

# Contenerización de aplicaciones web (React)

Fase	Semana	Día	Lección
6 - Bases de Datos	3	5	1-3

# Objetivos

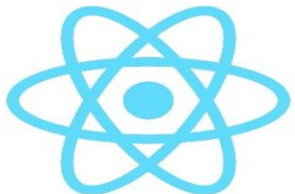
- Comprender la contenerización de una aplicación desarrollada en React
- Crear una imagen de docker que nos permita correr un contenedor para nuestra aplicación React
- Levantar/Correr nuestra aplicación de manera local usando docker compose
- Crear una nueva imagen para ambiente de producción usando NGINX para webserver para servir archivos estáticos.

# Creación de imagen docker y contenedor para nuestra aplicación React

# Contenerización de una aplicación desarrollada en React

Para dockerizar una aplicación desarrollada en React es necesario contar con:

1. Aplicación React existente (o [nueva](#))
1. Una imagen base que tenga instalado [Node.js](#) (necesario para levantar React apps)
1. Construir la imagen
1. Ejecutar la imagen (crear contenedor)



# Paso 1: Crear aplicación React básica

Crear aplicación llamada **react-docker** usando create-react-app (CRA <https://create-react-app.dev/>)

```
npx create-react-app react-docker
```

Ingresar al directorio e instalar dependencias

```
cd react-docker  
npm install
```

Iniciar la aplicación para asegurarnos que todo esté funcionando

```
npm start
```

Esto correrá el servidor local y abrirá nuestra aplicación en nuestro buscador en la dirección <http://localhost:3000/>

## Paso 2: Crear imagen de docker para nuestra aplicación React

En este paso crearemos una imagen de docker que nos funcione para levantar nuestra aplicación React.

**NOTA:** Esta primer imagen será para desarrollar de manera local y no se recomienda para ser utilizada en ambiente de producción.

### Primer etapa: Configuración

En esta etapa nos enfocaremos en definir nuestra imagen base. Como sabemos para utilizar React es necesario tener Node.js instalado en nuestro sistema por lo que lo primero en nuestra imagen debe de ser tomar una imagen de Node como nuestra base.

Podemos revisar en docker hub las versiones actuales y tomar la más reciente en caso de un proyecto nuevo o buscar la versión que nuestra aplicación ya está utilizando.

[https://hub.docker.com/\\_/node/](https://hub.docker.com/_/node/)

## Paso 2: Crear imagen de docker para nuestra aplicación React

Vamos a crear nuestro archivo Dockerfile en el root de nuestra aplicación y agregar la siguiente línea

```
🐳 Dockerfile > ...  
1  # Indica la imagen base para nuestra nueva imagen  
2  FROM node:20-alpine
```

**NOTA:** Usaremos la versión **alpine** ya que es una versión más “liviana/light” de node 20. Considerar que para aplicaciones complejas puede esta versión de node puede no tenga todas las funcionalidades necesarias

Ahora necesitamos definir el nombre de nuestro workspace (el directorio dentro del contenedor sobre el que se comenzará a trabajar de ahora en adelante)

```
4  # Directorio de trabajo dentro del contenedor  
5  WORKDIR /app
```

## Paso 2: Crear imagen de docker para nuestra aplicación React

Lo siguiente sería copiar nuestros archivos de dependencias del proyecto

```
7  # Copiamos los archivo package.json y package-lock.json a la carpeta /app
8  COPY package*.json ./
```

Una vez tenemos estos archivos dentro de /app (al igual que cuando creamos la nueva aplicación) debemos de instalar las dependencias

```
10  # Instalar dependencias
11  RUN npm install
```



## Paso 2: Crear imagen de docker para nuestra aplicación React

Una vez instaladas las dependencias podemos copiar los demás archivos de nuestra aplicación

```
# Copiar el resto de los archivos  
COPY . /app
```

**NOTA:** Cuando creamos una imagen queremos que esta tenga únicamente los archivos necesarios, por lo que se recomienda omitir aquellos archivos que no son necesarios o que necesitamos se generen de nuevo. Para esto podemos crear un archivo **.dockerignore** (en el mismo directorio donde tenemos el Dockerfile) e incluir lo siguiente:

```
📁 .dockerignore  
1  .dockerignore  
2  node_modules  
3  **/node_modules  
4  **/.git  
5  **/.DS_Store
```

## Paso 2: Crear imagen de docker para nuestra aplicación React

Los últimos dos pasos serían:


Exponer el puerto en el que vamos a correr nuestra aplicación

```
16  # Exponer el puerto 3000
17  EXPOSE 3000
```

Ejecutar la aplicación (el mismo comando que corremos de manera local para correr la aplicación React)

```
19  # Comando para ejecutar la aplicación
20  CMD ["npm", "start"]
```

## Paso 2: Versión final imagen de Docker (Dev)

 Dockerfile > ...

```
1  # Indica la imagen base para nuestra nueva imagen
2  FROM node:20-alpine
3
4  # Directorio de trabajo dentro del contenedor
5  WORKDIR /app
6
7  # Copiamos los archivo package.json y package-lock.json a la carpeta /app
8  COPY package*.json ./
9
10 # Instalar dependencias
11 RUN npm install
12
13 # Copiar el resto de los archivos
14 COPY . /app
15
16 # Exponer el puerto 3000
17 EXPOSE 3000
18
19 # Comando para ejecutar la aplicación
20 CMD ["npm", "start"]
```

## Paso 3: Construir nuestra imagen

Una vez tenemos nuestro Dockerfile listo con la “receta” para generar/construir nuestra imagen de docker lo que debemos de hacer es correr el siguiente comando:

```
docker build --tag nombre_de_imagen .
```

Explicación del comando:

- **docker build:** Indica a docker que vamos a construir una imagen
- **--tag:** Opción que indica a docker el nombre (y opcionalmente el tag) que vamos a poner a la imagen
- **nombre\_de\_imagen:** Nombre que queremos poner a nuestra imagen (no debe de existir anteriormente en nuestra computadora una imagen con el mismo nombre)
- El punto final indica a Docker donde se encuentra el Dockerfile a partir del cual queremos se construya nuestra nueva imagen

## Paso 3: Construir nuestra imagen

Vamos a crear nuestra imagen llamada **react-docker** y que tenga la versión/tag 1:

```
docker build --tag "react-docker:1" .
```

Al correr este comando se va a cargar el `.dockerfile` para asegurarse que docker ignore lo que anotamos anteriormente, leer lo anotado en el Dockerfile e iniciará a correr todos los comandos escritos en él como: usar imagen node, instalar dependencias, ...

# Paso 3: Construir nuestra imagen

```
react-docker|main ⚡ → docker build -t "react-docker:1" .  
[+] Building 37.8s (11/11) FINISHED  
=> [internal] load .dockerignore  
=> => transferring context: 1048  
=> [internal] load build definition from Dockerfile  
=> => transferring dockerfile: 1.17kB  
=> [internal] load metadata for docker.io/library/node:20-alpine  
=> [auth] library/node:pull token for registry-1.docker.io  
=> [1/5] FROM docker.io/library/node:20-alpine@sha256:002b6ee25b63b81dc4e47c9378ffe20915c3fa0e98e834c46584438468b1d0b5  
=> => resolve docker.io/library/node:20-alpine@sha256:002b6ee25b63b81dc4e47c9378ffe20915c3fa0e98e834c46584438468b1d0b5  
=> => sha256:8b8b60d56fb87c6d10ba362d45c153e8de47dcbe6463f9cc5343b11bae00f676 49.81MB / 49.81MB  
=> => sha256:97f8dfa93eef338011688eb86e78e5fb5f6feab4c24c3ca31a7853e832e0bcef 2.34MB / 2.34MB  
=> => sha256:002b6ee25b63b81dc4e47c9378ffe20915c3fa0e98e834c46584438468b1d0b5 1.43kB / 1.43kB  
=> => sha256:1ccb0c0ded3b21cee95fe6b6ce1ac23bd6680c8f152cbfb3047d5d9ea490b098 1.16kB / 1.16kB  
=> => sha256:cf2316e995eb236a3d42066d396685efb1333bd540aface0a9bfc4ff29ce030f 6.78kB / 6.78kB  
=> => sha256:96526aa774ef0126ad0fe9e9a95764c5fc37f409ab9e97021e7b4775d82bf6fa 3.40MB / 3.40MB  
=> => extracting sha256:96526aa774ef0126ad0fe9e9a95764c5fc37f409ab9e97021e7b4775d82bf6fa  
=> => sha256:aaa59b7b85b65010878890781e909cb3345c930a9942bbf8b319825c2a670236 452B / 452B  
=> => extracting sha256:8b8b60d56fb87c6d10ba362d45c153e8de47dcbe6463f9cc5343b11bae00f676  
=> => extracting sha256:97f8dfa93eef338011688eb86e78e5fb5f6feab4c24c3ca31a7853e832e0bcef  
=> => extracting sha256:aaa59b7b85b65010878890781e909cb3345c930a9942bbf8b319825c2a670236  
=> [internal] load build context  
=> => transferring context: 1.85kB  
=> [2/5] WORKDIR /app  
=> [3/5] COPY package*.json ./  
=> [4/5] RUN npm install  
=> [5/5] COPY . /app  
=> exporting to image  
=> => exporting layers  
=> => writing image sha256:0202397ce6d882b700d70950d64ce483eebfc73b50271fe40205385e7d69e64d  
=> => naming to docker.io/library/react-docker:1
```

## Paso 3: Construir nuestra imagen

Al terminar podemos listar nuestra imagen corriendo el comando

```
docker images react-docker
```

Esto nos mostrará información relevante como el ID de la imagen, tag y el tamaño de la imagen

```
react-docker|main ⚡ ⇒ docker images react-docker
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
react-docker	1	0202397ce6d8	6 minutes ago	485MB

## Paso 4: Ejecutar la imagen (crear contenedor)

Una vez tenemos nuestro Dockerfile listo con la “receta” para generar/construir nuestra imagen de docker lo que debemos de hacer es correr el siguiente comando:

```
docker run -p 3000:3000 nombre_de_imagen:tag
```

Explicación del comando:

- **docker run:** Indica a Docker que vamos iniciar/correr un contenedor
- **-p:** Le indicamos a Docker el puerto en el que nos vamos a poder conectar con la aplicación desde nuestra computadora y el puerto expuesto dentro del contenedor (el indicado en el Dockerfile 3000)
- **nombre\_de\_imagen:** Nombre de la imagen que utilizaremos para crear nuestro contenedor
  - Nota: si al crear la imagen indicamos un tag/versión, también debemos de indicar el tag al correrlo



## Paso 4: Ejecutar la imagen (crear contenedor)

Vamos a crear el contenedor basado en la imagen llamada **react-docker:1**

```
docker run -p 3000:3000 react-docker:1
```

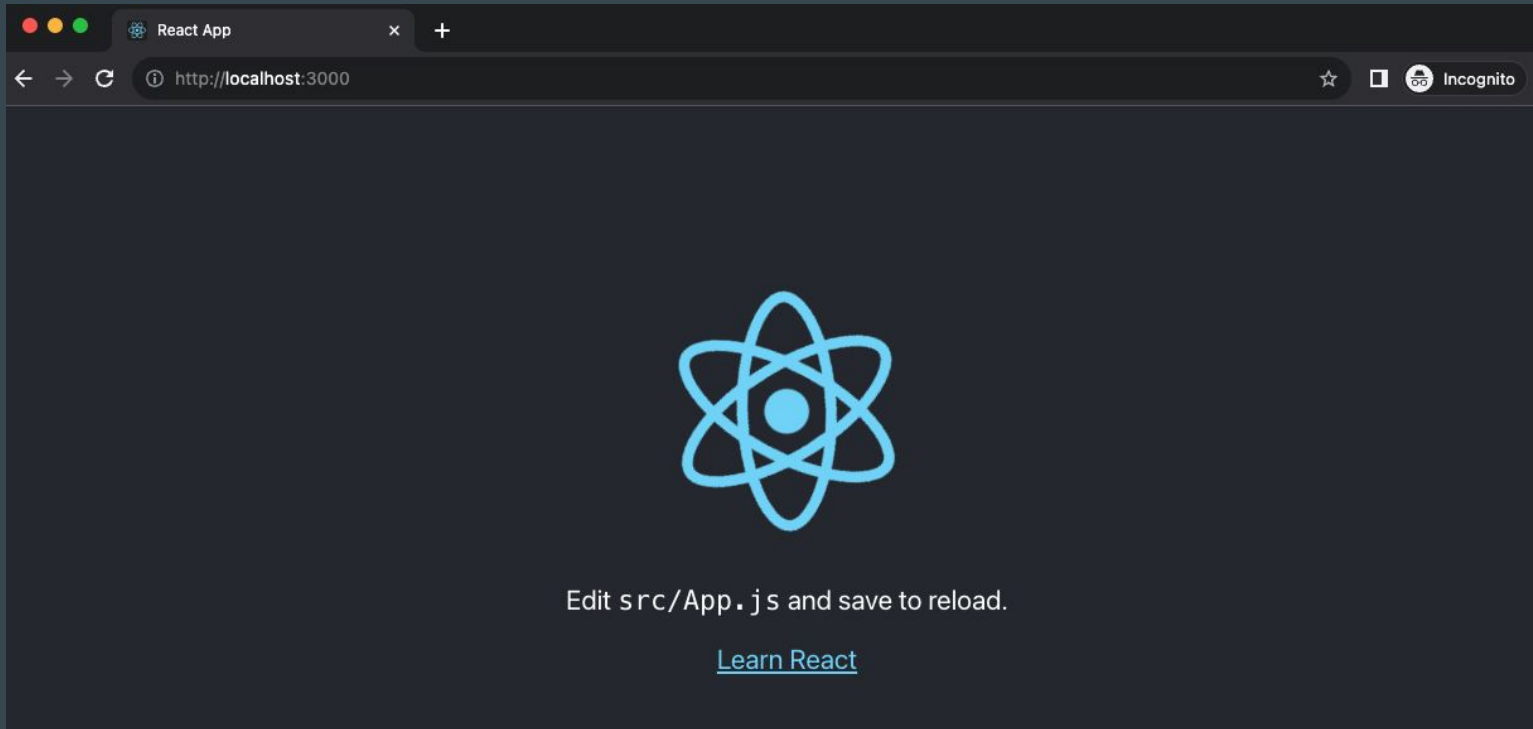
Esto iniciará nuestra aplicación desde un contenedor. Para comprobar que el contenedor está corriendo podemos correr **docker ps**

```
react-docker | main ⚡ → docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
7c2b32d479c2	react-docker:1	"docker-entrypoint.s..."	About a minute ago	Up About a minute	0.0.0.0:3000->3000/tcp	fervent_feynman

Una vez comprobamos esto, podemos ir al buscador e ingresar <http://localhost:3000> nuevamente para interactuar con la aplicación React

## Paso 4: Ejecutar la imagen (crear contenedor)



**Correr aplicación React usando Docker Compose**

# Configuración Docker Compose

Primeramente debemos crear un archivo llamado **docker-compose.yml** con la siguiente información:

- Versión del formato del archivo compose
- Lista de servicios (se utilizó **app** pero se puede usar el nombre que se desee)
- El nombre del contenedor (es opcional)
- La imagen que se utilizará para el contenedor
- Al hacer build/construir nuestro contenedor se puede indicar el contexto, ambiente, ...
- El puerto donde se va a exponer la aplicación

```
🔥 docker-compose.yml
1  version: "3.8"
2
3  services:
4    app:
5      container_name: react-docker
6      image: react-docker
7      build:
8        context: .
9      ports:
10       - 3000:3000
```

# Correr contenedor con Docker Compose

Ahora para levantar nuestra aplicación desde un contenedor usando docker compose debemos de correr únicamente el comando `docker-compose up`

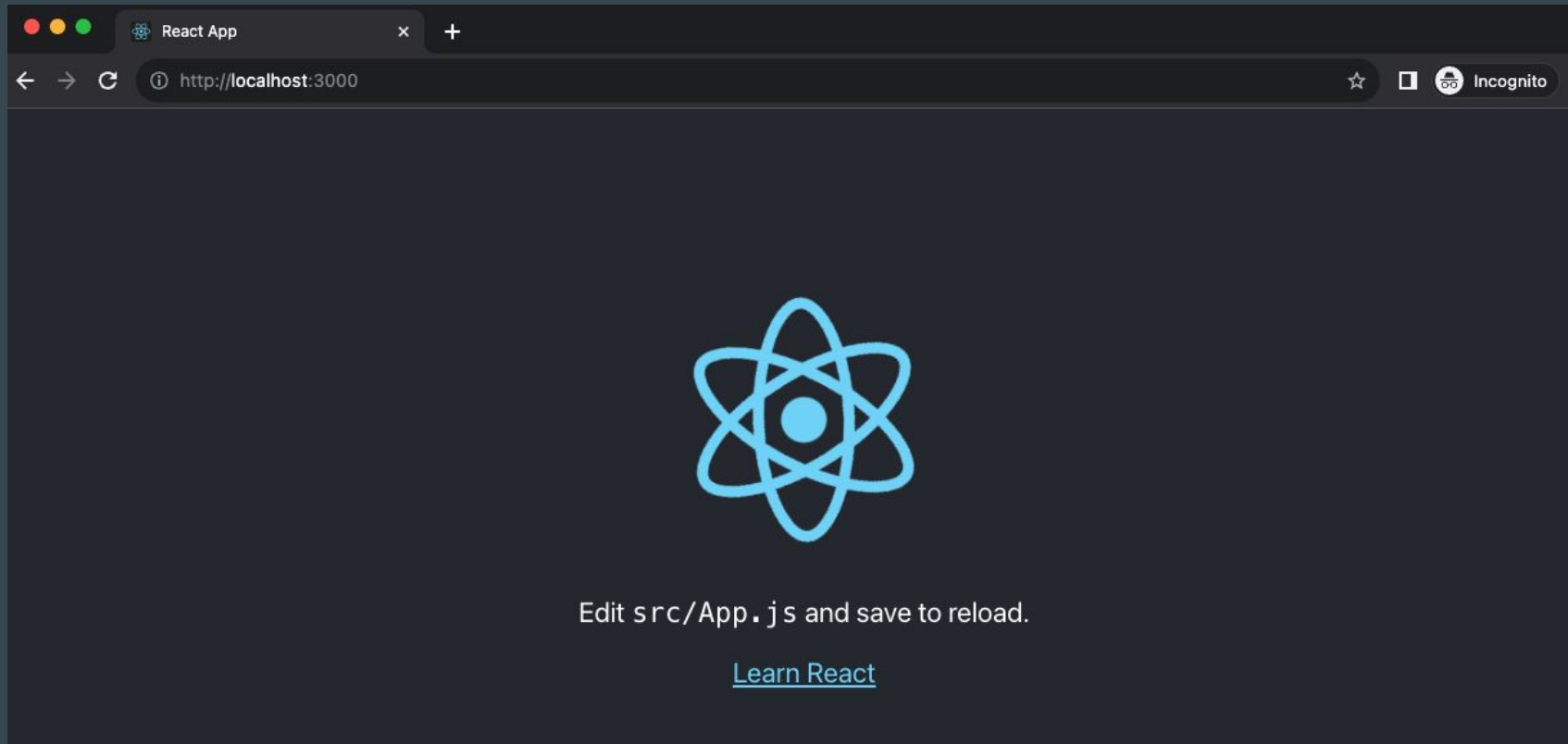
```
react-docker | main ⚡ → docker-compose up
[+] Building 0.0s (0/0)
[+] Running 1/0
✓ Container react-docker Created
Attaching to react-docker
react-docker |
react-docker | > react-docker@0.1.0 start
react-docker | > react-scripts start
```

Esto iniciará nuestra aplicación desde un contenedor. Para comprobar que el contenedor está corriendo podemos correr `docker ps`

```
react-docker | main ⚡ → docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
2aa53e499772	react-docker	"docker-entrypoint.s..."	9 minutes ago	Up About a minute	0.0.0.0:3000->3000/tcp	react-docker

Una vez comprobamos esto, podemos ir al buscador e ingresar <http://localhost:3000> nuevamente para interactuar con la aplicación React



# Creación de imagen docker para producción

# Imagen de Docker para Aplicación React en Producción

Para generar una imagen optimizada y lista para utilizarse en producción se deben de considerar aspectos como:

- Instalación de dependencias de la aplicación en modo producción
- Construir la aplicación en modo producción para optimización de archivos y generar archivos estáticos (html, css, js)
- “Alguien”/Programa que nos ayude a servir los archivos estáticos generados.
  - Al usar React (o cualquier otro framework Single Page Application que utilice lógica de ruteo) necesitamos un servidor que nos ayude a ejecutar este comportamiento con archivos estáticos.
  - En nuestro caso vamos a hacer uso de [NGINX](#)



# Qué es NGINX?

Es un software de código abierto para servicio web , proxy inverso, almacenamiento en caché, equilibrio de carga, transmisión de medios entre otros. Comenzó como un servidor web diseñado para un máximo rendimiento y estabilidad. Además de sus capacidades de servidor HTTP, NGINX también puede funcionar como un servidor proxy para correo electrónico (IMAP, POP3 y SMTP) y un proxy inverso y equilibrador de carga para servidores HTTP, TCP y UDP.

Sitio Nginx: <https://www.nginx.com/>

Documentación: <https://nginx.org/en/docs/>



# Configuración de NGINX

Para nuestra aplicación de React en producción necesitamos agregar la configuración de NGINX.

Primero vamos a crear un archivo llamado **nginx.conf** el cual vamos a crear dentro de un directorio llamado **nginx** en nuestro proyecto.

El archivo deberá de contener el siguiente código.

```
nginx > ≡ nginx.conf
1  server {
2      listen 80;
3
4      location / {
5          root /usr/share/nginx/html;
6          include /etc/nginx/mime.types;
7          try_files $uri $uri/ /index.html;
8      }
9  }
```

# Dockerfile para Producción

Vamos a crear un archivo llamado **Dockerfile.prod**

Este nuevo archivo va a tener un código distinto al que creamos anteriormente ya que vamos a incluir instrucciones para crear la imagen en múltiples etapas.

Documentación: <https://docs.docker.com/build/building/multi-stage/>

Para nuestra aplicación vamos a necesitar dos etapas:

- Etapa 1: Nos enfocamos en generar los archivos estáticos de nuestra aplicación (modo producción)
- Etapa 2: “Servimos” nuestros archivos con ayuda de NGINX en el puerto indicado en la configuración.

# Etapas/Stage 1

En esta etapa nos vamos a enfocar en generar los archivos estáticos de nuestra aplicación.

```
Dockerfile.prod > ...
1  # Indica la imagen base para la primer etapa
2  # A esta etapa le daremos el nombre de builder
3  FROM node:20-alpine AS builder
4
5  # Directorio de trabajo dentro del contenedor
6  WORKDIR /app
7
8  # Copiamos los archivos package.json y package-lock.json a la carpeta /app
9  COPY package*.json ./
10
11 # Instalar dependencias
12 RUN npm install --production
13
14 # Copiar el resto de los archivos
15 COPY . /app
16
17 # Comando hacer build de la aplicación
18 # Se generan los archivos estáticos dentro de app/build
19 RUN npm run build
```

# Etapa/Stage 1

**NOTA:** Algo importante a considerar es que al hacer build de nuestra aplicación (en nuestro ambiente local) se crearán los archivos estáticos dentro de un directorio llamado **/build** en el root de nuestro proyecto. Al igual que el directorio `node_modules`, este otro lo queremos omitir al generar la imagen por lo que debemos de agregarlo al archivo **.dockerignore**

```
👉 .dockerignore
1  .dockerignore
2  node_modules
3  **/node_modules
4  **/.git
5  **/.DS_Store
6  build
```

## Etapas/Stage 2

En esta etapa configuramos NGINX para la etapa 2 y final para nuestra imagen de producción

🔗 Dockerfile.prod > ...

```
21  # Segunda etapa/capa
22  # Indica la imagen base para la segunda etapa
23  FROM nginx:1.24-alpine AS production
24
25  # Copiar los archivos estáticos de la primer etapa
26  # a la carpeta /usr/share/nginx/html de donde NGINX
27  # tomará y servirá los archivos estáticos.
28  COPY --from=builder /app/build /usr/share/nginx/html
29
30  # Sobreescrir el archivo de configuración default de NGINX
31  # con nuestra configuración personalizada
32  COPY ./nginx/nginx.conf /etc/nginx/conf.d/default.conf
33
34  # Exponer el puerto 80
35  EXPOSE 80
36
37  # Comando para iniciar NGINX
38  CMD ["nginx", "-g", "daemon off;"]
```

# Construir Imagen para Producción

Creamos la nueva imagen

```
docker build -f ./Dockerfile.prod --tag react-docker:prod .
```

Listamos las imágenes para revisar que se creó

```
docker images react-docker
```

```
react-docker | main ⚡ ⇒ docker images react-docker
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
react-docker	prod	c423e9b326d6	5 minutes ago	46.5MB
react-docker	latest	cbaf4ed5a29f	About an hour ago	485MB
react-docker	1	0202397ce6d8	About an hour ago	485MB

Nótese la diferencia de tamaño entre la primer imagen del ambiente dev (485MB) y la de producción (46.5MB)

# Ejecutar Imagen de Producción (Container)

Corremos la imagen que creamos para producción

```
docker run -p 80:80 react-docker:prod
```

```
react-docker|main ⚡ ⇒ docker run -p 80:80 react-docker:prod
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform
configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.
sh
```

Listamos los contenedores corriendo con `docker ps`

```
react-docker|main ⚡ ⇒ docker ps
```

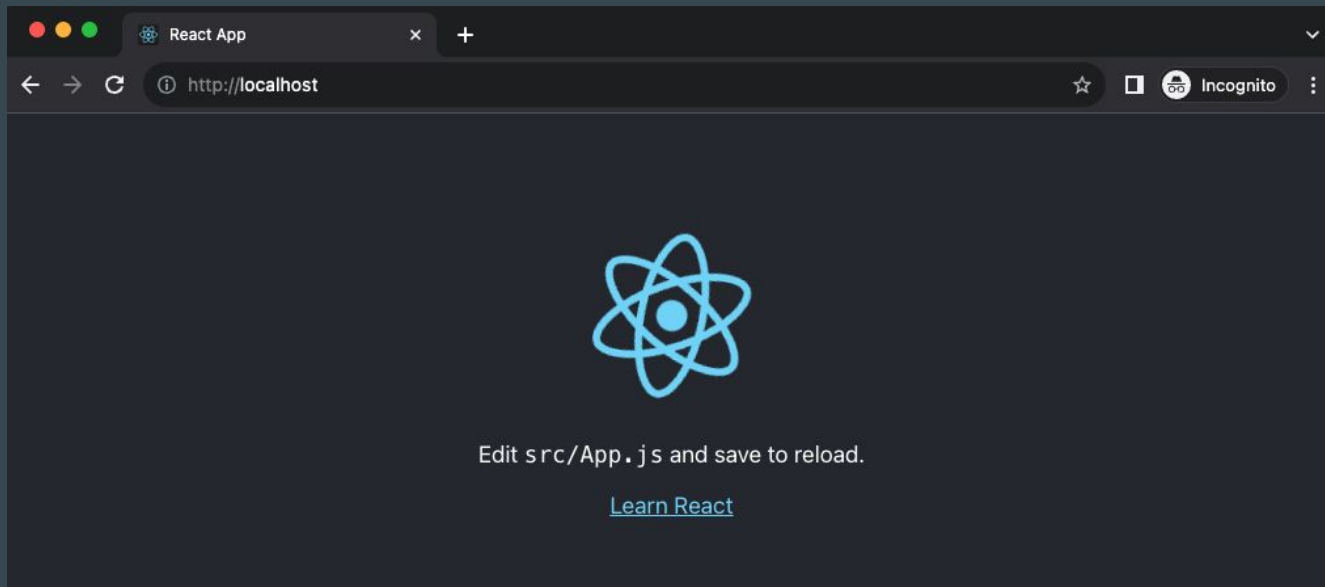
CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
ef4dfaa0d45a	react-docker:prod	"/docker-entrypoint...."	12 seconds ago	Up 12 seconds	0.0.0.0:80->80/tcp	intelligent_poitras



# Ejecutar Imagen de Producción (Container)

Podemos revisar que la aplicación funciona según lo esperado ingresando en el navegador a <http://localhost/>

**Nota:** Nótese que no se indica ningún puerto en la URL ya que en la configuración de NGINX y al correr el contenedor indicamos que ibamos a usar el 80, el cual es el default en el navegador.



# Docker Compose para Producción

Vamos a crear un archivo llamado `docker-compose-prod.yml` con el siguiente código:

**NOTA:** Este nuevo archivo no es totalmente necesario aunque se puede crear para revisar que nuestra nueva imagen y que la interacción entre varios servicios funciona según lo esperado.

```
docker-compose-prod.yml
1  version: "3.8"
2
3  services:
4    app:
5      container_name: react-docker-prod
6      image: react-docker:prod
7      build:
8        context: .
9        target: production
10       args:
11         - NODE_ENV=production
12       dockerfile: Dockerfile.prod
13     ports:
14       - 80:80
```

# Correr Docker Compose de Producción

Comandos de docker-compose para nuevo archivo:

- Hacer build de la imagen desde el docker compose  
`docker-compose -f docker-compose-prod.yml build`
- Correr el contenedor desde el docker-compose  
`docker-compose -f docker-compose-prod.yml up`

```
react-docker|main ⚡ → docker-compose -f docker-compose-prod.yml up

[+] Building 0.0s (0/0)
[+] Running 1/1
✓ Container react-docker Recreated
Attaching to react-docker-prod
react-docker-prod | /docker-entrypoint.sh: /docker-entrypoint.d/ i
```