UNIVERSIDAD NACIONAL DEL CENTRO DE LA PROVINCIA DE BUENOS AIRES

FACULTAD DE CIENCIAS EXACTAS

Informe Final

Taller de Desarrollo Web

(MODIFICADO PARA SISTEMAS OPERATIVOS, Tobias Romano)

Matias Fino

mfino@alumnos.exa.unicen.edu.ar

Legajo 249849

Tobias Romano

tobiasromano151@gmail.co

m Legajo 249871

Tomás Elordi

telordi@alumnos.exa.unicen.edu.a

r Legajo 249710

Introducción

Este proyecto tiene como objetivo principal proporcionar una experiencia de aprendizaje significativa en el desarrollo de aplicaciones web. Para lograrlo, se enfocó en la creación de una aplicación que ofrece información sobre los grupos de la Copa Libertadores 2023. Para la implementación de esta aplicación, se aprovecharon las herramientas proporcionadas por la cátedra, como Express, MongoDB y Mongoose, para el desarrollo del backend. Además, se optó por el uso del framework Next.js para el desarrollo del frontend, brindando una estructura sólida y eficiente para la aplicación web. Esta combinación de tecnologías permite explorar y comprender los aspectos fundamentales del desarrollo web mientras se crea una aplicación funcional y relevante.

Desarrollo

Para la implementación de dicho trabajo se optó por la utilización de express para el lado del servidor, este framework cuenta con las herramientas necesarias para el desarrollo de una api, la cual es consumida desde el frontend. Dicho frontend fue realizado con el framework Next.js, que es un framework realizado sobre React.

API

Para comenzar con el desarrollo de la api, se utilizó como motor de base de datos MongoDB, que es una base de datos de documentos que ofrece una gran escalabilidad y flexibilidad, y un modelo de consultas e indexación avanzado.Los datos de los distintos grupos fueron obtenidos a través de un api externa gratuita y cargados de manera dinámica a partir de código en node.js con la ejecución del archivo ‘cargarTabla.js’.

A manera de hacer un correcto uso del motor MongoDB se adiciono el uso de Mongoose, así a partir de los documentos de dicho motor y express se puede cargar los datos y manipularlos fácilmente.
Para la utilización de mongoose se implementó un esquema para los grupos, que es la representación de los datos en express.
Una vez representados los datos se realizaron controllers para acceder a dichos datos a través de funciones. Luego se crearon rutas por las cual el frontend se comunica por medio de peticiones HTTP y obtiene los datos requeridos.

Frontend

En el desarrollo de este componente como ya se dijo anteriormente se utilizó el framework Next.js. En este se obtuvieron los datos de los diferentes grupos y se los visualizaron en forma de tabla. Dichos grupos pueden ser obtenidos todos juntos en la página raíz o a través de su página específica (ejemplo: /Group A).
Así es como se ven el página raíz:

Y una vez que se hace click en un grupo específico se puede apreciar todos los equipos y resultados de los mismos.

Adaptación a Sistemas Operativos

Para involucrar al trabajo con la materia Sistemas Operativos se propuso adaptarlo a Docker, de esta manera aprendimos a comunicar tres servicios independientes a través de una red, en este caso local. Para lo mismo se crearon tres contenedores de Docker con sus especificaciones en la dockerfile.

- Back-end: Se conecta con Mongo y a su vez tiene un script JS que se comunica con una API externa y carga los datos en la DB
- MongoDB
- Front-end: Se conecta conecta con el back mediante http://backend/3001.

Dockerfile

```
FROM node:16-alpine
WORKDIR /API
COPY package*.json ./

RUN npm install
COPY . .
RUN node "Model/cargarTabla.js"
EXPOSE 3001

CMD ["node", "Controller/Api.js"]
```

```
FROM node:16-alpine

RUN apk add --no-cache bash
WORKDIR /front-end

COPY package.json package-lock.json ./
RUN npm config set timeout 600000
RUN npm install -g npm@8.5.1
RUN npm i next

COPY wait-for-it.sh /usr/local/bin/wait-for-it.sh
RUN chmod +x /usr/local/bin/wait-for-it.sh

COPY . .

RUN npm run build

EXPOSE 3000

# Comando para correr la aplicación en modo desarrollo
CMD ["npm", "run", "dev"]
# Comando para correr la aplicación en producción
# CMD ["npm", "start"]
```

PROBLEMÁTICA:

Una vez armados los dockerfile para la especificación de los contenedores se realizó el Docker-Compose, encargado de contener las especificaciones del entorno de desarrollo, servicios y volúmenes.

```yaml
services:
  backend:
    build: ./API
    ports:
      - "3001:3001"
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:3000/"]
      interval: 30s
      timeout: 10s
      retries: 3
    environment:
      - MONGO_URL=mongodb://mongo:27017/CopaLibertadores
    depends_on:
      - mongo



  frontend:
    build:
      context: ./front-end
      args:
        - NEXT_PUBLIC_API_URL=http://backend:3001
    ports:
      - "3000:3000"
    depends_on:
      - mongo
      - backend
    environment:
      - NEXT_PUBLIC_API_URL=http://backend:3001
    entrypoint: ["/usr/local/bin/wait-for-it.sh", "backend:3001", "--strict", "--timeout=300", "--", "npm", "run", "dev"]

  mongo:
    image: mongo:latest
    ports:
      - "27017:27017"
    volumes:
      - mongo-data:/data/db

volumes:
  mongo-data:
```

La problemática surge a partir de que para que el servicio que representa el Front-End funcione el Back-End y el servicio de MongoDB deben estar funcionando y la DB debe tener datos. Usualmente para que un servicio dependa de otro se utiliza la cláusula “depends_on”, a partir de esta cláusula tampoco funcionaba con normalidad porque si bien los dos servicios, Backend y Mongo se activaban antes, al momento que el Front se quería activar la DB no tenía datos.

¿A qué se debía esto?

La respuesta se encontraba dentro del Back-End, al momento de ejecutar el script que cargaba la DB, se hacía el request a la API externa que carga la tabla con datos de la copa libertadores, los datos tardaban en cargarse lo suficiente como para que el front-end no pueda realizar el fetch de los datos, entonces generaba un error.

“TypeError: fetch failed

at Object.fetch (/front-end/node_modules/next/dist/compiled/undici/index.js:1:26669)

at processTicksAndRejections (node:internal/process/task_queues:96:5)

- info Generating static pages (5/5)”

¿Cómo se soluciono el error?

La solución surgió de dos maneras distintas:

- Investigando la documentación de docker se llegó a de nuevo a la cláusula depends_on, pero esta cláusula posee más opciones entre ellas “service_started”, “service_healthy”, “service_completed_succesfully”, etc.

- La otra solución, un poco menos convencional pero la que fue utilizada, encontramos un script de bash que se encarga de esperar a que un servicio esté completamente funcionando para ejecutar otro servicio (En este caso se le pasa por parámetros que el servicio del BACK esté funcionando para luego ejecutar el FRONT).

```
entrypoint: ["/usr/local/bin/wait-for-it.sh", "backend:3001", "--strict", "--timeout=300", "--", "npm", "run", "dev"]
```

Adaptación a Podman

Primero y principal, para abordar este otro enfoque debemos comprender que es Podman y que tienen en común con DOCKER. En este momento de investigación de la herramienta es donde aparece el concepto de OCI (Open Container Initiative):

“La Open Container Initiative, también conocida por sus siglas OCI, es un proyecto de la Linux Foundation para diseñar un estándar abierto para virtualización a nivel de sistema operativo.1 El objetivo con estos estándares es asegurar que las plataformas de contenedores no estén vinculadas a ninguna empresa o proyecto concreto.”

Docker y Podman comparten este concepto con lo cual permite la interoperabilidad en la mayoría de los casos.

Adaptación

Para la compatibilidad entre lo ya creado para Docker y Podman, no hubo ningún tipo de problema en cuanto a los archivos ya que son compatibles.

Los problemas surgieron al tener ambos en una máquina, la problemática se debe generar en que están en una misma máquina que tiene Windows. Para esto se investigó y se llegó a la siguiente explicación:

“It seems like we are depending on whether there's a docker_engine named pipe already or not. But I guess the problem
is, when will the Docker engine start, and when will the Podman engine start, there's no guarantee. If Docker starts first,
The named pipe is taken, then it's all good. But if the Podman engine starts first, the name pipe is taken, then
the Docker engine is going to fail.”

Como esto sugiere que no hay una opcion en windows que permita determinar que programa toma el Pipe primero se llegó a la siguiente solución para poder ejecutar uno u el otro:

It turns out `win-sshproxy.exe` is still running and holding a handle to the named pipe, and that's probably why I see an access denied error. After killing the ssh proxy process, now my docker can start normally.

FYI for people who run into the same issue, a temporary solution is to:

1. Shutdown podman machine & desktop
2. Kill all win-sshproxy.exe which still holds a handle to the `docker_engine` named pipe (you can find it from Resource Monitor)
3. Start Docker desktop and it should works normally
4. Start Podman machine & desktop and it should works normally

Diferencias Podman / Docker

Seguridad en Docker

Privilegios de Superusuario

- Daemon Docker (dockerd):
  Requiere root: El daemon Docker debe ejecutarse como root para gestionar contenedores. Esto puede ser un riesgo, ya que comprometer el daemon Docker podría dar acceso completo al sistema.
  Seguridad del daemon: La superficie de ataque es mayor debido a que el daemon escucha por un socket y cualquier vulnerabilidad en Docker puede potencialmente escalar a permisos de superusuario.

Control de Acceso

- Roles y Permisos: Docker permite la gestión de usuarios y permisos, pero la mayoría de las acciones críticas requieren acceso root.
- Docker Content Trust (DCT): Utiliza firmas digitales para asegurar la integridad y la autenticidad de las imágenes de contenedor, pero requiere configuración adicional.

Aislamiento

- Namespaces: Docker usa namespaces del kernel de Linux para aislar procesos, redes y almacenamiento entre contenedores.
- Cgroups: Controla el uso de recursos (CPU, memoria, I/O) entre contenedores.
- Seccomp: Docker usa perfiles de seccomp (Secure Computing Mode) para restringir las llamadas al sistema que los contenedores pueden hacer, reduciendo el riesgo de exploits.
- AppArmor y SELinux: Docker puede usar AppArmor en sistemas Ubuntu y SELinux en sistemas Red Hat para aplicar políticas de seguridad adicionales.

Rootless Mode

- Docker Rootless: Introducido para permitir la ejecución de contenedores sin privilegios de superusuario. Aunque mejora la seguridad, es una característica relativamente nueva y puede tener limitaciones en comparación con Podman.

Adaptación ROOTLESS en Docker

Como parte del proyecto final, cualquier imagen creada en docker debería ser rootless de esta manera se investigó cómo llegar a esto.
Algunos de los videos consultados fueron los siguientes:

https://www.youtube.com/watch?v=0xUwaz0MD_E&ab_channel=PeladoNerd

https://www.youtube.com/watch?v=sXfaogNlc7Y&t=7s&ab_channel=NickJanetakis

Configurar Docker en modo rootless te permite ejecutar contenedores sin necesidad de privilegios de superusuario. Esto mejora la seguridad al reducir la posibilidad de que un contenedor comprometido tenga acceso completo al sistema operativo host. A continuación, demuestro como los dockerfiles ya existentes fueron modificados para volverlos rootless:

```
FROM node:16-alpine

RUN addgroup -S -g 1000 dbgroup && \
    adduser -S -u 1000 -G dbgroup dbuser || \
    echo "Usuario o grupo ya existe"

WORKDIR /API
COPY package*.json ./
RUN npm install

COPY . .
RUN node "Model/cargarTabla.js"
RUN chown -R 1000:1000 /API

EXPOSE 3001
USER 1000
CMD ["node", "Controller/Api.js"]
```

Este caso es el servicio que genera el backend, se le agregaron las lineas remarcadas en amarillo y la linea “ RUN chown -R 1000:1000 /API “

addgroup -S -g 1000 dbgroup:

- -S: Crea un grupo del sistema (system group) que normalmente no tiene una cuenta de usuario asociada y es utilizado para tareas específicas del sistema.
- -g 1000: Especifica el ID del grupo (GID) como 1000.
- dbgroup: Nombre del grupo.

adduser -S -u 1000 -G dbgroup dbuser:

- -S: Crea un usuario del sistema (system user).
- -u 1000: Especifica el ID de usuario (UID) como 1000.
- -G dbgroup: Añade el usuario al grupo dbgroup.
- dbuser: Nombre del usuario.

chown -R 1000:1000 /API:

- chown: Cambia el propietario de los archivos.
- -R: Recursivamente, aplica el cambio a todos los archivos y directorios dentro de /API.
- 1000:1000: Especifica el nuevo propietario y grupo como el usuario con UID 1000 y el grupo con GID 1000.
- /API: El directorio al que se aplicará el cambio de propietario.

USER 1000:

- Indica que cualquier comando subsiguiente (CMD, RUN, ENTRYPOINT, etc.) será ejecutado como el usuario con UID 1000.

Flujo Completo

1. **Creación del usuario y grupo**: Se establece un usuario no root específico para el contenedor.
2. **Cambio de propietario**: Asegura que el nuevo usuario tiene los permisos adecuados en el directorio de trabajo.
3. **Ejecución como usuario no root**: Ejecuta los procesos dentro del contenedor con menos privilegios, mejorando la seguridad.

Y de la misma manera se realizaron los mismos pasos para generar los contenedores del Front-End y la base de datos.

Seguridad en Podman

https://www.youtube.com/watch?v=5WML8gX2F1c&ab_channel=IBMTechnology

Sin Daemon (Daemonless)

- Sin Daemon Persistente: Al no tener un daemon en ejecución, Podman reduce la superficie de ataque. Cada contenedor se ejecuta como un proceso hijo del proceso Podman invocado, lo que mejora la seguridad y el control de procesos.
- Menos Superficie de Ataque: No hay un proceso de fondo escuchando peticiones, lo que reduce el riesgo de ataques remotos.

Rootless Mode

- Ejecución sin Root: Podman fue diseñado desde el principio para soportar la ejecución sin root. Utiliza user namespaces para mapear el ID del usuario dentro del contenedor a un rango diferente de IDs en el host, proporcionando un aislamiento robusto.

Ventajas:

- Mayor Seguridad: Los contenedores rootless no tienen acceso directo al sistema host como root, incluso si son comprometidos.
- Integración Completa: Las características rootless de Podman están mejor integradas y son más maduras en comparación con Docker.

Aislamiento

- Namespaces: Similar a Docker, Podman usa namespaces para aislar contenedores.
- Cgroups: Controla y limita el uso de recursos por parte de los contenedores.
- Seccomp: Podman también utiliza perfiles de seccomp para limitar las llamadas al sistema de los contenedores.
- SELinux: En sistemas Red Hat y derivados, Podman integra políticas de SELinux para aislar contenedores y proteger el sistema host.

Contenedores y Pods

- Seguridad de Pods: Podman puede gestionar pods de manera nativa, lo que permite agrupar contenedores que comparten namespaces y pueden ser gestionados conjuntamente. Esto es similar a Kubernetes y proporciona un modelo de seguridad más robusto para aplicaciones complejas.

Rendimiento y Uso de Recursos

- **Docker**:
  - **Daemon Persistente**: El daemon Docker consume recursos adicionales y puede introducir latencias debido a la comunicación cliente-servidor.
  - **Optimización**: A menudo optimizado para un alto rendimiento en entornos de producción.
- **Podman**:
  - **Sin daemon**: La falta de un daemon persistente reduce la sobrecarga de recursos y potencialmente mejora el rendimiento en ciertos casos.
  - **Eficiencia**: La gestión directa de contenedores puede resultar en una operación más eficiente en términos de recursos del sistema.

Casos de Uso y Escenarios

- **Docker**:
  - **Desarrollo Local**: Muy popular para entornos de desarrollo y CI/CD.
  - **Producción**: Usado en producción junto con orquestadores como Kubernetes y Docker Swarm.
- **Podman**:
  - **Seguridad**: Preferido en entornos donde la seguridad es crítica y se desea minimizar el uso de privilegios de superusuario.
  - **Integración con Kubernetes**: Facilita la transición a Kubernetes con herramientas integradas y generación de configuraciones YAML.

Conclusiones Web

En resumen, el desarrollo de esta API y frontend ha sido un proceso integral que combina diversas tecnologías para ofrecer una experiencia completa y funcional. La elección de MongoDB y Mongoose como base de datos y ORM, respectivamente, ha permitido una gestión eficiente de los datos, mientras que el uso de Next.js ha posibilitado la creación de un frontend dinámico y fácil de usar. Con este proyecto hemos aprendido los conceptos fundamentales del desarrollo de aplicaciones web y cómo las diferentes partes se comunican entre sí.

Conclusiones SO

Para concluir, la adaptación del proyecto a la materia de Sistemas Operativos abordó un enfoque que resultó ser de extremo interés, la adaptación no solo mejoró la portabilidad de nuestra aplicación sino que también aumentó su escalabilidad y facilitó procesos de integración y despliegue continuos. A través de esta experiencia, he ganado habilidades valiosas en el manejo de contenedores y pude introducirme en el concepto básico de OCI que probablemente siga investigando este tipo de tecnologías, que son esenciales en la infraestructura moderna de desarrollo de software y me resultan de gran interés.