Using Bind Mounts

In the previous chapter, we talked about and used a **named volume** to persist the data in our database. Named volumes are great if we simply want to store data, as we don't have to worry about *where* the data is stored.

With **bind mounts**, we control the exact mountpoint on the host. We can use this to persist data, but is often used to provide additional data into containers. When working on an application, we can use a bind mount to mount our source code into the container to let it see code changes, respond, and let us see the changes right away.

For Node-based applications, nodemon [https://npmjs.com/package/nodemon] is a great tool to watch for file changes and then restart the application. There are equivalent tools in most other languages and frameworks.

Quick Volume Type Comparisons

Bind mounts and named volumes are the two main types of volumes that come with the Docker engine. However, additional volume drivers are available to support other use cases (SFTP [https://github.com/vieux/docker-volume-sshfs], Ceph [https://ceph.com/geen-categorie/getting-started-with-the-docker-rbd-volume-plugin/], NetApp [https://netappdvp.readthedocs.io/en/stable/], S3 [https://github.com/elementar/docker-s3-volume], and more).

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

Starting a Dev-Mode Container

To run our container to support a development workflow, we will do the following:

- Mount our source code into the container
- Install all dependencies, including the "dev" dependencies
- Start nodemon to watch for filesystem changes

So, let's do it!

1. Make sure you don't have any previous `getting-started` containers running.
2. Also make sure you are in app source code directory, i.e. `/path/to/getting-started/app` . If you aren't, you can `cd` into it, .e.g:

   ```
   cd /path/to/getting-started/app
   ```

3. Now that you are in the `getting-started/app` directory, run the following command. We'll explain what's going on afterwards:

   ```
   docker run -dp 3000:3000 \
       -w /app -v "$(pwd):/app" \
       node:12-alpine \
       sh -c "yarn install && yarn run dev"
   ```

If you are using PowerShell then use this command.

```
docker run -dp 3000:3000 `
    -w /app -v "$(pwd):/app" `
    node:12-alpine `
    sh -c "yarn install && yarn run dev"
```

If you are using an Apple Silicon Mac or another ARM64 device then use this command.

```
docker run -dp 3000:3000 \
    -w /app -v "$(pwd):/app" \
    node:12-alpine \
    sh -c "apk add --no-cache python2 g++ make && yarn install
&& yarn run dev"
```

- `-dp 3000:3000` - same as before. Run in detached (background) mode and create a port mapping
- `-w /app` - sets the container's present working directory where the command will run from
- `-v "$(pwd):/app"` - bind mount (link) the host's present `getting-started/app` directory to the container's `/app` directory. Note: Docker requires absolute paths for binding mounts, so in this example we use `pwd` for printing the absolute path of the working directory, i.e. the `app` directory, instead of typing it manually
- `node:12-alpine` - the image to use. Note that this is the base image for our app from the Dockerfile
- `sh -c "yarn install && yarn run dev"` - the command. We're starting a shell using `sh` (alpine doesn't have `bash`) and running `yarn install` to install *all* dependencies and then running `yarn run dev`. If we look in the `package.json`, we'll see that the `dev` script is starting `nodemon`.

4. You can watch the logs using `docker logs -f <container-id>`. You'll know you're ready to go when you see this...

```
docker logs -f <container-id>
$ nodemon src/index.js
[nodemon] 1.19.2
```

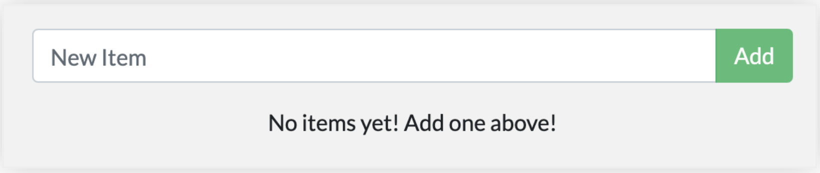
```
[nodemon] to restart at any time, enter `rs`
[nodemon] watching dir(s): *.*
[nodemon] starting `node src/index.js`
Using sqlite database at /etc/todos/todo.db
Listening on port 3000
```

When you're done watching the logs, exit out by hitting `Ctrl` + `C` .

5. Now, let's make a change to the app. In the `src/static/js/app.js` file, let's change the "Add Item" button to simply say "Add". This change will be on line 109 - remember to save the file.

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

6. Simply refresh the page (or open it) and you should see the change reflected in the browser almost immediately. It might take a few seconds for the Node server to restart, so if you get an error, just try refreshing after a few seconds.



No items yet! Add one above!

7. Feel free to make any other changes you'd like to make. When you're done, stop the container and build your new image using `docker build -t getting-started .` .

Using bind mounts is *very* common for local development setups. The advantage is that the dev machine doesn't need to have all of the build tools and environments installed. With a single `docker run` command, the dev environment is pulled and ready to go. We'll talk about Docker Compose in a future step, as this will help simplify our commands (we're already getting a lot of flags).

Recap

At this point, we can persist our database and respond rapidly to the needs and demands of our investors and founders. Hooray! But, guess what? We received great news!

Your project has been selected for future development!

In order to prepare for production, we need to migrate our database from working in SQLite to something that can scale a little better. For simplicity, we'll keep with a relational database and switch our application to use MySQL. But, how should we run MySQL? How do we allow the containers to talk to each other? We'll talk about that next!