

Tutorial: Create a Docker app with Visual Studio Code

Article • 04/12/2023

This tutorial is the beginning of a four-part series introducing [Docker](#) for use with Visual Studio Code (VS Code). You'll learn to create and run Docker containers, [persist data](#), and manage multiple containers with Docker Compose.

VS Code offers a Docker extension that lets you work with a local Docker Desktop service. Docker Desktop runs on your computer and manages your local containers, which are compact virtualized environments that provide a platform for building and running apps. Containers don't require the size and overhead of a complete operating system.

In this first tutorial, you learn how to:

- ✓ Create a Docker container.
- ✓ Build a container image.
- ✓ Start an app container.

Prerequisites

- [Visual Studio Code](#) installed.
- [Docker VS Code Extension](#) installed.
- [Docker Desktop](#) configured to use Linux containers.
- A [Docker Hub](#) account. You can create an account for free.

The tutorial works with Windows 10 or later and Docker Desktop configured to use Linux containers.

Create a container

A container is a process on your computer. It's isolated from all other processes on the host computer. That isolation uses kernel namespaces and control groups.

A container uses an isolated filesystem. This custom filesystem is provided by a *container image*. The image contains everything needed to run an application, such as all dependencies, configuration, scripts, and binaries. The image also contains other configuration for the container, such as environment variables, a default command to run, and other metadata.

After you install the Docker extension for VS Code, you can work with containers in VS Code. In addition to context menus in the Docker pane, you can select **Terminal > New Terminal** to open a command-line window. You can also run commands in a Bash window. Unless specified, any command labeled as **Bash** can run in a Bash window or the VS Code terminal.

1. Set Docker to Linux container mode. To switch to Linux containers if you are currently set to Windows containers, right-click on the Docker icon in the system tray while Docker Desktop is running, and choose **Switch to Linux containers**.
2. In VS Code, select **Terminal > New Terminal**.
3. In the terminal window or a Bash window, run this command.

Bash

```
docker run -d -p 80:80 docker/getting-started
```

This command contains the following parameters:

- `-d` Run the container in detached mode, in the background.
- `-p 80:80` Map port 80 of the host to port 80 in the container.
- `docker/getting-started` Specifies the image to use.

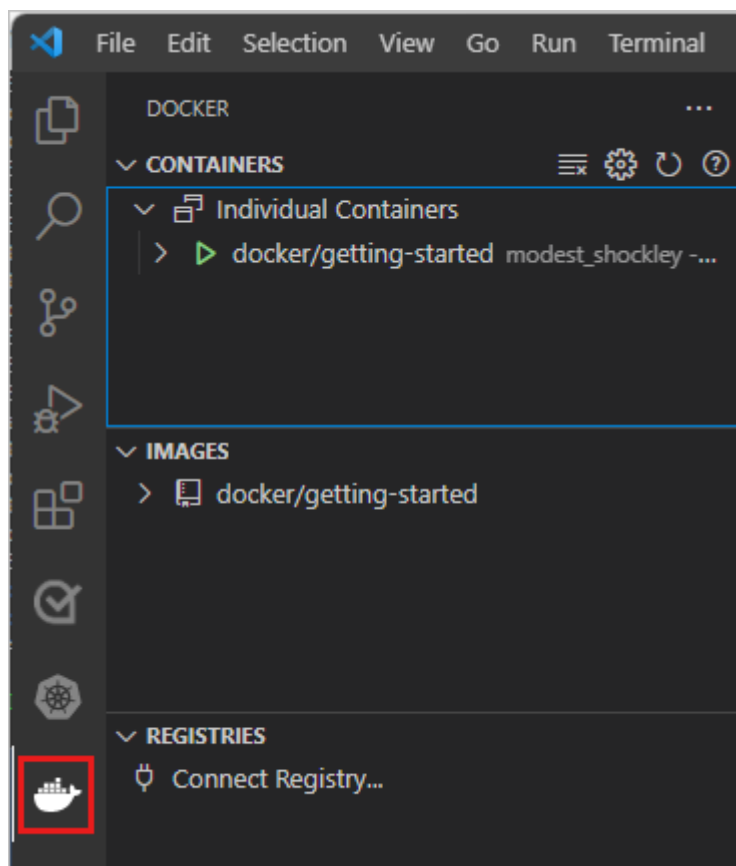
Tip

You can combine single character flags to shorten the full command. As an example, the command above could be written as:

Bash

```
docker run -dp 80:80 docker/getting-started
```

4. In VS Code, select the Docker icon on the left to view the Docker extension.



The Docker VS Code Extension shows you the containers running on your computer. You can access container logs and manage container lifecycle, such as stop and remove.

The container name, **modest_shockley** in this example, is randomly created. Yours will have a different name.

5. Right-click on **docker/getting-started** to open a context menu. Select **Open in Browser**.

Instead, open a browser and enter `http://localhost/tutorial/`.

You'll see a page, hosted locally, about DockerLabs.

6. Right-click on **docker/getting-started** to open a context menu. Select **Remove** to remove this container.

To remove a container by using the command line, run this command to get its container ID:

```
Bash
```

```
docker ps
```

Then stop and remove the container:

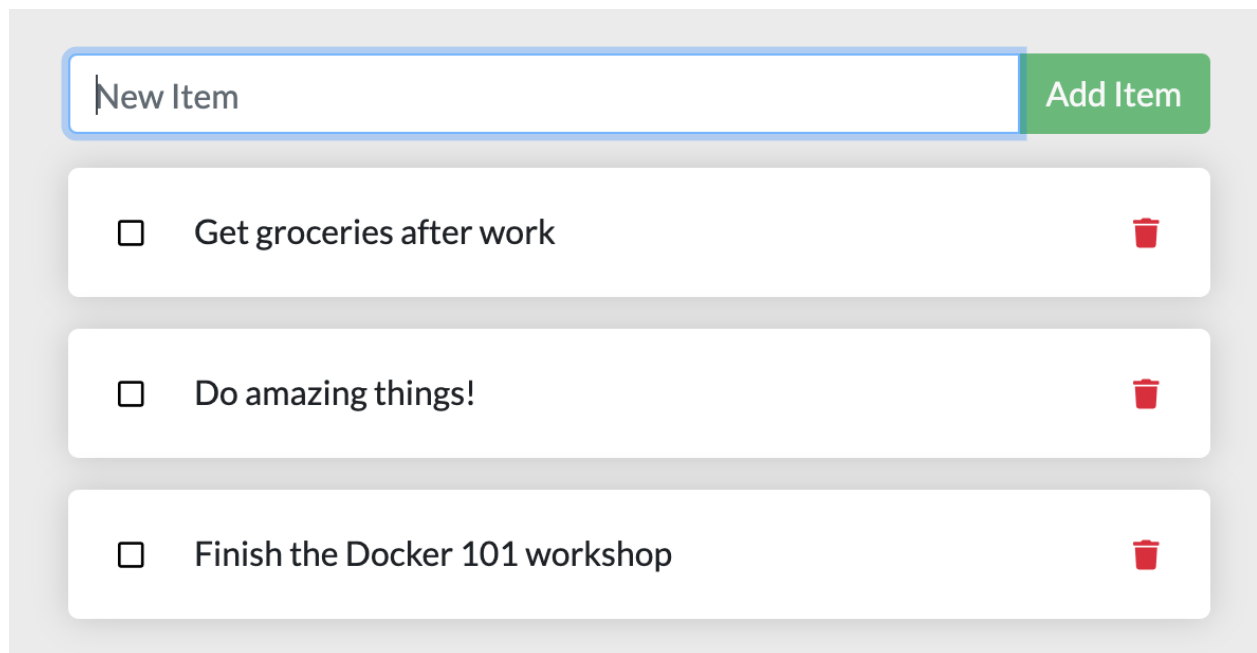
Bash

```
docker stop <container-id>  
docker rm <container-id>
```

7. Refresh your browser. The Getting Started page you saw a moment ago is gone.

Build a container image for the app

This tutorial uses a simple Todo application.

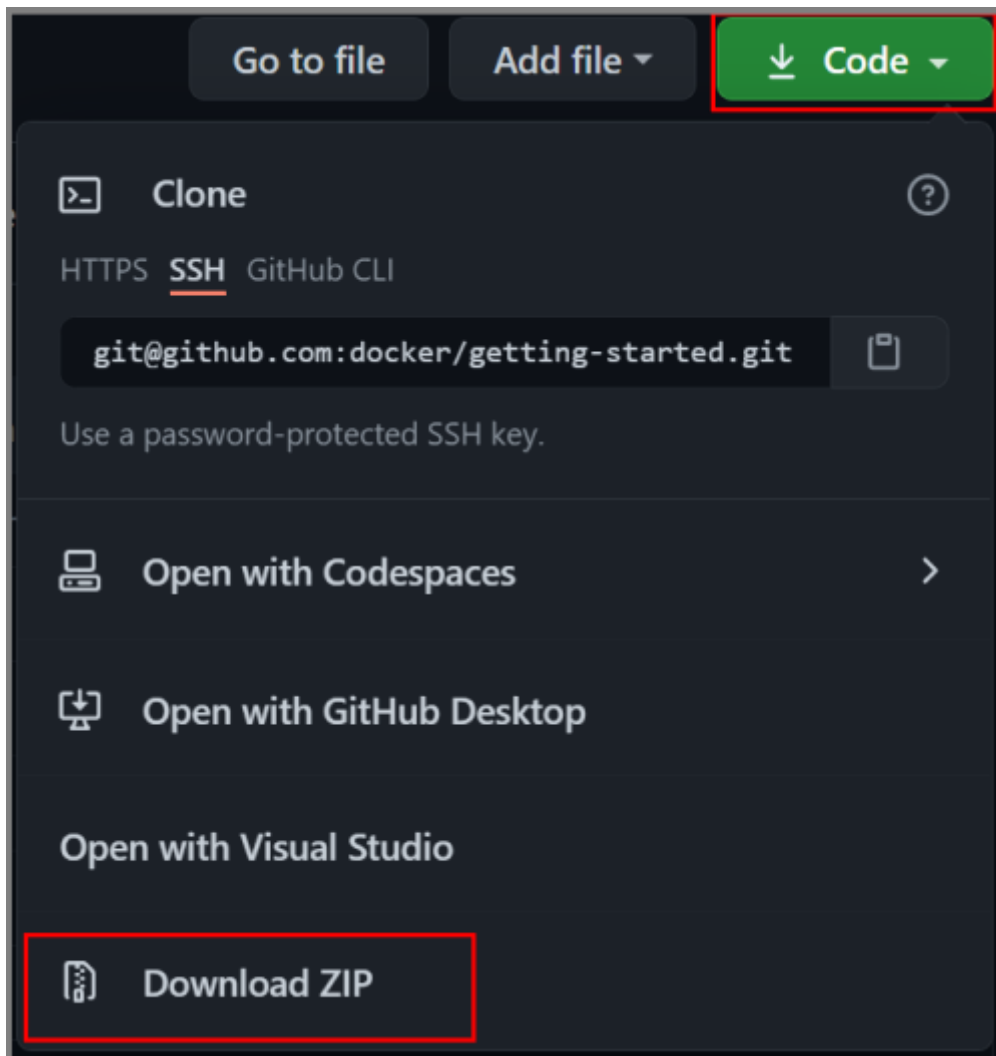
A screenshot of a web application interface for a todo list. At the top, there is a text input field with the placeholder text "New Item" and a green button labeled "Add Item". Below this, there is a list of three todo items, each in a white box with a light gray border. Each item has a checkbox on the left, the text of the item in the middle, and a red trash icon on the right. The items are: "Get groceries after work", "Do amazing things!", and "Finish the Docker 101 workshop".

Input	Action
New Item	Add Item
<input type="checkbox"/> Get groceries after work	
<input type="checkbox"/> Do amazing things!	
<input type="checkbox"/> Finish the Docker 101 workshop	

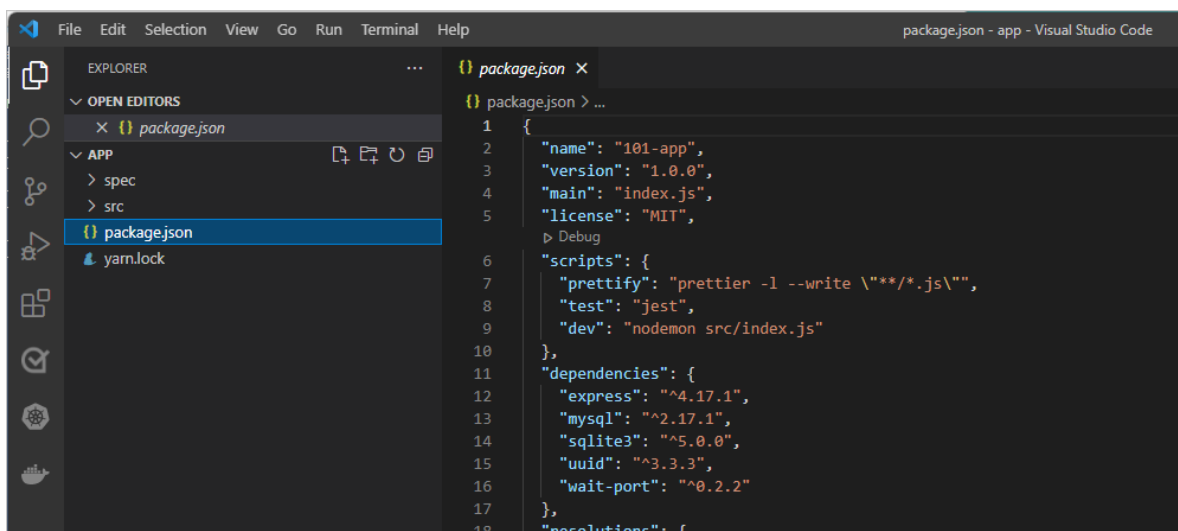
The app allows you to create work items and to mark them as completed or delete them.

In order to build the application, create a *Dockerfile*. A Dockerfile is a text-based script of instructions that is used to create a container image.

1. Go to the [Docker Getting Started Tutorial](#) repo, and then select **Code** > **Download ZIP**. Extract the contents to a local folder.



2. In VS Code, select **File > Open Folder**. Navigate to the *app* folder in the extracted project and open that folder. You should see a file called *package.json* and two folders called *src* and *spec*.



```
FROM node:20-alpine
RUN apk add --no-cache python3 g++ make
WORKDIR /app
COPY . .
RUN yarn install --production
CMD ["node", "/app/src/index.js"]
```

⚠ Note

Be sure that the file has no file extension like `.txt`.

4. In the file explorer, on the left in VS Code, right-click the *Dockerfile* and then select **Build Image**. Enter *getting-started* as the tag for the image in the text entry box.

The tag is a friendly name for the image.

To create a container image from the command line, use the following command.

Bash

```
docker build -t getting-started .
```

⚠ Note

In an external Bash window, go to the `app` folder that has the *Dockerfile* to run this command.

You've used the *Dockerfile* to build a new container image. You might have noticed that many "layers" were downloaded. The *Dockerfile* starts from the `node:20-alpine` image. Unless that image was on your computer already, that image needed to be downloaded.

After the image was downloaded, the *Dockerfile* copies your application and uses `yarn` to install your application's dependencies. The `CMD` value in the *Dockerfile* specifies the default command to run when starting a container from this image.

The `.` at the end of the `docker build` command tells that Docker should look for the *Dockerfile* in the current directory.

Start your app container

Now that you have an image, you can run the application.

1. To start your container, use the following command.

Bash

```
docker run -dp 3000:3000 getting-started
```

The `-d` parameter indicates that you're running the container in detached mode, in the background. The `-p` value creates a mapping between the host port 3000 and the container port 3000. Without the port mapping, you wouldn't be able to access the application.

2. After a few seconds, in VS Code, in the Docker area, under **CONTAINERS**, right-click **getting-started** and select **Open in Browser**. You can instead open your web browser to `http://localhost:3000`.

You should see the app running.

New Item

Add Item

No items yet! Add one above!

3. Add an item or two to test if it works as you expect. You can mark items as complete and remove items. Your frontend is successfully storing items in the backend.

Next steps

You've completed this tutorial and you have a running todo list manager with a few items. You've learned to create container images and run a containerized app.

Keep everything that you've done so far to continue this series of tutorials. Next, try part II of this series:

Update and share a Docker app

Here are some resources that might be useful to you:

- [Docker Cloud Integration](#) 
- [Examples](#) 

Tutorial: Share a Docker app with Visual Studio Code

Article • 06/13/2023

This tutorial is part two of a four-part series introducing [Docker](#) for use with Visual Studio Code (VS Code).

In this tutorial, you learn how to:

- ✓ Update the code and replace the container.
- ✓ Share your image.
- ✓ Run the image on a new instance.

Prerequisites

This tutorial continues the previous tutorial, [Create a Docker app with Visual Studio Code](#). To continue here, you'll need the running todo list manager from part 1.

Update the code and replace the container

Let's make a few changes and learn about managing your containers.

1. In the `src/static/js/app.js` file, update line 56 to use this new text label:

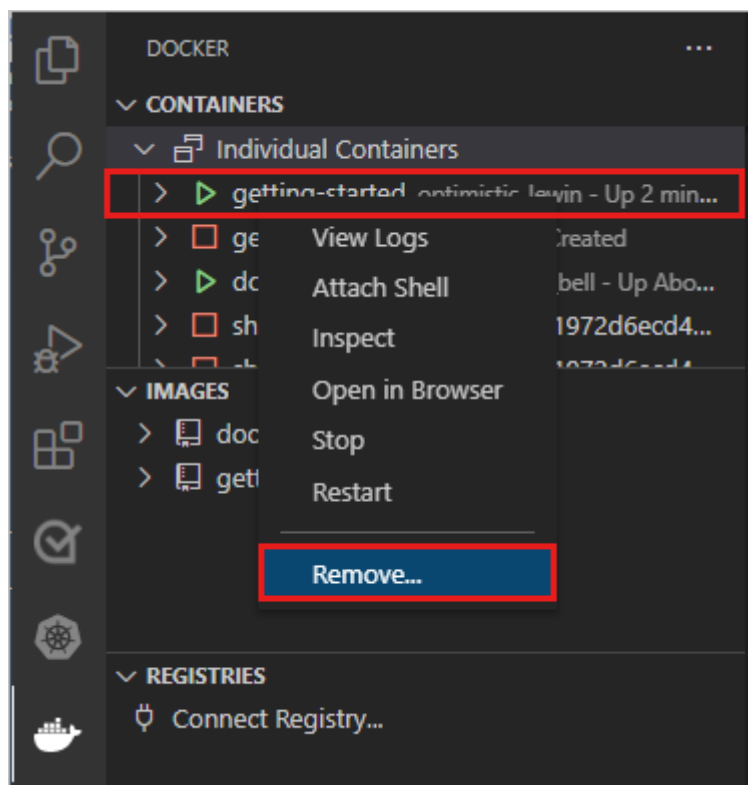
diff

```
- <p className="text-center">No items yet! Add one above!</p>  
+ <p className="text-center">You have no todo items yet! Add one above!  
</p>
```

Save your change.

2. Stop and remove the current version of the container. More than one container can't use the same port.

Right-click the **getting-started** container and select **Remove**.



Or, from the command line, use the following command to get the container ID.

```
Bash
```

```
docker ps
```

Then stop and remove the container:

```
Bash
```

```
docker stop <container-id>  
docker rm <container-id>
```

3. Build the updated version of the image. In the file explorer, right-click *Dockerfile*, then select **Build Image**.

Or, to build on the command line, change directory to the folder that contains the Dockerfile, and use the same command you used before.

```
Bash
```

```
docker build -t getting-started .
```

4. Start a new container that uses the updated code.

```
Bash
```

```
docker run -dp 3000:3000 getting-started
```

5. Refresh your browser on `http://localhost:3000` to see your updated help text.

Add Item

You have no todo items yet! Add one above!

Share your image

Now that you've built an image, you can share it. To share Docker images, use a Docker registry. The default registry is Docker Hub, which is where all of the images we've used have come from.

To push an image, first, you need to create a repo on Docker Hub.

1. Go to [Docker Hub](#) and sign in to your account.
2. Select **Create Repository**.
3. For the repo name, enter `getting-started`. Make sure that the **Visibility** is **Public**.
4. Select **Create**.

On the right of the page, you'll see a section named **Docker commands**. This section gives an example command to run to push to this repo.

Docker commands

Public View

To push a new tag to this repository,

```
docker push docker/getting-started:tagname
```

5. In VS Code, in the Docker view, under **REGISTRIES**, click the plug icon, to connect to a registry, and choose **Docker Hub**.

Enter your Docker Hub account name and password.

6. In the Docker view of VS Code, under **IMAGES**, right-click the image tag, and select **Push**. Enter the namespace and the tag, or accept the defaults.

7. To push to Docker Hub by using the command line, use this procedure.

Sign in to the Docker Hub:

Bash

```
docker login -u <username>
```

8. Use the following command to give the *getting-started* image a new name.

Bash

```
docker tag getting-started <username>/getting-started
```

9. Use the following command to push your container.

Bash

```
docker push <username>/getting-started
```

Run the image on a new instance

Now that your image has been built and pushed into a registry, try running the app on a brand new instance that has never seen this container image. To run your app, use Play with Docker.

1. Open your browser to [Play with Docker](#).
2. Sign in with your Docker Hub account.
3. Select **Start** and then select the + **ADD NEW INSTANCE** link in the left side bar.
After a few seconds, a terminal window opens in your browser.

03:56:49

CLOSE SESSION

Instances

+ ADD NEW INSTANCE

192.168.0.48
node1

br0lcrti_br0ldviosm4g00b718h0

IP
192.168.0.48

OPEN PORT

Memory
0.82% (32.98MiB / 3.906GiB)

CPU
0.54%

SSH
ssh ip172-18-0-38-br0lcrtim9m000a8ass0@direct.labs.plk

DELETE EDITOR

```
#####  
#                               WARNING!!!                               #  
# This is a sandbox environment. Using personal credentials             #  
# is HIGHLY! discouraged. Any consequences of doing so are              #  
# completely the user's responsibilities.                                #  
# The PWD team.                                                          #  
#####  
[node1] (local) root@192.168.0.48 ~  
$
```

4. In the terminal, start your app.

```
Bash
```

```
docker run -dp 3000:3000 <username>/getting-started
```

Play with Docker pulls down your image and starts it.

5. Select the **3000** badge, next to **OPEN PORT**. You should see the app with your modifications.

If the **3000** badge doesn't show up, select **OPEN PORT** and enter 3000.

Clean up resources

Keep everything that you've done so far to continue this series of tutorials.

Next steps

Congrats. You've completed part 2 and learned how to update your code and run your image on a new instance.

Here are some resources that might be useful to you:

- [Docker Cloud Integration](#)
- [Examples](#)

Next, try the next tutorial in this series:

Persist data and layer a Docker app

Tutorial: Persist data in a container app using volumes in VS Code

Article • 08/16/2023

In this tutorial, you'll learn to persist data in a container application. When you run it or update it, the data is still available. There are two main types of volumes used to persist data. This tutorial focuses on **named volumes**.

You'll also learn about *bind mounts*, which control the exact mountpoint on the host. You can use bind mounts to persist data, but it can also add more data into containers. When working on an application, you can use a bind mount to mount source code into the container to let it see code changes, respond, and let you see the changes right away.

This tutorial also introduces image layering, layer caching, and multi-stage builds.

In this tutorial, you learn how to:

- ✓ Understand data across containers.
- ✓ Persist data using named volumes.
- ✓ Use bind mounts.
- ✓ View image layer.
- ✓ Cache dependencies.
- ✓ Understand multi-stage builds.

Prerequisites

This tutorial continues the previous tutorial, [Create and share a Docker app with Visual Studio Code](#). Start with that one, which includes prerequisites.

Understand data across containers

In this section, you'll start two containers and create a file in each. The files created in one container aren't available in another.

1. Start a `ubuntu` container by using this command:

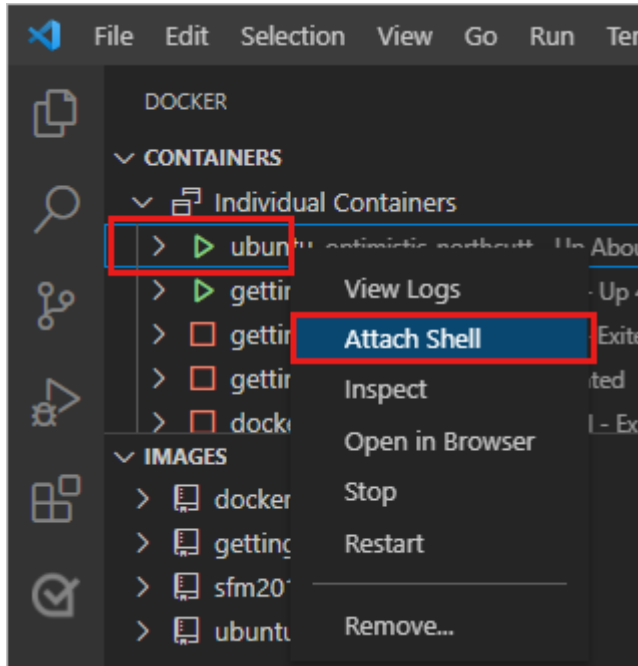
```
Bash
```

```
docker run -d ubuntu bash -c "shuf -i 1-10000 -n 1 -o /data.txt && tail
```

```
-f /dev/null"
```

This command starts invokes two commands by using `&&`. The first portion picks a single random number and writes it to `/data.txt`. The second command is watching a file to keep the container running.

2. In VS Code, in the **Docker** area, right-click the ubuntu container and select **Attach Shell**.



A terminal opens that is running a shell in the Ubuntu container.

3. Run the following command to see the contents of the `/data.txt` file.

```
Bash
```

```
cat /data.txt
```

The terminal shows a number between 1 and 10000.

To use the command line to see this result, get the container ID by using the `docker ps` command, and run the following command.

```
Bash
```

```
docker exec <container-id> cat /data.txt
```

4. Start another `ubuntu` container.

```
Bash
```

```
docker run -d ubuntu bash -c "shuf -i 1-10000 -n 1 -o /data.txt && tail -f /dev/null"
```

5. Use this command to look at the folder contents.

Bash

```
docker run -it ubuntu ls /
```

There should be no `data.txt` file there because it was written to the scratch space for only the first container.

6. Select these two Ubuntu containers. Right-click and select **Remove**. From the command line, you can remove them by using the `docker rm -f` command.

Persist your todo data using named volumes

By default, the todo app stores its data in a [SQLite Database](#) at `/etc/todos/todo.db`. SQLite Database is a relational database that stores data a single file. This approach works for small projects.

You can persist the single file on the host. When you make it available to the next container, the application can pick up where it left off. By creating a volume and attaching, or *mounting*, it to the folder that the data is stored in, you can persist the data. The container writes to the `todo.db` file and that data persists to the host in the volume.

For this section, use a **named volume**. Docker maintains the physical location the volume on the disk. Refer to the name of the volume, and Docker provides the right data.

1. Create a volume by using the `docker volume create` command.

Bash

```
docker volume create todo-db
```

2. Under **CONTAINERS**, select **getting-started** and right-click. Select **Stop** to stop the app container.

To stop the container from the command line, use the `docker stop` command.

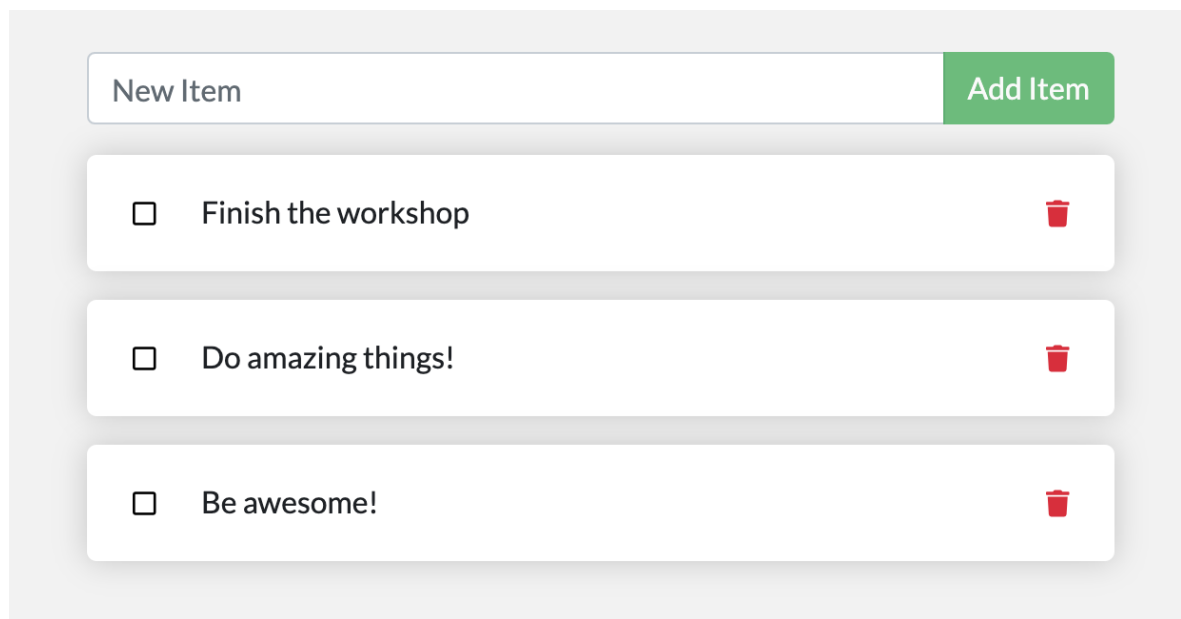
3. Start the **getting-started** container by using the following command.

Bash

```
docker run -dp 3000:3000 -v todo-db:/etc/todos getting-started
```

The volume parameter specifies the volume to mount and the location, `/etc/todos`.

4. Refresh your browser to reload the app. If you've closed the browser window, go to `http://localhost:3000/`. Add some items to your todo list.

A screenshot of a web application interface for a todo list. At the top, there is a text input field labeled "New Item" and a green button labeled "Add Item". Below this, there is a list of three todo items, each in a white box with a light gray border. The first item is "Finish the workshop" with an unchecked checkbox on the left and a red trash icon on the right. The second item is "Do amazing things!" with an unchecked checkbox on the left and a red trash icon on the right. The third item is "Be awesome!" with an unchecked checkbox on the left and a red trash icon on the right.

5. Remove the **getting-started** container for the todo app. Either right-click the container in the Docker area and select **Remove** or use the `docker stop` and `docker rm` commands.
6. Start a new container using the same command:

Bash

```
docker run -dp 3000:3000 -v todo-db:/etc/todos getting-started
```

This command mounts the same drive as before. Refresh your browser. The items you added are still in your list.

7. Remove the **getting-started** container again.

Named volumes and bind mounts, discussed below, are the main types of volumes supported by a default Docker engine installation.

Property	Named Volumes	Bind Mounts
Host Location	Docker chooses	You control
Mount Example (using <code>-v</code>)	my-volume:/usr/local/data	/path/to/data:/usr/local/data
Populates new volume with container contents	Yes	No
Supports Volume Drivers	Yes	No

There are many volume driver plugins available to support NFS, SFTP, NetApp, and more. These plugins are especially important to run containers on multiple hosts in a clustered environment such as Swarm or Kubernetes.

If you wonder where Docker *actually* stores your data, run the following command.

Bash

```
docker volume inspect todo-db
```

Look at the output, similar to this result.

Output

```
[
  {
    "CreatedAt": "2019-09-26T02:18:36Z",
    "Driver": "local",
    "Labels": {},
    "Mountpoint": "/var/lib/docker/volumes/todo-db/_data",
    "Name": "todo-db",
    "Options": {},
    "Scope": "local"
  }
]
```

The `Mountpoint` is the actual location where the data is stored. On most computers, you need root access to access this directory from the host.

Use bind mounts

With *bind mounts*, you control the exact mountpoint on the host. This approach persists data, but is often used to provide more data into containers. You can use a bind mount

to mount source code into the container to let it see code changes, respond, and let you see the changes right away.

To run your container to support a development workflow, you'll take the following steps:

1. Remove any `getting-started` containers.
2. In the `app` folder, run the following command.

Bash

```
docker run -dp 3000:3000 -w /app -v ${PWD}:/app node:20-alpine sh -c  
"yarn install && yarn run dev"
```

This command contains the following parameters.

- `-dp 3000:3000` Same as before. Run in detached mode and create a port mapping.
- `-w /app` Working directory inside the container.
- `-v ${PWD}:/app` Bind mount the current directory from the host in the container into the `/app` directory.
- `node:20-alpine` The image to use. This image is the base image for your app from the *Dockerfile*.
- `sh -c "yarn install && yarn run dev"` A command. It starts a shell using `sh` and runs `yarn install` to install all dependencies. Then it runs `yarn run dev`. If you look in the `package.json`, the `dev` script is starting `nodemon`.

3. You can watch the logs using `docker logs`.

Bash

```
docker logs -f <container-id>
```

Output

```
$ nodemon src/index.js  
[nodemon] 2.0.20  
[nodemon] to restart at any time, enter `rs`  
[nodemon] watching path(s): *.*  
[nodemon] watching extensions: js,mjs,json  
[nodemon] starting `node src/index.js`  
Using sqlite database at /etc/todos/todo.db  
Listening on port 3000
```

When you see the final entry on this list, the app is running.

When you're done watching the logs, select any key in the terminal window or select **Ctrl+C** in an external window.

4. In VS Code, open `src/static/js/app.js`. Change the text of the **Add Item** button on line 109.

diff

```
- {submitting ? 'Adding...' : 'Add Item'}  
+ {submitting ? 'Adding...' : 'Add'}
```

Save your change.

5. Refresh your browser. You should see the change.

New Item

Add

No items yet! Add one above!

View image layers

You can look at the layers that make up an image. Run the `docker image history` command to see the command that was used to create each layer within an image.

1. Use `docker image history` to see the layers in the *getting-started* image that you created earlier in the tutorial.

Bash

```
docker image history getting-started
```

Your result should resemble this output.

plaintext

IMAGE SIZE	CREATED COMMENT	CREATED BY
a78a40cbf866	18 seconds ago	/bin/sh -c #(nop) CMD ["node"]
"/app/src/ind... f1d1808565d6	0B 19 seconds ago	/bin/sh -c yarn install --
production	85.4MB	

```

a2c054d14948      36 seconds ago    /bin/sh -c #(nop) COPY
dir:5dc710ad87c789593... 198kB
9577ae713121      37 seconds ago    /bin/sh -c #(nop) WORKDIR /app
0B
b95baba1cfdb      13 days ago       /bin/sh -c #(nop)  CMD ["node"]
0B
<missing>         13 days ago       /bin/sh -c #(nop)  ENTRYPOINT
["docker-entry... 0B
<missing>         13 days ago       /bin/sh -c #(nop) COPY
file:238737301d473041... 116B
<missing>         13 days ago       /bin/sh -c apk add --no-cache -
-virtual .bui... 5.35MB
<missing>         13 days ago       /bin/sh -c #(nop)  ENV
YARN_VERSION=1.21.1 0B
<missing>         13 days ago       /bin/sh -c addgroup -g 1000
node && addu... 74.3MB
<missing>         13 days ago       /bin/sh -c #(nop)  ENV
NODE_VERSION=12.14.1 0B
<missing>         13 days ago       /bin/sh -c #(nop)  CMD
["/bin/sh"] 0B
<missing>         13 days ago       /bin/sh -c #(nop) ADD
file:e69d441d729412d24... 5.59MB

```

Each of the lines represents a layer in the image. The output shows the base at the bottom with the newest layer at the top. Using this information, you can see the size of each layer, helping diagnose large images.

2. Several of the lines are truncated. If you add the `--no-trunc` parameter, you'll get the full output.

Bash

```
docker image history --no-trunc getting-started
```

Cache dependencies

Once a layer changes, all downstream layers have to be recreated as well. Here's the *Dockerfile* again:

Dockerfile

```

FROM node:20-alpine
WORKDIR /app
COPY . .
RUN yarn install --production
CMD ["node", "/app/src/index.js"]

```

Each command in the *Dockerfile* becomes a new layer in the image. To minimize the number of layers, you can restructure your *Dockerfile* to support caching of dependencies. For Node-based applications, those dependencies are defined in the `package.json` file.

The approach is to copy only that file in first, install the dependencies, and *then* copy everything else. The process only recreates the yarn dependencies if there was a change to the `package.json`.

1. Update the *Dockerfile* to copy in the `package.json` first, install dependencies, and then copy everything else. Here's the new file:

Dockerfile

```
FROM node:20-alpine
WORKDIR /app
COPY package.json yarn.lock ./
RUN yarn install --production
COPY . .
CMD ["node", "/app/src/index.js"]
```

2. Build a new image using `docker build`.

Bash

```
docker build -t getting-started .
```

You should see output like the following results:

Output

```
Sending build context to Docker daemon 219.1kB
Step 1/6 : FROM node:12-alpine
---> b0dc3a5e5e9e
Step 2/6 : WORKDIR /app
---> Using cache
---> 9577ae713121
Step 3/6 : COPY package* yarn.lock ./
---> bd5306f49fc8
Step 4/6 : RUN yarn install --production
---> Running in d53a06c9e4c2
yarn install v1.17.3
[1/4] Resolving packages...
[2/4] Fetching packages...
info fsevents@1.2.9: The platform "linux" is incompatible with this
module.
info "fsevents@1.2.9" is an optional dependency and failed
compatibility check. Excluding it from installation.
```

```
[3/4] Linking dependencies...
[4/4] Building fresh packages...
Done in 10.89s.
Removing intermediate container d53a06c9e4c2
---> 4e68fbc2d704
Step 5/6 : COPY . .
---> a239a11f68d8
Step 6/6 : CMD ["node", "/app/src/index.js"]
---> Running in 49999f68df8f
Removing intermediate container 49999f68df8f
---> e709c03bc597
Successfully built e709c03bc597
Successfully tagged getting-started:latest
```

All layers were rebuilt. This result is expected because you changed the *Dockerfile*.

3. Make a change to the *src/static/index.html*. For instance, change the title to say "The Awesome Todo App".
4. Build the Docker image now using `docker build` again. This time, your output should look a little different.

```
plaintext

Sending build context to Docker daemon 219.1kB
Step 1/6 : FROM node:12-alpine
---> b0dc3a5e5e9e
Step 2/6 : WORKDIR /app
---> Using cache
---> 9577ae713121
Step 3/6 : COPY package* yarn.lock ./
---> Using cache
---> bd5306f49fc8
Step 4/6 : RUN yarn install --production
---> Using cache
---> 4e68fbc2d704
Step 5/6 : COPY . .
---> cccde25a3d9a
Step 6/6 : CMD ["node", "/app/src/index.js"]
---> Running in 2be75662c150
Removing intermediate container 2be75662c150
---> 458e5c6f080c
Successfully built 458e5c6f080c
Successfully tagged getting-started:latest
```

Because you're using the build cache, it should go much faster.

Multi-stage builds

Multi-stage builds are an incredibly powerful tool to help use multiple stages to create an image. There are several advantages for them:

- Separate build-time dependencies from runtime dependencies
- Reduce overall image size by shipping only what your app needs to run

This section provides brief examples.

Maven/Tomcat example

When you build Java-based applications, a JDK is needed to compile the source code to Java bytecode. That JDK isn't needed in production. You might be using tools like Maven or Gradle to help build the app. Those tools also aren't needed in your final image.

Dockerfile

```
FROM maven AS build
WORKDIR /app
COPY . .
RUN mvn package

FROM tomcat
COPY --from=build /app/target/file.war /usr/local/tomcat/webapps
```

This example uses one stage, `build`, to perform the actual Java build using Maven. The second stage, starting at "FROM tomcat", copies in files from the `build` stage. The final image is only the last stage being created, which can be overridden using the `--target` parameter.

React example

When building React applications, you need a Node environment to compile the JavaScript code, SASS stylesheets, and more into static HTML, JavaScript, and CSS. If you aren't doing server-side rendering, you don't even need a Node environment for the production build.

Dockerfile

```
FROM node:20-alpine AS build
WORKDIR /app
COPY package* yarn.lock ./
RUN yarn install
COPY public ./public
COPY src ./src
RUN yarn run build
```



```
FROM nginx:alpine
COPY --from=build /app/build /usr/share/nginx/html
```

This example uses a `node:20` image to perform the build, which maximizes layer caching, and then copies the output into an *nginx* container.

Clean up resources

Keep everything that you've done so far to continue this series of tutorials.

Next steps

You've learned about options to persist data for container apps.

What do you want to do next?

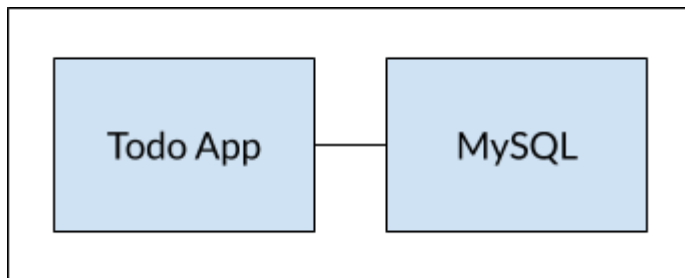
- Work with multiple containers by using Docker Compose:
 - [Create multi-container apps with MySQL and Docker Compose](#)
- Deploy to Azure Container Apps:
 - [Quickstart: Deploy to Azure Container Apps using Visual Studio Code](#)
 - [Tutorial: Deploy to Azure Container Apps](#)
- Deploy to Azure App Service
 - [Deploy a containerized app to Azure](#) [↗](#)

Tutorial: Create multi-container apps with MySQL and Docker Compose

Article • 05/31/2023

In this tutorial, you'll learn how to create multi-container apps. This tutorial builds on the getting started tutorials, [Get started with Docker and Visual Studio Code](#). In this advanced tutorial, you'll update your application to work as described in this diagram and learn how to:

- ✓ Start MySQL.
- ✓ Run your app with MySQL.
- ✓ Create the compose file.
- ✓ Run the application stack.



Using multiple containers allows you to dedicate containers for specialized tasks. Each container should do one thing and do it well.

Here are some reasons you might want to use multi-container apps:

- Separate containers you to manage APIs and front-ends differently than databases.
- Containers let you version and update versions in isolation.
- While you might use a container for the database locally, you may want to use a managed service for the database in production.
- Running multiple processes requires a process manager, which adds complexity to container startup/shutdown.

Prerequisites

This tutorial continues the series of tutorials, starting with [Create a container app](#). Start with that one, which includes prerequisites. Then do the tutorial [Persist data in your app](#).

You also need the following items:

- [Docker Compose](#) [↗](#).

Docker Desktop for Windows or Mac includes Docker Compose. Run this command to verify:

```
Bash
```

```
docker-compose version
```

If you use the Linux operating system, [Install Docker Compose](#).

As with the previous tutorials, you can accomplish most tasks from the VS Code **EXPLORER** view or the **DOCKER** view. You can select **Terminal > New Terminal** to open a command-line window in VS Code. You can also run commands in a Bash window. Unless specified, any command labeled as **Bash** can run in a Bash window or the VS Code terminal.

Start MySQL

Containers, by default, run in isolation. They don't know anything about other processes or containers on the same computer. To allow one container to talk to another, use networking.

If two containers are on the same network, they can talk to each other. If they aren't, they can't.

There are two ways to put a container on a network: assign it at start or connect an existing container. In this example, you create the network first and attach the MySQL container at startup.

1. Create the network by using this command.

```
Bash
```

```
docker network create todo-app
```

2. Start a MySQL container and attach it the network.

```
Bash
```

```
docker run -d
  --network todo-app --network-alias mysql
  -v todo-mysql-data:/var/lib/mysql
  -e MYSQL_ROOT_PASSWORD=<your-password>
  -e MYSQL_DATABASE=todos
  mysql:5.7
```

This command also defines environment variables. For more information, see [MySQL Docker Hub listing](#).

The command specifies a network alias, `mysql`.

3. Get your container ID by using the `docker ps` command.
4. To confirm you have the database up and running, connect to the database.

Bash

```
docker exec -it <mysql-container-id> mysql -p
```

Enter the password you used, above, when prompted.

5. In the MySQL shell, list the databases and verify you see the `todos` database.

SQL

```
SHOW DATABASES;
```

You should see the following output.

Output

```
+-----+
| Database          |
+-----+
| information_schema |
| mysql             |
| performance_schema |
| sys               |
| todos             |
+-----+
5 rows in set (0.00 sec)
```

6. Enter `exit` when you're ready to return to the terminal command prompt.

Run your app with MySQL

The todo app supports the setting of environment variables to specify MySQL connection settings.

- `MYSQL_HOST` The hostname for the MySQL server.
- `MYSQL_USER` The username to use for the connection.

- `MYSQL_PASSWORD` The password to use for the connection.
- `MYSQL_DB` The database to use once connected.

⚠ Warning

Using environment variables to set connection settings is acceptable for development. We recommend against this practice for running applications in production. For more information, see [Why you shouldn't use environment variables for secret data](#).

A more secure mechanism is to use the secret support provided by your container orchestration framework. In most cases, these secrets are mounted as files in the running container.

This procedure starts your app and connects that container to your MySQL container.

1. Use the following docker run command. It specifies the environment variables above.

Bash

```
docker run -dp 3000:3000
  -w /app -v ${PWD}:/app
  --network todo-app
  -e MYSQL_HOST=mysql
  -e MYSQL_USER=root
  -e MYSQL_PASSWORD=<your-password>
  -e MYSQL_DB=todos
  node:20-alpine
  sh -c "yarn install && yarn run dev"
```

2. In VS Code, in the Docker view, right-click the app container and select **View Logs**. To view the logs from the command line, use the `docker logs` command.

The result includes a line that indicates that the app is connected to the MySQL database.

Output

```
# Previous log messages omitted
$ nodemon src/index.js
[nodemon] 1.19.2
[nodemon] to restart at any time, enter `rs`
[nodemon] watching dir(s): *.*
[nodemon] starting `node src/index.js`
```

```
Connected to mysql db at host mysql
Listening on port 3000
```

3. Enter `http://localhost:3000` into your browser. Add some items to your todo list.
4. Connect to the MySQL database, as you did in the previous section. Run this command to verify that the items are being written to the database.

Bash

```
docker exec -ti <mysql-container-id> mysql -p todos
```

And in the MySQL shell, run the following commands.

SQL

```
use todos;
select * from todo_items;
```

Your result will look like the following output.

Output

```
+-----+-----+-----+
--+
| id                | name                | completed |
|
+-----+-----+-----+
--+
| c906ff08-60e6-44e6-8f49-ed56a0853e85 | Do amazing things! | 0         |
|
| 2912a79e-8486-4bc3-a4c5-460793a575ab | Be awesome!        | 0         |
|
+-----+-----+-----+
--+
```

At this point, you have an application that stores data in an external database. That database runs in a separate container. You learned about container networking.

Create a Docker Compose file

Docker Compose helps define and share multi-container applications. With Docker Compose, you can create a file to define the services. With a single command, you can spin up everything or tear it all down.

You can define your application stack in a file and keep that file at the root of your project repo, under version control. This approach enables others to contribute to your project. They would only need to clone your repo.

1. At the root of the app project, create a file named `docker-compose.yml`.
2. In the compose file, start by defining the schema version.

YAML

```
version: "3.7"
```

In most cases, it's best to use the latest supported version. For current schema versions and compatibility matrix, see [Compose file](#).

3. Define the services, or containers, you want to run as part of your application.

YAML

```
version: "3.7"
```

```
services:
```

Tip

Indentation is significant in `.yaml` files. If you're editing in VS Code, Intellisense indicates errors.

4. Here's the command you used to your app container. You'll add this information to your `.yaml` file.

Bash

```
docker run -dp 3000:3000
  -w /app -v ${PWD}:/app
  --network todo-app
  -e MYSQL_HOST=mysql
  -e MYSQL_USER=root
  -e MYSQL_PASSWORD=<your-password>
  -e MYSQL_DB=todos
  node:20-alpine
  sh -c "yarn install && yarn run dev"
```

Define the service entry and the image for the container.

YAML

```
version: "3.7"

services:
  app:
    image: node:20-alpine
```

You can pick any name for the service. The name automatically becomes a network alias, which is useful when defining the MySQL service.

5. Add the command.

YAML

```
version: "3.7"

services:
  app:
    image: node:20-alpine
    command: sh -c "yarn install && yarn run dev"
```

6. Specify the ports for the service, which correspond to `-p 3000:3000` in the command above.

YAML

```
version: "3.7"

services:
  app:
    image: node:20-alpine
    command: sh -c "yarn install && yarn run dev"
    ports:
      - 3000:3000
```

7. Specify the working directory and the volume mapping

YAML

```
version: "3.7"

services:
  app:
    image: node:20-alpine
    command: sh -c "yarn install && yarn run dev"
    ports:
      - 3000:3000
    working_dir: /app
```



```
volumes:
  - ./:/app
```

In Docker Compose volume definitions, you can use relative paths from the current directory.

8. Specify the environment variable definitions.

YAML

```
version: "3.7"

services:
  app:
    image: node:20-alpine
    command: sh -c "yarn install && yarn run dev"
    ports:
      - 3000:3000
    working_dir: /app
    volumes:
      - ./:/app
    environment:
      MYSQL_HOST: mysql
      MYSQL_USER: root
      MYSQL_PASSWORD: <your-password>
      MYSQL_DB: todos
```

9. Add the definitions for the MySQL service. Here's the command you used above:

Bash

```
docker run -d
  --network todo-app --network-alias mysql
  -v todo-mysql-data:/var/lib/mysql
  -e MYSQL_ROOT_PASSWORD=<your-password>
  -e MYSQL_DATABASE=todos
  mysql:5.7
```

Define the new service and name it *mysql*. Add your text after the `app` definition, at the same level of indentation.

YAML

```
version: "3.7"

services:
  app:
    # The app service definition
```

```
mysql:
  image: mysql:5.7
```

The service automatically gets the network alias. Specify the image to use.

10. Define the volume mapping.

Specify the volume with a `volumes:` section at the same level as `services:`. Specify the volume mapping under the image.

YAML

```
version: "3.7"

services:
  app:
    # The app service definition
  mysql:
    image: mysql:5.7
    volumes:
      - todo-mysql-data:/var/lib/mysql

volumes:
  todo-mysql-data:
```

11. Specify the environment variables.

YAML

```
version: "3.7"

services:
  app:
    # The app service definition
  mysql:
    image: mysql:5.7
    volumes:
      - todo-mysql-data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: <your-password>
      MYSQL_DATABASE: todos

volumes:
  todo-mysql-data:
```

At this point, the complete `docker-compose.yml` looks like this:

YAML

```
version: "3.7"

services:
  app:
    image: node:20-alpine
    command: sh -c "yarn install && yarn run dev"
    ports:
      - 3000:3000
    working_dir: /app
    volumes:
      - ./:/app
    environment:
      MYSQL_HOST: mysql
      MYSQL_USER: root
      MYSQL_PASSWORD: <your-password>
      MYSQL_DB: todos

  mysql:
    image: mysql:5.7
    volumes:
      - todo-mysql-data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: <your-password>
      MYSQL_DATABASE: todos

volumes:
  todo-mysql-data:
```

Run the application stack

Now that you have the `docker-compose.yml` file, try it.

1. Make sure no other copies of the app and database are running. In the Docker extension, right-click any running container and select **Remove**. Or, at the command line, use the command `docker rm` as in previous examples.
2. In the VS Code Explorer, right-click `docker-compose.yml` and select **Compose Up**. Or, at the command line, use this docker command.

Bash

```
docker-compose up -d
```

The `-d` parameter makes the command run in the background.

You should see output like the following results.

Output

```
[+] Building 0.0s (0/0)
[+] Running 2/2
✓ Container app-app-1    Started
0.9s
✓ Container app-mysql-1  Running
```

The volume was created as well as a network. By default, Docker Compose creates a network specifically for the application stack.

3. In the Docker extension, right-click the app container and select **View Logs**. To view the logs from the command line, use the `docker logs` command.

Output

```
mysql_1 | 2019-10-03T03:07:16.083639Z 0 [Note] mysqld: ready for
connections.
mysql_1 | Version: '5.7.27'  socket: '/var/run/mysqld/mysqld.sock'
port: 3306  MySQL Community Server (GPL)
app_1   | Connected to mysql db at host mysql
app_1   | Listening on port 3000
```

The logs from each of the services are interleaved into a single stream. With this behavior, you can watch for timing-related issues.

The service name is displayed at the beginning of the line to help distinguish messages. To view logs for a specific service, add the service name to the end of the logs command.

4. At this point, you should be able to open your app. Enter `http://localhost:3000` into your browser.

When you're done with these containers, you can remove them all simply.

- In VS Code Explorer, right-click **docker-compose.yml** and select **Compose Down**.
- At the command line, run `docker-compose down`.

The containers stop. The network is removed.

By default, named volumes in your compose file aren't removed. If you want to remove the volumes, run `docker-compose down --volumes`.

Clean up resources

The prerequisites you used in this tutorial series can be used for future Docker development. There's no reason to delete or uninstall anything.

Next steps

In this tutorial, you learned about multi-container apps and Docker Compose. Docker Compose helps dramatically simplify the defining and sharing of multi-service applications.

Here are some resources that might be useful to you:

- [Docker Cloud Integration](#) 
- [Examples](#) 