[Choose Your Docker Compose Version](https://developer.okta.com/blog/2017/10/11/developers-guide-to-docker-part-3" \l "choose-your-docker-compose-version)

The first line of any docker-compose.yml file is the version setting. In this case, you’ll use version 3.3, so just tell Docker Compose that.

version: '3.3'

You can see the documentation for docker-compose version 3 at <https://docs.docker.com/compose/compose-file/> and you can see what the differences are between versions.

[Define Node and Mongo Services](https://developer.okta.com/blog/2017/10/11/developers-guide-to-docker-part-3#define-node-and-mongo-services)

Services are how Docker refers to each container you want to build in the docker-compose file. In this case, you’ll create two services: one for the NodeJS application, and one for the MongoDB database.

services:

app:

db:

Remember that indention is how the YAML file format groups information, so indention is important. Here, you’ve indented the app and db services under the services tag. These can be named whatever you want. In this case app and ‘db` are just easiest to refer to. Now, you’ll put some meat on these two services.

First, tell Docker what image you want to build the app service from by specifying that you’ll be building from the sample:1.0image. So you’ll specify that indented under the app tag.

app:

image: sample:1.0

Of course that image doesn’t exist, so you’ll need to let Docker know where to find the Dockerfile to build it by setting a build context. If you don’t, Docker will try to pull the image from Docker Hub and when that fails it will fail the docker-composecommand altogether.

app:

image: sample:1.0

build: .

Here, you’ve specified that the build context is the current directory, so when Docker can’t find the sample:1.0 image locally, it will build it using the Dockerfile in the current directory.

Next, you’ll tell Docker what the container name should be once it’s built the image to create the container from.

app:

image: sample:1.0

container\_name: sample\_app

build: .

Now, when Docker builds the image, it will immediately create a container named sample\_app from that image.

By default, NodeJS apps run on port 3000, so you’ll need to map that port to 80, since this is the “production” docker-compose file. You do that using the ports tag in YAML.

app:

image: sample:1.0

container\_name: sample\_app

build: .

ports:

- 80:3000

Here, you’ve mapped port 80 on the host operating system, to port 3000 from the container. That way, when you’ve moved this container to a production host, users of the application can go to the host machine’s port 80 and have those requests answered from the container on port 3000.

Your application will be getting data from a MongoDB database and to do that the application will need a connection string that it will get from an environment variable called “MONGO\_URI”. To set environment variables in the container once it’s built, use the environment tag in the YAML file.

app:

image: sample:1.0

container\_name: sample\_app

build: .

ports:

- 80:3000

environment:

- MONGO\_URI=mongodb://sampledb/sample

[Create a Docker Network](https://developer.okta.com/blog/2017/10/11/developers-guide-to-docker-part-3#create-a-docker-network)

For the application service to actually be able to reach the sample database, it will need to be on the same network. To put both of these services on the same network, create one in the docker-compose file by using the networks tag at the top level (the same level of indention as the services tag.

version: '3.3'

services:

app:...

db:...

networks:

samplenet:

driver: bridge

This creates a network called “samplenet” using a bridge type network. This will allow the two containers to communicate over a virtual network between them.

Back in the app section of the file, join the app service to the “samplenet” network:

app:

image: sample:1.0

container\_name: sample\_app

build: .

ports:

- 80:3000

environment:

- MONGO\_URI=mongodb://sampledb/sample

networks:

- samplenet

[Create the MongoDB Service](https://developer.okta.com/blog/2017/10/11/developers-guide-to-docker-part-3#create-the-mongodb-service)

Now the app service is ready, but it won’t be much good without the db service. So add the same kinds of things in the next section for the db service.

db:

image: mongo:3.0.15

container\_name: sample\_db

networks:

samplenet:

aliases:

- "sampledb"

This service builds from the official MongoDB 3.0.15 image and creates a container named “sample\_db”. It also joins the “samplenet” network with an alias of “sampledb”. This is like a DNS name on a physical network, it allows other services on the “samplenet” network to refer to it by its alias. This is important because without it, the app service would have a much harder time talking to it. (I don’t know that it *couldn’t*, it would just probably have to use the container’s full hash!)

[Use Docker Volumes](https://developer.okta.com/blog/2017/10/11/developers-guide-to-docker-part-3#use-docker-volumes)

You’ll also want to create a volume mount in the database service. Volumes allow you to mount folders on the host machine to folders in the container. Meaning, when something inside the container refers to a folder, it will actually be accessing a folder on the host machine. This is especially helpful for database containers because containers are meant to be disposable. With a mount to the physical folder on the host machine, you’ll be able to destroy a container and rebuild it and the data files for the container will still be there on the host machine. So add a volume tag in the db section mounting the /data/db folder in the container (where Mongo stores its data) to the db folder in your application’s root folder so that the final db section looks like the following.

db:

image: mongo:3.0.15

container\_name: sample\_db

volumes:

- ./db:/data/db

networks:

samplenet:

aliases:

- "sampledb"

[Review Your Docker Compose File](https://developer.okta.com/blog/2017/10/11/developers-guide-to-docker-part-3#review-your-docker-compose-file)

With all that done, your final docker-compose.yml file should look like:

version: '3.3'

services:

app:

image: sample:1.0

container\_name: sample\_app

build: .

ports:

- 80:3000

environment:

- MONGO\_URI=mongodb://sampledb/sample

depends\_on:

- db

networks:

- samplenet

db:

image: mongo:3.0.15

container\_name: sample\_db

volumes:

- ./db:/data/db

networks:

samplenet:

aliases:

- "sampledb"

networks:

samplenet:

driver: bridge

With that all done, you should be able to save the file and run docker-compose up -d in the folder where you docker-compose.yml file is and watch Docker build and start your environment for you.

If everything completes successfully, you can then go to http://localhost/users and see something like the image below.

