

Práctica docker-compose

Documentar correctamente y de forma clara el proceso de dockerización de una aplicación de servidor utilizando docker-compose, podéis utilizar la aplicación de la asignatura de servidor.

Así pues, partiendo de la misma rama main, crearéis una nueva rama denominada main_docker_compose que subiréis al repositorio remoto. Sobre esta y en la carpeta raíz de vuestro código, deberá existir un fichero docker-compose.yml que implementará lo siguiente:

- Servicio de mongodb
 - Utilizará la imagen docker oficial de mongo ([link](#))
 - El contenedor asociado se denominará mongo_container
 - Será el primer servicio en arrancar
 - La primera tarea que hará nada más arrancar será crear las tablas necesarias ([link](#)) y realizar una restauración de datos ([link](#)) partiendo de un fichero dump ([link](#)) que previamente habréis generado y almacenado en una carpeta mongo de vuestro proyecto.
 - Todos los ficheros de configuración necesarios residirán en una carpeta mongo de nuestro repositorio
- Servicio de backend
 - Igual que el servicio anterior, utilizará una multi-stage build (al menos tendrá dos etapas) para generar la imagen de la parte backend de vuestro proyecto implementada con express. Partirá de una imagen node:19-alpine
 - No arrancará hasta que el servicio de mongodb no esté preparado completamente
 - El contenedor asociado se denominará backend_container
 - Ejecutará como primer comando nada más arrancar: npm start
- Servicio de frontend
 - Utilizará una multi-stage build (al menos tendrá dos etapas) para generar la imagen de vuestra parte implementada en angularjs o otro framework. Partirá de una imagen node:19-alpine
 - Arrancará tras el servicio de backend
 - El contenedor asociado se denominará frontend_container
 - Ejecutará como primer comando nada más arrancar: npm start
- Servicio mongo-express.
 - Nos permitirá administrar la base de datos mongo. Utilizará la imagen oficial de mongo-express ([link](#))
 - El contenedor asociado se denominará adminMongo_container
 - Arrancará después del servicio de mongodb
- Servicio de loadbalancer de nginx
 - Nos permitirá implementar un sistema de balanceo de carga/proxy en nuestro sistema
 - Partirá de la imagen oficial de nginx

- Asociará un fichero de configuración de nginx (nginx.conf) que tendremos en la carpeta loadbalancer de nuestro repositorio con el mismo fichero de la carpeta /etc/nginx/ de la imagen. Este fichero nos permitirá implementar el balanceador de carga y su contenido será similar al adjuntado a esta tarea (realizando las modificaciones oportunas para adecuarlo a vuestras necesidades).

- Ejecutará como primer comando nada más arrancar: `nginx -g daemon off`

- Servicios de monitorización (opcionales)

- Servicio Prometheus

Prometheus es una aplicación que nos permite recoger métricas de una aplicación en tiempo real. Como veréis en el ejemplo de app.js, se incluye una dependencia en el código (prom-client) que permite crear contadores de peticiones que podemos asociar fácilmente a nuestros endpoints de manera que podemos saber cuántas veces se ha llamado a una función de nuestra api. En nuestro caso, el servicio de prometheus se encargará de arrancar en el puerto 9090 de nuestro host un contenedor (prometheus_practica) basado en la imagen prom/prometheus:v2.20.1. Para poder configurar correctamente este servicio, será necesario realizar además dos acciones:

- Copiar el fichero adjunto prometheus.yml al directorio /etc/prometheus del contenedor
- Ejecutar el comando `--config.file=/etc/prometheus/prometheus.yml`

- Servicio Grafana

Este servicio será el encargado de graficar todas las métricas creadas por el servicio de Prometheus. Por tanto, siempre arrancará tras el de prometheus. En nuestro caso, el servicio de grafana se encargará de arrancar en el puerto 3500 de nuestro host un contenedor (grafana_practica) basado en la imagen grafana/grafana:7.1.5 que, además, se caracterizará por:

- Establecer las variables de entorno necesarias para:
 - Deshabilitar el login de acceso a Grafana
 - Permitir la autenticación anónima
 - Que el rol de autenticación anónima sea Admin
 - Que instale el plugin grafana-clock-panel 1.0.1
- Dispondrá de un volumen nombrado (myGrafanaVol) que permitirá almacenar los cambios en el servicio ya que se asociará con el directorio /var/lib/grafana

Además, para una correcta configuración de Grafana, será necesario realizar la copia del fichero adjunto datasources.yml al directorio del contenedor /etc/grafana/provisioning/datasources/.

Como ayuda aquí tenéis un ejemplo de proyecto realizado en años anteriores que se integraba Grafana y prometheus con la aplicación desarrollada en el módulo de servidor:

- https://vicnx.github.io/Metrics_Prometheus_Grafana_Nodejs/

A tener en cuenta:

- Con toda seguridad necesitareis modificar a nivel de código las referencias a endpoints o url's que tengan que ver con la base de datos para que la aplicación funcione correctamente.
- Necesitareis que todos los servicios estén conectados en una misma red (practica_net). Por supuesto, cada uno de ellos gestionará unos puertos que, de no indicarse uno específico, podéis elegir libremente.
- Todas las variables de entorno que necesiten los servicios estarán en un fichero .env común del proyecto que cada servicio leerá.

Como respuesta a la actividad planteada, entregaréis el link a vuestro repositorio github. En el README.md de la rama creada deberéis documentar los principales pasos/cambios realizados para poder implementar la configuración pedida y los pasos necesarios para poder poner en marcha el proyecto.

Para empezar con este proyecto, vamos a crear el repositorio github y después lo clonaremos en el ordenador para empezar a trabajar desde ahí:

The screenshot shows the GitHub 'Create a new repository' page. The browser address bar shows 'github.com/new'. The page title is 'Create a new repository'. Below the title, there is a note: 'A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)'. A note below that says 'Required fields are marked with an asterisk (*)'. The form has two main sections: 'Owner' and 'Repository name'. The 'Owner' dropdown is set to 'vaoenriquejordi'. The 'Repository name' text input contains 'PracticaDockerCompose2'. Below these, there is a green checkmark message: 'Your new repository will be created as PracticaDockerCompose2024JordiVa-Enrique. The repository name can only contain ASCII letters, digits, and the characters ., -, and _.' Below this, there is a note: 'Great repository names are short and memorable. Need inspiration? How about [urban-waffle](#) ?'. The 'Description (optional)' text input contains 'Práctica Final Docker Compose 2024 - Jordi Vañó Enrique'. Below the description, there are two radio buttons for 'Visibility': 'Public' (selected) and 'Private'. The 'Public' option has a sub-note: 'Anyone on the internet can see this repository. You choose who can commit.' The 'Private' option has a sub-note: 'You choose who can see and commit to this repository.' Below the visibility options, there is a section 'Initialize this repository with:'. It has two checkboxes: 'Add a README file' (checked) and 'Add .gitignore' (unchecked). Below the 'Add a README file' checkbox, there is a note: 'This is where you can write a long description for your project. [Learn more about READMEs.](#)'. Below the 'Add .gitignore' checkbox, there is a dropdown menu for '.gitignore template: None'. At the bottom, there is a link: '[Choose which files not to track from a list of templates. Learn more about ignoring files.](#)'

Una vez hecha la clonación, accedemos al terminar y creamos una nueva rama en git mediante los siguientes comandos “git checkout main”, “git pull origin main” y “git checkout -b main_docker_compose” tal y como se muestra en la siguiente pantalla:

```
PS C:\Users\vano\ Desktop \PracticaDockerCompose2024JordiVa-Enrique> git checkout main
Already on 'main'
Your branch is up to date with 'origin/main'.
PS C:\Users\vano\ Desktop \PracticaDockerCompose2024JordiVa-Enrique> git pull origin main
From https://github.com/vanoenriquejordi/PracticaDockerCompose2024JordiVa-Enrique
* branch      main      -> FETCH_HEAD
Already up to date.
PS C:\Users\vano\ Desktop \PracticaDockerCompose2024JordiVa-Enrique> git checkout -b main_docker_compose
Switched to a new branch 'main_docker_compose'
PS C:\Users\vano\ Desktop \PracticaDockerCompose2024JordiVa-Enrique>
```

Una vez creada la rama, nos aseguramos que estamos dentro de la rama creada “main_docker_compose” mediante el comando “git branch -a” y seguimos con la elaboración del proyecto:

```
PS C:\Users\vano\ Desktop \PracticaDockerCompose2024JordiVa-Enrique> git branch -a
main
* main_docker_compose
remotes/origin/HEAD -> origin/main
remotes/origin/main
PS C:\Users\vano\ Desktop \PracticaDockerCompose2024JordiVa-Enrique>
```

Acto seguido, procedemos a realizar (tras añadir los archivos principales de app, datasources, nginx.conf y prometheus) el archivo docker-compose.yml con los siguientes datos:

```
version: "3"

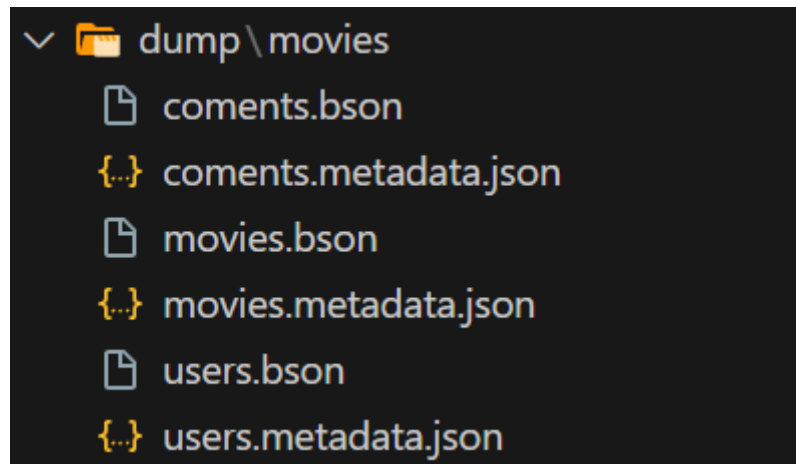
services:
  # app:
  #   container_name: exampleapp
  #   restart: always
  #   build: .
  #   ports:
  #     - "4000:3000"
  #   depends_on:
  #     - mongo
  #   volumes:
  #     - ../usr/src/app

  mongo:
    container_name: mymongodatabase
    image: mongo:latest
    restart: always
```

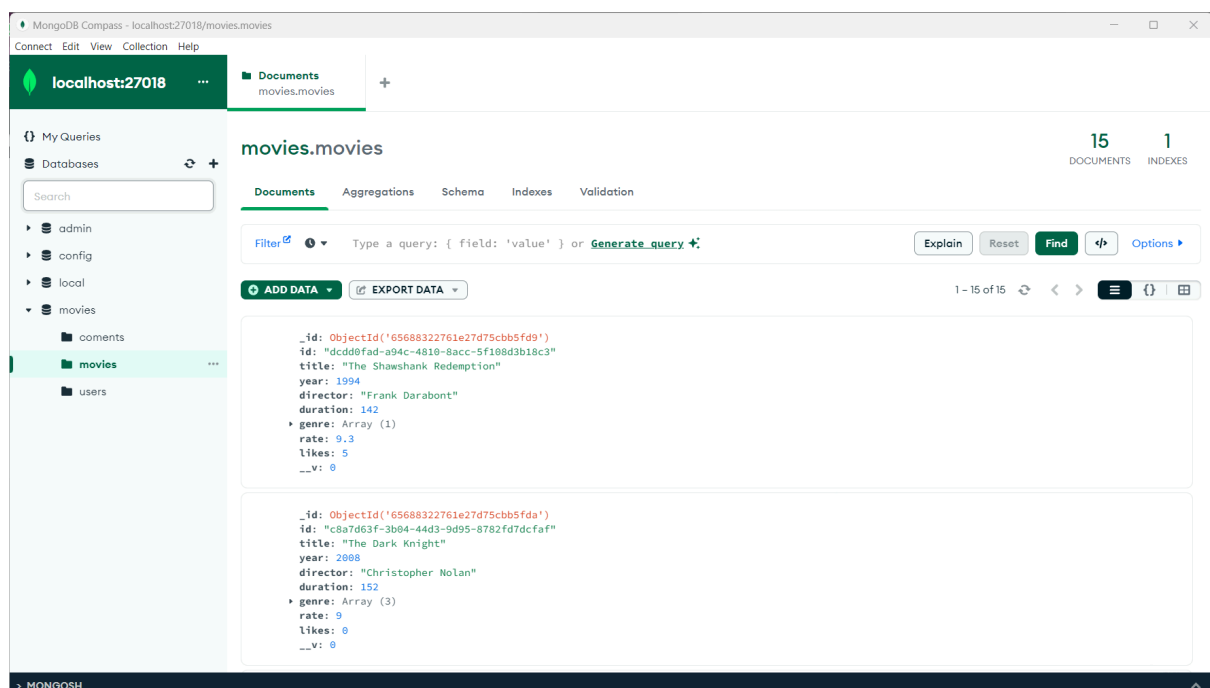
```
# command: mongorestore -d movies ./db-dump
environment:
    MONGO_INITDB_DATABASE: movies
ports:
    - "27018:27017"
volumes:
    - ./mongorestore.sh:/docker-entrypoint-initdb.d/mongorestore.sh
    - ./dump:/dump
```

Tras esto, realizamos el uso del comando “docker-compose build” de forma que se implementará el dump para la conexión de la base de datos:

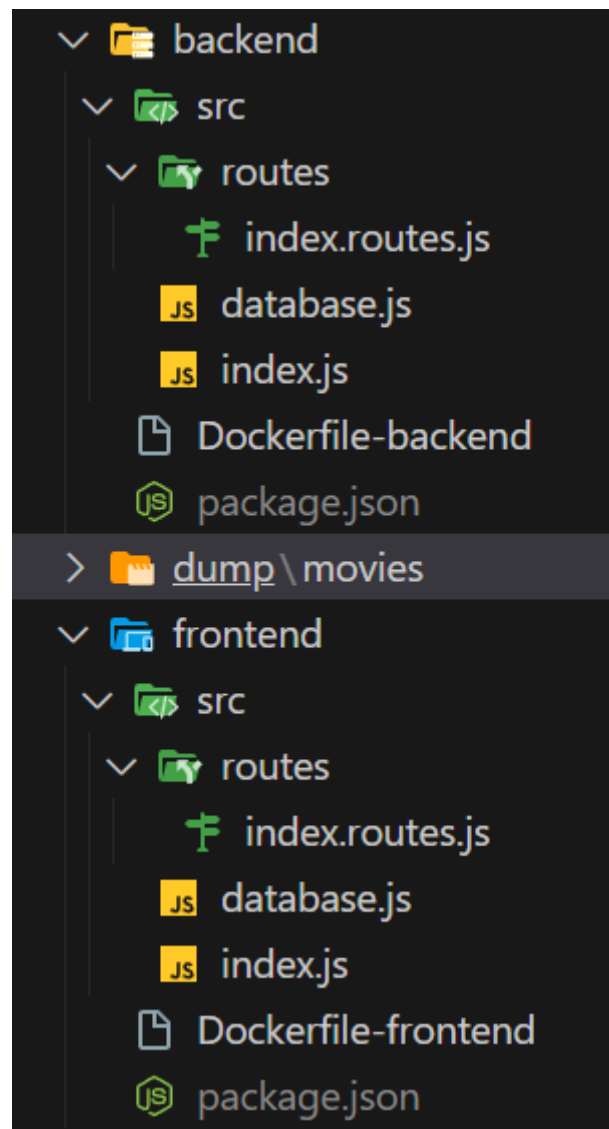
```
PS C:\Users\vano\ Desktop\ Practica Docker Compose 2024 Jordi Va-Enrique> docker-compose build
[+] Building 0.0s (0/0) docker:default
```



Seguidamente abrimos mongoDB para ver la conexión de la misma.



Finalmente tenemos todo comprobado y en funcionamiento, por lo que pasamos a la realización del backend y del frontend creando una carpeta para cada una de las partes donde situaremos los mismos archivos en cada carpeta a excepción del archivo dentro de rutas:



Empezamos mostrando los archivos “index.js”, “database.js” y “index.routes.js” de la carpeta “backend”:


```
backend > src > index.js > ...  
  Click here to ask Blackbox to help you code faster  
1  const express = require('express')  
2  
3  const app = express();  
4  
5  
6  require('./database')  
7  
8  app.use(require('./routes/index.routes'))  
9  
10 app.listen(3000)  
11 console.log("server en el puerto: ", 3000);
```


```
backend > src > JS database.js > ...
  ⚡ Click here to ask Blackbox to help you code faster
1  const mongoose = require('mongoose')
2
3  mongoose.connect('mongodb://mongo/movies', {
4    useNewUrlParser: true,
5    useUnifiedTopology: true
6  })
7    .then(db => console.log('Db is conected to', db.connection.host))
8    .catch(err => console.error(err))
```

```
backend > src > routes > TS index.routes.js > ...
  ⚡ Click here to ask Blackbox to help you code faster
1  const { Router } = require('express');
2  const router = Router()
3
4  router.get('/', (req, res) => {
5    res.send("Hola mundo backend")
6  })
7
8  router.get('/api', (req, res) => {
9    res.send("Hola mundo api")
10 })
11
12
13 module.exports = router;
```



Seguimos mostrando los archivos de la carpeta "frontend":

```
frontend > src > JS index.js > ...
  ⚡ Click here to ask Blackbox to help you code faster
1  const express = require('express')
2
3  const app = express();
4
5
6  require('./database')
7
8  app.use(require('./routes/index.routes'))
9
10 app.listen(4000)
11 console.log("server en el puerto: ", 4000);
```

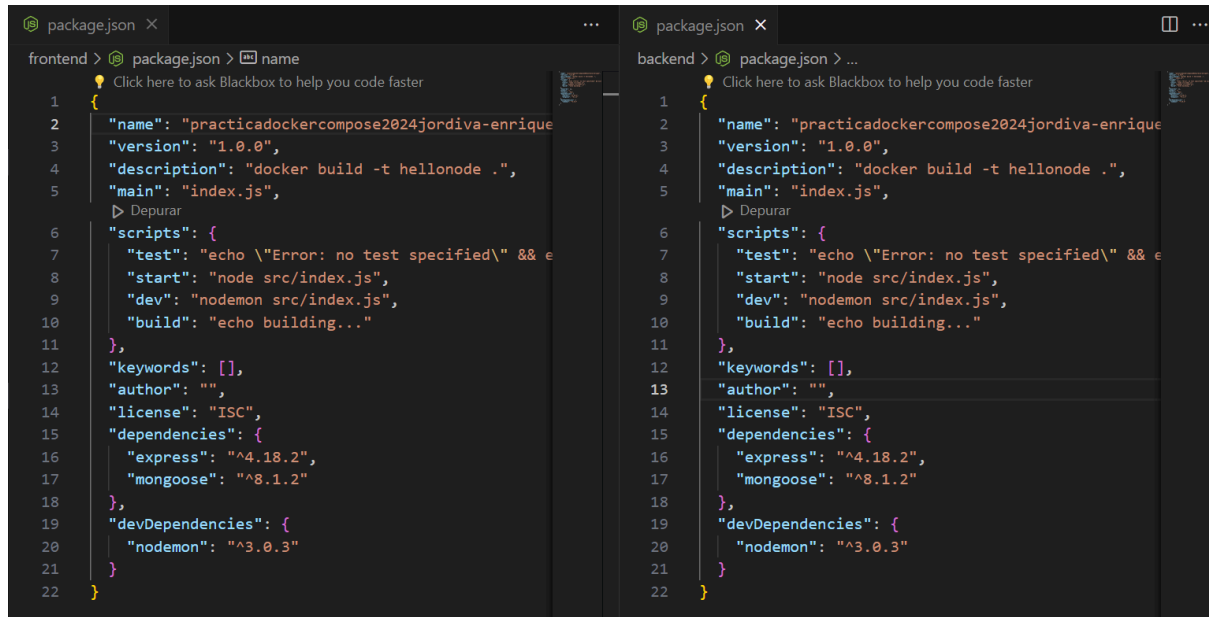
```
frontend > src >  database.js > ...
  ⚡ Click here to ask Blackbox to help you code faster
1  const mongoose = require('mongoose')
2
3  mongoose.connect('mongodb://mongo/movies', {
4    useNewUrlParser: true,
5    useUnifiedTopology: true
6  })
7    .then(db => console.log('Db is conected to', db.connection.host))
8    .catch(err => console.error(err))
```

```
frontend > src > routes >  index.routes.js > ...
  ⚡ Click here to ask Blackbox to help you code faster
1  const { Router } = require('express');
2  const router = Router()
3
4  router.get('/', (req, res) => {
5    res.send("Hola mundo frontend")
6  })
7
8  module.exports = router;
```

Vistos estos archivos, hay que darse cuenta que “index.js” y “database.js” son iguales pero que “index.routes.js” sí es diferente. Acto seguido, revisamos el archivo “Dockerfile-backend” y “Dockerfile-frontend”, que tienen la misma información pero cada uno tiene su nombre:

Dockerfile-frontend	Dockerfile-backend
frontend >  Dockerfile-frontend > ...	backend >  Dockerfile-backend > ...
⚡ Click here to ask Blackbox to help you code faster	⚡ Click here to ask Blackbox to help you code faster
1 # Utiliza una imagen base con Node.js	1 # Utiliza una imagen base con Node.js
2 FROM node:19-alpine AS build	2 FROM node:19-alpine AS build
3	3
4 # Establece el directorio de trabajo en el contenedor	4 # Establece el directorio de trabajo en el contenedor
5 WORKDIR /usr/src/app	5 WORKDIR /usr/src/app
6	6
7 # Copia los archivos de configuración y dependencia	7 # Copia los archivos de configuración y dependencia
8 COPY package*.json ./	8 COPY package*.json ./
9 COPY . .	9 COPY . .
10	10
11 # Instala las dependencias del proyecto	11 # Instala las dependencias del proyecto
12 RUN npm install	12 RUN npm install
13	13
14 # Compila la aplicación para producción	14 # Compila la aplicación para producción
15 RUN npm run build	15 RUN npm run build
16	16
17 FROM node:19-alpine	17 FROM node:19-alpine
18	18
19 WORKDIR /usr/src/app	19 WORKDIR /usr/src/app
20	20
21 COPY --from=build /usr/src/app ./	21 COPY --from=build /usr/src/app ./
22	22
23 COPY package*.json ./	23 COPY package*.json ./
24	24
25 RUN npm install	25 RUN npm install
26	26
27 # Comando para iniciar el servidor web del frontend	27 # Comando para iniciar el servidor web del frontend
28 CMD ["npm", "start"]	28 CMD ["npm", "start"]
29	29

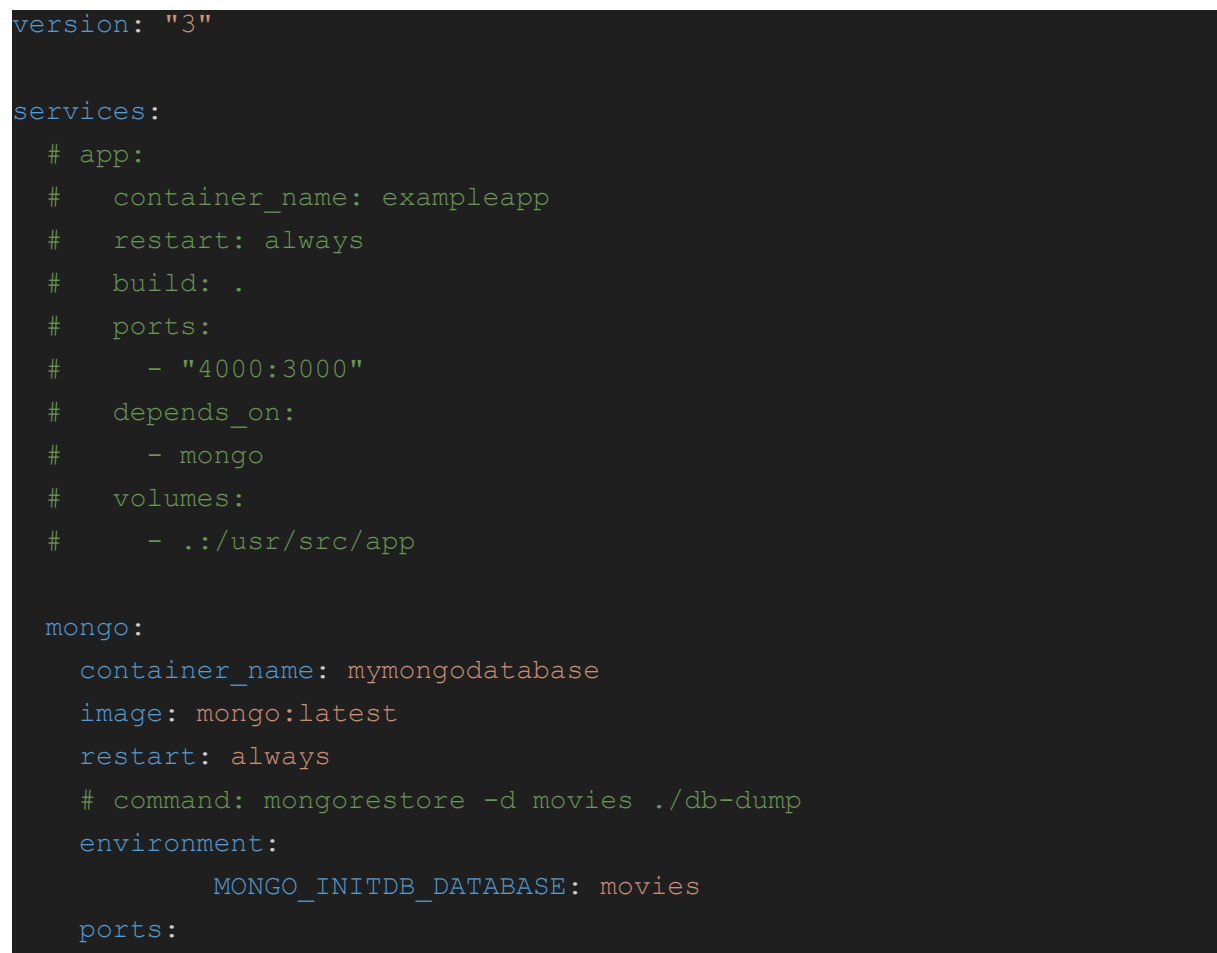
Y finalmente mostramos los archivos “package.json” que son los mismos tanto en una carpeta como en la otra:



The image shows two side-by-side code editors. The left editor is titled 'package.json' and shows the 'frontend' directory. The right editor is also titled 'package.json' and shows the 'backend' directory. Both editors display the same JSON content for their respective package.json files.

```
1 {
2   "name": "practicadockercompose2024jordiva-enrique",
3   "version": "1.0.0",
4   "description": "docker build -t hellonode .",
5   "main": "index.js",
6   "scripts": {
7     "test": "echo \\\"Error: no test specified\\\" && exit 1",
8     "start": "node src/index.js",
9     "dev": "nodemon src/index.js",
10    "build": "echo building..."
11  },
12  "keywords": [],
13  "author": "",
14  "license": "ISC",
15  "dependencies": {
16    "express": "^4.18.2",
17    "mongoose": "^8.1.2"
18  },
19  "devDependencies": {
20    "nodemon": "^3.0.3"
21  }
22 }
```

Una vez terminadas las carpetas de “backend” y “frontend”, completamos el archivo “docker-compose.yml” para añadir ambas partes quedando este archivo de la siguiente forma:



The image shows a code editor with the content of a docker-compose.yml file. The file defines two services: 'app' and 'mongo'.

```
version: "3"

services:
  # app:
  #   container_name: exampleapp
  #   restart: always
  #   build: .
  #   ports:
  #     - "4000:3000"
  #   depends_on:
  #     - mongo
  #   volumes:
  #     - ./usr/src/app

  mongo:
    container_name: mymongodatabase
    image: mongo:latest
    restart: always
    # command: mongorestore -d movies ./db-dump
    environment:
      MONGO_INITDB_DATABASE: movies
    ports:
```

```

- "27018:27017"
volumes:
- ./mongorestore.sh:/docker-entrypoint-initdb.d/mongorestore.sh
- ./dump:/dump

backend:
  container_name: backend_container
  build:
    context: ./backend
    dockerfile: Dockerfile-backend
  depends_on:
    - mongo
  ports:
    - "3000:3000"

frontend:
  container_name: frontend_container
  build:
    context: ./frontend
    dockerfile: Dockerfile-frontend
  depends_on:
    - backend
  ports:
    - "4000:4000"

```

Una vez finalizado todo el proceso, reconstruimos el docker eliminando los contenedores y volviendo a ejecutar el comando “docker-compose build” y reiniciamos el contenedor con “docker-compose up -d”:

```

PS C:\Users\vanoe\Desktop\PracticaDockerCompose2024JordiVa-Enrique> docker-compose build
[+] Building 0.0s (0/0)  docker:default

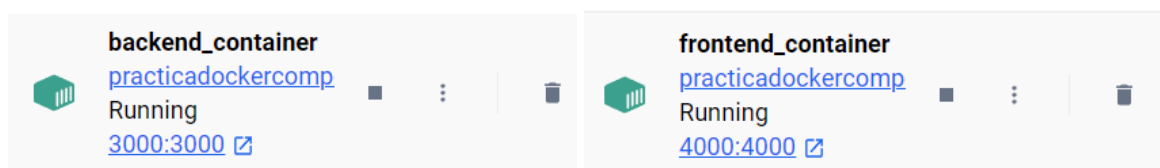
```

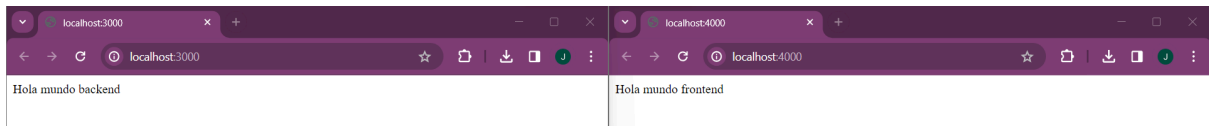
```

PS C:\Users\vanoe\Desktop\PracticaDockerCompose2024JordiVa-Enrique> docker-compose up -d

```

Una vez ejecutado, comprobamos Docker desktop para comprobar que funcionan los contenedores:





De esta forma, comprobamos la funcionalidad tras comprobar que en cada puerto se establece el mensaje de prueba que establecimos en cada fichero de prueba.

Tras la comprobación, pasamos a realizar el apartado del contenedor mongo, para ello, añadimos los siguientes datos al archivo “docker-compose.yml”:

```
version: "3"

services:
  # app:
  #   container_name: exampleapp
  #   restart: always
  #   build: .
  #   ports:
  #     - "4000:3000"
  #   depends_on:
  #     - mongo
  #   volumes:
  #     - ./usr/src/app

  mongo:
    container_name: mymongodatabase
    image: mongo:latest
    restart: always
    # command: mongorestore -d movies ./db-dump
    environment:
      MONGO_INITDB_DATABASE: movies
    ports:
      - "27018:27017"
    volumes:
      - ./mongorestore.sh:/docker-entrypoint-initdb.d/mongorestore.sh
      - ./dump:/dump

  backend:
    container_name: backend_container
    build:
      context: ./backend
      dockerfile: Dockerfile-backend
    depends_on:
```

```

    - mongo
  ports:
    - "3000:3000"

frontend:
  container_name: frontend_container
  build:
    context: ./frontend
    dockerfile: Dockerfile-frontend
  depends_on:
    - backend
  ports:
    - "4000:4000"

mongo-express:
  container_name: adminMongo_container
  image: mongo-express:latest
  environment:
    - ME_CONFIG_MONGODB_SERVER=mongo
    - ME_CONFIG_MONGODB_PORT=27017
  ports:
    - "8081:8081"
  depends_on:
    - mongo

```

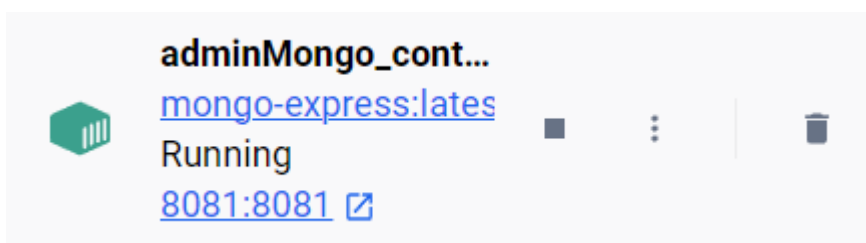
Una vez añadida la información, volvemos a ejecutar “docker-compose up -d”:

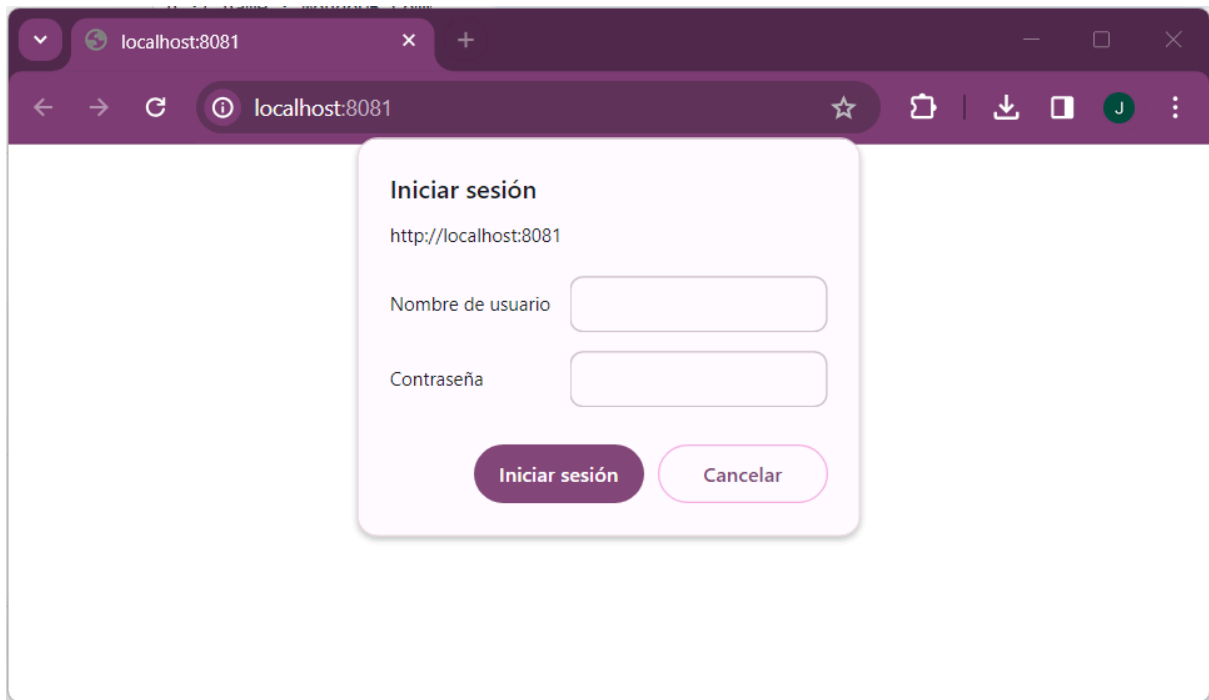
```

PS C:\Users\vanoe\Desktop\PracticaDockerCompose2024JordiVa-Enrique> docker-compose up -d

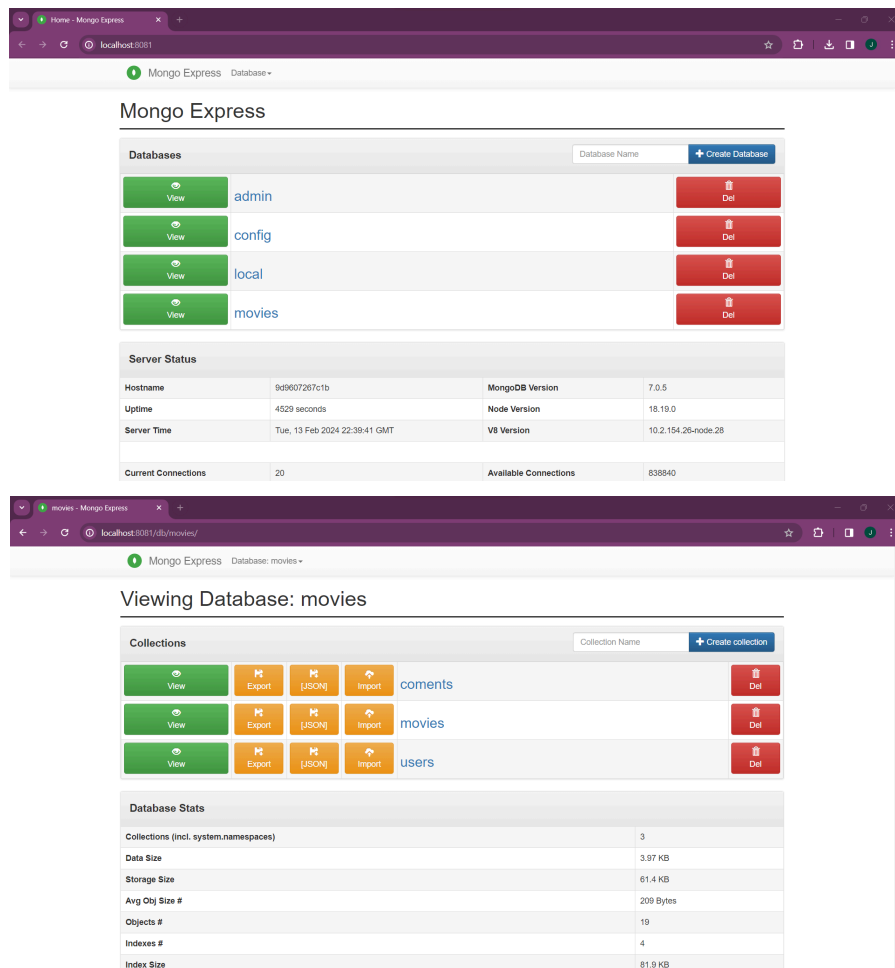
```

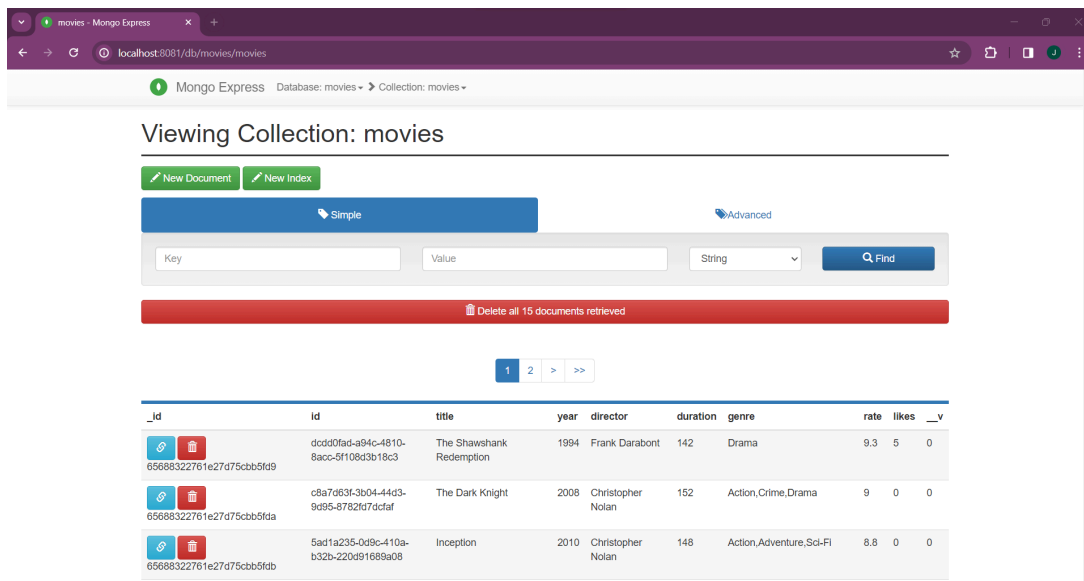
Comprobamos el Docker Desktop para ver que se ha creado el contenedor y realizamos la comprobación en el navegador con la ruta del localhost y el puerto (localhost:8081):





Al acceder a la ruta nos pedirá los credenciales para acceder. Tras seguir la documentación, añadimos “admin” como usuario y “pass” como la contraseña.





Tras la comprobación y verificación del funcionamiento, pasamos al apartado del contenedor “nginx”. Para ello, creamos el archivo “nginx.conf” el cual debe contener tanto la parte backend como la parte frontend. Este archivo debe quedar de la siguiente forma:

```
events {
    worker_connections 1024;
}

http {
    upstream frontend {
        server frontend:4000;
    }
    upstream backend {
        server backend:3000;
    }

    server {
        listen 80;

        location / {
            proxy_pass http://frontend;
            proxy_set_header Host $host;
        }

        location /api {
            proxy_pass http://backend;
            proxy_set_header Host $host;
        }
    }
}
```

Y añadimos la última parte al archivo “docker-compose.yml” que quedará de definitivamente de la siguiente manera:

```
version: "3"

services:
  # app:
  #   container_name: exampleapp
  #   restart: always
  #   build: .
  #   ports:
  #     - "4000:3000"
  #   depends_on:
  #     - mongo
  #   volumes:
  #     - ./usr/src/app

  mongo:
    container_name: mymongodatabase
    image: mongo:latest
    restart: always
    # command: mongorestore -d movies ./db-dump
    environment:
      MONGO_INITDB_DATABASE: movies
    ports:
      - "27018:27017"
    volumes:
      - ./mongorestore.sh:/docker-entrypoint-initdb.d/mongorestore.sh
      - ./dump:/dump

  backend:
    container_name: backend_container
    build:
      context: ./backend
      dockerfile: Dockerfile-backend
    depends_on:
      - mongo
    ports:
      - "3000:3000"

  frontend:
```

```

  container_name: frontend_container
  build:
    context: ./frontend
    dockerfile: Dockerfile-frontend
  depends_on:
    - backend
  ports:
    - "4000:4000"

mongo-express:
  container_name: adminMongo_container
  image: mongo-express:latest
  environment:
    - ME_CONFIG_MONGODB_SERVER=mongo
    - ME_CONFIG_MONGODB_PORT=27017
  ports:
    - "8081:8081"
  depends_on:
    - mongo

nginx:
  container_name: nginx_loadbalancer
  image: nginx:latest
  volumes:
    - ./nginx.conf:/etc/nginx/nginx.conf
  command: ["nginx", "-g", "daemon off;"]
  ports:
    - "80:80"
  depends_on:
    - backend
    - frontend

```


























De tal forma, volvemos a ejecutar el comando “docker-compose up -d”:

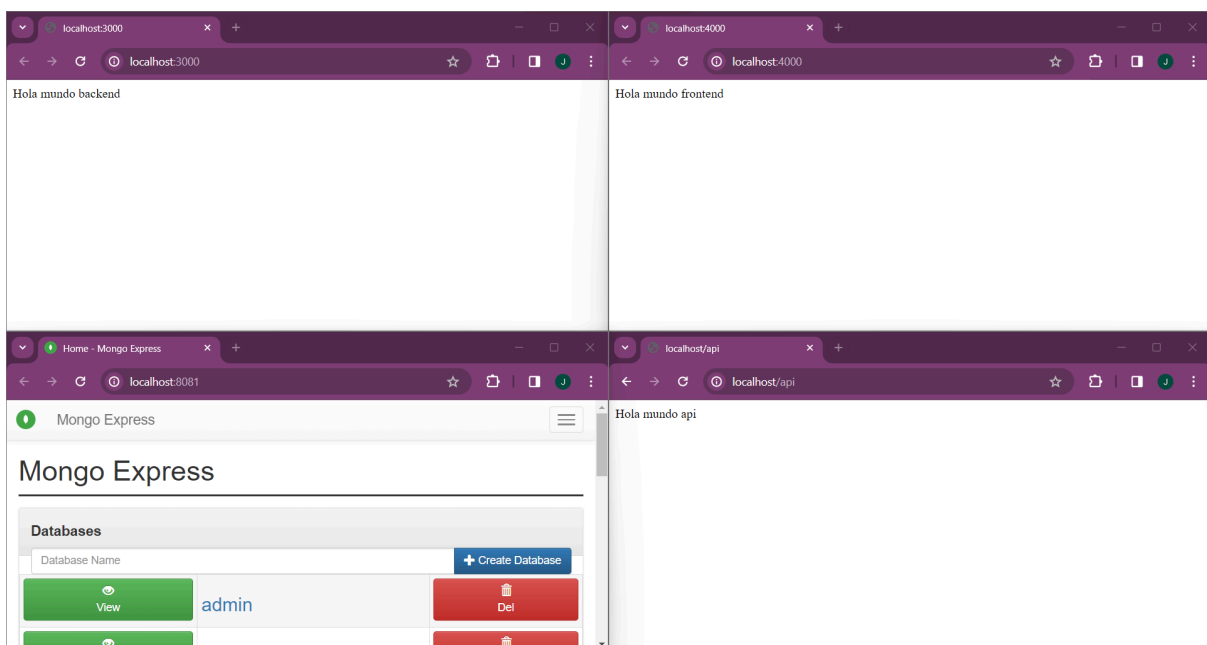
```

PS C:\Users\vano\ Desktop \PracticaDockerCompose2024JordiVa-Enrique> docker-compose up -d

```

Y comprobamos en Docker desktop que los contenedores se hayan creado y funcionen correctamente tras verificarlo en el navegador usando el localhost y los puertos correspondientes:

	mymongodatabase mongo:latest Running 27018:27017 			
	adminMongo_cont... mongo-express:lates Running 8081:8081 			
	backend_container practicadockercomp Running 3000:3000 			
	nginx_loadbalancer nginx:latest Running 80:80 			
	frontend_container practicadockercomp Running 4000:4000 			



[Enlace a repositorio github](#)