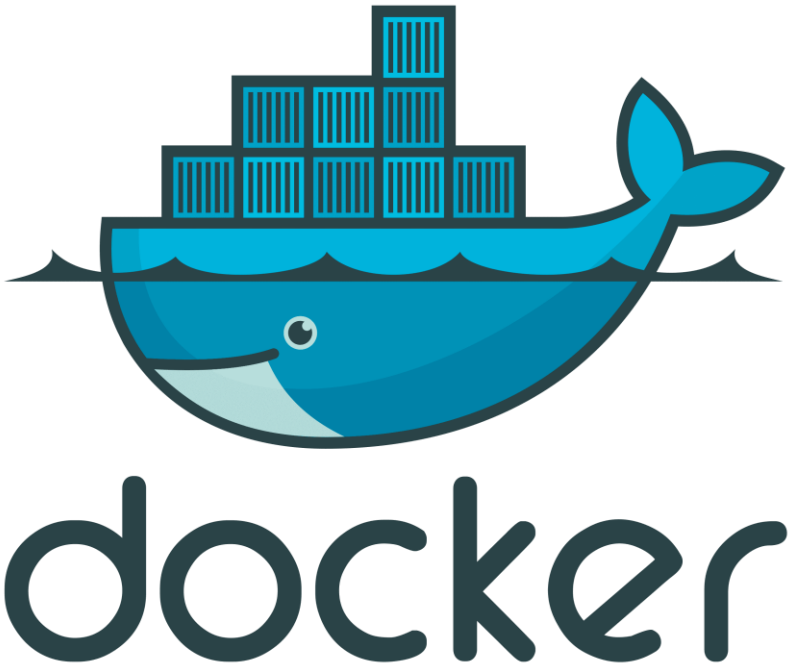
SEGUIMIENTO



Enrique José Carmona Cerrato

PRESENTACIÓN

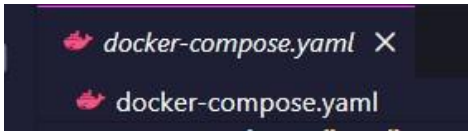
En este documento explicaremos principalmente la configuración de un entorno LAMP en Docker donde los principales servicios que utilizaremos serán Apache(Nginx en estecaso), MySQL, PHP y nosotros añadiremos REDIT; Todo estos convivirán en un entorno para la creación de un servicio web donde tendremos una base de datos interrelacionada y que la gestiona y otra base de datos para la cache de la misma página, todo esto programado en PHP.

Comenzaremos con la instalación de Docker y los servicios que vamos a utilizar dentro del mismo, para ello deberemos tener un entorno Linux/Debian o en su defecto WSL (Windows Subsystem For Linux) diseñada para poder utilizar las herramientas de linux en Windows.

Una vez configurada comenzaremos con la instalación de Docker con el comando:

-Sudo apt install Docker –y

Después crearemos una carpeta en nuestro directorio donde vamos a configurar todos nuestros archivos, la llamaremos en este caso DOCKER-PHP



Dentro de ella crearemos dos archivos fundamentales para la configuración de Docker nuestro Dockercompose y nuestro Dockerfile, dentro de los mismo
hallaremos configuraciones básicas para la ejecución de nuestros contenedores, A continuación, os enseño las configuraciones que he creado para nuestro entorno, iremos explicando cada una paso a paso y dado incisos en los errores que puede salir a lo largo de la configuración.

Bien comenzamos con la configuración principal de nuestros “docker.compose,yaml, en el configuraremos parámetros como: Puertos, versiones, rutas... abajo lo especificamos mucho más:

Configuración de Docker-compose-yaml:

Configuración para NGINX:

Versión: Versión de docker compose que vamos a utilizar para nuestra configuración

Services: Servicios que se van a utilizar **Paginaweb:**

Nombre de nuestro servicio **imagen:** nginx:latest

Ultima versión de la imagen del servicio Nginx

Ports: Puertos que vamos a utilizar en este caso el 8080 en nuestro local y el 80 para el contenedor

Volumes: Ruta donde van a volcarse los archivos locales a los archivos del contenedor

Configuración de PHP:

docker-php: Nombre del servicio que ejecuta PHP.

build: Indica que la imagen de este servicio se construye usando el Dockerfile ubicado en el directorio actual.

ports: Sección para exponer puertos del contenedor. "9000:9000" Expone el puerto 9000 del contenedor PHP-FPM para que Nginx pueda comunicarse con él.

volumes: Define un volumen compartido para el código PHP *./php:/www/public/* Comparte el código PHP entre el equipo anfitrión y el contenedor PHP.

Configuración de la base de datos:

basededatos: Nombre del servicio que ejecuta el servidor MySQL.

image: mysql:8.1 Usa la imagen oficial de MySQL versión 8.1. **volumes:**

Define volúmenes persistentes para la base de datos.

- mysqldata:/var/lib/mysql Guarda los datos de MySQL en un volumen para que no se pierdan al apagar el contenedor.

ports: Sección donde se exponen los puertos de MySQL.

- "4306:3306" Permite conectarse a MySQL desde el equipo anfitrión usando el puerto 4306.

restart: unless-stopped Configura el contenedor para que se reinicie automáticamente si falla, excepto cuando se detiene manualmente. **environment:** Define variables de entorno para configurar MySQL automáticamente.

MYSQL_ROOT_PASSWORD: kike1234 Establece la contraseña del usuario root de MySQL.

MYSQL_USER: usuario Crea un usuario adicional llamado `usuario`.

MYSQL_PASSWORD: 1234 Establece la contraseña del usuario `usuario`.

MYSQL_DATABASE: docker-php Crea automáticamente una base de datos llamada docker-php.

Configuración de Redit:

cache: Nombre del servicio que ejecuta Redis.

image: redis:latest Indica que se usará la imagen oficial de Redis en su versión más reciente. **container_name:** redis-server Asigna un nombre fijo al contenedor Redis.

ports: Define los puertos expuestos por Redis.

- "6379:6379 Expone el puerto estándar de Redis para conexiones externas.

command: ["redis-server", "--appendonly", "yes"] Ejecuta Redis con persistencia activada para guardar los datos en disco. **volumes:** Sección donde se definen los

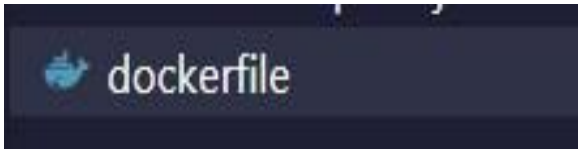
volúmenes Docker. **mysqldata:** {} Crea un volumen persistente llamado `mysqldata`.

redisdata: Declara un volumen llamado `redisdata`, aunque no está siendo utilizado en ningún servicio. **networks:** Sección para definir redes personalizadas. **mi-red:** Nombre de la red personalizada.

driver: bridge Usa el driver `bridge`, que permite la comunicación entre contenedores en la misma red.

Configuración de Dockfile

Una vez creada la configuración de nuestro docker-compose.yaml proseguiremos con la configuración de nuestro otro archivo importante, nuestro Dockfile, el archivo contendrá configuraciones de plugin, instalaciones de servicio..., lo explicamos más a fondo a continuación:



FROM php:8.2-fpm Indica la imagen base del contenedor; usa PHP versión 8.2 con PHP-FPM instalado.

RUN apt-get update && apt-get install -y \ Ejecuta comandos dentro de la imagen y actualiza la lista de paquetes disponibles. **libzip-dev** \ Instala las librerías necesarias para trabajar con archivos ZIP en PHP.

libonig-dev \ Instala la librería necesaria para el manejo de expresiones regulares (usada por PHP).

&& docker-php-ext-install pdo pdo_mysql Compila e instala las extensiones **PHP:**

pdo: interfaz genérica para bases de datos **pdo_mysql:**

permite conectar PHP con MySQL usando PDO

RUN apt-get update \ Actualiza nuevamente la lista de paquetes del sistema.

&& apt-get install -y default-mysql-client \ Instala el cliente de MySQL para poder conectarse a bases de datos desde el contenedor.

&& docker-php-ext-install mysql \ Instala la extensión mysqli, otra forma de conectar PHP con MySQL (distinta de PDO).

RUN apt-get update && apt-get install -y \ Actualiza paquetes e inicia otra instalación de dependencias.

libzip-dev \ Instala nuevamente soporte para ZIP (está repetido respecto a bloques anteriores).

unzip \ Instala la herramienta para descomprimir archivos ZIP. **git** \

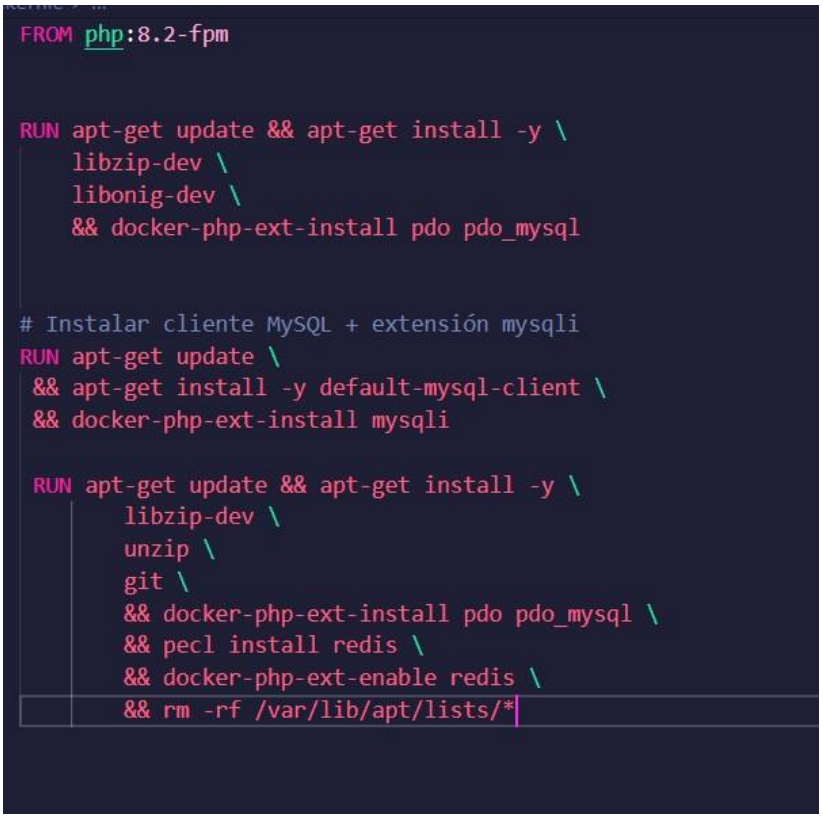
Instala Git para poder clonar repositorios dentro del contenedor.

&& docker-php-ext-install pdo pdo_mysql \ Vuelve a instalar las extensiones PDO y PDO MySQL (también repetido).

&& pecl install redis \ Instala la extensión Redis para PHP usando PECL.

&& docker-php-ext-enable redis \ Habilita la extensión Redis en PHP para que pueda usarse.

&& rm -rf /var/lib/apt/lists/* Elimina archivos temporales de instalación para reducir el tamaño final de la imagen.



```
FROM php:8.2-fpm


RUN apt-get update && apt-get install -y \
    libzip-dev \
    libonig-dev \
    && docker-php-ext-install pdo pdo_mysql


# Instalar cliente MySQL + extensión mysqli
RUN apt-get update \
 && apt-get install -y default-mysql-client \
 && docker-php-ext-install mysqli

 RUN apt-get update && apt-get install -y \
        libzip-dev \
        unzip \
        git \
        && docker-php-ext-install pdo pdo_mysql \
        && pecl install redis \
        && docker-php-ext-enable redis \
        && rm -rf /var/lib/apt/lists/*
```

Errores comunes en la configuración de estos documentos:

Los principales errores que puedes encontrar a la hora de configurar de estos archivos es la sintaxis y la estipulación de las rutas ya que si una ruta está mal escrita o existe una mala configuración tanto en el "compose" como en el "Dockfile" al levantar nuestro contenedor no funcionará correctamente y no se ejecutará, siempre hay que establecer rutas claras a la hora de estipular en el apartado "volume" ya que si no es así nuestros contenedores no tendrán la información necesaria y no se ejecutarán o no funcionarán correctamente.

Una vez creada esta configuración, pasaremos a ejecutar nuestros contenedores de dockfile, para ellos pondremos una serie de comando que son:

Docker-compose build:

```
ovimatica@ROVSEV130:~/DOCKER-PHP$ docker-compose build
WARNING: Some networks were defined but are not used by any service: mi-red
paginaweb uses an image, skipping
basededatos uses an image, skipping
cache uses an image, skipping
Building docker-php
[+] Building 34.3s (8/8) FINISHED
```

Este comando construirá los contenedores cargando así toda la información de nuestros archivos configurados anteriormente.

Docker-compose up:

```
ovimatica@ROVSEV130:~/DOCKER-PHP$ docker-compose up
WARNING: Some networks were defined but are not used by any service: mi-red
Creating network "docker-php_default" with the default driver
Creating docker-php_paginaweb_1    ... done
Creating docker-php_docker-php_1   ... done
Creating redis-server              ... done
Creating docker-php_basededatos_1 ... done
```

Este comando no montara nuestros contenedores dándolos de “alta” para su funcionamiento

Para comprobar que los contenedores están en funcionamiento utilizaremos:

Docker ps:

```
CONTAINER ID   IMAGE                  COMMAND                  CREATED         STATUS         PORTS
                                      NAMES
a0fa7b02ac50   mysql:8.1              "docker-entrypoint.s…"   2 minutes ago   Up 2 minutes   33060/tc
p, 0.0.0.0:4306->3306/tcp, [::]:4306->3306/tcp   docker-php_basededatos_1
fec9ea47fc47   docker-php_docker-php   "docker-php-entrypoi…"   2 minutes ago   Up 2 minutes   0.0.0.0:
9000->9000/tcp, [::]:9000->9000/tcp              docker-php_docker-php_1
a88998fb5c12   redis:latest           "docker-entrypoint.s…"   2 minutes ago   Up 2 minutes   0.0.0.0:
6379->6379/tcp, [::]:6379->6379/tcp              redis-server
3293efa4a1b5   nginx:latest           "/docker-entrypoint.…"   2 minutes ago   Up 2 minutes   0.0.0.0:
8080->80/tcp, [::]:8080->80/tcp                  docker-php_paginaweb_1
```

Muestra un listado de nuestro container.

Si deseamos detener el container deberemos poner:

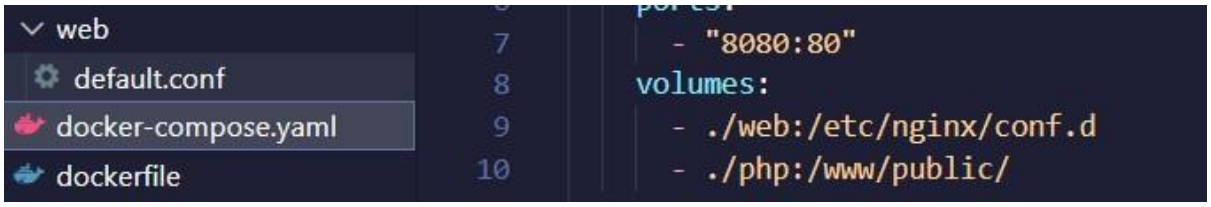
Docker-compose down:

```
WARNING: Some networks were defined but are not used by any service: mi-red
Stopping docker-php_basededatos_1 ... done
Stopping docker-php_docker-php_1  ... done
Stopping redis-server             ... done
Stopping docker-php_paginaweb_1   ... done
Removing docker-php_basededatos_1 ... done
Removing docker-php_docker-php_1  ... done
Removing redis-server             ... done
Removing docker-php_paginaweb_1   ... done
Removing network docker-php_default
ovimatica@ROVSEV130:~/DOCKER-PHP$
```

Detendrá todos los servicios creados en nuestro container

Configuración de .conf para el servidor Nginx

Bien, ahora después de enseñaros de cómo se configuran los archivos y de cómo se ejecutan los containeres, deberemos crear un archivo .conf para la configuración de nuestro servidor web, si nos fijamos en la configuración de nuestro compose tenemos estipulado que utilice la configuración de este mismo archivo.



En este caso nosotros hemos creado una carpeta y lo hemos alojado ahí el direccionamiento que tiene la raíz.

Explicaremos la configuración de este archivo paso por paso para que ver cómo funciona en nuestro servicio web:

server { Inicio del bloque de configuración de un servidor virtual en Nginx. **listen**

80; Indica que el servidor web escuchará las peticiones HTTP en el puerto 80.

server_name LOCALHOST; Define el nombre del servidor; en este caso responde a peticiones hechas a `localhost`.

root /www/public; Establece el directorio raíz donde se encuentran los archivos del sitio web dentro del contenedor.

index index.php index.html; Define los archivos que se cargarán por defecto cuando se acceda a un directorio.

location ~ .php$ { Bloque que indica cómo procesar los archivos con extensión .php.

include fastcgi_params; Incluye las variables necesarias para que Nginx se comunique con PHP-FPM mediante FastCGI.

fastcgi_pass docker-php:9000; Envía las peticiones PHP al contenedor PHP-FPM llamado docker-php usando el puerto 9000.

fastcgi_index index.php; Indica cuál es el archivo PHP principal que se ejecutará si se solicita un directorio.

fastcgi_param SCRIPT_FILENAME$document_root$fastcgi_script_name; Especifica la ruta completa del archivo PHP que debe ejecutar PHP-FPM.

default.conf

```
server {
    listen 80;
    server_name LOCALHOST;

    root /www/public;
    index index.php index.html;

    location / {
        try_files $uri $uri/ /index.php?$query_string;
    }

    location ~ \.php$ {
        include fastcgi_params;
        fastcgi_pass docker-php:9000;
        fastcgi_index index.php;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    }
}
```

Configuración de la red Docker para la interconexión de servicios:

Una vez hayamos terminado la configuración de todo nuestro archivo para que docker cree redireccione y gestione correctamente nuestros contenedores, pasaremos a la conexión de nuestros servicios a través de una red interna que crearemos para ello lo haremos a través de línea de comandos

La creación de nuestra red la tenemos estipulada en nuestro compose por lo cual debería crearse sola, pero si deseamos crearla a mano podemos hacerlo a travéx del comando:

-docker network create “nombre de la red”

```
ovimatica@ROVSEV130:~/DOCKER-PHP$ docker network create prueba1
386ce9ff6feec1072db0ed923dd9b99fc19ae8706871378f603db20593d8fe9e
ovimatica@ROVSEV130:~/DOCKER-PHP$
```

Para ver si la red existe, utilizaremos el comando:

-docker network ls

```
ovimatica@ROVSEV130:~/DOCKER-PHP$ docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
67438701c0dc   backend   bridge    local
e455413bdecc   bridge    bridge    local
0b977b7eebd3   host      host      local
28c1c3c4e509   mi-red    bridge    local
dde1ab2f4546   none      null      local
386ce9ff6fee   prueba1   bridge    local
ovimatica@ROVSEV130:~/DOCKER-PHP$
```

Como podemos observar la red se ha creado.

Vale, ahora comprobaremos la interconexión entre ambos servicios en este caso entre el servicio de php y nuestra base de datos ya que como más adelante veremos configuraremos nuestro php para que gestione las bases de datos y la página web que crearemos

Para comprobarlo, debemos utilizar una serie de comandos, que se utilizaran para acceder a nuestros contenedores, configurarlos con los servicios que sean necesarios y probar su interconexión dentro de la red docker.

Una vez creada añadiremos nustros contenedores a la red que hemos creado con el comando:

-docker network connect mi-red nombre_contenedor

```
ovimatica@ROVSEV130:~/DOCKER-PHP$ docker network connect mi-red nombre_contenedor
```

Nombre_contenedor se sustituye por el nombre que deseemos

Una vez añadidos nuestros servicios a la red comprobaremos si están dentro de la misma con el comando:

-docker network inspect mi-red

```
{
    "Name": "mi-red",
    "Id": "28c1c3c4e50989342ae9b86634fd67eb60c4be22578628b96226123b335ec991",
    "Created": "2026-01-13T10:03:40.596167788+01:00",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv4": true,
    "EnableIPv6": false,
    "IPAM": {
        "Driver": "default",
        "Options": {},
        "Config": [
            {
                "Subnet": "172.20.0.0/16",
                "Gateway": "172.20.0.1"
            }
        ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "ConfigFrom": {
        "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {
        "018963c564567b8c44b417ae1f21938ccc3067be98580b944b0e58f72ea2b4c9": {
            "Name": "docker-php_paginaweb_1",
            "EndpointID": "c86d30a549115b3d4d27e4b0da3a2f68d1cb25604fa24dc6adc8d8468015f573",
            "MacAddress": "ba:36:12:6a:ed:4b",
            "IPv4Address": "172.20.0.2/16",
            "IPv6Address": ""
        },
        "9ed804e8042f9b8b4109ea48febe14eea36bdaa417cbe46ed6a74dd47c86c755": {
            "Name": "docker-php_basededatos_1",
            "EndpointID": "f2b78d82ee90d69b9952a46b178c8a4d9965258d9115f96dc8e08b0590172e91",
            "MacAddress": "d2:58:6d:cc:9b:10",
            "IPv4Address": "172.20.0.3/16",
            "IPv6Address": ""
        },
        "dfdc8174b8018907c19b30c722646b0e3263329aff32e4ff64ce48ea124401e7": {
            "Name": "docker-php_docker-php_1",
            "EndpointID": "1c16945a617d7a8ca0ee3614281c5a6d5c9a5a64eed351a192473b9c06e9a0e6",
            "MacAddress": "a2:4c:bc:a7:d1:73",
            "IPv4Address": "172.20.0.4/16",
            "IPv6Address": ""
        }
```


Como podemos ver los tres contenedores aparecen dentro de nuestra red, en el apartado “container”

Vale, ya hemos creado la red y hemos añadido los contenedores a la misma, ahora procederemos a ver si tiene interconexión:

Para ello lo haremos con el comando:

docker exec -it docker-php_docker-php_1 bash

Con este comando accederemos dentro del container en específico al container de PHP y ejecutaremos un comando para que nos resuelva los nombres a traves de las DNS

```
ovimatica@ROVSEV130:~/DOCKER-PHP$ docker exec -it docker-php_docker-php_1 bash
root@ba73b1db596c:/var/www/html# getent hosts docker-php_basededatos_1
172.23.0.5      docker-php_basededatos_1
root@ba73b1db596c:/var/www/html# getent hosts docker-php_cache_1
root@ba73b1db596c:/var/www/html#
```

Para asegurarnos también haremos una petición al servidor SQL para ver si nos la devuelve

```
root@686a4e63e380:/var/www/html# getent hosts docker-php_basededatos_1
172.23.0.2      docker-php_basededatos_1
```

Existen diversas maneras de ver la interconexión en la red con ping, haciendo peticiones SQL desde PHP...

Problemas relacionados con la interconexión:

Los principales problemas que tenemos a la hora de interconectar son los fallos de autenticación SLL o TLS los cuales MySQL los lleva instalado de fábrica y son necesarios detener motificando nuestro compose y en la parte de base de datos colocar el comando MYSQL_SLL_MODE:DISABLED así podremos habilitar la entrada al sin SSL o TLS, también puede ser que no tenga el cliente de SQL instalado por lo cual deberás instalarlo en conteiner que sea necesario.

Configuración de Index.php

Finalmente, una vez que hemos realizado todos los pasos procederemos a la configuración de nuestro index.php en el cual deberemos programas la creación de nuestras tablas, la interconexión con MySQL, el formato al que le daremos a nuestra página...

PROPOSITO DE PAGINA:

El propósito de nuestra página será un traductor en el cual deberemos colocar a modo de prueba la palabra “hello” y nos devolverá la palabra traducida en los tres idiomas que hemos configurado.

La explicaremos por bloques para fácil desglose y entendimiento de este código:

```php
<?php
// Configuración MySQL (Docker)
$host = 'basededatos';
$user = 'usuario';
$password = '1234';
$database = 'docker-php';

// Conexión MySQL
mysqli_report(MYSQLI_REPORT_ERROR | MYSQLI_REPORT_STRICT);
$conn = new mysqli($host, $user, $password);
$conn->query("CREATE DATABASE IF NOT EXISTS `$database` CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci");
$conn->select_db($database);
```

En este apartado configuramos los parametros que iran conectados a la base de datos con docker en uno estipulamos los datos que tenemos en nuestro compose ya que tiene que ir correlacionados y tenemos la lineas de código donde creamos el login a nuestro servicio de datos.

```php
// Crear tablas si no existen
$conn->query("
    CREATE TABLE IF NOT EXISTS language (
        id INT AUTO_INCREMENT PRIMARY KEY,
        name VARCHAR(50) NOT NULL
    )
");
$conn->query("
    CREATE TABLE IF NOT EXISTS translation (
        id INT AUTO_INCREMENT PRIMARY KEY,
        phrase VARCHAR(255) NOT NULL,
        translation VARCHAR(255) NOT NULL,
        language_id INT,
        FOREIGN KEY (language_id) REFERENCES language(id) ON DELETE CASCADE
    )
");

// Insertar datos de ejemplo si no existen
$conn->query("
    INSERT INTO language (name)
    SELECT * FROM (
        SELECT 'Spanish' UNION
        SELECT 'French' UNION
        SELECT 'German'
    ) AS tmp
    WHERE NOT EXISTS (SELECT 1 FROM language LIMIT 1)
");
$conn->query("
    INSERT INTO translation (phrase, translation, language_id)
    SELECT * FROM (
        SELECT 'hello', 'hola', 1 UNION
        SELECT 'hello', 'bonjour', 2 UNION
        SELECT 'hello', 'hallo', 3
    ) AS tmp
    WHERE NOT EXISTS (SELECT 1 FROM translation LIMIT 1)
");
```

En esta parte tenemos la parte donde se estipula la creación de las tablas basándose en la manera que se diseña unas tablas a través de las sentencias de MYSQL

```php
// Conexión Redis
$redis = new Redis();
$redisConnected = false;
try {
    $redisConnected = $redis->connect('cache', 6379); // "cache" es el nombre del servicio en docker-compose
} catch (Exception $e) {
    $redisConnected = false;
}

$resultado = "";
$usandoCache = false;

if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $palabra = $_POST["palabra"];
    $idioma  = $_POST["idioma"];

    $cacheKey = "traduccion:$palabra:$idioma";

    // Comprobar si existe en Redis
    if ($redisConnected && $redis->exists($cacheKey)) {
        $resultado = $redis->get($cacheKey);
        $usandoCache = true;
    } else {
        // Consultar MySQL
        $stmt = $conn->prepare("
            SELECT t.translation
            FROM translation t
            JOIN language l ON t.language_id = l.id
            WHERE t.phrase = ? AND l.name = ?
            LIMIT 1
        ");
        $stmt->bind_param("ss", $palabra, $idioma);
        $stmt->execute();
        $stmt->bind_result($traduccion);

        if ($stmt->fetch()) {
            $resultado = $traduccion;

            // Guardar en Redis por 1 hora
            if ($redisConnected) {
                $redis->setex($cacheKey, 3600, $resultado);
            }
        } else {
            $resultado = "No se encontró traducción.";
        }

        $stmt->close();
    }
}
?>
```

-En este fragmento de código estipulamos la parte donde redit almacena la cache de nuestro uso de la página web, tiene estipulado un tiempo de guardado de cache y gestiona las peticiones de MySQL para corroborar que todo está correcto y devolver la información correspondiente, en el caso contrario de que no funcione devolverá el mensaje “No se encontró traducción”

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Traductor</title>
    <style>
        body { font-family: Arial, sans-serif; margin: 40px; text-align: center; }
        h2 { color: #333; }
        form { margin-top: 20px; }
        input, select, button { padding: 8px; margin-top: 5px; width: 200px; }
        button { cursor: pointer; }
        h3 { margin-top: 30px; color: #007BFF; }
        p { font-size: 1.1em; }
        .cache { color: green; font-weight: bold; margin-bottom: 20px; }
    </style>
</head>
<body>

<h2>Traductor Inglés</h2>

<?php if ($usandoCache): ?>
    <div class="cache">✅ Resultado obtenido de la cache Redis</div>
<?php endif; ?>

<form method="POST">
    <label>Palabra en inglés:</label><br>
    <input type="text" name="palabra" required><br><br>

    <label>Idioma:</label><br>
    <select name="idioma">
        <option value="Spanish">Español</option>
        <option value="French">Francés</option>
        <option value="German">Alemán</option>
    </select><br><br>

    <button type="submit">Traducir</button>
</form>

<?php if ($resultado): ?>
    <h3>Resultado:</h3>
    <p><?= htmlspecialchars($resultado) ?></p>
<?php endif; ?>

</body>
</html>
```

En esta última sección está el código HTML que da formato a la página creado así una selección múltiple con botones y enunciando dando un formato centrado y colores a los apartados con Styles(CSS) para mejor presentación de la página.

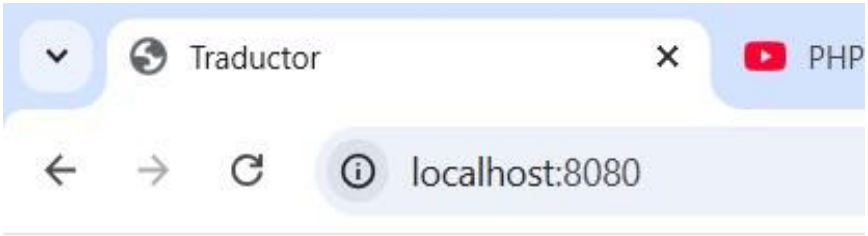
Errores comunes en la programación del Index.php:

Los errores que he visualizado a la hora de crear este index.php es que si previamente no tiene configurado el entorno de Docker te dará problema en el código a la hora de la conexión con el servido MySQL eso se solucionará en el momento que tenga la correclacion de datos correcta con el compose de docker normalmente el problema suele estar en estas lineas, pero no descartes una sintexis del codigo.

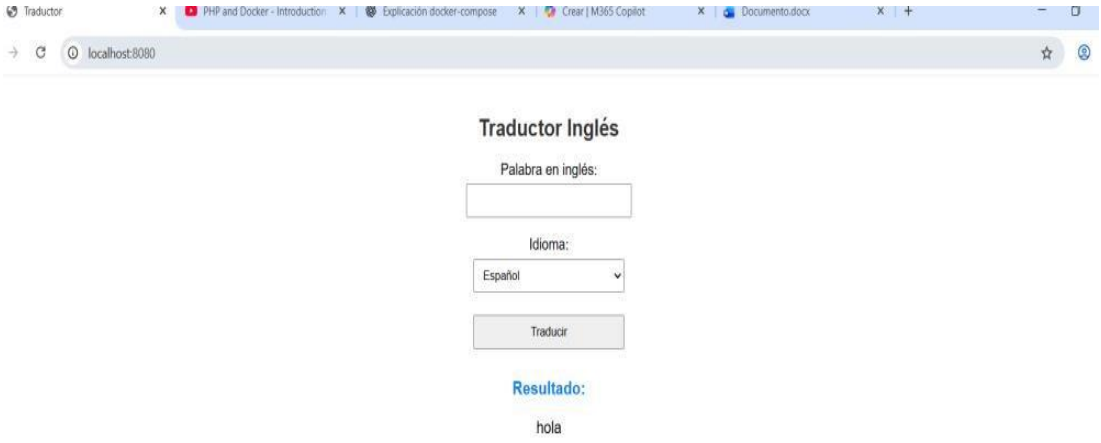
```php
// Configuración MySQL (Docker)
$host = 'basededatos';
$user = 'usuario';
$password = '1234';
$database = 'docker-php';

// Conexión MySQL
mysqli_report(MYSQLI_REPORT_ERROR | MYSQLI_REPORT_STRICT);
$conn = new mysqli($host, $user, $password);
$conn->query("CREATE DATABASE IF NOT EXISTS `$database` CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci");
$conn->select_db($database);
```

Resultado final:

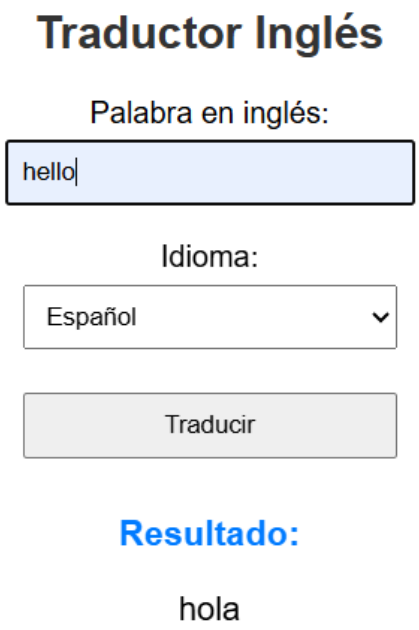
Escribimos “Localhost:8080”



Y nos redirige a nuestra página



Hacemos una consulta y cómo podemos ver nuestra página funciona y nos devuelve las peticiones que le hacemos a la misma, también los comunica que redis ha guardado el resultado de la petición. En su cache



Implementación de Node.js:

Después de crear Nuestro entorno, ahora en este caso aplicaremos nuestro entorno con Node.js. trabajaremos en este caso con JavaScrip, la funcionalidad de este Scrip será ampliar nuestra base de datos que ya anteriormente habíamos creado.

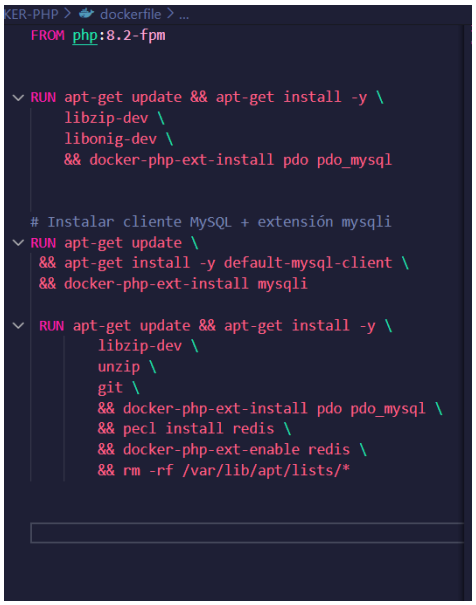
Para ello deberemos instalar en nuestro entorno la misma aplicación de Node y pm2 encargado de gestionar los servicios que proporcione Node, para ello comenzaremos configurando nuestro servicio a través de nuestro Docker-Compose y nuestro Dockerfile.

Aquí explicaremos paso a paso que funcionalidad tiene tanto los archivos de Docker como los que adicionalmente creamos para configuraciones tanto del propio servicio como de las interconexiones entre servicio o la ejecuciones de los scrip.

Configuración de Docker-Compose:

- **node:** Nombre del servicio, representa la app Node.js
- **container_name:** node_app Nombre personalizado del contenedor
- **build:** Instrucciones para construir la imagen Docker
- **context**: ./JavaScript Carpeta donde está el código fuente y el Dockerfile
- **dockerfile:** Dockerfile Archivo Dockerfile que define cómo construir la imagen
- **ports**:Puertos expuestos entre el host y el contenedor
- **"3000:3000":**Mapea el puerto 3000 del host al 3000 del contenedor
- **environment:**Variables de entorno dentro del contenedor
- **NODE_ENV:** production Indica que la app corre en modo producción
- **restart:** always Reinicia el contenedor si se detiene o falla
- **depends_on:** Define los servicios que deben levantarse antes
- **basededatos** Base de datos dependiente (por ejemplo, MySQL)
- **cache** Servicio de caché dependiente (por ejemplo, Redis)
- **networks:** Redes a las que pertenece este contenedor
- **- mi-red** Nombre de la red de conexión entre servicios

Configuración de Dockerfile:

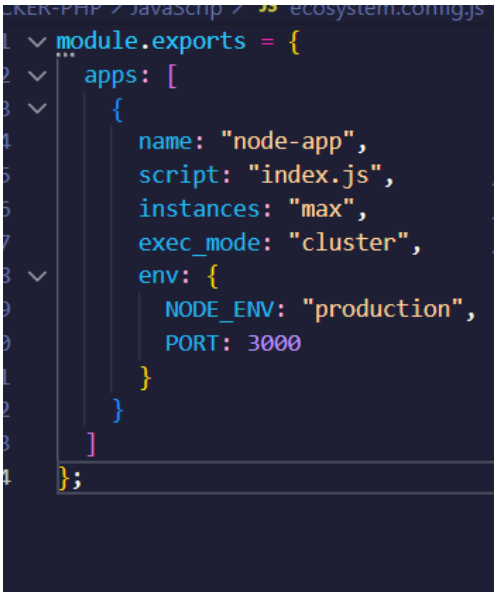


- **FROM php:8.2-fpm** Usa la imagen base de PHP 8.2 con FPM (modo FastCGI, ideal para servidores web)
- **RUN apt-get update && apt-get install -y** \ Actualiza los paquetes y luego instala librerías necesarias
- **libzip-dev** \ Librería para manejar archivos ZIP
- **libonig-dev** \ Librería usada por funciones de expresiones regulares de PHP (mbstring)
- **&& docker-php-ext-install pdo pdo_mysql** Instala las extensiones PHP PDO y PDO_MySQL para conexión a bases de datos MySQL
- **RUN apt-get update** \ Actualiza la lista de paquetes
- **&& apt-get install -y default-mysql-client** \ Instala el cliente MySQL para poder conectarse a bases de datos externas
- **&& docker-php-ext-install mysqli** Instala la extensión PHP “mysqli” para conexión MySQL clásic
- **RUN apt-get update && apt-get install -y** \ Nueva instalación combinada y más complet
- **libzip-dev** \ Soporte ZIP
- **unzip** \ Herramienta para descomprimir archivos ZIP
- **git** \ Instala Git, útil para clonar repositorios dentro del contenedor
- **&& docker-php-ext-install pdo pdo_mysql** \ Extensiones PHP para conexión a MySQL mediante PDO
- **&& pecl install redis** \ **nstala** la extensión Redis desde PECL (repositorio oficial de extensiones PHP)
- **&& docker-php-ext-enable redis** \ Activa la extensión Redis en PHP
- **&& rm -rf /var/lib/apt/lists/*** Limpia los archivos temporales de instalación para reducir el tamaño de la imagen Docker

Configuración de los archivos ecosystem.config,js y package.json:

Ecosystem.config.js

El archivo ecosystem.config.js es un archivo de configuración para PM2, un gestor de procesos avanzado para aplicaciones Node.js en producción. Su función principal es definir cómo ejecutar, monitorear y mantener viva tu aplicación Node.js de forma automática y escalable.



- **"name":** "mi-app",Nombre único de tu aplicación en PM2 (para identificarla en comandos como pm2 list, pm2 stop, etc.)
- **"version":** "1.0.0",Versión de tu aplicación (útil para tracking y deployments)
- "**dependencies":** { Paquetes Node.js que necesita tu aplicación
- "**pm2":** "^5.3.0", PM2 mismo (gestor de procesos para mantener la app viva)
- **"mysql2":** "^3.3.1", Cliente MySQL moderno y rápido para Node.js
- **"redis":** "^4.6.7" Cliente Redis oficial para Node.js (caché/sesiones)
- **"scripts":** { Scripts ejecutables (como en package.json normal)
- **"start":** "node index.js" Comando para iniciar la aplicación principal

Package.json:

El archivo package.json es el corazón de cualquier proyecto Node.js. Sirve para definir toda la información del proyecto, gestionar dependencias (paquetes externos) y crear comandos personalizados

- "**name":** "mi-app", Nombre único del proyecto (se usa para publicar en npm o identificar localmente)
- **"version":** "1.0.0", Versión del proyecto (formato semántico: mayor.menor.parche)
- "**dependencies":** Paquetes NECESARIOS para que la app funcione en producción
- "**pm2":** "^5.3.0", Gestor de procesos (mantiene la app viva, reinicios automáticos, cluster)
- **"mysql2":** "^3.3.1", Cliente MySQL moderno y rápido para Node.js
- "**redis":** "^4.6.7" Cliente Redis para caché, sesiones y colas
- **"scripts":** Comandos personalizados que puedes ejecutar con `npm run`
- **"start":** "node index.js" Inicia la aplicación ejecutando el archivo index.js

Configuración del Archivo Index,js:

Dividiremos el código en distintas partes para más fácil entendimiento de este

```
const mysql = require('mysql2/promise');
const redis = require('redis');

const dbConfig = {
    host: 'basededatos',
    user: 'usuario',
    password: '1234',
    database: 'docker-php',
    port: 3306
};

const redisClient = redis.createClient({
    url: 'redis://cache:6379'
});

async function waitForMySQL() {
    while (true) {
        try {
            const conn = await mysql.createConnection(dbConfig);
            await conn.end();
            console.log("✅ MySQL listo");
            break;
        } catch (e) {
            console.log("⏳ Esperando MySQL...");
            await new Promise(r => setTimeout(r, 2000));
        }
    }
}
```

Este código en Node.js configura la conexión a una base de datos MySQL y a un servidor Redis (pensados para un entorno con Docker), y define una función asíncrona que espera activamente a que MySQL esté disponible. La función waitForMySQL intenta conectarse repetidamente a la base de datos usando mysql2/promise; si la conexión falla, muestra un mensaje de espera y reintenta cada 2 segundos hasta lograrlo. Cuando la conexión es exitosa, la cierra inmediatamente y muestra un mensaje indicando que MySQL está listo, lo que resulta útil para asegurar que el servicio de base de datos esté operativo antes de que la aplicación continúe su ejecución.

```
async function main() {
    await waitForMySQL();

    const connection = await mysql.createConnection(dbConfig);
    await redisClient.connect();

    // ✅ CREAR TABLAS (CLAVE)
    await connection.query(`
        CREATE TABLE IF NOT EXISTS language (
            id INT AUTO_INCREMENT PRIMARY KEY,
            name VARCHAR(50) UNIQUE NOT NULL
        )
    `);

    await connection.query(`
        CREATE TABLE IF NOT EXISTS translation (
            id INT AUTO_INCREMENT PRIMARY KEY,
            phrase VARCHAR(255) NOT NULL,
            translation VARCHAR(255) NOT NULL,
            language_id INT NOT NULL,
            UNIQUE KEY unique_translation (phrase, language_id),
            FOREIGN KEY (language_id) REFERENCES language(id)
        )
    `);

    // Idiomas base
    await connection.query(`
        INSERT IGNORE INTO language (id, name) VALUES
        (1, 'Spanish'),
        (2, 'French'),
        (3, 'German')
    `);

    // Función insertar
    async function addWord(phrase, translations) {
        for (const t of translations) {
            await connection.execute(
                `INSERT INTO translation (phrase, translation, language_id)
                 VALUES (?, ?, ?)
                 ON DUPLICATE KEY UPDATE translation = VALUES(translation)`,
                [phrase, t.text, t.langId]
            );

            const cacheKey = `traduccion:${phrase}:${t.langId}`;
            await redisClient.del(cacheKey); // invalidar cache
        }

        console.log(`✅ Palabra "${phrase}" insertada/actualizada`);
    }
```

Este fragmento define la función principal de la aplicación, que primero espera a que MySQL esté disponible y luego establece las conexiones a MySQL y Redis. A continuación, crea las tablas language y translation si no existen, incluyendo claves únicas y una relación por clave foránea para mantener la integridad de los datos. Inserta algunos idiomas base evitando duplicados y define la función addWord, que permite insertar o actualizar traducciones de una frase según el idioma usando ON DUPLICATE KEY UPDATE.Cada vez que se modifica una traducción, se invalida la clave correspondiente en Redis para asegurar que el caché no quede desactualizado.

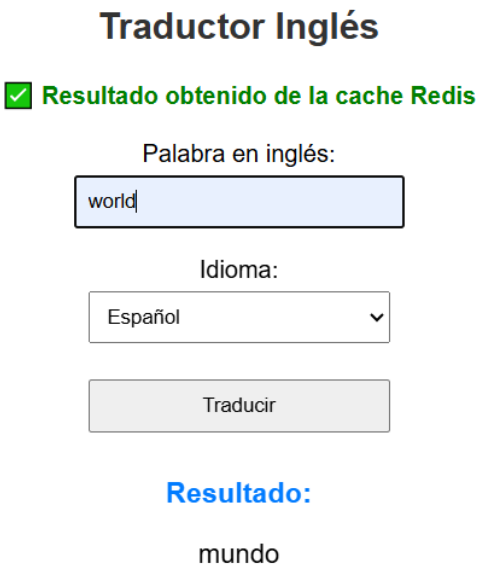
```
    await addWord('world', [
        { langId: 1, text: 'mundo' },
        { langId: 2, text: 'monde' },
        { langId: 3, text: 'welt' }
    ]);

    await connection.end();
    await redisClient.quit();
}

main().catch(console.error);
```

Este código utiliza la función addWord para insertar o actualizar la palabra **"world"** junto con sus traducciones en español, francés y alemán, asociándolas a sus respectivos identificadores de idioma. Tras completar la inserción, se cierran correctamente la conexión a MySQL y la conexión con Redis para liberar recursos. Finalmente, se ejecuta la función main y se captura cualquier error que ocurra durante el proceso, mostrándolo en consola, lo que asegura un cierre controlado y una gestión básica de errores en la aplicación.

Interconexión de Node con MySQL y NginX:

Una vez configurado todos nuestros archivos, debemos añadirlo a la red para que Node pueda ejecutar el Scrip y añadir la información que pedimos a la página web para ello, como previamente hemos configurado en el compose la red a la que se va a conectar nuestro servicio, como podemos comprobar a continuación la todo esta dentro de la misma:

Como comprobamos tenemos los servicios funcionando dentro de la misma red por lo cual Node.js cuando ejecute el Scrip y como hemos visto actualiza y redireccionada la información para que una vez levantemos nuestros contenedores se actualice en nuestra página web



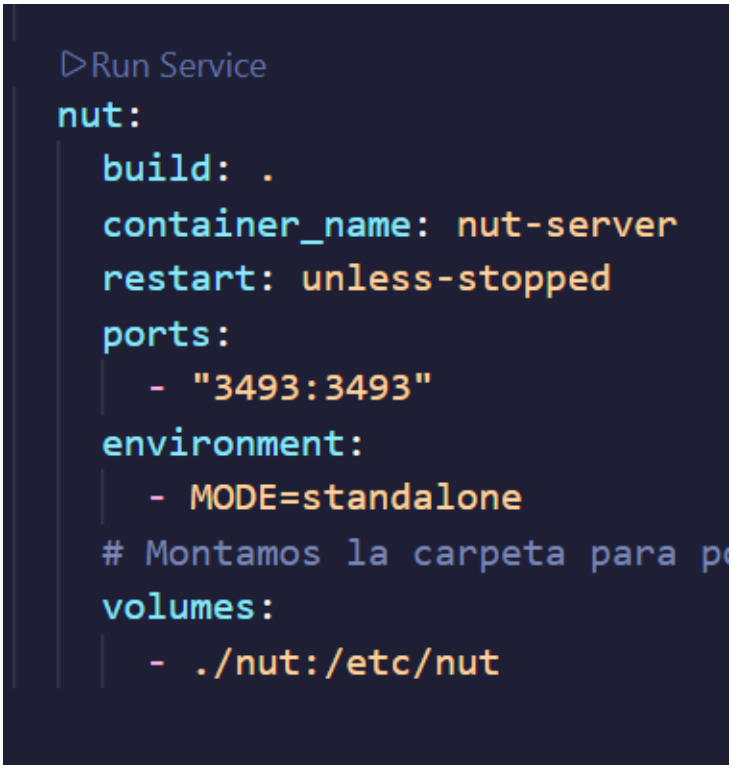
Como vemos una vez ejecutado los containeres con Docker.compose up automáticamente nuestro scrip ha actualizado las palabras y ahora tenemos la palabra world.

Configuración de NUT:

NUT significa Network UPS Tools:Es un software que permite monitorear y controlar sistemas de alimentación ininterrumpida (UPS) desde un servidor o red,Por ejemplo, puede alertarte cuando la batería del UPS está baja, registrar eventos de corte de energía o apagar servidores de forma segura.

Para comenzar con la configuración de NUT debemos de crear distintos archivos encargados de la configuración de los drivers de NUT y también como es lógico su dockerfile dedicado y su apartado en nuestro Docker-compose .

Configuración de Docker-compose-yaml (NUT)



Nut: Define un servicio llamado nut, que será gestionado por Docker Compose y representa una instancia de la aplicación NUT dentro de un contenedor.

Build: La imagen del contenedor se construye a partir del Dockerfile ubicado en el directorio actual del proyecto.

container_name: El contenedor tendrá el nombre fijo nut-server, evitando que Docker asigne un nombre automático.

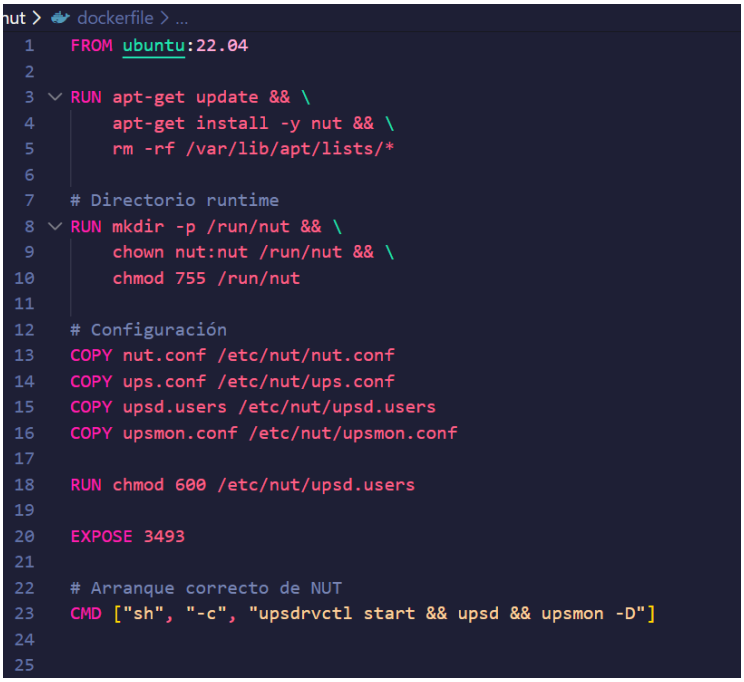
Restart: El contenedor se reiniciará automáticamente si se detiene de forma inesperada o si el sistema se reinicia, excepto cuando sea detenido manualmente.

Ports: Se expone el puerto 3493, enlazando el puerto 3493 del host con el puerto 3493 del contenedor para permitir el acceso al servicio desde el exterior.

Environment: Se define la variable de entorno MODE con el valor standalone, que controla el modo de funcionamiento de la aplicación dentro del contenedor.

Volúmenes: monta la carpeta local ./nut dentro del contenedor en la ruta /etc/nut, permitiendo editar los archivos

Configuración de Dockerfile:



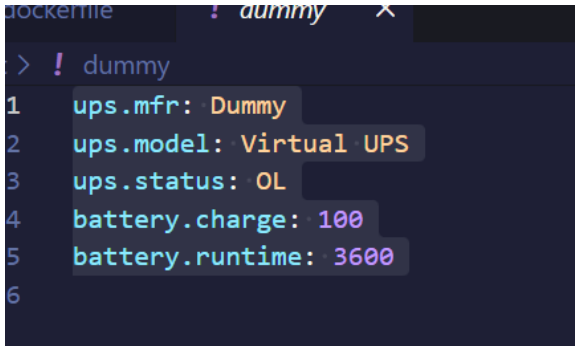
- **FROM ubuntu:22.04** Imagen base del contenedor, en este caso Ubuntu 22.04.
- **apt-get update** Actualiza la lista de paquetes disponibles en el sistema.
- **apt-get install -y nut** Instala NUT (Network UPS Tools) sin preguntar confirmación.
- **rm -rf /var/lib/apt/lists/** Limpia la caché de apt para reducir el tamaño de la imagen.**mkdir -p /run/nut** Crea el directorio donde NUT guarda archivos temporales en ejecución.
- **chown nut:nut /run/nut** Asigna al usuario y grupo nut la propiedad del directorio.
- **chmod 755 /run/nut** Permisos de lectura/ejecución para todos, escritura solo para el propietario.
- **COPY nut.conf /etc/nut/nut.conf** Copia la configuración general de NUT al contenedor.
- **COPY ups.conf /etc/nut/ups.conf** Copia la configuración de las UPS definidas.
- **COPY upsd.users /etc/nut/upsd.users** Copia los usuarios y contraseñas para el demonio upsd.
- **COPY upsmon.conf /etc/nut/upsmon.conf** Copia la configuración del monitor de UPS.
- **chmod 600 /etc/nut/upsd.users** Permisos restrictivos para proteger las credenciales.

- **EXPOSE 3493** Documenta y permite mapear el puerto donde NUT escucha conexiones de clientes.
- **upsdrvctl start** →Inicia el controlador de UPS.
- **upsd** Inicia el demonio de NUT que permite la comunicación con clientes.
- **upsmon -D** Arranca el monitor de UPS en primer plano (modo debug).
- **CMD ["sh", "-c", "..."]** Ejecuta todos los comandos de arranque en una sola línea de shell al iniciar el contenedor.

Archivos de configuración de NUT:

Los archivos que configuraremos en este caso son los archivos de gestión de driver, de monitoreo… pasaremos por cada uno explicando el código y para que sirven cada uno.

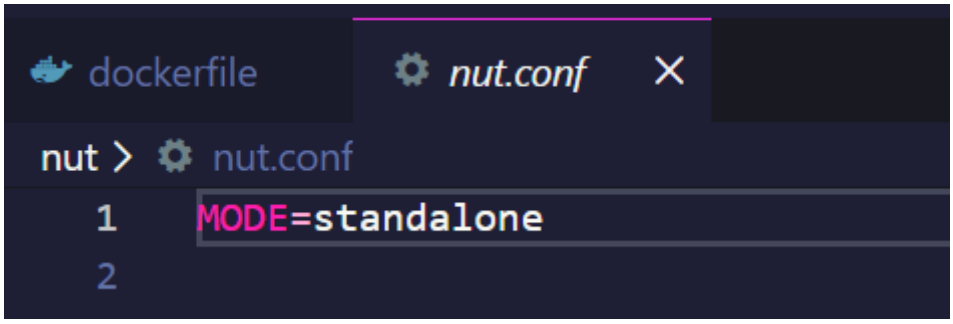
DUMMY



- **ups.mfr: Dummy** Fabricante de la UPS; aquí es “Dummy” porque no es real, es soloparasimulación.
- **ups.model: Virtual UPS** Modelo de la UPS; se indica que es virtual, no hardware real.
- **ups.status: OL** Estado de la UPS; “OL” significa **Online**, que la UPS está alimentando la carga normalmente con corriente de red.
- **battery.charge: 100** Porcentaje de batería; indica que está completamente cargada.
- **battery.runtime: 3600** Tiempo estimado de autonomía en segundos; aquí 3600 s = 1 hora.

NUT.CONF

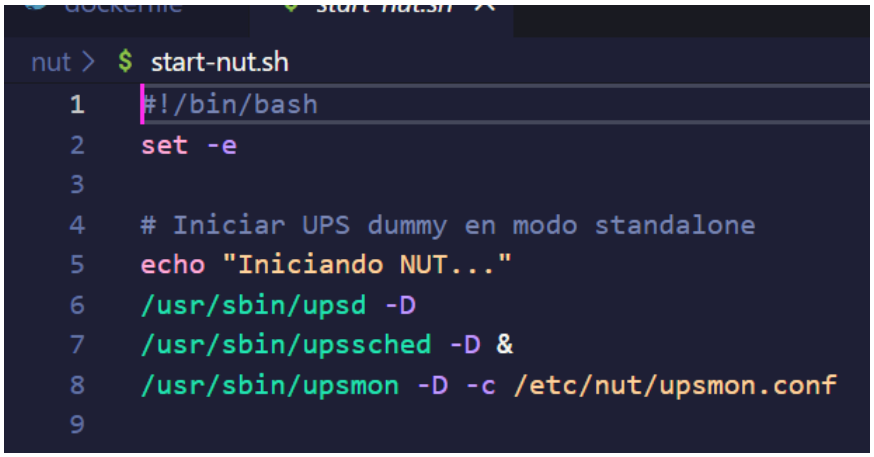
Es el archivo principal de configuración de NUT (Network UPS Tools). Define cómo se va a ejecutar NUT en el sistema, es decir, si trabajará en modo standalone, como servidor (master), cliente (slave), etc



standalone Significa que el demonio NUT se ejecuta localmente y gestiona la UPS directamente sin depender de otros servidores o clientes.

START-NUSH.SH

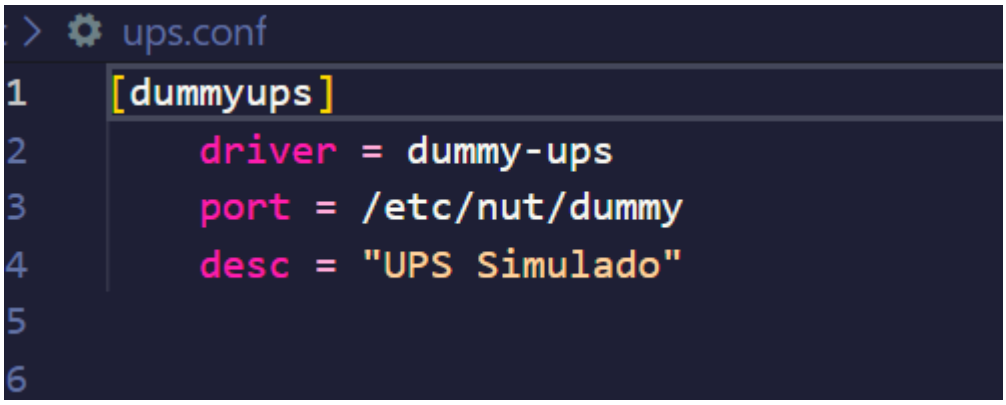
El archivo start-nut.sh es un script de arranque para NUT, pensado para iniciar todos los servicios necesarios de NUT en un contenedor o sistema local. Es útil porque agrupa varios comandos en un solo script y asegura que NUT se ejecute correctamente en modo standalone



- **set -e** hace que el script se detenga inmediatamente si algún comando devuelve un error.
- **echo "Iniciando NUT..."** imprime un mensaje en la terminal informando al usuario que se está iniciando NUT (Network UPS Tools).
- **/usr/sbin/upsd -D** inicia el demonio del servidor UPS en segundo plano, con depuración activada para mostrar mensajes de registro.
- **/usr/sbin/upssched -D** & ejecuta el programador de eventos de NUT en segundo plano con depuración, permitiendo gestionar acciones automáticas del UPS.
- **/usr/sbin/upsmon -D** -c /etc/nut/upsmon.conf inicia el monitor de UPS con depuración, usando el archivo de configuración /etc/nut/upsmon.conf para supervisar el UPS y tomar acciones según el estado.
-

UPS.CONF

El archivo ups.conf es el archivo de configuración principal de NUT (Network UPS Tools) donde se definen los UPS que el sistema va a manejar. Cada UPS se declara con un nombre único y se especifican parámetros como el driver, el puerto y una descripción.

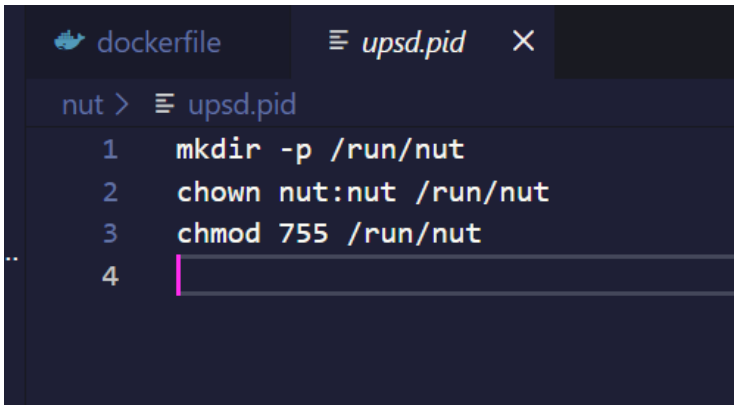


- **[dummyups]** define el nombre del UPS. Este es el identificador que NUT usará para referirse a este UPS dentro del sistema.
- **driver = dummy-ups** indica qué controlador usar para este UPS. En este caso, dummy-ups es un controlador simulado que no requiere hardware real, útil para pruebas o entornos de desarrollo.

- **port = /etc/nut/dummy** especifica el puerto o archivo que el UPS usará para comunicarse. Para un UPS real, aquí se pondría el puerto serie, USB o IP; en este caso se usa un archivo simulado.
- **desc =** "UPS Simulado" proporciona una descripción legible del UPS, que aparece en los registros o interfaces de NUT para identificarlo fácilmente como un UPS de prueba.

UPSD.PID

El archivo upsd.pid almacena el PID (Process ID) del demonio upsd cuando se ejecuta. Esto permite que otros procesos, como upsmon o scripts de administración, sepan si el demonio está activo y puedan controlarlo



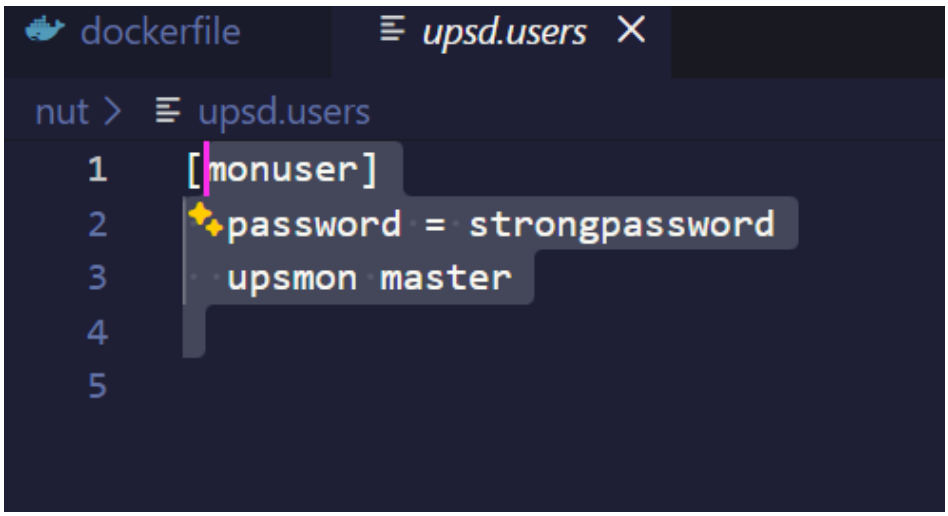
kdir -p /run/nut crea el directorio /run/nut si no existe. La opción -p asegura que no falle si el directorio ya existe y que cree todos los directorios intermedios necesarios.

chown nut:nut /run/nut cambia el propietario y el grupo del directorio /run/nut al usuario y grupo nut, asegurando que solo el servicio NUT tenga permisos de escritura allí.

chmod 755 /run/nut establece los permisos del directorio /run/nut para que el propietario tenga permisos de lectura, escritura y ejecución, y los demás usuarios solo puedan leer y ejecutar, pero no escribir. Esto garantiza que NUT pueda crear su archivo upsd.pid de manera segura, mientras que otros usuarios no puedan modificarlo.

UPSD.USER

El archivo upsd.users define los usuarios que pueden conectarse al demonio upsd para monitorear o controlar los UPS. Cada usuario tiene un nombre, una contraseña y permisos específicos



[monuser] define el nombre del usuario que se conectará al demonio upsd. Este nombre será usado por upsmon u otras herramientas de NUT para autenticar la conexión.

password = strongpassword asigna la contraseña para este usuario. Esta contraseña se usa para autenticar la conexión al demonio upsd y proteger el acceso al UPS.

upsmon master indica que este usuario tiene permisos de **master**, lo que significa que puede ejecutar comandos críticos sobre el UPS (como apagar el sistema si hay un corte de energía prolongado). También permite que upsmon funcione como monitor principal de este UPS.

UPSMON.CONF

El archivo upsmon.conf configura el monitor de UPS (upsmon), que es el componente de NUT encargado de supervisar el estado del UPS y ejecutar acciones automáticas (como apagar el sistema en caso de corte de energía)

upsmon.conf

```
MONITOR dummyups@localhost 1 monuser strongpassword master
```

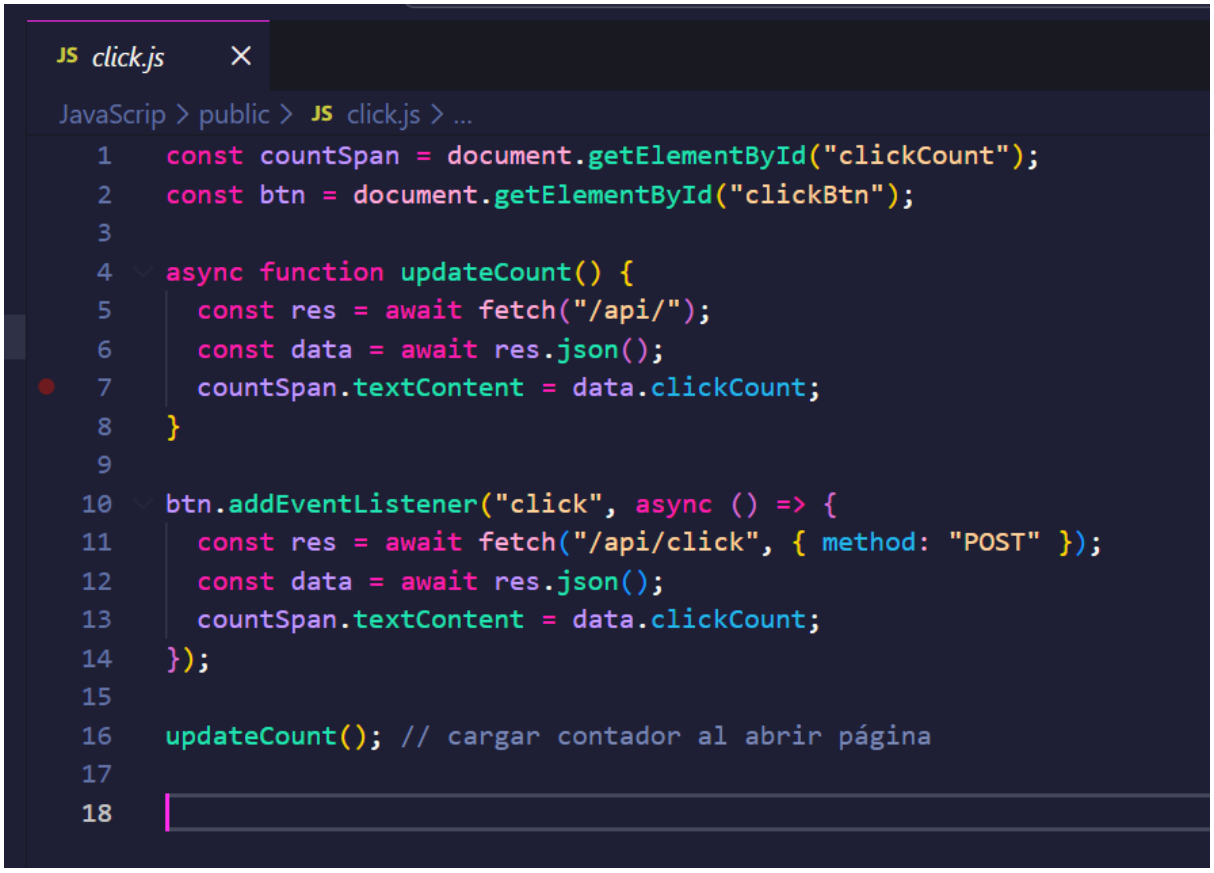
- **MONITOR dummyups@localhost 1 monuser strongpassword master** indica que upsmon debe monitorear el UPS llamado dummyups que se encuentra en el host localhost. El número 1 representa la prioridad o nivel de alerta para este monitor. monuser es el usuario que se usará para autenticarse con upsd, strongpassword es la contraseña de ese usuario, y master indica que este monitor tiene permisos de master, lo que le permite recibir información completa del UPS y ejecutar acciones críticas si es necesario.

Ampliación de Node.js con mas funcionalidades:

A continuación, procederemos a la ampliación de complementos de nuestra pagina web a través de java Scrip, esto lo implementaremos a través de Node.js, lo que requerimos en este caso es la colocación de un contador que vaya guardando la información en nuestra base de datos y a la vez un listado de logs, para ello comenzaremos con una seria de configuraciones que veremos ahora:

Configuración de PUBLIC:

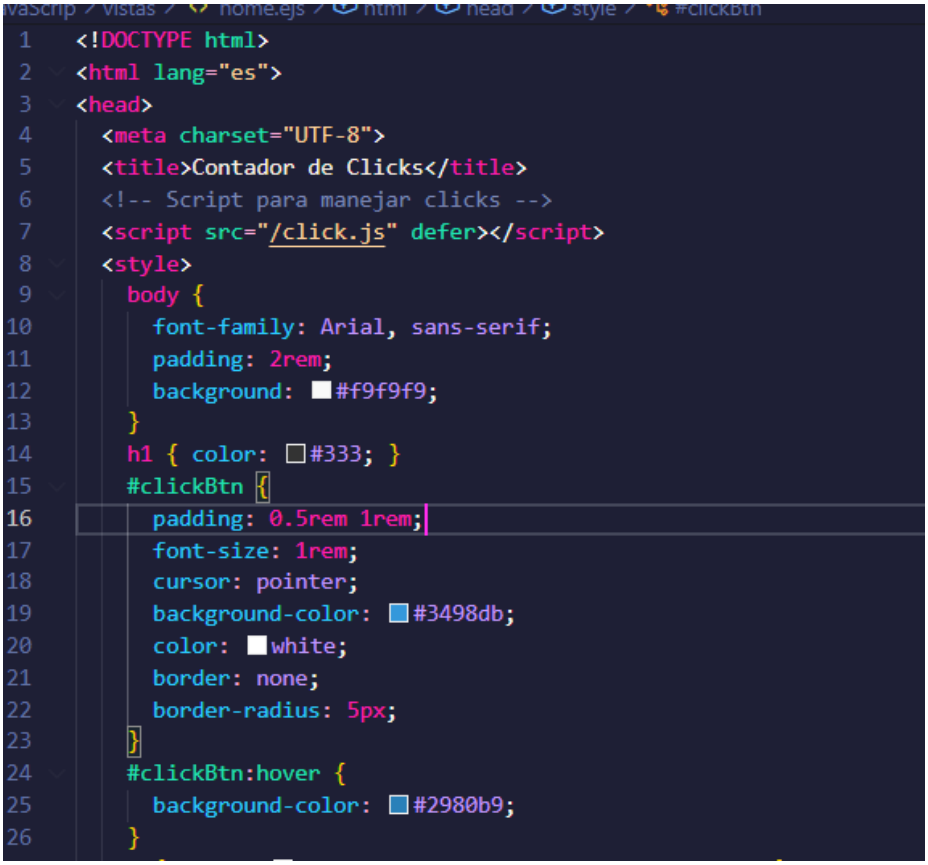
Principalmente lo que haremos será crear una carpeta la cual llamaremos “public”, esta carpera albergara el código de sincronizar nuestro contador de click con nuestro servidor, haciéndolo a su vez persistente en el Backend para así poder compartise entre usuarios sin ningún problema aparente:



Configuración de VIEW (home.EJS/logs.EJS)

HOME.EJS:

Este código HTML junto con CSS crea una página web sencilla llamada “Contador de Clicks”. Define la estructura básica con un título y carga un archivo JavaScript externo (click.js) que presumiblemente maneja la lógica del contador de clics



Este código crea una página con un contador de clicks que muestra su valor inicial desde el servidor, un botón para incrementarlo y un enlace a los logs. Usa un script que actualiza el contador desde la API (/api/) y maneja errores si falla la carga.

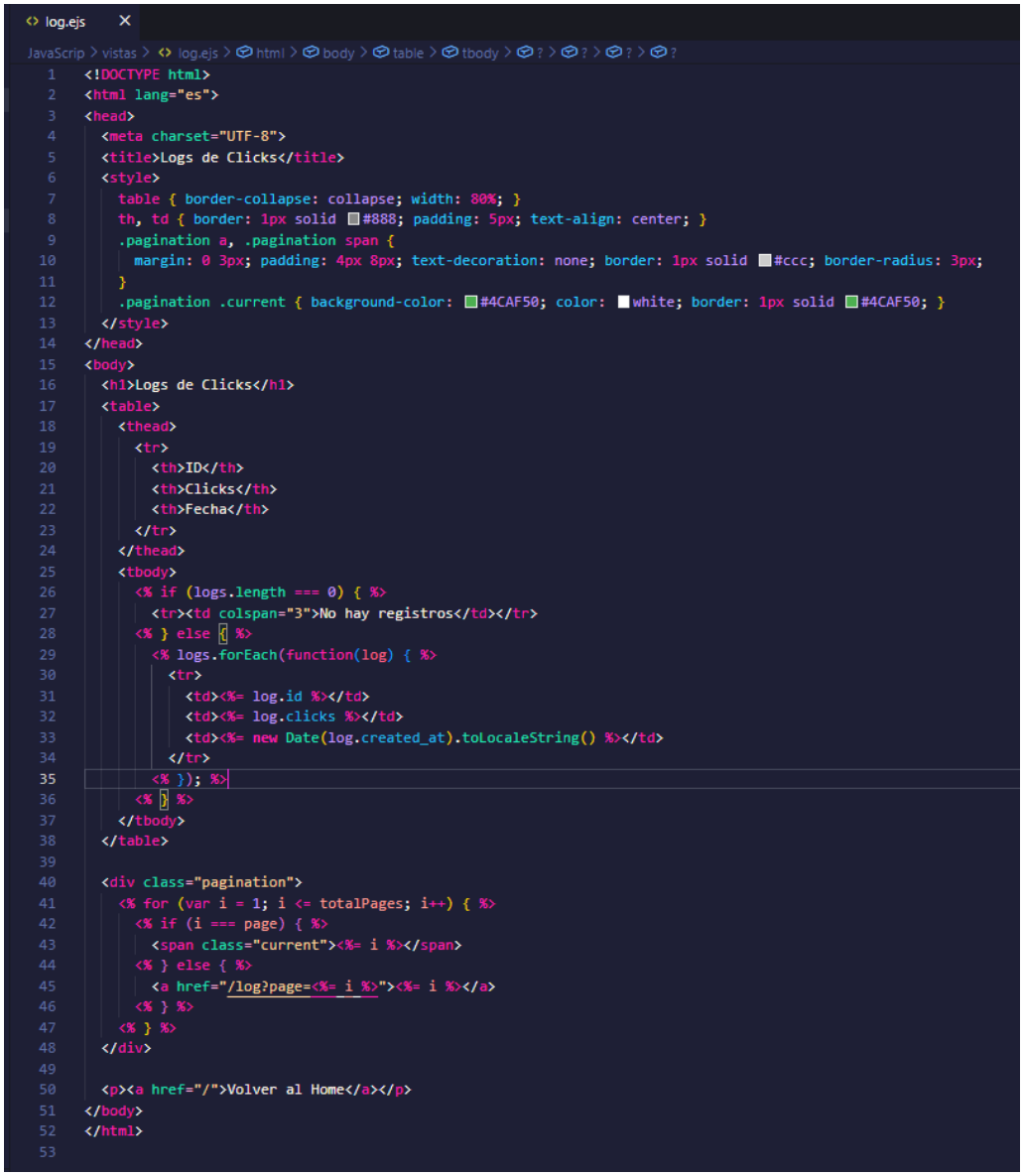
```
    }
    a { color: #3498db; text-decoration: underline; }
  </style>
</head>
<body>
  <h1>Contador de Clicks</h1>

  <!-- Mostrar el contador inicial -->
  <p>Clicks totales: <span id="clickCount"><%= clickCount %></span></p>

  <!-- Botón para incrementar clicks -->
  <button id="clickBtn">Haz Click</button>

  <!-- Enlace a los logs -->
  <p><a href="/log.html">Ver logs</a></p>

  <!-- Script inline para actualizar contador al cargar -->
  <script>
    const countSpan = document.getElementById("clickCount");
    const btn = document.getElementById("clickBtn");

    // Función para actualizar contador desde la API
    async function updateCount() {
      try {
        const res = await fetch("/api/");
        const data = await res.json();
        countSpan.textContent = data.clickCount;
      } catch (err) {
        console.error("Error cargando contador:", err);
      }
    }
```

Este código hace que el botón de la página registre clicks enviando un POST a /api/click cada vez que se pulsa, actualizando el con el total recibido del servidor y mostrando errores en la consola si algo falla; además, al cargar la página, llama a updateCount() para mostrar desde el inicio el contador actual de clicks.

```
    // Evento click
    btn.addEventListener("click", async () => {
      try {
        const res = await fetch("/api/click", { method: "POST" });
        const data = await res.json();
        countSpan.textContent = data.clickCount;
      } catch (err) {
        console.error("Error enviando click:", err);
      }
    });

    // Inicializa el contador al cargar la página
    updateCount();
  </script>
</body>
</html>
```

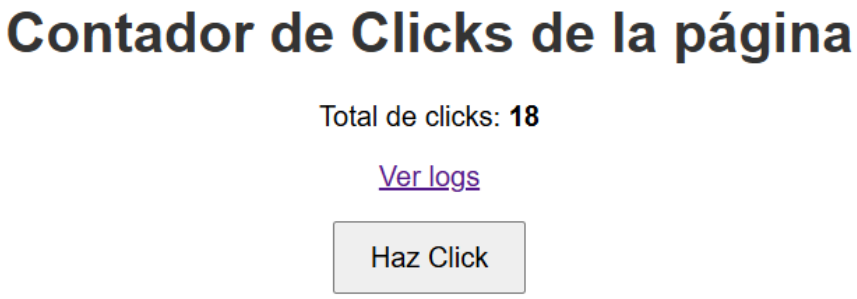
LOGS.EJS:

Este código genera una página de logs de clicks usando un motor de plantillas (como EJS), mostrando una tabla con las columnas ID, clicks y fecha de cada registro; si no hay logs, muestra un mensaje indicando “No hay registros”. Además, incluye paginación que resalta la página actual y permite navegar entre ellas mediante enlaces, y un enlace al inicio de la web. El estilo define bordes y espaciado para la tabla y los botones de paginación, haciendo la presentación clara y ordenada.

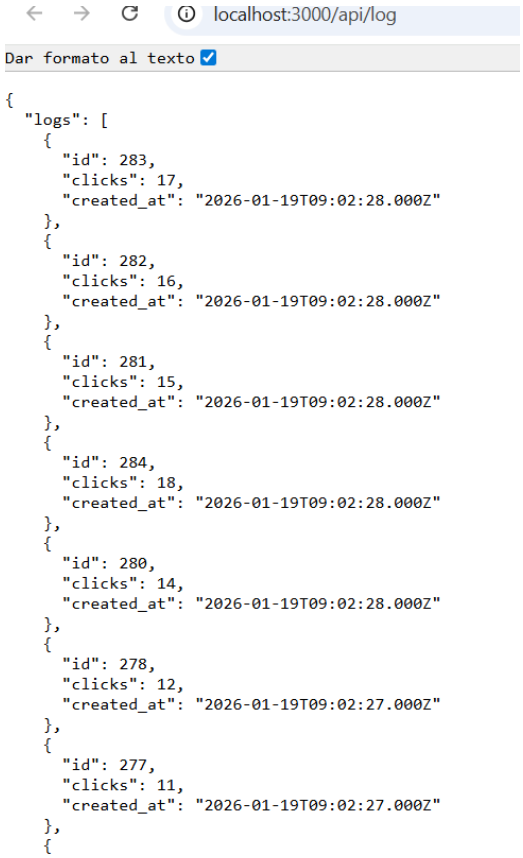


Resultado Final:

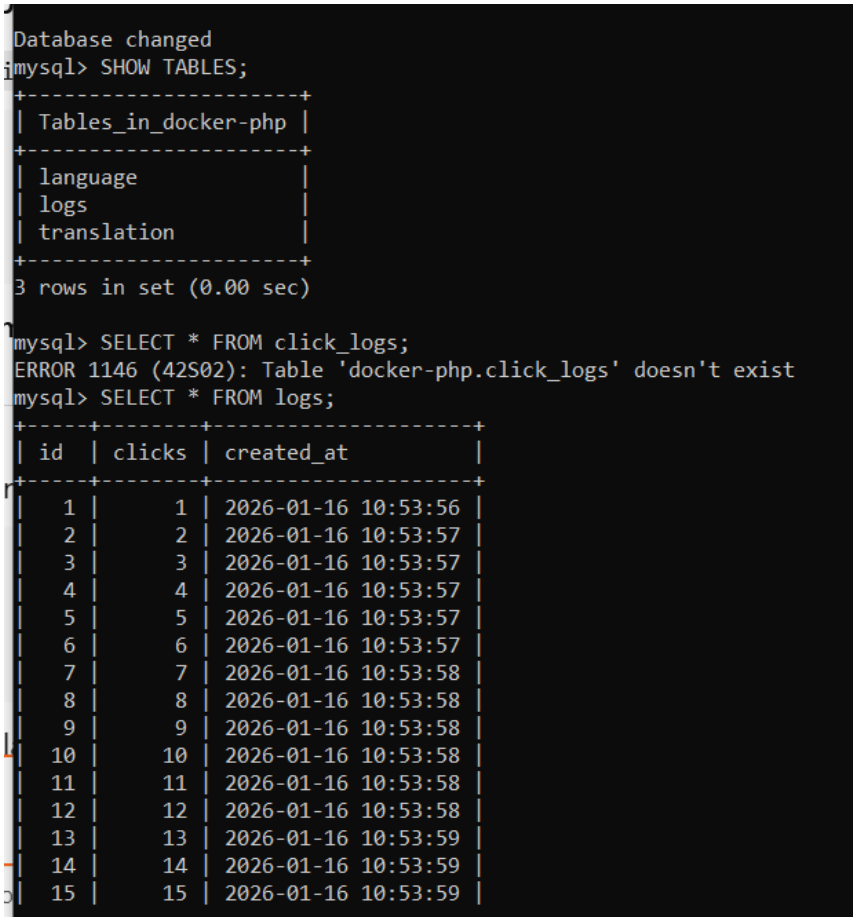
Como hemos visto al principio del apartado, vemos que hemos creado botón contador, que actualiza la información dentro de mi MySQL y también que genera logs que guardan hora/fecha del registro.



Cuando le damos a “Haz click” el contador se eleva, también esta implementado que si recargamos la página vuelva a 0. Ahora veremos como se guardan los logs:



Finalmente, como podemos observar, nuestro registro de logs también se guarda en nuestra base de datos:



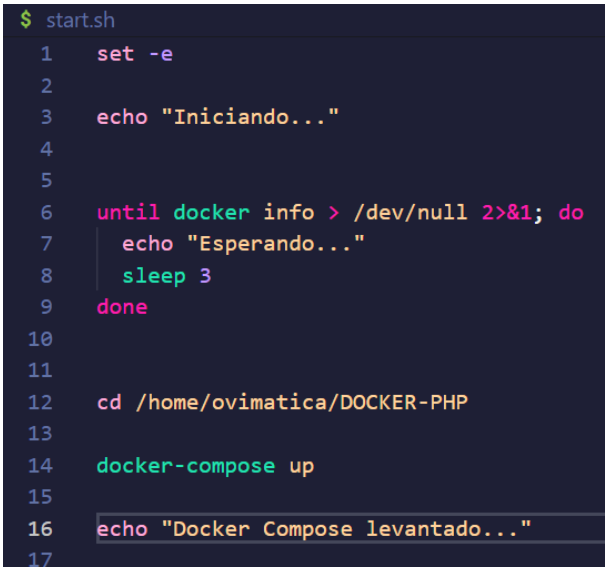
```
Database changed
mysql> SHOW TABLES;
+----------------------+
| Tables_in_docker-php |
+----------------------+
| language             |
| logs                 |
| translation          |
+----------------------+
3 rows in set (0.00 sec)

mysql> SELECT * FROM click_logs;
ERROR 1146 (42S02): Table 'docker-php.click_logs' doesn't exist
mysql> SELECT * FROM logs;
+-----+--------+---------------------+
| id  | clicks | created_at          |
+-----+--------+---------------------+
|   1 |      1 | 2026-01-16 10:53:56 |
|   2 |      2 | 2026-01-16 10:53:57 |
|   3 |      3 | 2026-01-16 10:53:57 |
|   4 |      4 | 2026-01-16 10:53:57 |
|   5 |      5 | 2026-01-16 10:53:57 |
|   6 |      6 | 2026-01-16 10:53:57 |
|   7 |      7 | 2026-01-16 10:53:58 |
|   8 |      8 | 2026-01-16 10:53:58 |
|   9 |      9 | 2026-01-16 10:53:58 |
|  10 |     10 | 2026-01-16 10:53:58 |
|  11 |     11 | 2026-01-16 10:53:58 |
|  12 |     12 | 2026-01-16 10:53:58 |
|  13 |     13 | 2026-01-16 10:53:59 |
|  14 |     14 | 2026-01-16 10:53:59 |
|  15 |     15 | 2026-01-16 10:53:59 |
```

Configuración de Start.sh y Stop.sh

Finalmente, y para concluir este seguimiento, crearemos 2 Scripts simple para el apagado y el encendido de nuestro entorno, con ellos facilitaremos su manejo a la hora de utilizarlo, para ello dentro del directorio donde tenemos configurado nuestro entorno y por consecuente nuestro Docker-compose, crearemos nuestro dos Scripts

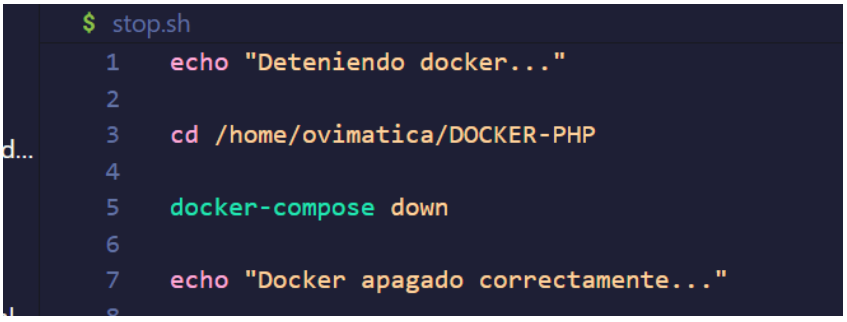
Start.sh:

Como podemos ver en este Scripts primero activa set -e para que el script se detenga si ocurre algún error, luego imprime “Iniciando...”. Usa un bucle until que espera hasta que Docker esté listo (docker info), mostrando “Esperando...” y esperando 3 segundos entre intentos. Después cambia al directorio del proyecto (/home/ovimatica/DOCKER-PHP) y ejecuta docker-compose up para levantar los contenedores definidos en el docker-compose.yml. Al finalizar, imprime “Docker Compose levantado” para indicar que todo está funcionando.

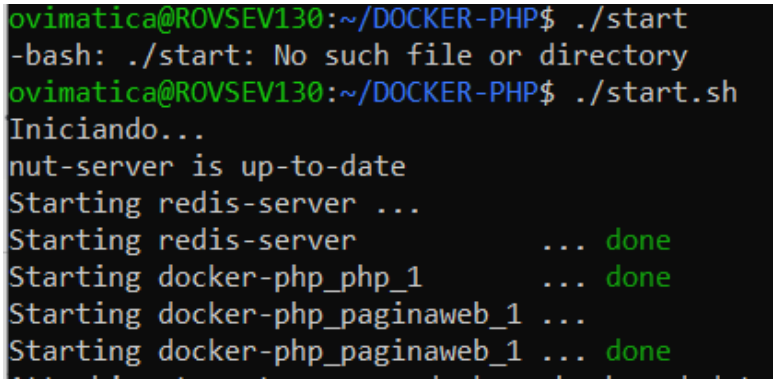


Stop.sh

Este script de bash se encarga de detener los contenedores Docker de un proyecto. Primero imprime “Deteniendo docker…”, luego cambia al directorio del proyecto (/home/ovimatica/DOCKER-PHP) y ejecuta docker-compose down, que apaga y elimina los contenedores, redes y volúmenes temporales creados por Docker Compose. Finalmente muestra “Docker apagado correctamente…” para indicar que todo se detuvo con éxito.



Para la comprobación de su funcionalidad tenemos que poner el comando “./start.sh”



E igual para “./Stop.sh”