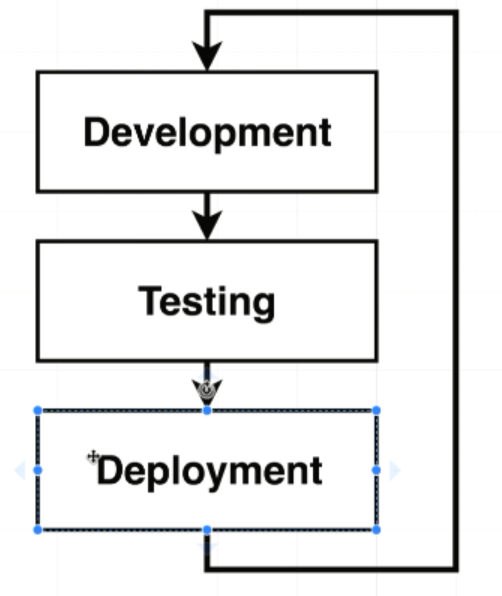
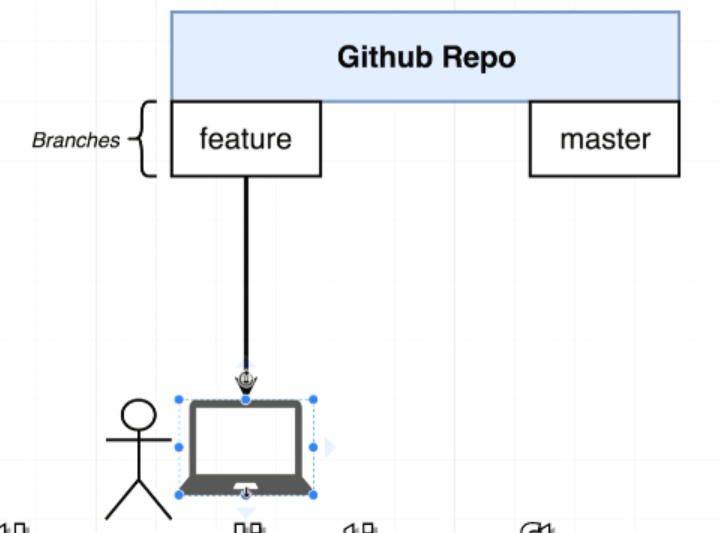
V—62:(Development Workflow):

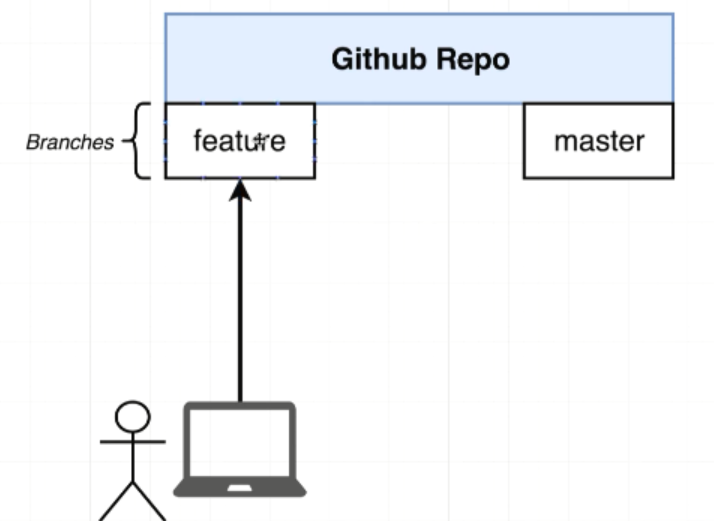


V—63:(Flow Specifics):



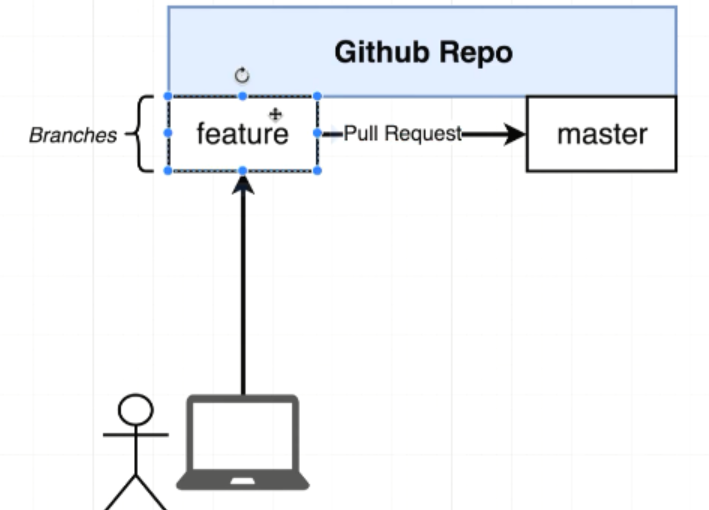
Pull the code to our local machine from git

Do some changes then

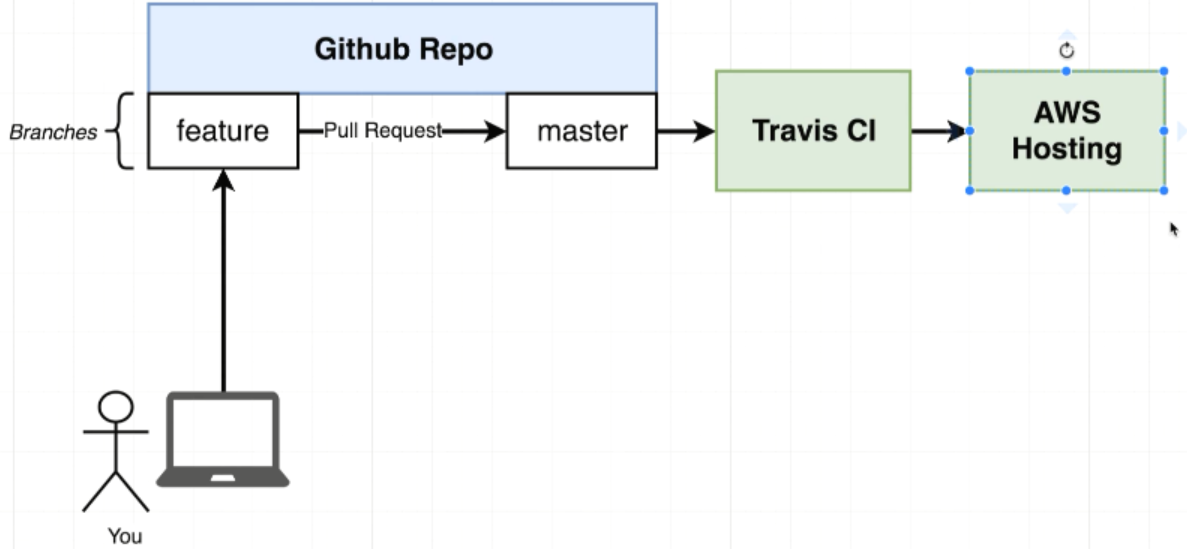


Push those changes back to the git repository.

Once we push our changes to the feature branch. You then need to create a pull request. Pull request to take all the changes you have added to the feature branch and merge them all over to the master branch.



Travis-CI is able to pull the code from master branch and run tests on it successfully. Travis CI is going to set upto automatically take our codebase take our entire project and push it over some Amazon Web Service hosting.



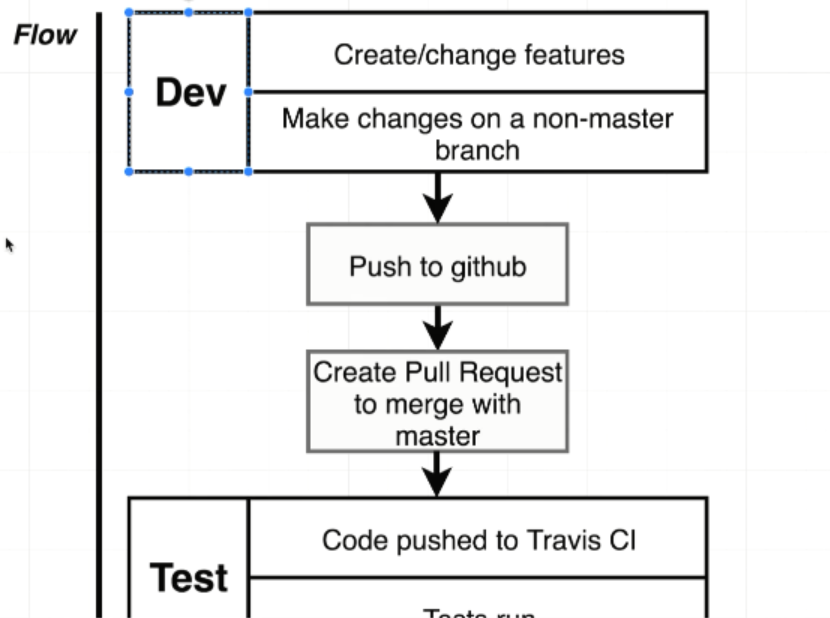
This entire flow is depending upon you pushing some code up to the feature branch

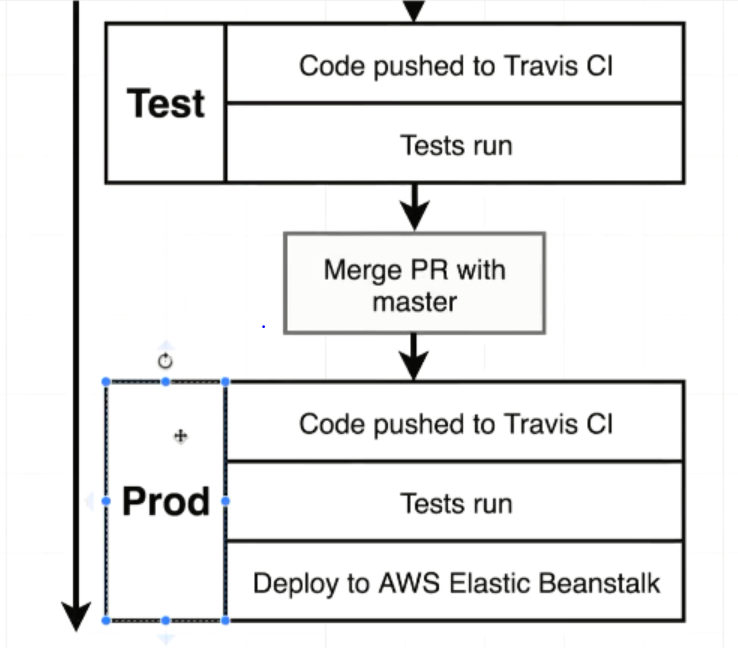
Creating the pull request

Merging the pull request with the master branch

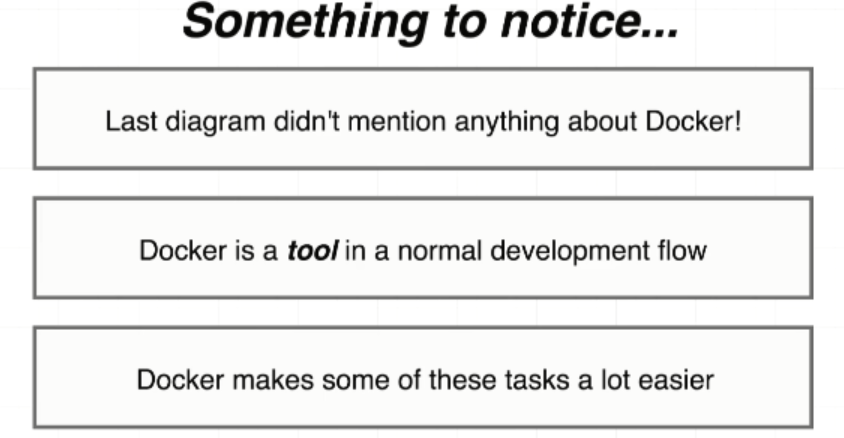
Then we would run our test cases

If the test cases are successful Travis CI will automatically deploy your application to AWS.





V—64:(Dockers Purpose):



V—65:(Project Generation):

Wrap react-front end inside a docker container and learn how to test it and deploy it automatically.

V—66:(React React app Generation):

**updated 8-4-2020**

In the next lecture, Stephen will be going over how to install Create React App globally and generate the application. This method of generating a React project is no longer recommended.

**Instead of this:**

npm install -g create-react-app

create-react-app frontend

**We need to run this command:**

npx create-react-app frontend

**Documentation:**

<https://create-react-app.dev/docs/getting-started#npx>

If you've previously installed create-react-app globally via npm install -g create-react-app, we recommend you uninstall the package using npm uninstall -g create-react-app to ensure that npx always uses the latest version.

V—67:(More on Project Generation):

Installing nodejs on linux(ubuntu):

sudo apt-get install curl

curl -sL https://deb.nodesource.com/setup\_15.x | sudo -E bash -

sudo apt-get install nodejs

**node -v**

v15.6.0

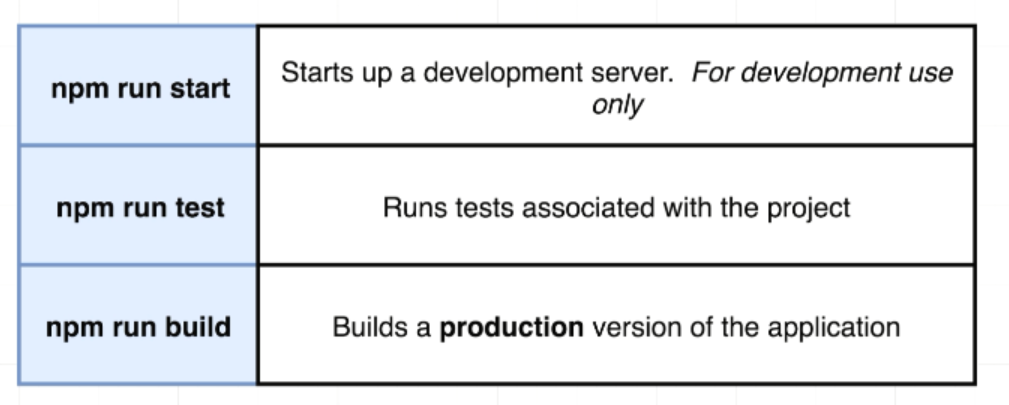
**npm -v**

7.4.0

sudo npm install -g create-react-app

create-react-app frontend

V—68:(Necessary Commands):



Project generation on Ubuntu:

npx create-react-app frontend

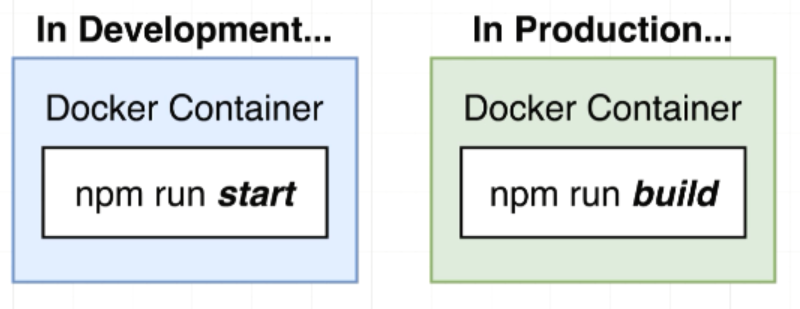
cd into the generated project

npm run test

npm run build

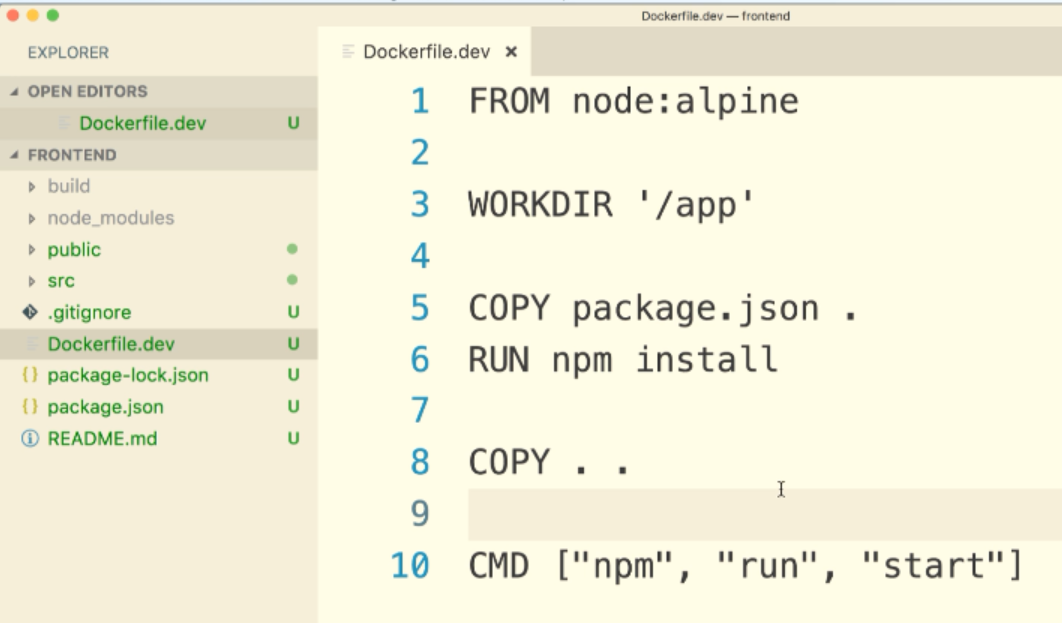
npm run start

V—69:(Creating the Dev docker file):

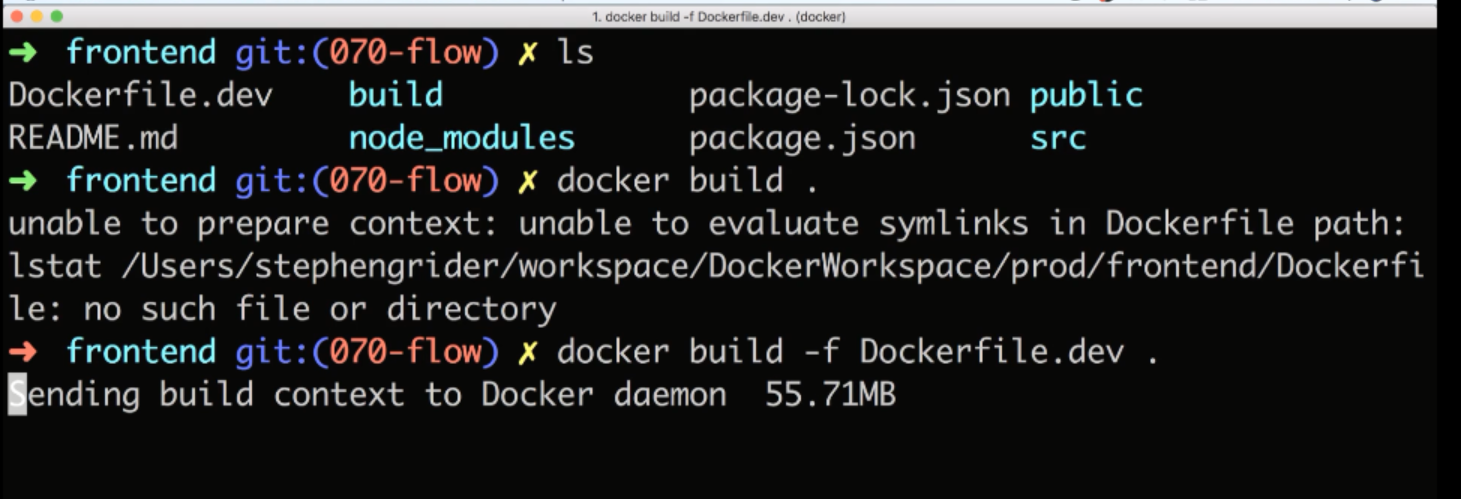


Dockerfile.dev

.dev indicates that we are going to use the dockerfile when we are running our application in development environment.



-f to specify the file name to build the container



V—70:(Duplicating Dependencies):

Delete the nodu modules folder in before copying because npm install will install all the node module dependicies.

V—71:(React App Exits immediately with docker run command):

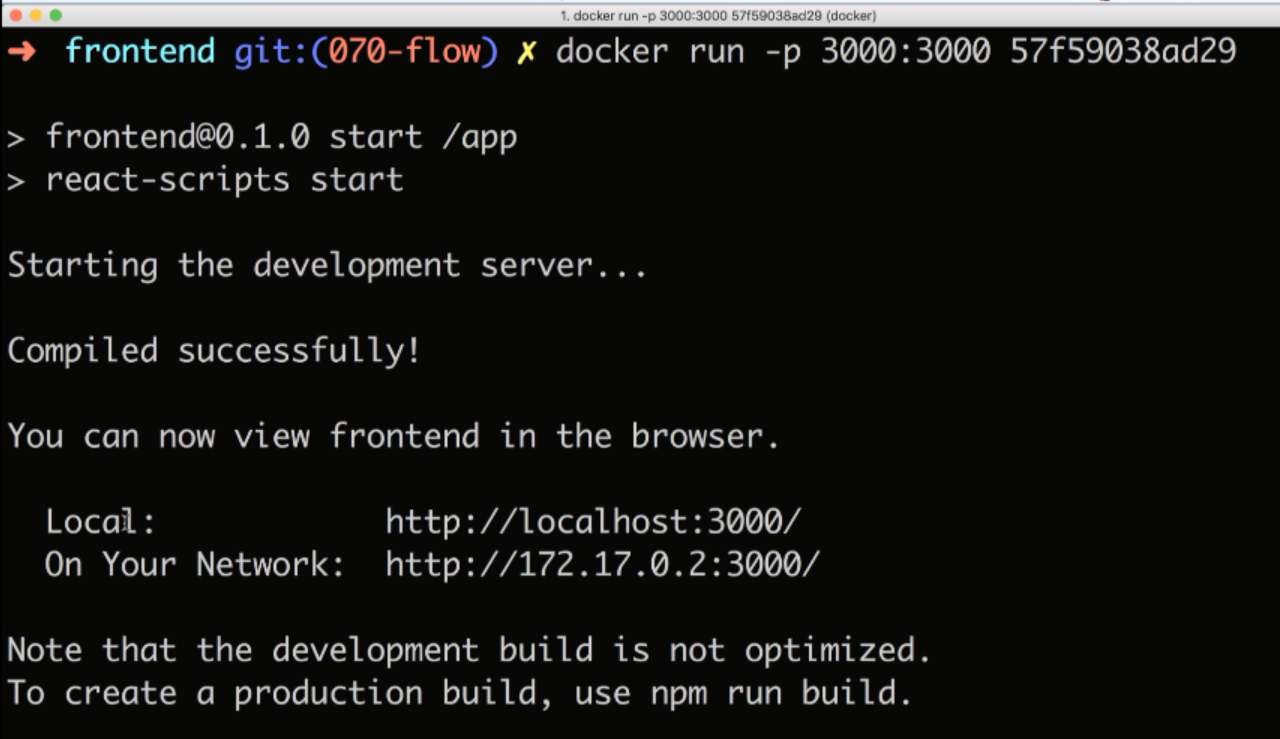
Due to a recent update in the Create React App library, we will need to change how we start our containers.

In the upcoming lecture, you'll need to add the -it flag to run the container in interactive mode:

docker run -it -p 3000:3000 CONTAINER\_ID

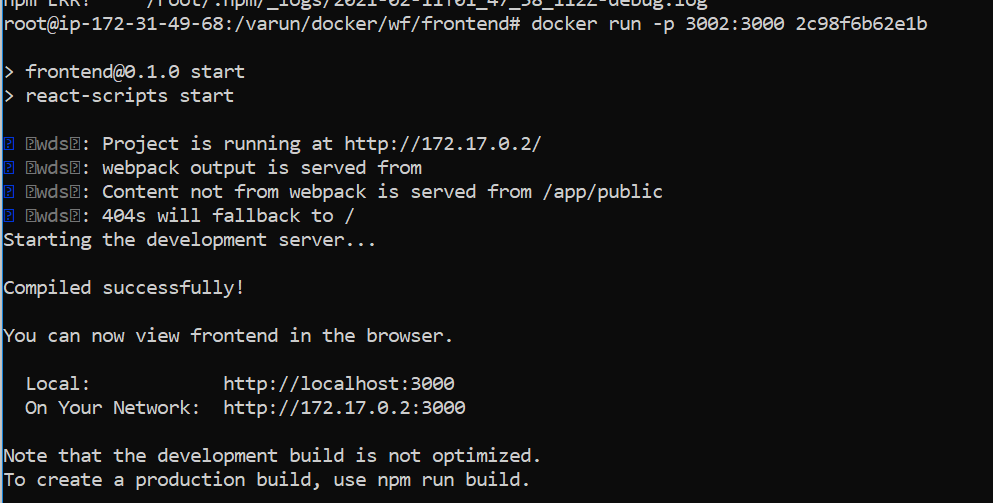
V—72:(Starting the container):

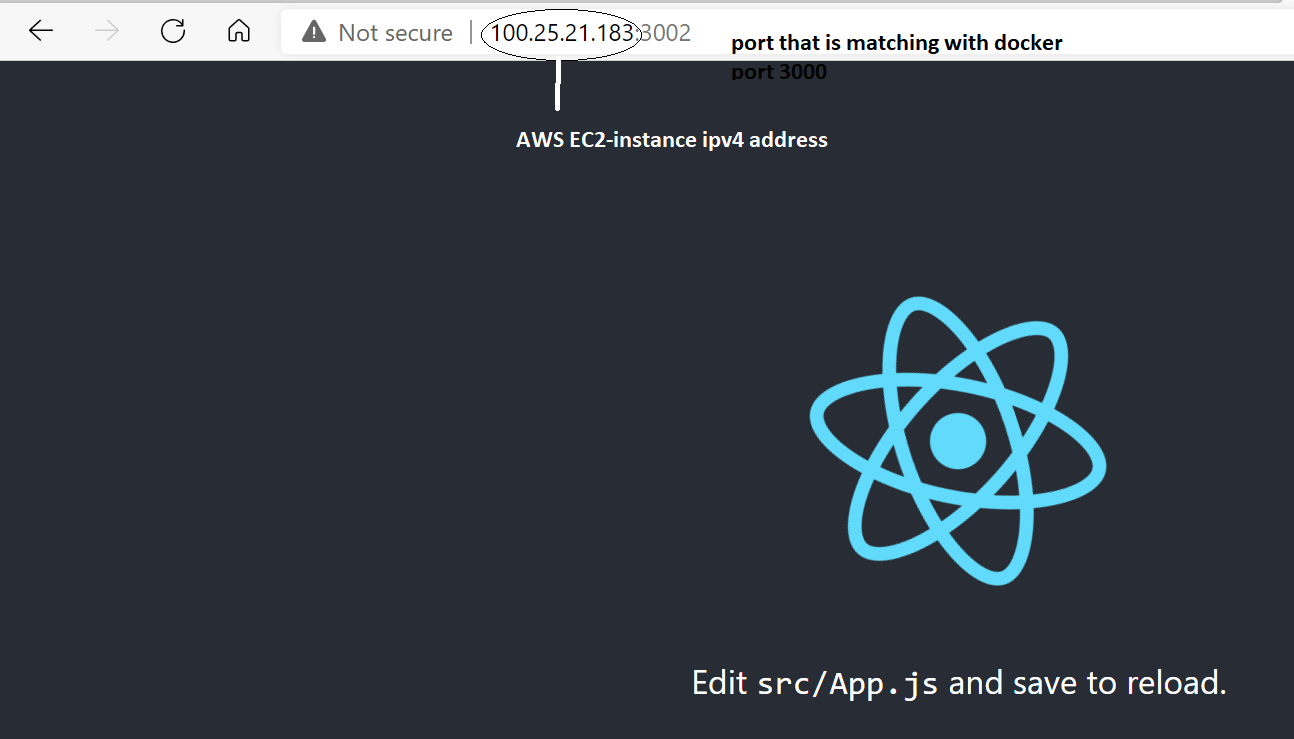
To expose port from our docker image or the docker container to our machine, we have to add -P flag to map out the port.



The command I gave to start the container :

docker run -p 3002:3000 2c98f6b62e1b

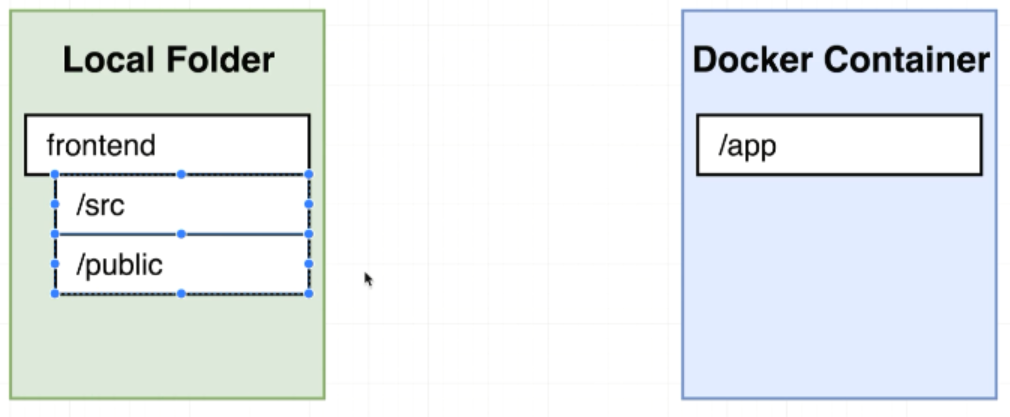


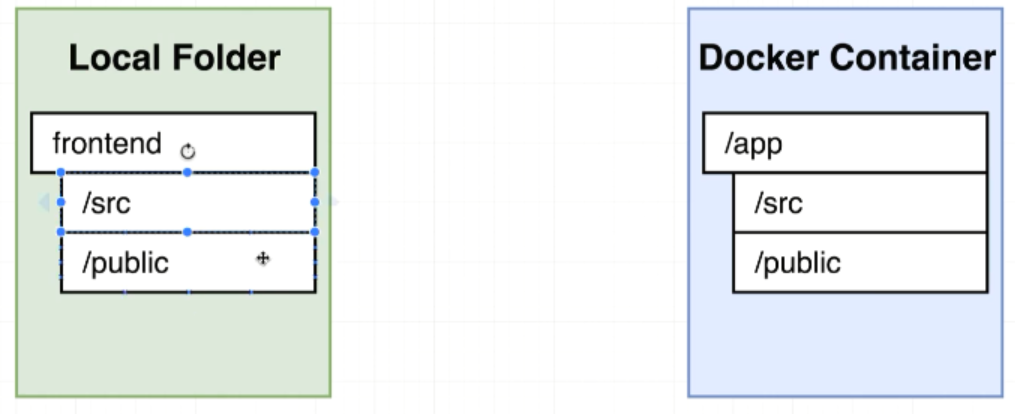


V—73:(Docker Volume):

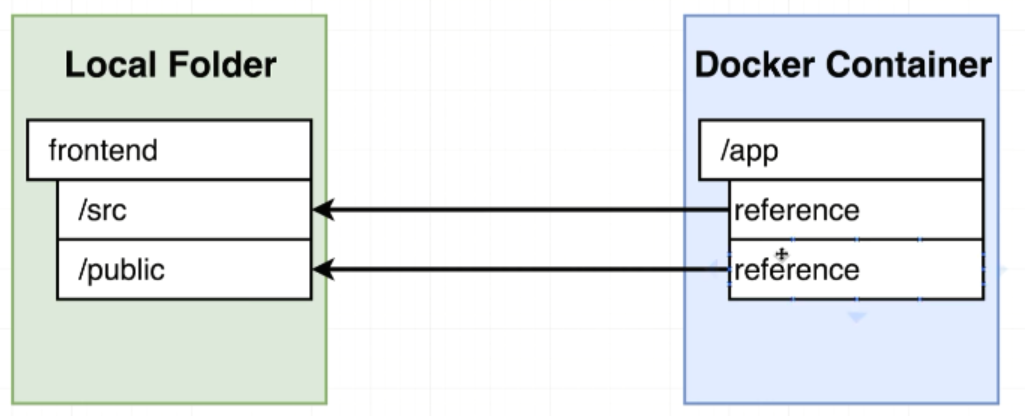
Docker volumes to automatically get changes that we made locally(to the continer).

A cleaver solution to make sure that any changes that we make to our source code get automatically propagated into the container as well without stop it rrebuild the image and then restart the container.



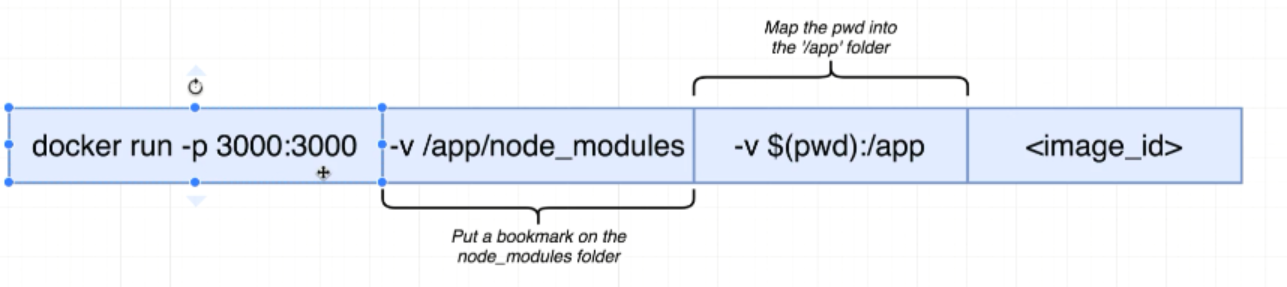


Instead of doing the above, we make use of volumes in docker



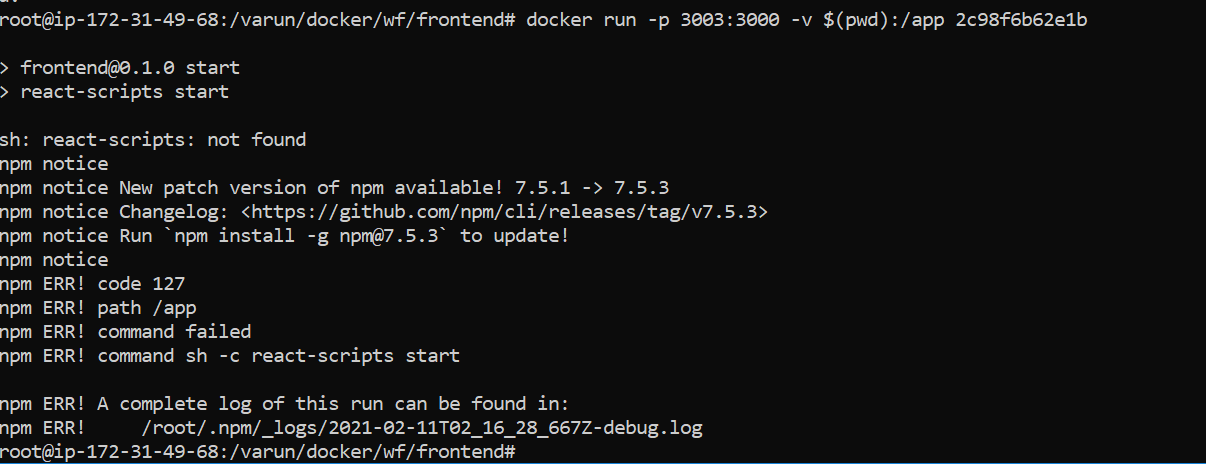
The volume is essentially going to set up a reference that going to point back to our local machine and give us access to the files and folders inside of these folders on the local machine.

We are mapping a folder inside a container to the outside the container like how we are mapping a port which is inside the container to the outside the container with -p flag.

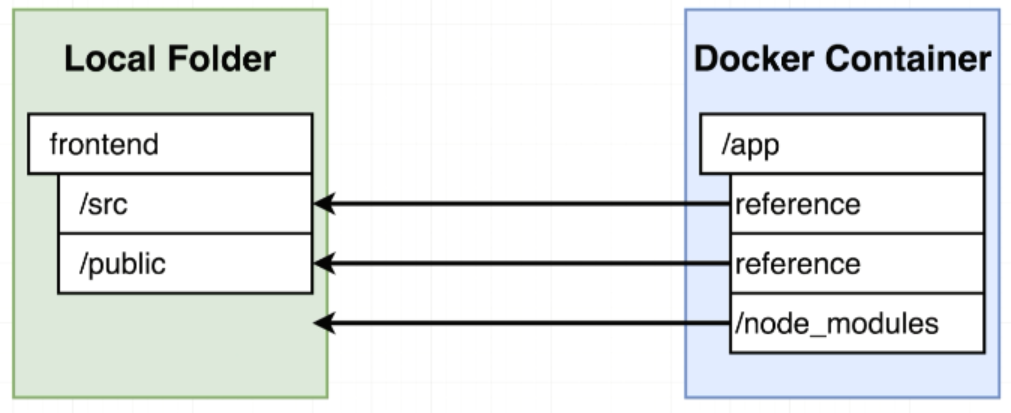


Without first argument:

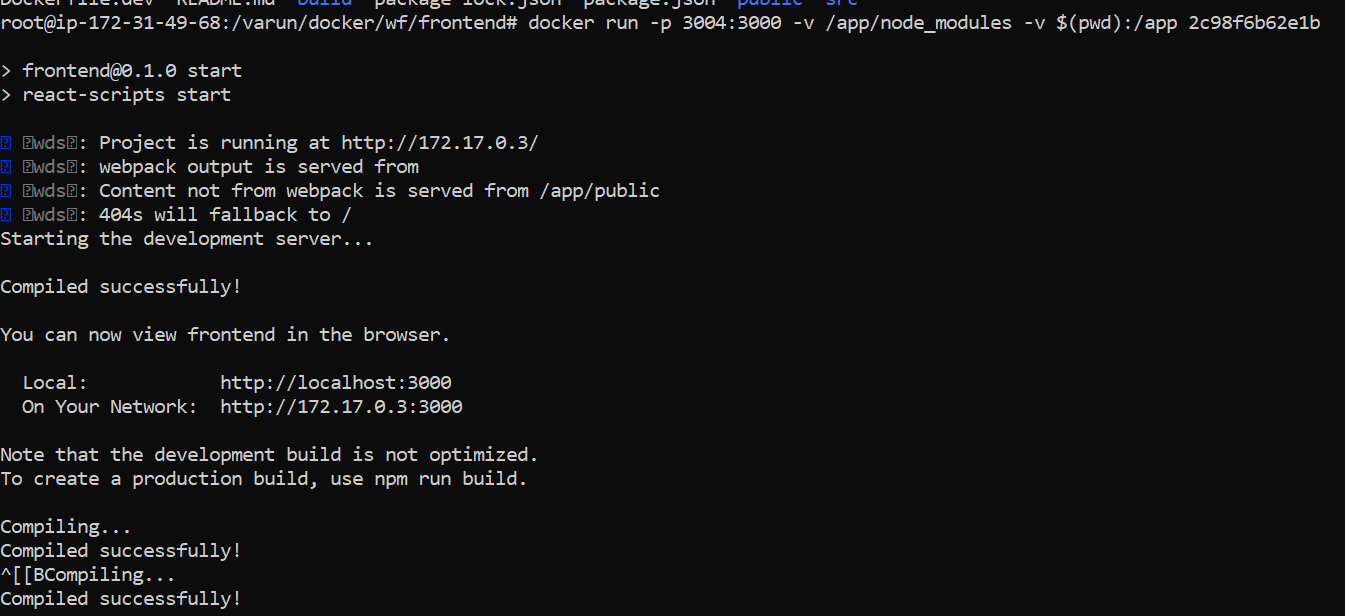
-v indicates volume



V—75:(Bookmarking Volumes):



node\_modules are referencing back to the blank(nothing). Blank because we don’t have node\_modules outside of the container.



V—76:(React App Exited with Code 0):

Recently, a bug was introduced with the latest Create React App version that is causing the React app to exit when starting with Docker Compose.

To Resolve this:

**Add stdin\_open property to your docker-compose.yml file**

web:  
 stdin\_open: true

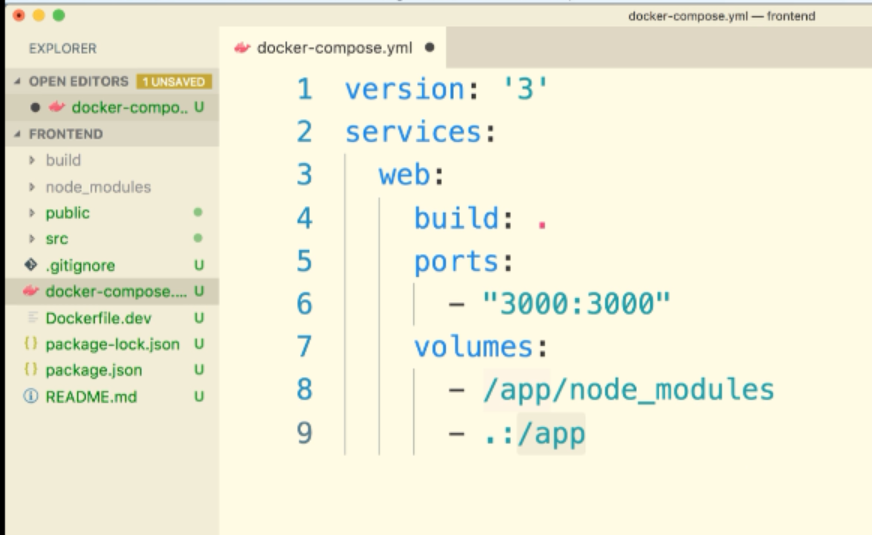
Make sure you rebuild your containers after making this change with  docker-compose down && docker-compose up --build

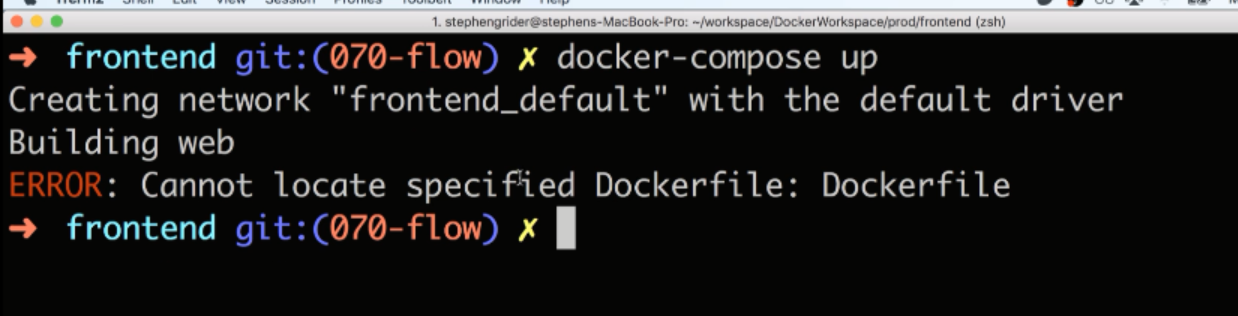
<https://github.com/facebook/create-react-app/issues/8688>

<https://stackoverflow.com/questions/60790696/react-scripts-start-exiting-in-docker-foreground-cmd>

V—77:(Shorthand with Docker Compose):

The purpose of docker compose is to make executing docker run command easier.



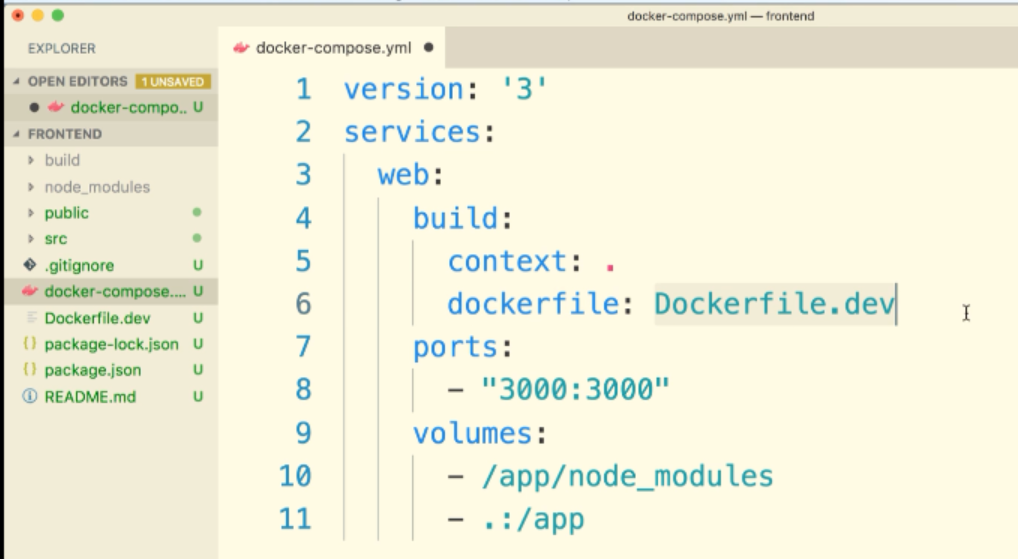


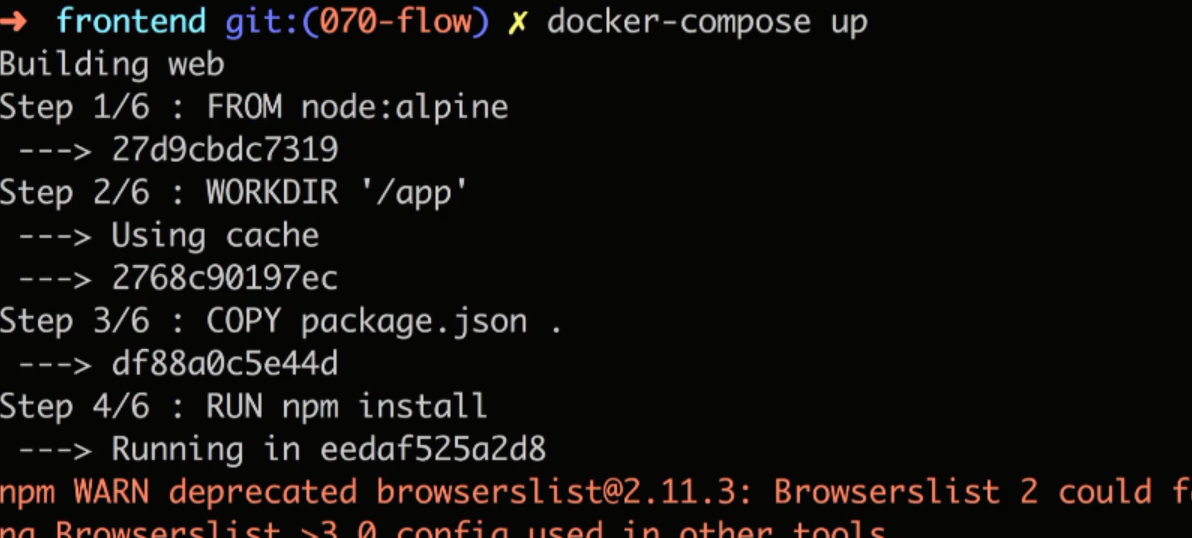
We don’t have Dockerfile we have Dockerfile.dev

V—78:(Overriding docker file Selection):

Context specifies where we want all the fiels and folders for this image to be pulled from.

We want them come from the same location where docker-compose present.





V—79:(Windows not detecting changes):

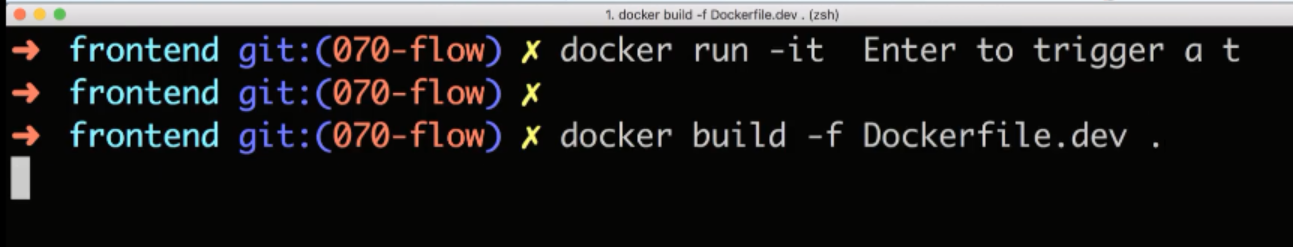
If you are using any version of Windows and your React app is not automatically reloading after a code change, you can add this environment variable to your compose file:

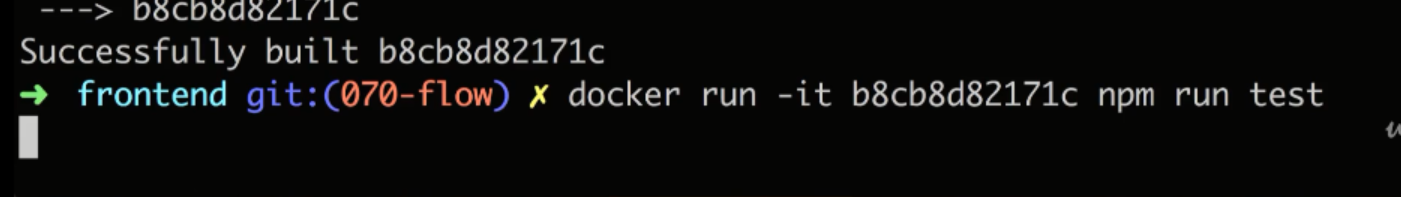
services:  
 web:  
 environment:  
 - CHOKIDAR\_USEPOLLING=true

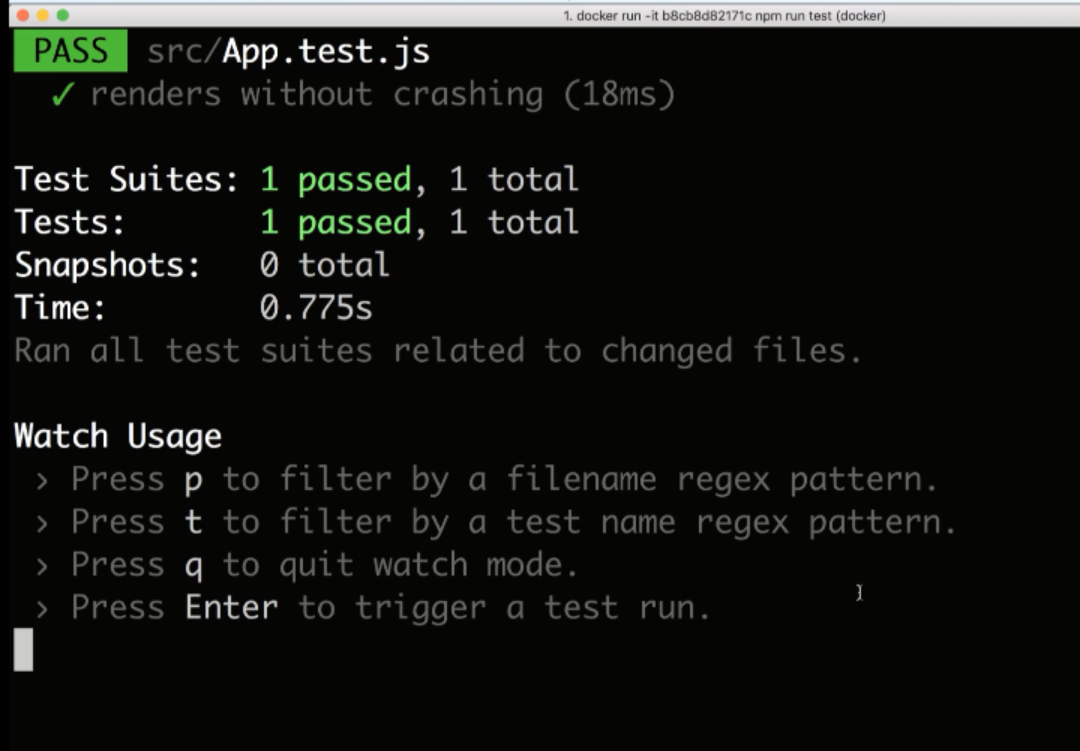
V—80:(Do We Need Copy):

Yes

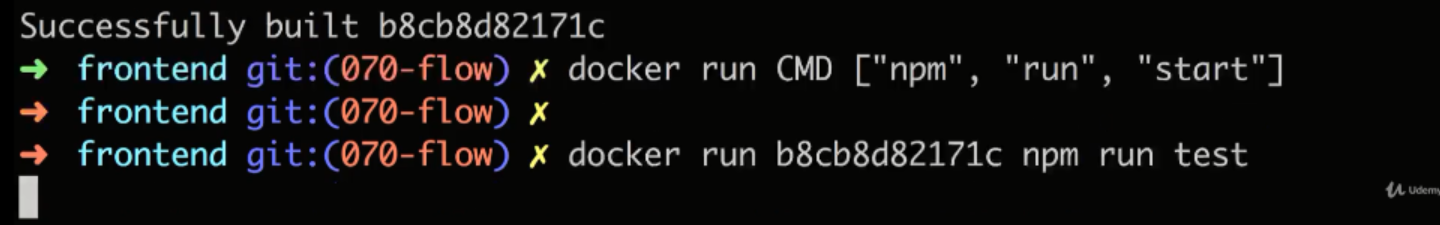
V—81:(Executing Tests):

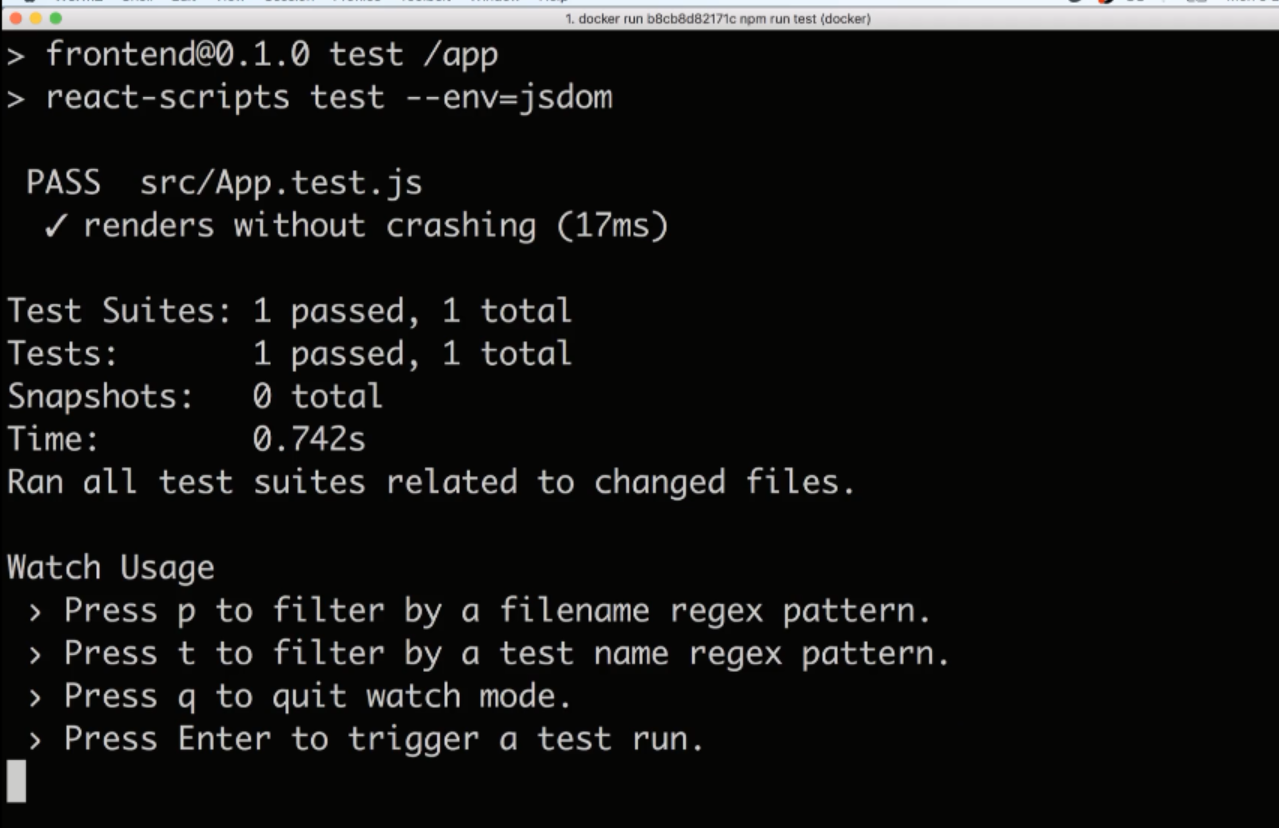






We can run in this way also, but it is not much more good to see and work

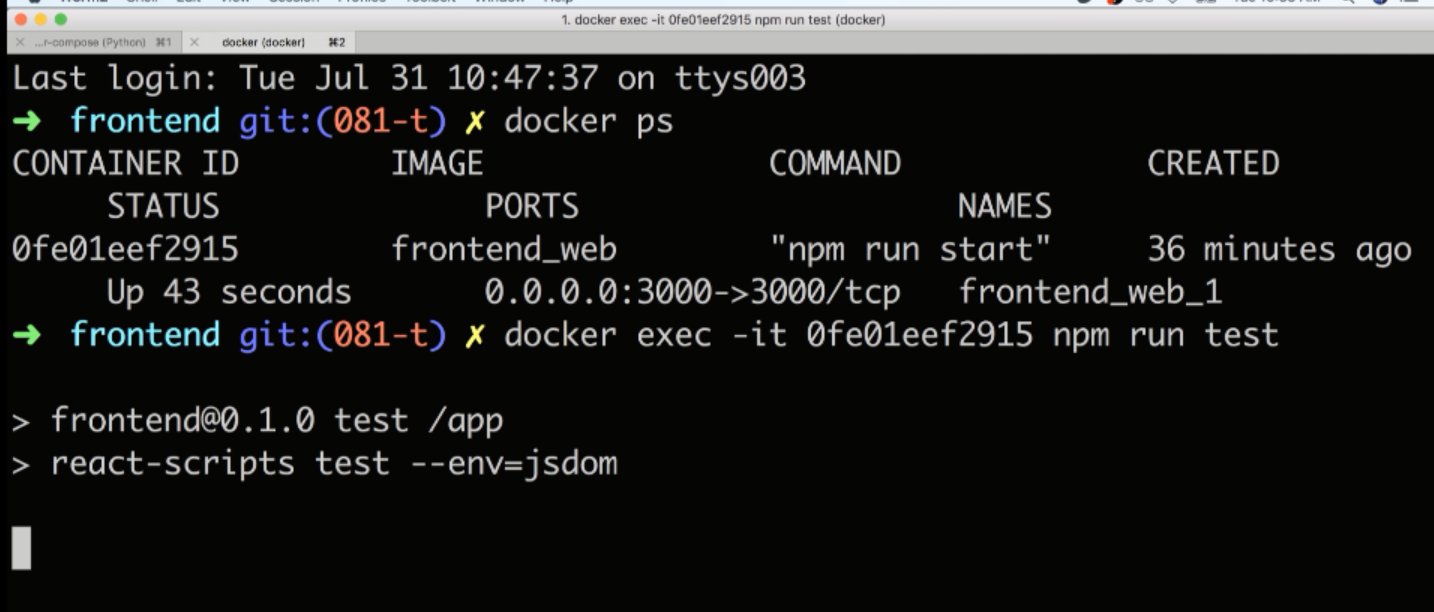




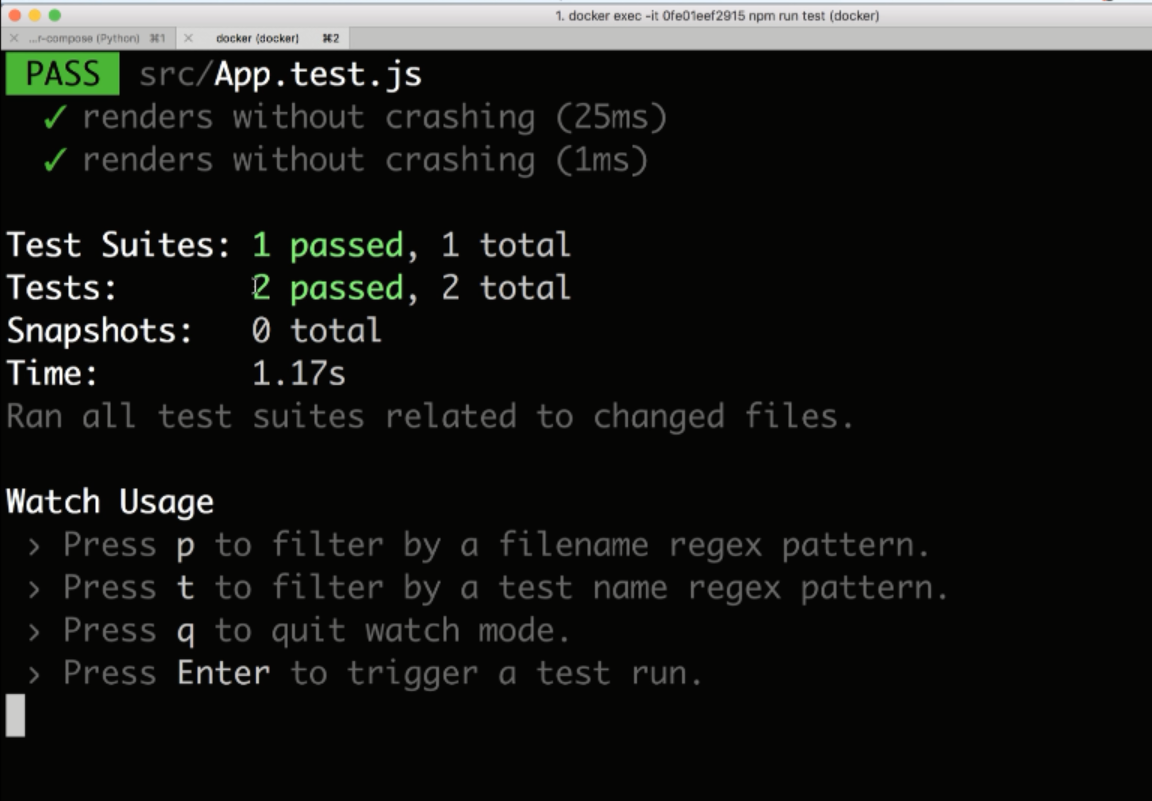
Above Enter, q,t,p will not work. They work if we add -it flag

V—82:(Live Updating Tests):

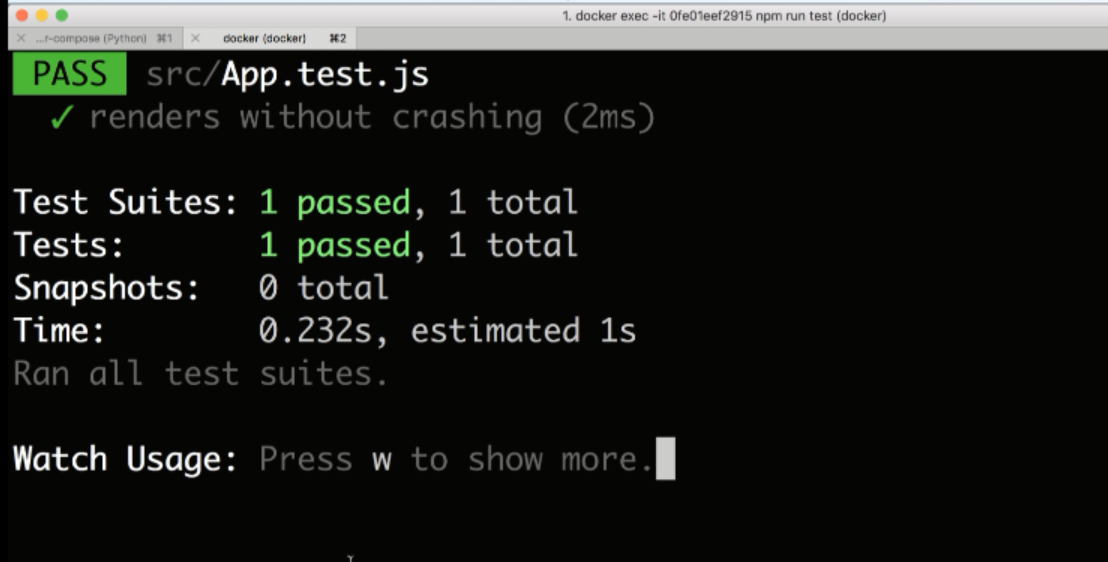




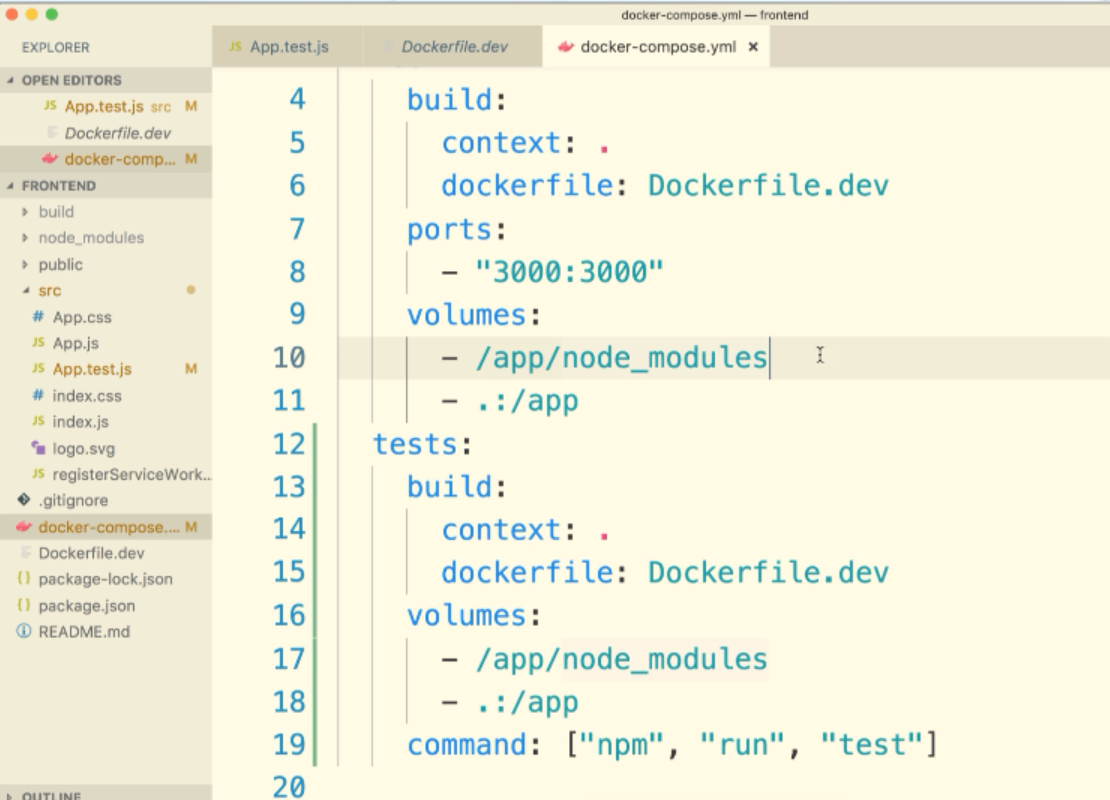
Here we have two test cases so

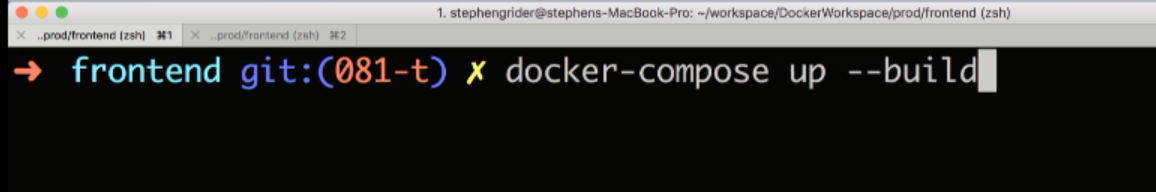


Now we have deleted one test case from App.test.js file, then open this cmd then it will be changed automatically like this



V—83:(Docker compose for running Tests):





After adding one more test in app.test.js. It will automatically gets reloaded.

V—84:(Tests not running in windows):

If you are using Windows Home you may have noticed that when adding a test or making a change to the App.test.js the tests are not re-running inside the container.

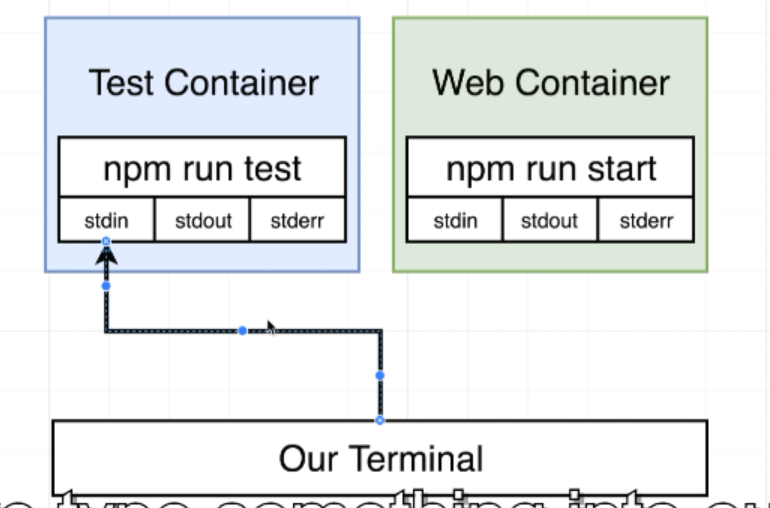
While this works on macOS (and likely Linux), Jest watchers seem to be completely broken on certain versions of Windows. We are looking into a potential fix or hack to get this working again and will update this note if we find one.

**Note** - Since the latest release of Create React App and Docker Desktop v2.2 this does not appear to be an issue for Windows Pro or Enterprise users.

V—86:(Shortcomings on Testing):

Dockar attac does:

We are attaching the stdin, stdout, stdout of the primary process inside that container.

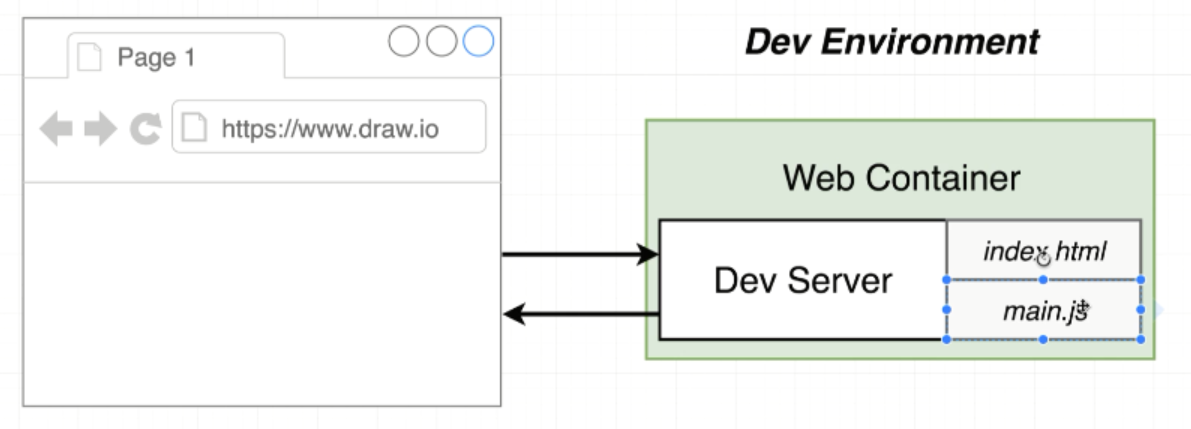


If we type something in our terminal, it will be sent to stdin on that primary process.

***Revisit – 86***

V—87:(Need for Nginix):

In normal environment:

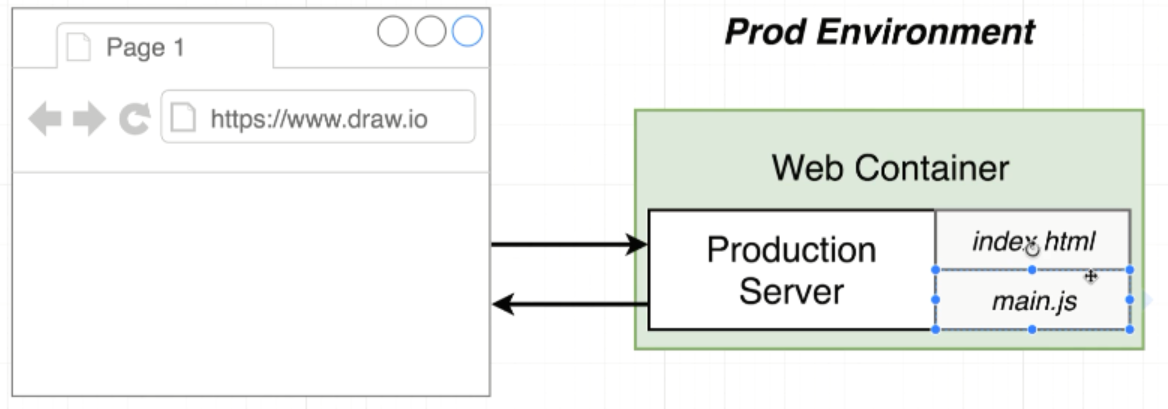


In production environment:

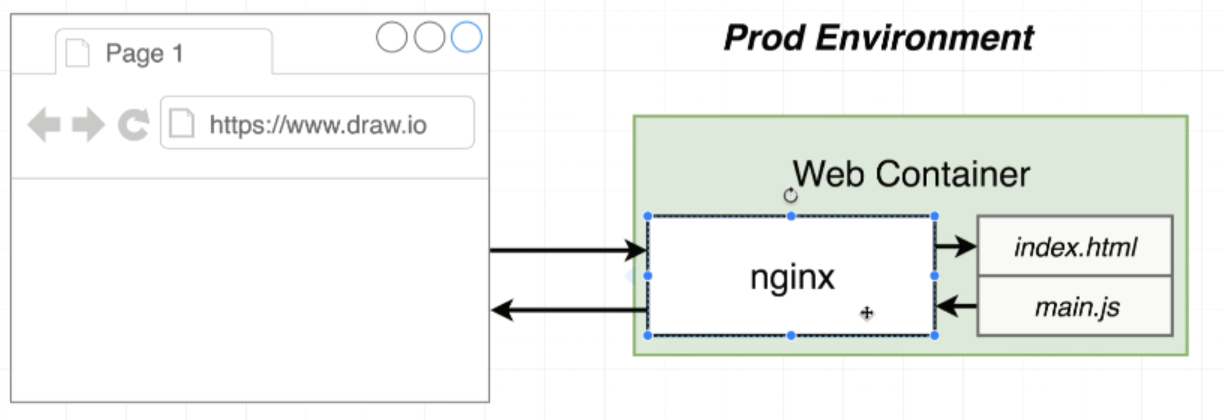
Production environment means

After giving npm run build command npm will take all javascripts and process them all together and make all of them as a single file and spit it in to the folder.

In production environment, dev server will not be present.



For production environment we need some type of server, that sole purpose is going to respond to browser requests with some index.html file and random js file that contains all the react-application code.

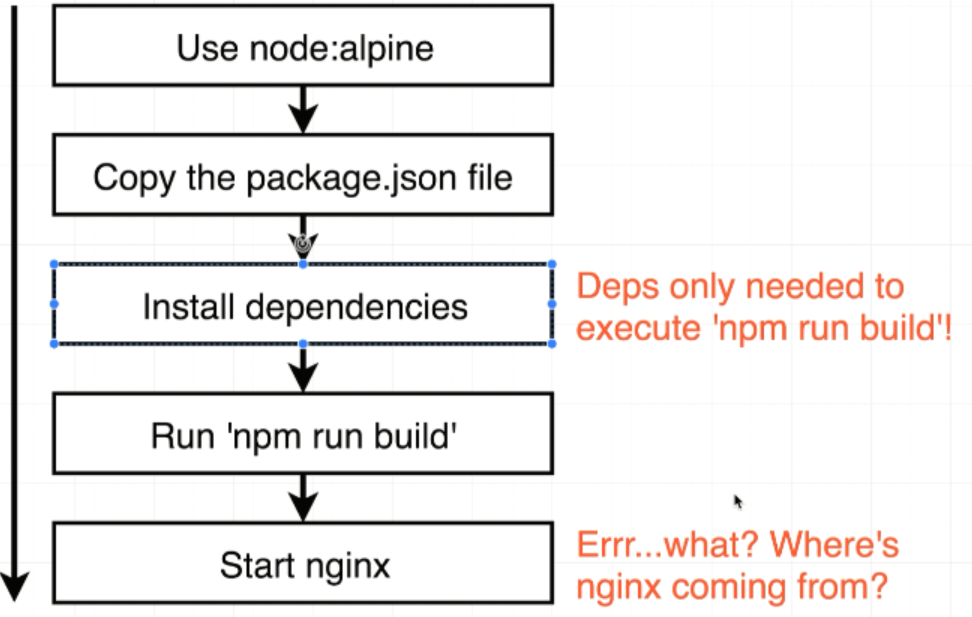


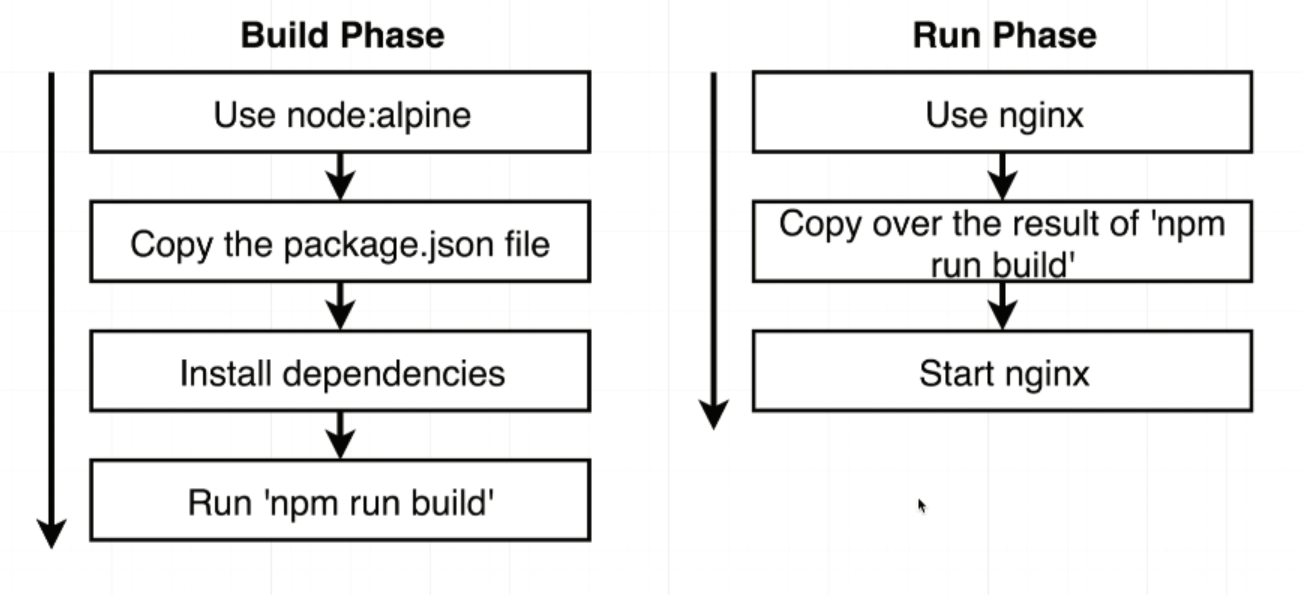
Nginix is extremely popular web server. It takes incoming traffic and route it and respond with some static files.

We are going to create a separate docker file that is going to create a production version of our web container. This production version of web container is gonna start up an nginx instance.

V—88:(Multi step docker Builds):







V—89:(Named Builders and AWS):

In the next lecture, we will be creating a multi-step build in our production Dockerfile. AWS currently will fail if you attempt to use a named builder as shown.

To remedy this, we should create an unnamed builder like so:

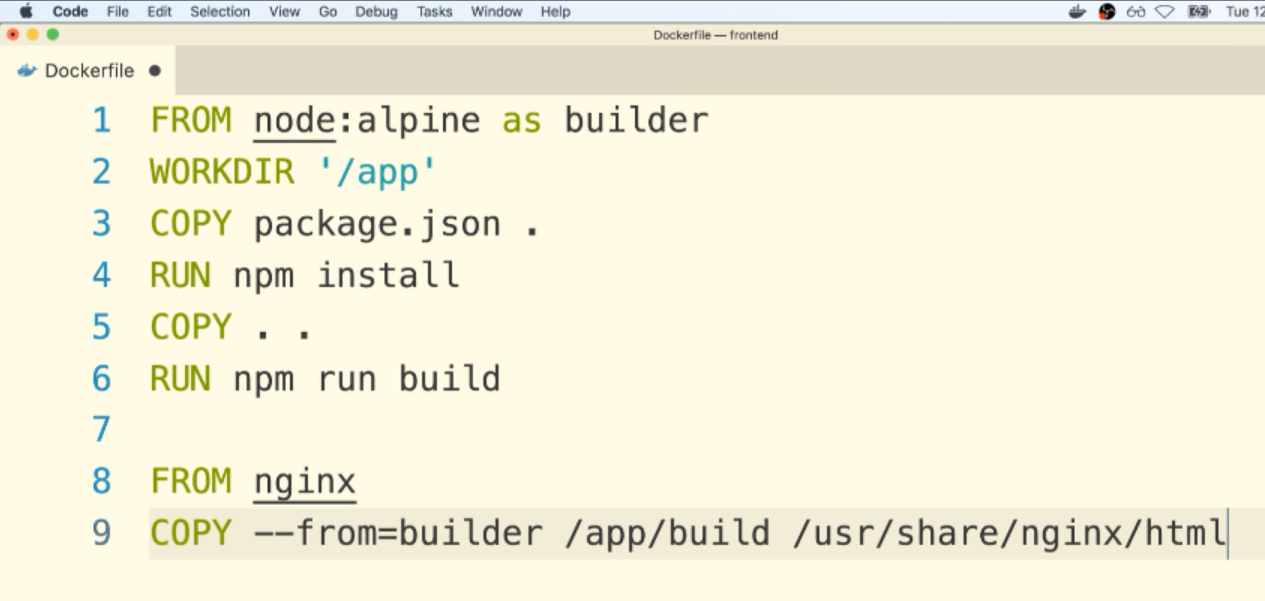
Instead of this:

FROM node:alpine as builder  
WORKDIR '/app'  
COPY package.json .  
RUN npm install  
COPY . .  
RUN npm run build  
FROM nginx  
COPY --from=builder /app/build /usr/share/nginx/html

Do this:

FROM node:alpine  
WORKDIR '/app'  
COPY package.json .  
RUN npm install  
COPY . .  
RUN npm run build  
FROM nginx  
COPY --from=0 /app/build /usr/share/nginx/html

V—90:(Implementing Multi-step Builds):



V—91:(Running Nginx):

