INFORME PROYECTO FINAL - INFRAESTRUCTURAS PARALELAS Y DISTRIBUIDAS

Nicolás Mauricio Rojas Mendoza
2259460 - Grupo 51

Víctor Manuel Hernández Ortiz
2259520 - Grupo 50

Esteban Alexander Revelo Salazar
2067507 - Grupo 50

Jhon Alejandro Martinez Murillo
2259565 - Grupo 50

Juan Miguel Posso Alvarado
2259610 - Grupo 50

I. INTRODUCCIÓN

A. Descripción

En este proyecto se busca realizar una separación de componentes, principalmente Backend, Base de Datos y Frontend, lo que se busca es que exista una comunicación efectiva entre estos tres componentes pero que cada uno se mantenga independiente de los otros en cuanto a funcionamiento, lo que se busca integrar dichos componentes bajo arquitecturas locales y en la nube. En este caso especifico proponemos una aplicación llamada **cocoon-home** la cual fue desarrollada usando como backend **Django rest-framework** y como frontend **React**, ademas de integrar un almacenamiento en base de datos. Con estas caracteristicas se busca separar la aplicación en los componentes mencionados y permitir que estos se comuniquen entre ellos.

B. Objetivos

1) **Separación de Componentes:** Dividir la aplicación en Backend, Base de Datos y Frontend para lograr escalabilidad, flexibilidad y mantenibilidad.
2) **Contenerización Local:** Usar Docker Compose para gestionar los componentes de la aplicación de manera modular, garantizando la independencia y escalabilidad de cada uno.
3) **Despliegue en la Nube:** Replicar la arquitectura local en un entorno en la nube (como Azure), incluyendo configuraciones para balanceadores de carga, almacenamiento persistente y servicios gestionados.
4) **Automatización y Escalabilidad:** Configurar scripts y herramientas que faciliten el despliegue y escalado de los componentes tanto localmente como en la nube.
5) **Cumplimiento de Operaciones CRUD:** Implementar una API REST que conecte el Frontend y el Backend para realizar operaciones CRUD sobre la base de datos.

II. SOLUCIÓN LOCAL

A. Descripción detallada de la arquitectura local usando Docker Compose o Kubernetes.

Para la arquitectura local decidimos usar Docker Compose debido a su sencillez y facilidad de implementación. Nuestra solución se compone de cuatro servicios principales, cada uno desempeñando un rol clave dentro del sistema:

1) **Frontend:** Este servicio contiene la interfaz de usuario desarrollada en React. Se comunica con la API para obtener y enviar datos.
2) **API (Backend):** Implementada con Django REST Framework, la API es la parte central de la aplicación. Es responsable de manejar las operaciones CRUD solicitadas por el frontend, así como de procesar las peticiones y respuestas realizando las respectivas modificaciones dentro de la base de datos.
3) **DB (Base de Datos):** Utilizamos PostgreSQL como solución de almacenamiento de datos. Este servicio se configura con persistencia de datos mediante volúmenes de Docker, asegurando que la información se mantenga incluso si el contenedor se reinicia.
4) **Nginx:** En este caso, Nginx cumple el rol de servidor para archivos estáticos, imágenes y videos. Su función es manejar la entrega eficiente de estos recursos al frontend y a los usuarios finales. Al actuar como servidor dedicado para este propósito, garantiza un tiempo de respuesta rápido y descarga la carga de trabajo del backend, optimizando el rendimiento del sistema.

La integración entre estos servicios está orquestada en un archivo **docker-compose.yml**, que define las relaciones, dependencias y configuraciones de red entre los contenedores. Esta separación clara de responsabilidades asegura la independencia y comunicación de los servicios ademas que también permite un despliegue más rápido.

B. Diseño y separación de componentes: backend, base de datos y frontend.

A continuación veremos un diagrama en el que se especifica como es la interacción y separación de estos componentes:

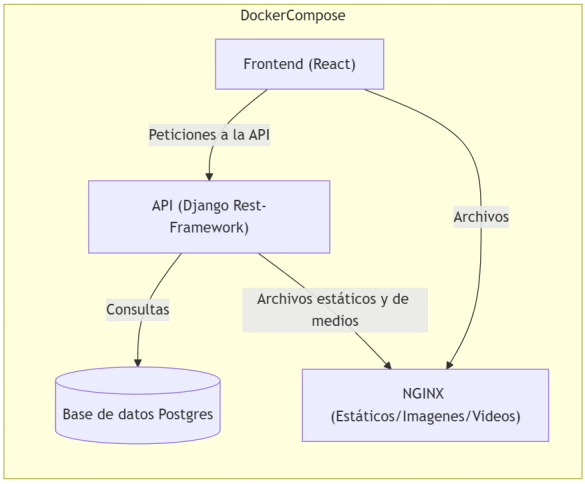


Fig. 1: Separación e interacción de componentes

Aca podemos observar como es el diseño de la arquitectura local implementada usando Docker Compose.

C. Configuración de cada contenedor y cómo estos interactúan.

```
FROM python:3.11-slim

WORKDIR /app

COPY requirements.txt ./

# instalo dependencias
RUN pip install --no-cache-dir -r requirements.txt

# establezco la variable de entorno Deploy para que se
    detecte en settings.py
ENV DEPLOY=true
COPY . .
RUN python manage.py collectstatic --noinput # recolecto
    archivos estaticos

EXPOSE 8000
# comando de ejecucion que incluye migraciones de modelos a
     la base de datos
CMD ["sh", "-c", "python manage.py migrate && python manage
    .py runserver 0.0.0.0:8000"]
```

Dockerfile del backend

```
FROM node:18-alpine
WORKDIR /app/react-app

COPY package.json package-lock.json ./
RUN npm install

COPY ./ ./

RUN npm run build # buildeo la aplicacion

EXPOSE 3000

# ejecuto la preview
CMD [ "npm", "run", "preview" ]
```

Dockerfile del frontend

Los anteriores dos son los contenedores que requieren de una imagen propia y por ello se crea el archivo Dockerfile, en este caso ambos interactuan mediante los puertos expuestos al host, pero esto se especifica mejor dentro del archivo **docker-compose**. A continuación se especifica cada servicio dentro del docker composse

```
frontend:
  build:
    context: ./frontend
  container_name: react-app
  ports:
    - "3000:3000"
  command: "npm run preview"
  depends_on:
    - api
  networks:
    - cocoon_network
```

Configuración del frontend dentro de docker-compose

```
api:
  build:
    context: ./backend
  container_name: django_app
  command: >
    sh -c "python manage.py migrate &&
           python manage.py collectstatic --noinput &&
           python manage.py runserver 0.0.0.0:8000"
  ports:
    - "8000:8000"
  environment:
    DEBUG: '1'
    SECRET_KEY: 'your-secret-key' # llave secreta,
        idealmente cambiarla en produccion
    DJANGO_SETTINGS_MODULE: cocoonbackend.settings
    DATABASE_URL: postgres://django_user:1234ch@db:5432/
        django_db
  depends_on:
    db:
      condition: service_healthy
  volumes:
    - static_volume:/app/staticfiles  # Montar los
        archivos estaticos generados
    - media_volume:/app/media
  networks:
    - cocoon_network
```

Configuración del backend dentro de docker-compose

```
  db:
  image: postgres:15
  container_name: postgres_db
  environment:
    POSTGRES_USER: django_user
    POSTGRES_PASSWORD: 1234ch
    POSTGRES_DB: django_db
  ports:
    - "5433:5432" #  Expongo el puerto 5433 en el host (
        debido a conflictos con postgres local)
  volumes:
    - db_data:/var/lib/postgresql/data
  networks:
    - cocoon_network
  healthcheck: # tiempo de espera para terminar la
      ejecucion correcta de la base de datos
    test: ["CMD-SHELL", "pg_isready -U django_user -d
        django_db"]
    interval: 5s
    retries: 5
```

Configuración de la base de datos dentro del docker compose

```
    nginx: # servicio de nginx para redireccionar las
        peticiones de archivos
    image: nginx:latest
    container_name: nginx_server
    ports:
      - "80:80"
    volumes:
      - static_volume:/app/staticfiles:ro
      - media_volume:/app/media:ro
      - ./nginx/nginx.conf:/etc/nginx/conf.d/default.conf:
          ro
    depends_on:
      - api
    networks:
      - cocoon_network
```

Configuración de Nginx dentro del docker-compose

```
    volumes:
  db_data:  # almacena los datos persistentes de la base de
       datos
  static_volume:  # almacena archivos estaticos
      recolectados
  media_volume: # almacena archivos de medios cargados

networks:
  cocoon_network:
    driver: bridge
```

Configuración de volumenes y red dentro de docker-compose

La comunicación/interacción entre los contenedores se da de la siguiente forma:

- **Backend (api) y Base de Datos (db)**: El backend se conecta a la base de datos mediante la URL proporcionada en la variable de entorno DATABASE_URL *(postgres://django_user:1234ch@db:5432/django_db)*, donde db es el nombre del servicio en docker-compose. La resolución de nombres de los servicios es posible gracias a que están en la misma red **(cocoon_network)**. Ademas La base de datos expone el puerto 5432 internamente, y está mapeado al puerto 5433 en el host para evitar conflictos locales. El backend no necesita usar este puerto externo ya que usa la red interna.
- **Backend (api) y Frontend (frontend)**: El frontend interactúa con el backend a través de la API, en el endpoint /api puerto 8000. Esto ocurre mediante las configuraciones de depends_on y la red compartida. La configuración del frontend puede incluir llamadas al host api directamente, ya que comparten la misma red.
- **Backend (api) y Nginx (nginx)**:: Nginx utiliza volúmenes compartidos (static_volume y media_volume) para servir archivos estáticos y multimedia generados por el backend. Las solicitudes HTTP son redirigidas a través de Nginx al backend (puerto 8000) configurado en el archivo nginx.conf.

También tenemos otras interacciones importantes como en la cuestión de volúmenes que permiten la persistencia de datos, estos volúmenes aseguran que los datos importantes, sean archivos, multimedia o datos de la base de datos persistan incluso cuando se reinicien los contenedores. Y por ultimo tenemos la red compartida **(cocoon_network)**. Todos los servicios están conectados a la misma red tipo bridge. Esto permite que los servicios se comuniquen utilizando sus nombres definidos en docker-compose (por ejemplo, db para la base de datos o api para el backend).

III. Solución en la Nube

A. Replicación de la arquitectura local en la Nube

En este caso decidimos optar por replicar la arquitectura en **Google Kubernetes Engine**, para esto creamos los deployments y servicios necesarios como manifiestos de kubernetes.

Tenemos la configuración de pvc (Persistent Volume Claim) para la creación de los volumenes:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: static-volume
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: standard
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: db-data2
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: standard
```

Configuración de volumenes

También tendremos un contenedor que se encargara de nuestra base de datos, a continuación podemos ver el deployment y service para la base de datos postgres.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: db
spec:
  replicas: 1
  selector:
    matchLabels:
      app: db
  template:
    metadata:
      labels:
        app: db
    spec:
      containers:
      - name: postgres-db
        image: postgres:15
        env:
        - name: POSTGRES_USER
          value: "django_user"
        - name: POSTGRES_PASSWORD
          value: "1234ch"
        - name: POSTGRES_DB
          value: "django_db"
        ports:
        - containerPort: 5432
        volumeMounts:
        - name: db-data2
          mountPath: /data/postgres
        livenessProbe:
          exec:
            command:
            - pg_isready
            - -U
            - django_user
```

```
            - -d
            - django_db
          initialDelaySeconds: 5
          periodSeconds: 5
      volumes:
      - name: db-data2
        persistentVolumeClaim:
          claimName: db-data2
---
apiVersion: v1
kind: Service
metadata:
  name: db
spec:
  selector:
    app: db
  ports:
  - protocol: TCP
    port: 5432
    targetPort: 5432
  type: ClusterIP
```

Configuración de la base de datos

También tenemos la configuración de la api, en este caso a diferencia de la arquitectura local, no usaremos el volumen de medios y por ende tampoco necesitaremos servicio de Nginx para servir estos archivos, debido a que en este caso implementamos un almacenamiento de archivos en la nube gracias a un Bucket en google cloud, esta configuración se hizo de manera interna dentro de la api en django.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api
spec:
  replicas: 3
  selector:
    matchLabels:
      app: api
  template:
    metadata:
      labels:
        app: api
    spec:
      containers:
      - name: django-app
        image: vicmah/django_api:latest
        command: ["/bin/sh", "-c", "python manage.py
             migrate && python manage.py collectstatic --
             noinput && python manage.py runserver
             0.0.0.0:8000"]
        env:
        - name: FRONTEND_URL_MANIFEST
          value: "http://35.184.196.156"
        - name: DEBUG
          value: "1"
        - name: SECRET_KEY
          value: "your-secret-key"
        - name: DJANGO_SETTINGS_MODULE
          value: "cocoonbackend.settings"
        - name: DATABASE_URL
          value: "postgres://django_user:1234ch@db:5432/
               django_db"
        volumeMounts:
        - name: static-volume
          mountPath: /app/staticfiles
        # - name: media-volume
        #   mountPath: /app/media
        ports:
        - containerPort: 8000
      volumes:
      - name: static-volume
        persistentVolumeClaim:
          claimName: static-volume
      # - name: media-volume
      #   persistentVolumeClaim:
      #     claimName: media-volume
---
apiVersion: v1
kind: Service
metadata:
  name: api
spec:
  selector:
    app: api
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8000
  type: LoadBalancer
```

Configuración del backend

Y por ultimo tenemos la configuración de nuestro frontend, en este caso se ha creado un script de ejecución con la finalidad de poder pasar de forma dinámica la url de la api a la build de la aplicación react

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - name: react-app
        image: vicmah/frontend_react:latest
        # command: ["npm", "run", "preview"]
        env:
        - name: VITE_API_URL
          value: "http://34.31.232.213"
        ports:
        - containerPort: 3000
---
apiVersion: v1
kind: Service
metadata:
  name: frontend
spec:
  selector:
    app: frontend
  ports:
  - protocol: TCP
    port: 80
    targetPort: 3000
  type: LoadBalancer
```

Configuración del frontend

B. Descripción del flujo para subir la aplicación a la nube

Como en nuestro caso usamos **Google Kubernetes Engine** primero debemos iniciar un proyecto en Google Cloud para poder usar el Engine de Kubernetes. Una vez con nuestro proyecto creado, los pasos son los siguientes:

1) Debemos crear un cluster, en este caso no requeriremos de configuraciones avanzadas, solo con asignar un nombre es suficiente.
2) Posteriormente podemos hacer uso del entorno de desarrollo y la terminal que nos brinda Google Cloud en su web o podemos enlazar nuestra terminal local de Google Cloud a el proyecto para poder introducir los comandos necesarios.

3) Despues de tener ya una terminal funcional, debemos aplicar los manifiestos anteriores para comenzar con la creación de los deployments y servicios los comandos serian los siguientes:

```
kubectl apply -f pvc.yaml
kubectl apply -f db.yaml
kubectl apply -f api.yaml
kubectl apply -f frontend.yaml
```

Es importante tener en cuenta los nombres correctos de los archivos ya que pueden variar. En este caso usamos los que se encuentran en nuestro repositorio del proyecto https://github.com/vicmaHo/proyecto-infra-local

4) Posteriormente podemos comprobar el estado de nuestros pods

```
kubectl get pods
```

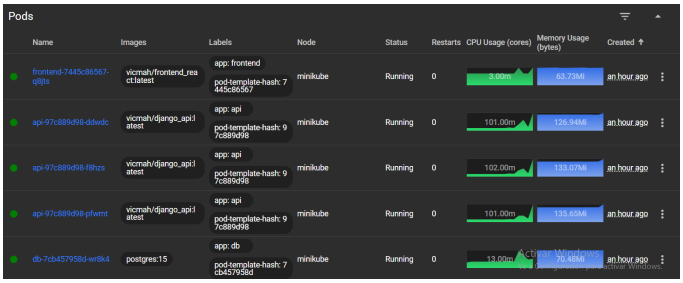
IV. Análisis y Conclusiones

A. Analizar el rendimiento de la aplicación en ambiente local vs en la nube.

Al hacer una prueba de carga, enviando peticiones simulando usuarios simulataneos obtuvimos lo siguiente:

```
--- Resultados del test de carga ---
URL probada: http://127.0.0.1:57258/
Número total de peticiones: 10000
Usuarios concurrentes: 100
Tiempo total: 38.06 segundos
Peticiones exitosas: 10000
Peticiones fallidas: 0
Tiempo promedio de respuesta: 0.35 segundos
```

En este caso se enviaron la cantidad de peticiones especificada a el proyecto corriendo localmente



Podemos observar como localmente subió el uso de CPU de la api, distribuyéndose entre los pods, lo que evidencia que se realizaron las peticiones, ademas en general el tiempo promedio para responder de las peticiones totales fue de 0.35 segundos.

Haciendo lo mismo pero con la aplicación desplegada en google cloud

```
--- Resultados del test de carga ---
URL probada: http://34.31.232.213/
Número total de peticiones: 10000
Usuarios concurrentes: 100
Tiempo total: 28.45 segundos
Peticiones exitosas: 10000
Peticiones fallidas: 0
Tiempo promedio de respuesta: 0.28 segundos
```

Podemos observar un tiempo total de ejecución menor, lo que nos lleva a tener un tiempo promedio de respuesta de 0.28 segundos.

En un inicio podríamos pensar que el entorno local sería más rápido, pero esta prueba demuestra que el rendimiento del entorno en la nube es un poco superior, esto puede deberse a distintos factores como la velocidad de internet, los recursos locales o de nube dispuestos a atender la arquitectura entre otros.

Al ejecutar arquitecturas como esta en la nube, tenemos que tener en cuenta los recursos, ya que en su mayoria los costos operativos dependen de los recursos que designemos, es decir que si necesitaramos más cpu o memoria tendriamos que pagar más. También debemos de tener en cuenta el escalado automatico con el que Kubernetes agrega más nodos en función de la carga. Aunque esto es útil, sin límites configurados, puede generar costos inesperados. Adicionalmente como costos esperados, debemos considerar que tambíen existen los almacenamientos persistentes y las redes o balanceadores de carga, todo esto debe especificarse para saber el costo ”real”.

Aun así no podemos hablar de costo real sin tomar en cuenta variables que afectan y generan costos ocultos. Por ejemplo el Overprovisioning que ocurre cuando los recursos asignados son mayores a lo necesario, puede ser causado por una configuración excesiva de pods con requests y limits de CPU/memoria o por mantenimiento de nodos inactivos para ”picos de tráfico” que no suceden. El impacto de esto radica en que se paga por capacidad no utilizada, especialmente con nodos grandes y caros. Como la anterior, existen otras circunstancias como el almacenamiento de logs, el trafico entre zonas distintas, un sobrecrecimiento en el trafico etc, las cuales pueden afectar los costos finales.

Como optimización local de recursos no se implementa algo especifico, a diferencia de en la nube, en la cual dejamos de lado el volumen de datos de medios para que al final los archivos se almacenen en un bucket optimizando el espacio requerido en el cluster y retirando carga de trabajo adicional del mismo cluster.

Para finalizar con el análisis, en cuanto a seguridad, se implementa en ambos entornos, seguridad básica de autenticación mediante token por parte de la api, seguridad básica de protección de datos mediante hash en la base de datos. Ademas con fines de prueba se expone el backend pero esto en realidad no es lo correcto, a diferencia del contenedor de la base de datos, ya que a este no se tiene acceso de forma externa.

B. Interpretación de posibles retos y cómo fueron abordados

Los siguientes son los principales retos a los que nos enfrentamos durante este proyecto:

- El almacenamiento de archivos local fue una de las dificultades debido a su implementación, inicialmente pensamos en contar con un almacenamiento híbrido entre nube (bucket) y local para que cuando desplegáramos en nube pudiéramos usar el almacenamiento en nube y localmente almacenar los archivos de forma local,

pero esto suposo grandes dolores de cabeza en cuanto a configuración y fue una de las cosas que más consumió tiempo, al final decidimos dejar solo el almacenamiento de archivos local en el despliegue local y el de nube cuando usáramos la nube.

- La configuración de los manifiestos de kubernetes, sobretodo los servicios para que los contenedores pudieran comunicarse entre si nos dio mucho problema, ya que en ocasiones esta comunicación funcionaba de manera interna en la red de kubernetes pero no a la hora de exponerla, lo que nos llevaba a tener problemas desplegando el entorno de kubernetes localmente.

C. Conclusiones finales: Reflexión sobre el uso de tecnologías como Docker y Kubernetes.

Es muy interesante cómo gracias a Docker podemos desplegar aplicaciones en poco tiempo y sin necesidad de ahondar en configuraciones extra como descarga de dependencias, librerías u otros recursos, lo que brinda una facilidad increíble, además ahorrando muchos problemas de compatibilidad entre sistemas. En general, pensamos que a través de los contenedores la vida de los desarrolladores de todo tipo se hace más sencilla.

Por otro lado, Kubernetes complementa esta facilidad al permitir la orquestación de múltiples contenedores, logrando no solo simplificar su despliegue, sino también facilitar la gestión de la escalabilidad, el balanceo de carga y la recuperación ante fallos. Esto representa un avance un gran avance en el desarrollo, ya que antes todos estos aspectos requerían configuraciones manuales extensas, complejas y que acarreaban muchos errores.

Además, la combinación de Docker y Kubernetes no solo permite mejorar los flujos de trabajo en los equipos de desarrollo, sino también en los de operaciones, gracias a la automatización que proporciona. Estas herramientas garantizan entornos más estandarizados y reproducibles, eliminando las preocupaciones típicas de ”funciona en mi máquina pero no en la tuya”. Esto nos lleva a pensar y reflexionar sobre cómo estas tecnologías han cambiado las dinámicas entre los equipos de trabajo, permitiendo una mayor colaboración y reduciendo las barreras entre desarrollo y operaciones.

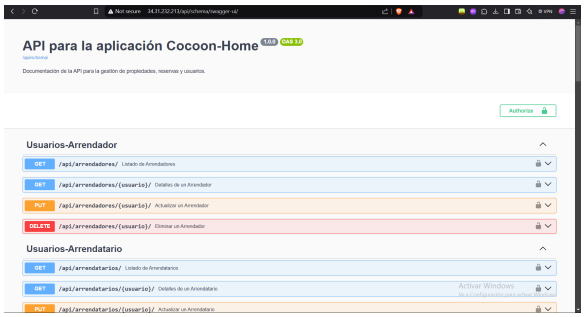
Finalmente, vemos que el uso conjunto de estas herramientas, aunque requiere una curva inicial de aprendizaje, es totalmente justificable debido a sus amplios beneficios obtenidos a largo plazo, tanto en la estabilidad de las aplicaciones como en la eficiencia de los equipos. Esto nos invita a continuar explorando y adoptando estas tecnologías en proyectos futuros, confiando en que nos permitirán construir soluciones más robustas y escalables.

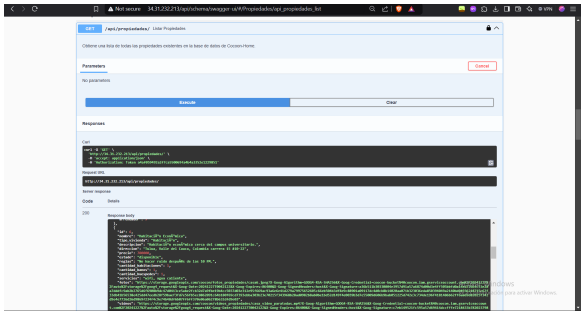
V. Enlaces importantes

- Repositorio del proyecto
- Enlace de la api desplegada con GKE
- Enlace de la documentación de la api desplegada con GKE
- Enlace de el FrontEnd desplgegado con GKE

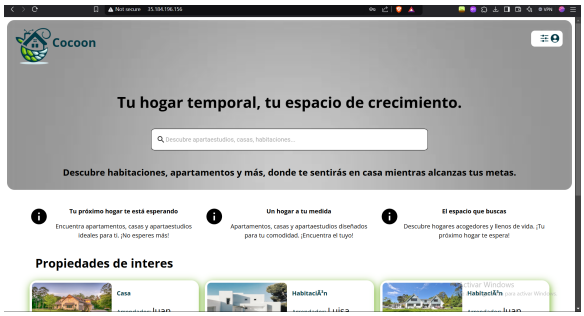
VI. Capturas de funcionamiento de la aplicación en la nube

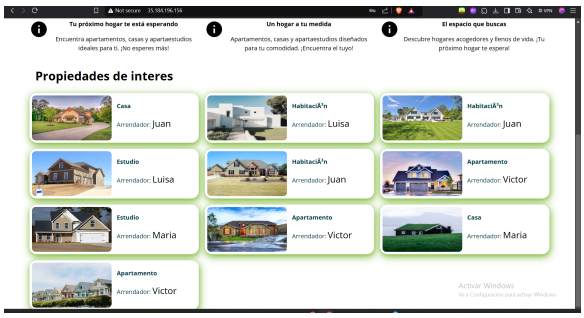
Funcionamiento de la API, en este caso se presenta la documentación de la misma con OpenAPI:

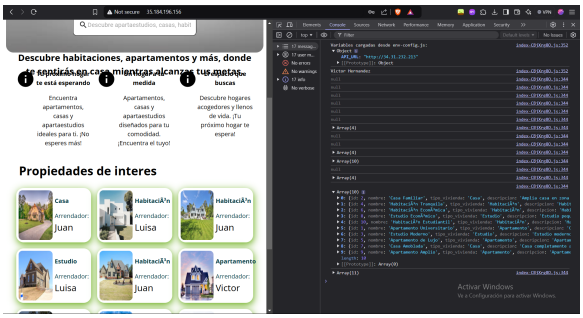




Funcionamiento del frontend, vemos la pagina principal, algunas propiedades y la consola con algunos comandos de depuración en los que podemos ver las peticiones realizadas:






Not secure 35.594.196.156
Descubre apartaestudios, casas, habit
Descubre habitaciones, apartamentos y más, donde te encticás en casa mientras alcanzas tus metas.
Encuentra el lugar que te está esperando
Encuentra apartamentos, casas y apartaestudios ideales para ti. ¡No esperes más!
Propiedades a la medida
Apartamentos, casas y apartaestudios diseñados para tu comodidad. ¡Encuentra el tuyo!
Encuentra lo que buscas
Descubre hogares acogedores y llenos de vida. ¡Tu próximo hogar te espera!
Propiedades de interes
Casa
Arrendador: Juan
HabitaciÃ³n
Arrendador: Luisa
HabitaciÃ³n
Arrendador: Juan
Estudio
Arrendador: Luisa
HabitaciÃ³n
Arrendador: Juan
Apartamento
Arrendador: Victor
Elements
Console
Sources
Network
Performance
Memory
Application
Security
Activar Windows