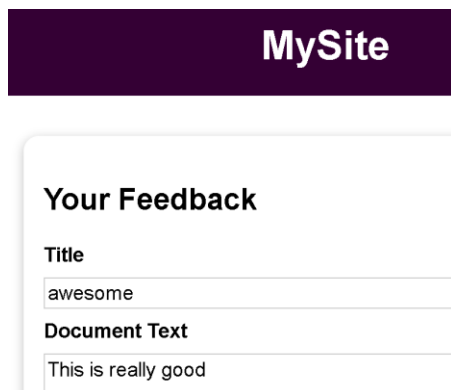
```
node server.js
```

Access the app on your browser at:

<http://localhost:3000>

enter some random feedback

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.

The screenshot shows a web application with a dark purple header bar containing the text "MySite" in white. Below the header is a white feedback form with a rounded top. The form has a title "Your Feedback" in bold. It contains two input fields: "Title" with the value "awesome" and "Document Text" with the value "This is really good".

MySite

Your Feedback

Title
awesome

Document Text
This is really good

You can subsequently retrieve the contents of the files saved so far at a specific API endpoint exposed by the app for this particular 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. It first stores in the content for these files 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.

You can check the root project folder to confirm the app behavior as explained above.

4 Creating custom image from Dockerfile

Before creating a Dockerfile to generate a custom image based on the contents of the current project root folder, we first delete all the files and directories that do not need to go into the custom image, which are:

- `node_modules`
- `feedback`
- `temp`
- `package-lock.json`

The remaining items in the project root folder which will be used during the Dockerfile build process are:

- `pages`
- `public`
- `server.js`
- `package.json`

Open the project root folder in VS Code and create the Dockerfile for building this project

Dockerfile

```
FROM node:lts-bookworm

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 3000

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

In a Powershell terminal in this particular directory, build an image from the Dockerfile with a custom tag:

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

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 and with the `--rm` option to delete the container after it stops. We will publish to the same localhost port as the container port (you can use a different port if you wish):

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

Check that the app is running at `localhost:3000` and enter some random feedback. The app should function exactly as it did previously when you ran it directly on your local machine.

For e.g. 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. 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 3000:3000 -d --name feedback-app feedback-node:volumes
```

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 an 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, verify that you are able to access the files corresponding to feedback titles that we had entered previously because the container was just stopped, not deleted.

In this case, 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`)

Data is **persisted between container restarts but is lost permanently when the container is removed.**

5 Named volumes

To persist data permanently after container removals, we will need to use **named volumes**, which must be **explicitly specified as an option at the CLI**.

Run the container with a new option `-v` to specify a named volume called `feedback` which maps to the `/app/feedback` directory in the container file system:

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

The option `-v` maps a new named volume called `feedback` to the directory `/app/feedback` on the container file system.

```
-v feedback:/app/feedback
```

After starting, check that the named volume exists with:

```
docker volume ls
```

You can inspect the volume as well with:

```
docker inspect volume feedback
```

If you have Docker Extension for VS Code installed, you should be able to view the mapping from the running container to the volume by hovering over the item with your mouse in the Containers section. Docker Desktop UI shows the volumes in a separate section, and you can check which container a volume is currently linked with.

The info returned should include the mountpoint, which is where the volume is stored in the underlying host system. This will be a directory in the [WSL2 file system](#) (`/var/lib/docker/volumes/feedback/_data`) that the Docker engine is running on, which in turn maps to different host file locations depending on whether you are using [Rancher Desktop](#) or [Docker Desktop](#).

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 after container deletion 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 3000:3000 -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: stop the container (thereby deleting it in `--rm` mode), 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. You can think of this named volume as conceptually similar to a virtual hard disk that is attached to the container.

When you are satisfied, remove the running container with:

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

5.1 Inspecting contents of named volumes

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. This could be for performance reasons, for e.g. the container might take a long time to start.

To do this, you could use a lightweight 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). Here we mount the `feedback` volume to the `/mnt` top level directory (the conventional directory path for external drives in a typical Linux file system).

```
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
```

5.2 Changing between different volumes with a container

We can create multiple volumes and swap between them when working with containers. This could be useful if we wish to persist data for different use cases or situations, but using the sample application. A simple example follows as below:

We can create an empty named volume beforehand 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 3000:3000 -d --name feedback-app -v second-feedback:/app/feedback 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. You can use different topic posts for different file names from the ones that you used earlier for `feedback`

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 3000:3000 -d --name feedback-app -v feedback:/app/feedback 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 lightweight 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"
```

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

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

6 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 be synchronized to 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 have **updates we make to our source code in the IDE 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 (synchronized) to 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.

Before demonstrating the use of a bind mount, start up a container using the previous image and named volume mapping again to verify that any changes you make to a source code file will not take effect immediately.

```
docker run -p 3000:3000 -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.

To see these latest changes take effect, we would have to rebuild the image using the same Dockerfile to incorporate these source code changes, and then start a brand new container with that image, as we have seen before.

Remove the container with:

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

6.1 Specific bind mount

We will now specify a bind mount so that it only involves 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.

To do this, run this command in a PowerShell terminal in the project root folder:

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

The bind mount is performed in this option

```
-v ${pwd}/pages:/app/pages
```

which binds the directory `${pwd}/pages` on the native host file system to the directory `/app/pages` on the container file system.

In PowerShell, the expression `${pwd}` is an automatic variable that holds the full path to the current working directory — similar to how `pwd` works as a command in Linux shells. We can use it here as shortcut if your PowerShell terminal is already in the project root folder.

Otherwise, you would need to explicitly specify the full complete path to the directory on the native host file system in the form as below:

```
-v "complete path to root folder:/pages:/app/pages"
```

If you have Docker extension for VS Code IDE or Docker Desktop installed, you should be able to see the bind mount mapping from `/pages` in the project root folder to `/app/pages` in the container, so this confirms bind mount mapping is effective.

Access the app in the container from your browser at:

<http://localhost:3000>

enter some random feedback and verify that the app is working as expected.

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 changes 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`

NOTE: If you have the Docker Extension for the VS Code IDE or Docker Desktop installed, you can also alternatively inspect the content of `/app/pages/feedback.html` in the running container, but this may not necessarily always reflect correctly the live mapping between the same file in the project folder on the native file system due to refresh issues in the IDE. However, if you download this file instead of viewing it, it should show the correct content.

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

```
docker volume ls
```

We will not see any info pertaining to the bind mount (`-v ${pwd}/pages:/app/pages`) 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 volumes).

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

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

6.2 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 named and bind mounts.

```
docker run -p 3000:3000 -d --name feedback-app -v
feedback:/app/feedback -v ${pwd}/pages:/app/pages 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");

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

.....
```

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

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 start the server with nodemon.

`package.json`

```
....  
  
  "author": "workshop",  
  "license": "ISC",  
  "scripts": {  
    "start": "nodemon -L server.js"  
  },  
  "devDependencies": {  
    "nodemon": "2.0.4"  
  },  
  "dependencies": {  
    "express": "^5.1.0"  
  }  
  
.....
```

Then we modify our Dockerfile to use this script to start with `nodemon` instead of starting with the standard Node runtime.

```
FROM node:lts-bookworm  
  
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 3000  
  
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 3000:3000 -d --name feedback-app -v  
feedback:/app/feedback -v ${pwd}/server.js:/app/server.js -v  
${pwd}/pages:/app/pages feedback-node:volumes
```

Notice now that we add another bind mount option

```
-v ${pwd}/server.js:/app/server.js
```

so that the contents of server.js in the project root folder is synchronized to the work directory in the container.

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");  
...  
...
```

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](#)).

Nodemon can also be used to [monitor live changes](#) to a Java application

When done, remove the container again:

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

6.3 Issues with generic bind mounts

In the previous approach, we specified explicit bind mounts for folders and files that we wanted to synchronize to the container file system, for e.g.

```
-v ${pwd}/server.js:/app/server.js
-v ${pwd}/pages:/app/pages
```

This approach is useful 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.

To simplify matters, we could attempt a bind mount of the entire project root folder (`${pwd}`) to the working directory (`/app`) in the container with:

```
docker run -p 3000:3000 -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 had expected it to.

```
docker ps -a
```

If you have Docker extension for VS Code IDE or Docker Desktop installed, you should be able to see the bind mount mapping from the project root folder to `/app` in the container, so this confirms bind mount mapping is effective.

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

```
docker logs feedback-app
```

```
> data-volume-example@1.0.0 start
> nodemon -L server.js

sh: 1: nodemon: not found
.....
```

All Node applications (for our case: Express.js and nodemon) 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 entire contents of the `/app` folder, thereby inadvertently deleting the `node_modules` folder there. This results in any Node applications (for our case: Express.js and nodemon) we using no longer being able to function.

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 just provided.

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 `npm start`, which attempts to run the script of the same name in `package.json` which subsequently crashes immediately when the required module is not found due to the missing `node_modules` folder.

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. This is something that you will do often practically in order to troubleshoot crashed containers that cannot start.

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
```

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.

If you have Node installed on your local machine, test this out by running in the project root folder:

```
npm install
```

Notice that this generates the `node_modules` folder as well as `package-lock.json`

Now repeat the previous attempt to bind mount the entire project root folder (`${pwd}`) to the working directory (`/app`) in the container with:

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

This time it should start as normal.

Access the app at:

```
localhost:3000
```

and verify that it is working properly now and you can enter a few random titles and associated content, and also ensure that you can retrieve their content with:

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

Just like before, make random modifications to `pages/feedback.html` on your local project folder which determines what is shown on the main page of this webapp. Refresh the webpage at `localhost:3000` 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, as before.

Now create an interactive shell into the container and verify that `node_modules` actually exists in the working directory: `/app`

```
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 23:09 feedback
drwxr-xr-x 66 root root 4096 Aug 3 23:05 node_modules
-rwxrwxrwx 1 root root 260 Oct 7 2020 package.json
....
....

```

While still in the main `/app` folder (which is linked by the bind mount to the project root folder in the host file system) create a file there dynamically, and it will automatically be synchronized to the local root project folder. Try this out for yourself in the interactive shell in the container:

```

root@beec5ece23dc:/app # 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 via the IDE, 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 from the IDE. They 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
...
...
...

```

One important point to keep in mind when using bind mount and named volumes in combination as we have done here.

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# 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 here even though the bind mount option maps this folder to the entire contents of the `/app` folder in the container file system.

```
-v ${pwd}:/app
```

The reason for this is that also have an existing named volume mapping for this folder as well when we started the container:

```
-v feedback:/app/feedback
```

When there is an existing **named volume mapping for any folder on the container file system, that folder will be automatically excluded from any bind mount operation**

6.4 Bind mount exclusion with anonymous volumes

In the previous example, we saw that if we wish to perform a bind mount of the entire project root folder (`${pwd}`) to the working directory (`/app`) in the container with:

```
-v ${pwd}:/app
```

We first need to ensure that all the folder dependencies that are required for the application source code in the container are present in the project root folder, and the way we initially solved it was to create the `node_modules` folder beforehand with `npm install`.

There are many problems with this approach:

- a) The number of dependencies in the `node_modules` folder is extremely large, and copying all of these from the host file system to the container file system as a result of the bind mount 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) We may also have a different version of Node installed on the host system for our local work, and we may not wish to uninstall this version to simply install the version required by the local container (as specified in the Dockerfile that generates its image: `FROM node:lts-bookworm`).
- c) Needing to install Node beforehand on our local machine in order to run **`npm install`** to create the `node_modules` folder 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**, without any need for any additional installations .

Therefore, to solve the original problem, which is to have the `node_modules` generated in the container file system from the Dockerfile not be overwritten by anything from the local host file system as a result of the bind mount, we can use the principle we saw earlier which is:

When there is an existing **named volume mapping for any folder on the container file system, that folder will be automatically excluded from any bind mount operation**

We can thus do a named volume mapping for `node_modules` folder on the container file system, which then solves our problem.

If the container is still running, remove it first with:

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

Next, delete the files and directories from the local project root folder that were generated from `npm install`:

- `node_modules`
- `package-lock.json`

Then start the container again with:

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

Here, we include a mapping from a randomly named volume to the `/app/node_modules` folder

```
-v dummy:/app/node_modules
```

This ensures that the folder is excluded from being affected / deleted by the generic bind mount operation

```
-v ${pwd}:/app
```

After starting, check on the 2 named volumes we have created so far (`dummy` and `feedback`) with:

```
docker volume ls
```

You can inspect the volume as well with:

```
docker inspect volume dummy
```

If you have Docker Extension for VS Code installed, you should be able to view the mapping from the running container to the volume by hovering over the item with your mouse in the Containers section. Docker Desktop UI shows the volumes in a separate section, and you can check which container a volume is current linked with.

Access the app in the container from your browser at:

<http://localhost:3000>

enter some random feedback and verify that the app is working as expected.

Here, we have given the named volume a random name (`dummy`) which really has not consequence, since we will never ever access or use this volume outside of its sole purpose of excluding the mapped local container folder from being synchronized in the bind mount operation.

A simpler alternative then is to use an anonymous volume, which functions similar to a named volume and accomplishes exactly the same objective except that its name is automatically assigned by Docker, rather than user defined.

Remove the running container with:

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

Start the container again with:

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

Access the app in the container from your browser at:

<http://localhost:3000>

and verify that it is working as usual.

The anonymous volume mapping option here is:

```
-v /app/node_modules
```

In this example, we combine the use of

- Named volume mapping (`-v feedback:/app/feedback`)
- Anonymous volume mapping (`-v /app/node_modules`)
- Bind mount (`-v ${pwd}:/app`)

In a single `docker run` command.

Now if you check on the existing volumes with:

```
docker volume ls
```

You should see a volume with a long random name (a hash value) which is the anonymous volume that was created in the mapping earlier.

Remove the running container with:

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

Now if you check again on existing volumes with:

```
docker volume ls
```

You will see that the anonymous volume has been automatically removed, unlike the named volumes we created earlier.

Start the container again with the same command:

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

Now if you check again on existing volumes with:

```
docker volume ls
```

You will see that a new anonymous volume with a different name (hash) has been created. The named volumes still stay the same.

You can remove the dummy named volume which you are no longer using any more with:

```
docker volume rm dummy
```

Check that it is removed with:

```
docker volume ls
```

Remove the running container with:

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

NOTE: You can [use the VOLUME instruction](#) in the Dockerfile as an alternative to create an anonymous volume to map to a specific folder. For e.g.

```
...
COPY . /app
VOLUME ["app/node_modules"]
EXPOSE 3000
....
```

When you generate a container from the custom file from building this particular Dockerfile, you won't need to specify the anonymous volume anymore yourself with:

```
-v /app/node_modules
```

6.5 Comparing named volumes, anonymous volumes and bind mounts

Named volumes have **names defined by a user** and are **mapped to a specific path in the container file system and will retain all files created in that path**. Named volumes **stay the same between container removal and regeneration** and 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.

Anonymous volumes are also mapped to a specific path in the container file system, but they are always **newly generated with a random name (hash)** assigned by Docker 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.

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 used to reflect dynamic source code changes** to a running container.

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

7 Using bind mounts with utility containers

One of the key use cases for Docker is to be able to run a container with a specific version of a programming language, library or tool and then access that language / library / tool without having to install it on your host machine. This is very helpful when you are working with many projects that require different versions of a particular language, library or tool in order to work properly.

Installing all of these different versions on your host machine and using them at the same time can increase the risk of conflicts and unintended consequences from failing to understand which particular version a piece of code or command is utilizing at any particular point in time.

The root folder for this project is: `utility`

One example of the above use case is running a particular source code file with different versions of a particular programming language. It is common that source code files from a certain project will only run on a specific version of that language.

For example, we can check which particular version of Python is required to run the code in `demo.py` by starting a container from that particular image and then attempting to execute `demo.py` with that version of Python. The use of the bind mount allows us to easily transfer our source code file (and other related files) into the container for quick testing without the need to perform a `docker cp` and then establish an interactive terminal with `docker exec -it` into the container.

For the file `demo.py`, we can quickly check its compatibility with different major versions of Python (3.12, 3.11, 3.10, 3.9) by running these Docker commands in the `utility` folder:

```
docker run --rm -v ${pwd}:/app python:3.12-alpine python /app/demo.py
docker run --rm -v ${pwd}:/app python:3.11-alpine python /app/demo.py
docker run --rm -v ${pwd}:/app python:3.10-alpine python /app/demo.py
docker run --rm -v ${pwd}:/app python:3.9-alpine python /app/demo.py
```

Here we use the lightweight Alpine image with the appropriate Python version installed. Notice that we did not name the container and also used the `--rm` flag since this is a one-off use container that we can dispose off immediately after it is completed and stops.

The example demonstrated here is equally applicable to other programming languages as well as different versions of different libraries / packages / frameworks in those languages. We can also specify these different images as base images in a Dockerfile, if we need a more container with more complex file system changes that must be generated from a custom image.

Another related example of this use case is the `grep` tool in Linux which is very popular and widely used too for performing complex searches for specific string patterns in a large text file. On Windows, to accomplish such complex searches, you would need to implement this custom search functionality in a program, or use an equivalent tool such as `Select-String` in Powershell. However, we can now just simply use a Linux container to access the `grep` tool, perform the search that we want and output the results in a file which we can access on our host machine

For example, to find all the sentences in `sample.txt` which start with the word `cats`, regardless of its case:

```
docker run --rm -v ${pwd}:/app alpine /bin/sh -c 'grep -i "^cats" /app/sample.txt > /app/results.txt'
```

When the container completes the execution of the `grep` command, the output is piped to `results.txt` which now becomes available on our local file system through the bind mount to the `/app` folder.

Similarly, to find all the sentences in `sample.txt` which end with the word `cats`, regardless of its case, and store the results in `results.txt`:

```
docker run --rm -v ${pwd}:/app alpine /bin/sh -c 'grep -i "cats$" /app/sample.txt > /app/results.txt'
```