DESPLIEGUE EN CONTENEDORES

Juan Pablo Gómez Triana[1]

Resumen

Se busca desarrollar habilidades, así como poner en práctica, los conceptos de virtualización y contenedores para el despliegue de aplicaciones seguras que utilizan el protocolo https, a la vez que se usa una arquitectura basada en Cloud Computing. Se utilizan tecnologías como Docker y AWS para poner en práctica dichos conceptos. También se utiliza un despliegue de un CMS como lo es WordPress y de una aplicación que utiliza el stack MERN. Para el aseguramiento de las aplicaciones se utiliza la Certificate Authority (CA) Let's Encrypt.

Palabras Clave: Virtualización, Contenedores, HTTPS, Cloud Computing, Docker, AWS, CMS, WordPress, MERN, Certificate Authority, Let's Encrypt.

Introducción

En este documento se numeran los pasos requeridos para desplegar ambas aplicaciones planteadas en los laboratorios 1 y 2 de la unidad 2.

[1] Estudiante de Ingeniería de Sistemas de la Universidad EAFIT, Medellín, Antioquia. E-mail: jpgomezt@eafit.edu.co

Laboratorio 1: CMS

En esta sección mostraremos los pasos que fueron requeridos para la elaboración del laboratorio 1.
Por medio de este enlace, se podrá acceder al servicio web desplegado:
www.wordpress-pampa.tk

Instancia EC2 y Freenom

Para iniciar se instancio una EC2 con S.O Ubuntu, la cual posee los puertos 80 y 443 (http y https) abiertos. A esta instancia se le asocio una IP elástica para evitar que su IP publica cambie, y poderla asociar al DNS de Freenom como se muestra a continuación:

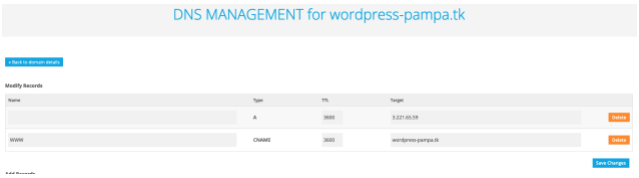

Figura 1. Configuración DNS Freenom.

Cabe resaltar que solo es necesario crear un registro A que apunte a nuestra instancia EC2, y un registro CNAME con el subdominio *"www"*.

Configuración EC2

Solo se requiere instalar Docker y Docker-Compose en esta instancia, ya que el demás software que se utilizaran será por medio de imágenes disponibles en Docker.
Para instalar Docker en Ubuntu seguimos los siguientes pasos:

```
$ sudo apt-get update

$ sudo apt-get install \
    ca-certificates \
    curl \
    gnupg \
    lsb-release
```

Esto es para configurar los repositorios de apt.
Luego agregamos la GPG key oficial de Docker:

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
```

Una vez tenemos la key configurada, podemos configurar el repositorio de Docker:

```
$ echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

Con esto ya Podemos instalar Docker en nuestra maquina Ubuntu:

```
$ sudo apt-get update

$ sudo apt-get install docker-ce docker-ce-cli containerd.io
```

Una vez instalado, podemos correr la aplicación de Docker:

```
$ sudo systemctl start docker

$ sudo systemctl enable docker
```

Finalmente, instalamos Docker-Compose:

```
$ sudo curl -L "https://github.com/docker/compose/releases/download/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

$ sudo chmod +x /usr/local/bin/docker-compose

$ sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose
```

Ya con esto tenemos tanto Docker como Docker-Compose instalado en nuestra instancia EC2.

Creación del archivo docker-compose.yml

Primero, seguimos la guía de laboratorio propuesta para crear los servicios de WordPress y MySQL,

utilizando las respectivas imágenes disponibles en Docker Hub:

```
version: '3.1'
services:
  wordpress:
    image: wordpress
    restart: always
    environment:
      WORDPRESS_DB_HOST: db
      WORDPRESS_DB_USER: exampleuser
      WORDPRESS_DB_PASSWORD: examplepass
      WORDPRESS_DB_NAME: exampledb
    volumes:
      - wordpress:/var/www/html
  db:
    image: mysql:5.7
    restart: always
    environment:
      MYSQL_DATABASE: exampledb
      MYSQL_USER: exampleuser
      MYSQL_PASSWORD: examplepass
      MYSQL_RANDOM_ROOT_PASSWORD: '1'
    volumes:
      - db:/var/lib/mysql
volumes:
  wordpress:
  db:
```

Cabe resaltar que se hace uso de los volúmenes, los cuales, según Docker, son el mecanismo preferido para conservar los datos generados y utilizados por los contenedores.

Con esto, ya tenemos el Compose listo para levanta las aplicaciones de WordPress y MySQL, y ya podríamos levantar los contenedores para tener nuestro servicio web. Ahora solo queda asegurar nuestro sitio web, utilizando el CA Let's Encrypt.

Dockerizando un servidor SSL con SWAG

Normalmente, tendríamos que generar un servidor SSL descargando Cerbot y configurando a Let's Encrypt como CA, pero ya que estamos utilizando Docker-Compose para configurar y levantar nuestros servicios, lo aprovecharemos para levantar nuestro servidor SSL. Para esto, utilizamos la imagen disponible en Docker Hub de SWAG (Secure Web Application Gateway). Este servicio configura un servidor web Nginx y un reverse proxy con un cliente Certbot integrado que automatiza los procesos de generación y renovación de certificados del servidor SSL utilizando a Let's Encrypt.

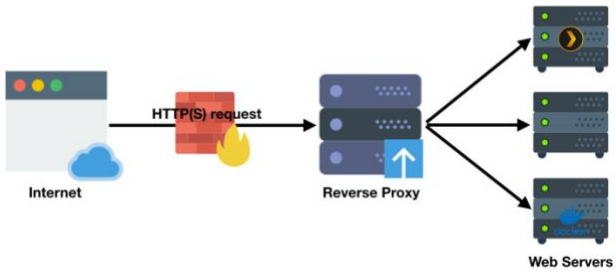

Figura 2. Funcionamiento del Reverse Proxy (tomado de LinuxServer.io).

En nuestro caso, no estaríamos tratando con una arquitectura de servidores distribuidos. Estaríamos trabajando con una arquitectura basada en contenedores, que simularían esos servidores distribuidos. Un contenedor que haría de Reverse Proxy, y otro que haría de Web Server.
Lo único que debemos de hacer es agregar el servicio en nuestro archivo docker-compose.yml, el cual si utilizamos la documentación de LinuxServer.io, nos quedaría así:

```
swag:
  image: linuxserver/swag
  container_name: swag
  depends_on:
    - wordpress
  volumes:
    - ./config:/config
    - ./default:/config/nginx/site-confs/default
  environment:
    - EMAIL=jpgomezt@eafit.edu.co
    - URL=wordpress-pampa.tk
    - SUBDOMAINS=www
    - VALIDATION=http
    - TZ=America/Bogota
    - PUID=500
    - PGID=500
  ports:
    - "443:443"
    - "80:80"
```

Podemos ver que es similar a los otros servicios que ya habíamos instanciado, en el cual cabe resaltar algunos puntos. Primero utilizamos la configuración *"depends on"* para decirle al SWAG a que servidores va a dirigir las peticiones después de pasar por el Reverse Proxy. En este caso, son dirigidas a nuestra aplicación WordPress.
En la configuración del "*environment"*, es importante configurar algunas variables que pide el SWAG para su correcto funcionamiento. En primer lugar, un email y el respectivo dominio y subdominios. Let's Encrypt se encarga de verificar que el correo y los respectivos dominios sean verídicos y estén en funcionamiento. También definimos el Time Zone (TZ), para esto, usamos nuestro respectivo Time Zone. Finalmente, LinuxServer.io recomienda la utilización del PUID y PGID (Process User ID y Process Group ID) para reducir los permisos que puede tener el contenedor al realizar la configuración de la red. Como no nos preocupamos por esto (es un servidor sencillo) se puede usar un valor por defecto 500. También configuramos los volúmenes. El config es solo para guardar en nuestra maquina EC2 el fichero de configuración que crea SWAG cuando levanta el contenedor, pero el default, si es importante configurarlo, ya que usaremos nuestro propio archivo default para reemplazar el archivo de site-confs de Nginx. En este archivo default, vamos a realizar la redirección de todas las peticiones realizadas por el puerto 80 al puerto 443, y además le decimos al Nginx donde están los archivos de configuración del SSL. Finalmente le decimos donde se encuentra ubicada nuestra aplicación a la que se redireccionara la petición del Proxy server. Para esto, utilizamos el *"depends on"* que ya habíamos configurado, en este caso wordpress:

```
server {
  listen 80;
  listen [::]:80;
  server_name _;
  return 301 https://$host$request_uri;
}

server {
  listen 443 ssl http2 default_server;
  listen [::]:443 ssl http2 default_server;

  root /var/www/html/example;
  index index.html index.htm index.php;

  server_name _;

  include /config/nginx/proxy-confs/*.subfolder.conf;

  include /config/nginx/ssl.conf;

  client_max_body_size 0;

  location / {
        try_files $uri $uri/ /index.php?$args @app;
  }

  location @app {
        proxy_pass http://wordpress;
        proxy_set_header Host $host;
        proxy_redirect off;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto https;
        proxy_set_header X-Real-IP $remote_addr;
  }

}
```

Así quedaría nuestro archivo *"default"*.

Finalmente, en nuestro servicio SWAG en el docker-compose.yml, mapeamos los puertos 443 y 80 de la maquina EC2, al contenedor SWAG.

Ya con esto, podemos lanzar nuestras aplicaciones (SWAG, WordPress y MySQL). Para esto, levantamos los contenedores utilizando Docker-Compose:

```
$ sudo docker-compose up -d
```

La opción -d es para correr el proceso en el *"background"*. Si queremos ver como corren y los mensajes que lanzan cada contenedor, podemos correr el comando sin esta opción.

Ya podemos visitar nuestra página:

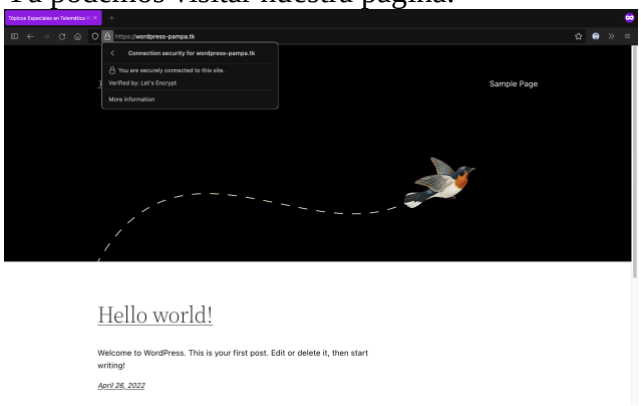

Figura 3. Página en WordPress desplegada

Laboratorio 2: MERN

En esta sección mostraremos los pasos que fueron requeridos para la elaboración del laboratorio 1.
Por medio de este enlace, se podrá acceder al servicio web desplegado:
www.bookstore-pampa.tk

Despliegue

Es una aplicación web que emplea un despliegue en nube, utilizando AWS, la cual emplea un estilo arquitectónico monolítico y de división en capas, teniendo 3 capas. Contamos con la capa de presentación, lógica de negocio y persistencia de datos.

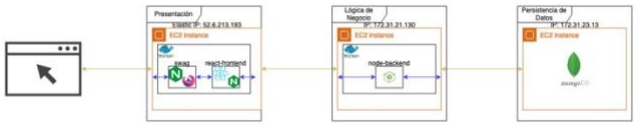

Figura 4. Despliegue en AWS.

A continuación, vamos a ver los pasos requeridos para desplegar cada capa:

Persistencia de Datos

En esta capa es donde la información es gestionada y almacenada. Los datos se almacenan en MongoDB, un motor de bases de datos orientado a documentos. Para esto, utilizaremos una maquina EC2 Amazon Linux.

Configuración EC2

Para iniciar se instancio una EC2 con S.O Amazon Linux, la cual posee el puerto 27017 (puerto por defecto para MongoDB) abierto. Una vez hecho esto, ya instalamos el motor de Mongo. Para esto seguimos el tutorial dispuesto en la página oficial de MongoDB:

Primero debemos crear el archivo de configuración de los repositorios de yum, para que este pueda instalar Mongo. Este archivo será *"/etc/yum.repos.d/mongodb-org-5.0.repo"*. Una vez lo creamos, debemos de insertar la siguiente información:

```
[mongodb-org-5.0]
name=MongoDB Repository
baseurl=https://repo.mongodb.org/yum/amazon/2/mongodb-org/5.0/x86_64/
gpgcheck=1
enabled=1
gpgkey=https://www.mongodb.org/static/pgp/server-5.0.asc
```

Una vez configurado el repositorio, ya podemos instalar mongo.

```
$ sudo yum install -y mongodb-org
```

Una vez instalado, podemos correr la aplicación de MongoDB.

```
$ sudo systemctl start mongod
```

Ya una vez inicializada la aplicación, lo único que tenemos que hacer es crear la base de datos, con la respectiva colección y documentos.

```
$ sudo mongosh
```

Para crear la base de datos *"bookstore"*, simplemente la tenemos que instanciar.

```
test> use bookstore
```

Para que la base de datos quede creada, debemos insertar datos en ella, para ello, insertaremos en la colección *"books"* los datos de los 2 libros de Leonardo da Vinci.

*bookstore> db.books.insert(<**json libro 1**>)*

*bookstore> db.books.insert(<**json libro 2**>)*

Finalmente, creamos el usuario que utilizaremos para conectarnos a la Base de Datos. Este usuario se crea en la base de datos por defecto *"admin"*, ya que tendrá permisos de lectura y escritura en todo el Mongo:

bookstore> db.createUser({user: "jpgomezt", pwd: "password", roles:[{role: "readWriteAnyDatabase" , db:"admin"}]})

Con esto, ya tenemos la base de datos configurada. Ahora tenemos que configurar a Mongo para que permita conexiones de cualquier dirección, y que tenga el módulo de autenticación activo para asegurar las conexiones que se intentan hacer. Este archivo se encuentra en *"/etc/mongod.conf"*. Una vez entramos en esta, debemos de ir al campo *"net"* y abrir el campo *"bindIp"*. Este por defecto trae la dirección loopback. Para abrirlo a cualquiera, usamos la wildcart *"0.0.0.0"*. Para activar la verificación de usuario, descomentamos el módulo *"security"*, y le agregamos la siguiente configuración *"authorization: 'enabled'"*. El archivo debe quedar así:

Finalmente, reiniciamos la aplicación de MongoDB para que los cambios hagan efecto.

$ sudo systemctl restart mongod

Con esto, terminamos la capa de Persistencia de Datos.

Lógica de Negocio (BackEnd)

En esta capa se ejecuta y lleva a cabo la lógica de negocio. El backend se encuentra desarrollado empleando tecnología de scripting del lado del servidor, para este caso Node.JS. Igualmente utilizamos un framework de desarrollo de aplicaciones web denominado Express, así como Mongoose para la conexión con la base de datos. Para esto, utilizaremos una maquina EC2 Amazon Linux.

Configuración EC2

Para iniciar se instancio una EC2 con S.O Amazon Linux, la cual posee el puerto 5000 (puerto por donde el servidor escuchara las peticiones) abierto. Una vez hecho esto, ya instalamos Docker, para realizar el despliegue de este servidor. Para esto seguimos el tutorial dispuesto en la página oficial de Amazon para la instalación de Docker. Solo hace falta instalar el paquete Docker Engine más reciente (el repo de estas instancias ya lo tienen listo).

$ sudo amazon-linux-extras install Docker

Una vez instalado, podemos correr la aplicación de Docker:

$ sudo systemctl start docker

$ sudo systemctl enable docker

Ya debemos cambiar el URL de conexión con la base de datos, que se encuentra en el archivo *".env"* del proyecto backend entregado en el laboratorio:

URL_DB_CONNECTION = mongodb://jpgomezt:password@172.31.23.13/bookstore

Finalmente, debemos crear nuestro archivo Dockerfile.

FROM node:14.19.1-alpine3.14

WORKDIR /usr/src/backend

COPY . .

RUN npm install

EXPOSE 5000

CMD ["node", "server.js"]

Para este, utilizamos la imagen de Node.JS que contiene la versión 14.19 (la misma versión utilizada en el laboratorio para evitar problemas de versiones). Configuramos el directorio donde trabajamos, que será *"usr/src/backend"*. Luego copiamos todo el contenido de nuestro fichero backend dentro del espacio de trabajo en el contenedor. Instalamos los módulos de Node.JS que utiliza el proyecto corriendo *"npm install"*, exponemos el puerto 5000 (el puerto que se tiene configurado para escuchar las peticiones) y ponemos a correr el servidor utilizando el archivo *"server.js"*.
Ya con esto, tenemos todo lo necesario para correr nuestro servidor dockerizado. Pasamos el fichero a nuestra instancia EC2 utilizando Git o SCP (secure copy).
Una vez poseemos el fichero en nuestra instancia EC2, solo debemos crear la imagen utilizando nuestro Dockerfile.

$ sudo docker build -t node-backend .

Y levantamos nuestro contenedor utilizando la respectiva imagen.

$ sudo docker run -dit --name node-backend-dockerize -p 5000:5000 node-backend

Con esto ya tenemos nuestra aplicación BackEnd desplegada.

Presentación (FrontEnd)

Esta capa se encarga de todos los aspectos relacionados con la interfaz de usuario. A esta capa se accede a través del browser. Igualmente, se encarga de la comunicación con el backend. Esta se encuentra desarrollado utilizando React. Para esto, utilizaremos una maquina EC2 Amazon Linux.

Configuración EC2

Para iniciar se instancio una EC2 con S.O Amazon Linux, la cual posee los puertos 80 y 443 (http y https) abiertos. A esta instancia se le asocio una IP elástica para evitar que su IP publica cambie, y poderla asociar al DNS de Freenom como se muestra a continuación:

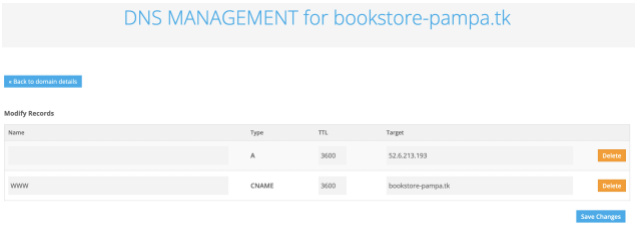

Figura 5. Configuración DNS Freenom.

Una vez hecho esto, ya instalamos Docker y Docker-Compose (mismos pasos señalados anteriormente) para realizar el despliegue de este servidor.

React con Dockerfile

Una vez instalados estos, debemos de acceder a nuestro fichero frontend dispuesto por el laboratorio, en donde crearemos otros archivos adicionales de configuración. Primero, crearemos nuestro archivo Dockerfile, el cual se encarga de correr la aplicación hecha en React y disponerla en un servidor Nginx.
#React build
FROM node:14.19.1-alpine3.14 as react-build

WORKDIR /usr/src/frontend

COPY . ./

RUN npm install

RUN npm run build

#Nginx setup
FROM nginx:1.12.0

RUN rm -f /etc/nginx/nginx.conf

COPY ./nginx.conf /etc/nginx/nginx.conf

COPY --from=react-build /usr/src/frontend/build /usr/share/nginx/html/bookstore

Vemos que el archivo cuenta con dos bloques, *"React build"*, donde creamos el build de la aplicación React, y *"Nginx setup"*, donde configuramos nuestro servidor Nginx. Para el bloque del React, utilizamos la imagen de Node.JS que contiene la versión 14.19 y le asociamos a este bloque el tag de *"react-build"*. Configuramos el directorio donde trabajamos, que será *"usr/src/frontend"*. Luego copiamos todo el contenido de nuestro fichero frontend dentro del espacio de trabajo en el contenedor. Instalamos los módulos de Node.JS que utiliza el proyecto corriendo *"npm install"*, y creamos el built corriendo *"npm run build"*. Ahora configuramos el Nginx, utilizando la imagen que contiene la versión 1.12 (la misma utilizada en el laboratorio). Primero eliminamos el archivo de configuración por defecto del Nginx y luego, lo reemplazamos por la configuración planteada en el laboratorio (se muestra más adelante). Finalmente, ubicamos el proyecto React dentro del servidor Nginx en el fichero *"bookstore"*.
En el archivo de configuración de Nginx, es necesario que el web server reciba las peticiones dirigidas al api: /api/books y las redirecciones al backend. Para esto agregamos la directiva upstream con la dirección IP del backend, antes de la de server en el archivo nginx.conf.

```
...
upstream backend {
        server 172.31.21.130:5000;
   }
...
```

Ahora se configura el servidor indicándole la dirección root en donde se encuentra el build en React.

```
...
server {
      listen      80 default_server;
      listen      [::]:80 default_server;
      server_name  _;
      root        /usr/share/nginx/html/bookstore;
...
```

Finalmente, en esta sección *"server"*, debemos de indicarle al servidor la ubicación de los recursos *"/api/books"*.

```
location /api/books {
        proxy_pass http://backend;
}
```

Con esto ya tendríamos todo lo necesario para desplegar nuestro servicio web con React. Ahora nos dispondremos a utilizar Docker-Compose para poder levantar este servicio junto con el servicio SWAG, para poder asegurar nuestro sitio web.

Creación del archivo docker-compose.yml

En esta instancia EC2, solo levantaremos 2 servicios, nuestro servicio de React que construimos con el Dockerfile, y el servicio SWAG.

```
version: '3'
services:
  swag:
    image: linuxserver/swag
    container_name: swag
    depends_on:
      - react-frontend
    volumes:
      - ./config:/config
      - ./default:/config/nginx/site-confs/default
    environment:
      - EMAIL=jpgomezt@eafit.edu.co
      - URL=bookstore-pampa.tk
      - SUBDOMAINS=www
      - VALIDATION=http
      - TZ=America/Bogota
      - PUID=500
      - PGID=500
    ports:
      - "443:443"
      - "80:80"
  react-frontend:
    build:
      context: .
```

El primer servicio de SWAG, es exactamente igual al que desplegamos en el laboratorio 1 con WordPress. El único cambio es el dominio y subdominio. Para construir nuestra aplicación React, solo hace falta definir el nombre del servicio *"react-frontend"* y decirle a Docker que, para construir el contenedor, use el Dockerfile que construimos y se encuentra en el mismo directorio que el Compose. Volvemos a definir el archivo default que utilizara el servidor Nginx en el SWAG.

```
server {
  listen 80;
  listen [::]:80;
  server_name _;
  return 301 https://$host$request_uri;
}

server {
  listen 443 ssl http2 default_server;
  listen [::]:443 ssl http2 default_server;

  root /var/www/html/example;
  index index.html index.htm index.php;

  server_name _;

  include /config/nginx/proxy-confs/*.subfolder.conf;

  include /config/nginx/ssl.conf;

  client_max_body_size 0;

  location / {
        try_files $uri $uri/ /index.php?$args @app;
  }

  location @app {
        proxy_pass http://react-frontend;
        proxy_set_header Host $host;
              proxy_redirect off;
              proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
              proxy_set_header X-Forwarded-Proto https;
              proxy_set_header X-Real-IP $remote_addr;
  }

}
```

Podemos ver que es similar al que definimos en el laboratorio 1. El único cambio es el nombre de la aplicación a la que redireccionaremos, ya que ahora toma el nombre de nuestro servicio *"react-frontend"*.

Ya con esto, podemos lanzar nuestras aplicaciones (SWAG, WordPress y MySQL). Para esto, levantamos los contenedores utilizando Docker-Compose:

```
$ sudo docker-compose up -d
```

Ya podemos visitar nuestra página:

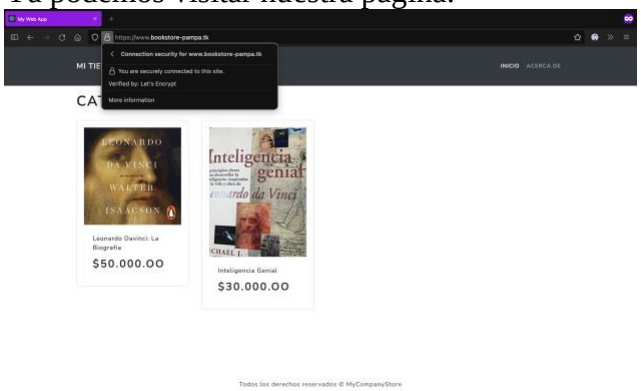

Figura 6. Página MERN desplegada

Referencias

[1] AWS. (s. f.). Creating a container image for use on Amazon ECS - Amazon Elastic Container Service. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/create-container-image.html

[2] Docker. (s. f.-a). Install Docker Compose. Docker Documentation. https://docs.docker.com/compose/install/

[3] Docker. (s. f.-b). Install Docker Engine on Ubuntu. Docker Documentation.

https://docs.docker.com/engine/install/ubuntu/

[4] Docker. (s. f.-c). Use volumes. Docker Documentation. https://docs.docker.com/storage/volumes/

[5] LinuxServer. (s. f.). SWAG setup - LinuxServer.io. LinuxServer.io. https://docs.linuxserver.io/general/swag#reverse-proxy

[6] London, I. (2016, 23 mayo). How to connect to your remote MongoDB server. Ian London's Blog. https://ianlondon.github.io/blog/mongodb-auth/

[7] MongoDB. (s. f.-a). Create A MongoDB Database. https://www.mongodb.com/basics/create-database

[8] MongoDB. (s. f.-b). Install MongoDB Community Edition on Amazon Linux — MongoDB Manual. MongoDB Documentation. https://www.mongodb.com/docs/manual/tutorial/install-mongodb-on-amazon/

[9] OpenJS Foundation. (s. f.). Dockerizing a web app. Node.Js. https://nodejs.org/en/docs/guides/nodejs-docker-webapp/

[10] Sokola, M. (2021, 17 marzo). How to Deploy a React App to Production Using Docker and NGINX with API Proxies. freeCodeCamp.Org. https://www.freecodecamp.org/news/how-to-deploy-react-apps-to-production/

[11] TIBCO. (s. f.). How to Do a Clean Restart of a Docker Instance. TIBCO Docs. https://docs.tibco.com/pub/mash-local/4.1.0/doc/html/docker/GUID-BD850566-5B79-4915-987E-430FC38DAAE4.html

[12] Willimott, C. (2021, 11 diciembre). Quickly setup WordPress & SSL via Let's Encrypt and Certbot using Docker Compose. Medium. https://carlwillimott.medium.com/quickly-setup-wordpress-ssl-via-lets-encrypt-and-certbot-b29e8abf2072