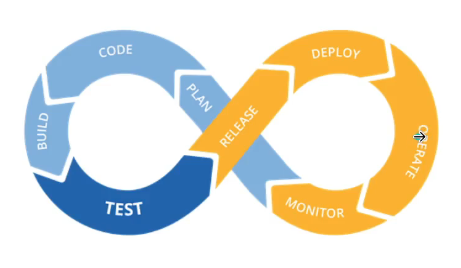
**Section 1, Before You begin.**

DevOps: Es la unión de personas, procesos y tecnología, con el fin de proporcionar valor continuamente a los clientes.



DevOps se encaga de la comunicación entre desarrollo y operaciones.

**FAUNDAMENTALAES:** Control de versiones.

Integracion continua: Pipelines Jenkins

Entrega continua.

Infrestructura como código.

Supervicion y registro.

Aprendizaje valido.

**Section 2:**

**Contenedores:** Empaqueta el codigo y todas las dependencias necesarias de una aplicación. (En

Ejecución).

**Image:** Contiene todo lo necesario para la aplicación (Archivo).

**Docker engine:** Motor de ejecucion de contenedores.

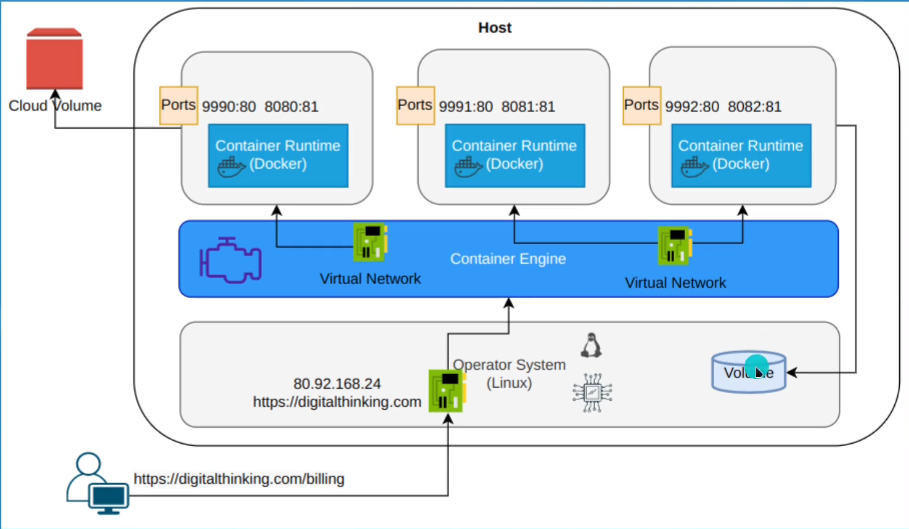
**Docker hub:**  Especie de github pero para imágenes y contenedores.

**Podman:** Open source Docker.

**Docker compose:** Orquestador ligero, para soluciones pequeñas.

**Docker swarm:** Orquestador robusto, administrar contenedores. Maneja maquinas en un cluster

**Kubernetes:** Solucion para administrar un cluster.

****

**Herramientas:**

**Linux:**

* Docker Engine.
* Docker ce cli.
* Docker Desktop.
* Docker Compose.

**Windows:**

* Docker desktop.

**Section 3:**

* Instalamos Docker desktop.
* Todo lo que requiera interfaz gráfica noes buena práctica meterla a contenedores.
* Buscamos mejorar la eficiencia para no instalar lo mismo en todos los servidores, por ejemplo.

**Nos compartiremos el contenedor parecido a git pero sin código.**

**Instalar paso a paso la app que usaremos.**

* docker image ls
* docker ps
* docker run -p 8080:80 -p 7080:7080 --name conBilling sotobotero/billingapp
  + **docker run**

Es el comando para crear y ejecutar un nuevo contenedor a partir de una imagen.

* + **-p 8080:80**

Significa que el puerto 8080 de tu computadora (host) se conecta al puerto 80 del contenedor.

Así, si abres localhost:8080 en tu navegador, estarás accediendo al servicio que corre en el puerto 80 dentro del contenedor.

* + **-p 7080:7080**

Similar al anterior, pero conecta el puerto 7080 de tu computadora al puerto 7080 del contenedor.

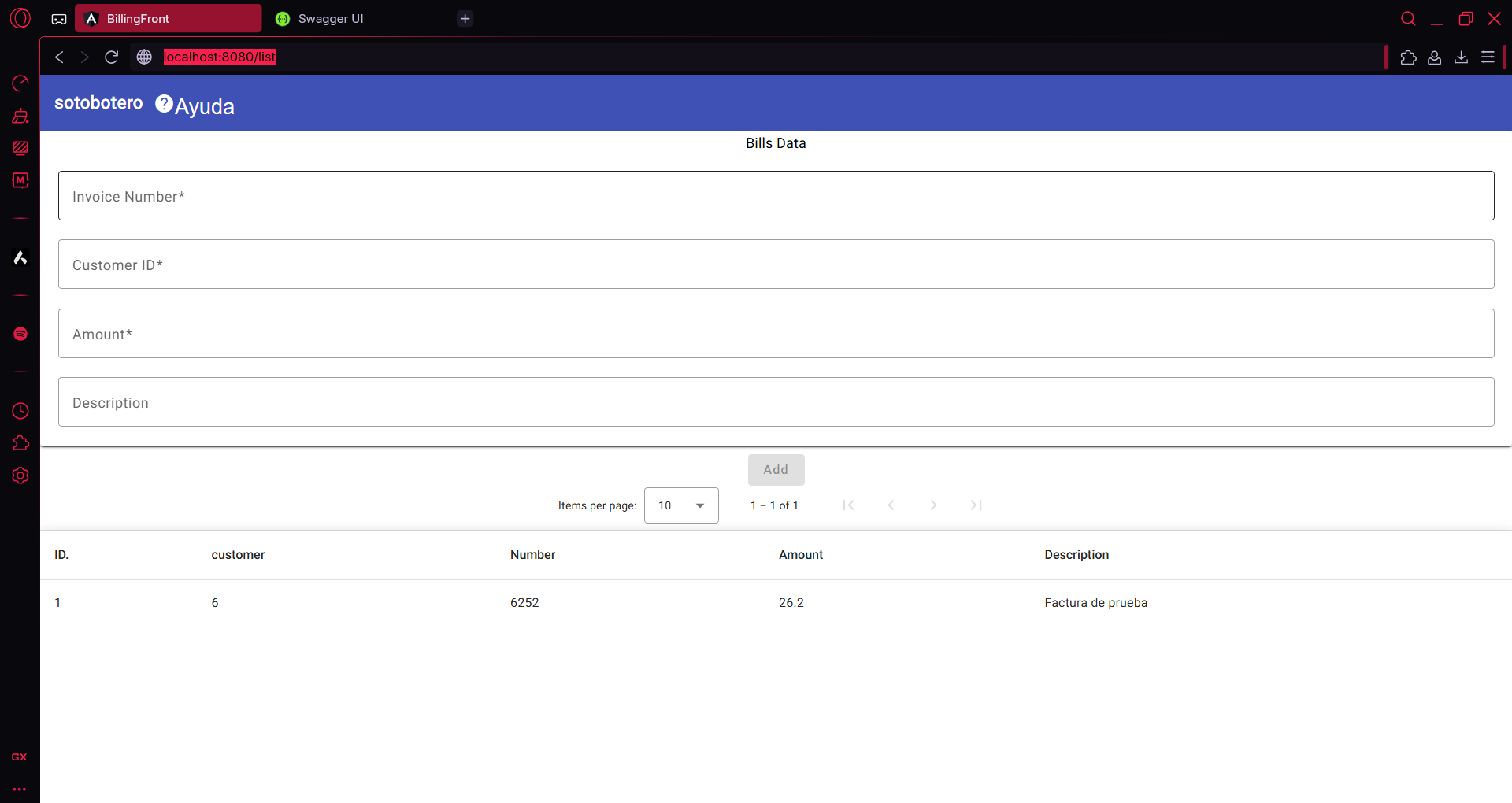
* + **--name conBilling**

Le da el nombre conBilling al contenedor, para que puedas identificarlo fácilmente y usar ese nombre en otros comandos (por ejemplo, para detenerlo o eliminarlo).

* + **sotobotero/billingapp**

Es la imagen de Docker que se va a utilizar para crear el contenedor. En este caso, corresponde a una aplicación llamada "billingapp" publicada por el usuario "sotobotero" en Docker Hub.

**Si ya ingresamos a nuestro local host y los puesrtos que asignamos ahí funcionara perfectamente.**

****

**Podemos probar en consola la conexión con:** telnet localhost “puertos”

**EJEMPLO:**

* telnet localhost 8080

**Como eliminarlo y crearlo**

* Se detiene con (ctrl + C)
* docker ps -a (ver los que tengo activos)
* Docker system prune –all **(Mucho Cuidado elimina todo lo inactivo)**
* Se cre usnado el mismo comando desde el inicio.
* Si no lo eliminas y solo lo detuviste usa:
  + Docker start “name”

**Practica:**

* Buscamos una imagen
  + Ejemplo: de postgres.
* Una vez dentro del repositorio de Docker hub es importante leer la documentación para saberlo usar.
* docker pull postgres:13.2
  + docker run --ulimit memlock=-1:-1 -d --name postgres -e POSTGRES\_USER=sa -e POSTGRES\_PASSWORD=admin -e POSTGRES\_DB=product\_db -p 5432:5432 postgres:13.12
  + **docker run**

Inicia un nuevo contenedor a partir de una imagen.

* + **--ulimit memlock=-1:-1**

Ajusta los límites de memoria del contenedor.

Aquí le dices a Docker que no limite la memoria que puede ser bloqueada por el proceso dentro del contenedor (útil para algunas configuraciones de bases de datos como Postgres).

* + **-d**

Significa "detached mode", es decir, el contenedor se ejecuta en segundo plano (no te bloquea la terminal).

* + **--name postgres**

Le das el nombre postgres al contenedor, lo que facilita identificarlo y gestionarlo después.

* + **-e POSTGRES\_USER=sa**

Crea una variable de entorno dentro del contenedor para definir el usuario principal de la base de datos con el nombre sa.

* + **-e POSTGRES\_PASSWORD=admin**

Define la contraseña del usuario principal de la base de datos (admin).

* + **-e POSTGRES\_DB=product\_db**

Indica que se debe crear una base de datos llamada product\_db al iniciar el contenedor.

* + -**p 5432:5432**

Abre el puerto 5432 de tu computadora y lo conecta al puerto 5432 del contenedor, que es el puerto por defecto de PostgreSQL.

Así podrás conectarte a la base de datos desde tu computadora usando ese puerto.

* + **postgres:13.12**

Es la imagen de Docker que se usará para crear el contenedor, en este caso, PostgreSQL versión 13.12.

**Si quieres que solo te muestre tus imágenes ejecuta:**docker image ls | Select-String -NotMatch 'docker/desktop|registry.k8s.io|coredns|pause'

**Para los contenedores son:**docker ps -a | Select-String -NotMatch 'k8s|docker-desktop|vpnkit|storage-provisioner'

Con docker log “name” te muestra todo lo que hace internamete la app.

**Pasos previos que realizamos:**

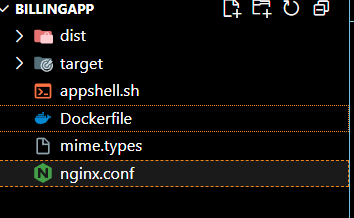
Detener los contenedores:  
 docker stop $(docker ps -q)

Eliminar los que están detenidos

docker system prun --all

**Ahora vamos a crear imágenes.**

En la carpeta del archivo creamos uno que se llame **Dockerfile**



#Buscamos una imagen que se ajuste a nuestras necesidades (el servidor, el lenguaje, etc)

#Utlilizaremos servidor ngix y juntaremos el back node y el front angular

* **FROM nginx:stable-alpine**

#Usamos el volumen mas eficiente para nuestro proyecto

#es una carpeta temporal que se utiliza para almacenar archivos temporales

* **VOLUME /tmp**

#Instalamos las dependencias necesarias

#Eliminamos todo lo que tengamos en la carpeta del servido

* **Run rm -rf /usr/share/nginx/html/\***

#Copiamos el archivo en la ruta del servidor

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

#Tomamos lo que esta en dist y lo copiamos en la carpeta del servidor

* **COPY dist/billingApp /usr/share/nginx/html**

#Copiamos el archivo mime.types en la carpeta del servidor

* **COPY mime.types /etc/nginx/mime.types**

**IMPORTANTE**

En el archivo “nginx.cof” del creador o de nuestro sercio esta definido el puerto.

**Nos falta la aplicación java ya hicimos lo de angular.**

#Instalammos OpenJDK 17

**RUN apk --no-cache add openjdk17-jre**

#Definimos la variable de entorno JAVA\_HOME

# Definimos la ruta de instalación de Java (JAVA\_HOME)

**ENV JAVA\_HOME /usr/lib/jvm/java-17-openjdk**

# Añadimos la carpeta de binarios de Java al PATH del sistema

**ENV PATH $JAVA\_HOME/bin:$PATH**

#Verificamos la versión de Java instalada

**RUN java -version**

#Instalamos microservice

# Variable de entorno para pasar opciones a la JVM, se deja vacía por defecto

**ENV JAVA\_OPTS=""**

# Argumento de construcción para indicar el archivo JAR a copiar

**ARG JAR\_FILE**

# Copiamos el archivo JAR al contenedor y lo renombramos como app.jar

**ADD ${JAR\_FILE} app.jar**

#Copiamos el archivo de configuración de la aplicación

**COPY appshell.sh appshell.sh**

#Los puertos que se van a exponer

#Puerto 80 para el servidor web y puerto 7080 para el back end

**EXPOSE 80 7080**

#entrypoint para ejecutar el script de inicio

**ENTRYPOINT ["sh", "appshell.sh"]**

Ahora vamos a ejecutarlo

Primero nos vamos a la carpeta del archivo desde la terminal.

Y lo creamos

* + docker build -t billingapp --no-cache --build-arg JAR\_FILE=target/\*.jar .
    - 1.**docker build**   
      Inicia la construcción de una imagen Docker usando el Dockerfile del directorio actual.
    - 2. **-t billingapp**  
      Le pone el nombre (tag) billingapp a la imagen resultante.
    - 3. **--no-cache**  
      Construye la imagen sin usar la caché previa de Docker (todo se hace desde cero).
    - 4. **--build-arg JAR\_FILE=target/\*.jar**  
      Pasa como argumento el archivo JAR encontrado en target/ para usarlo en el Dockerfile.

**Una vez creado lo corremos:**docker run -d -p 80:80 -p 7080:7080 --name localbillingApp billingapp

**En docker hub creamos el repositorio**

**Re etiquetamos la imagen con el de el repositorio**

docker tag billingapp damianhz/udemy-billingapp

Si queremos darle version es:  
docker tag billingapp damianhz/udemy-billingapp:”tagname”

**Hacemos push**

docker push damianhz/udemy-billingapp

Si especificamos una versión  
docker push damianhz/udemy-billingapp:”tagname”

**LISTO ESTA PUBLICADA**

Practica 2.

Descargamos los recursos necesarios.

**Creamos un archivo yml**

# Versión de la sintaxis de Docker Compose

version: '3.8'

# Sección donde se definen los servicios del proyecto

services:

  # Servicio para la base de datos PostgreSQL

  postgres\_db:

    # Nombre que tendrá el contenedor

    container\_name: postgres\_db

    # Imagen de Docker que se usará (PostgreSQL oficial)

    image: postgres:latest

    # El contenedor se reiniciará automáticamente si se detiene

    restart: always

    # Mapea el puerto 5432 del contenedor al 5432 de tu máquina

    ports:

      - "5432:5432"

    # Volúmenes para almacenar datos y archivos de inicialización

    volumes:

      # Carpeta local para scripts de inicialización

      - ./dbfiles:/docker-entrypoint-initdb.d

      # Carpeta local donde se guardan los datos de la base de datos

      - ./postgres\_data:/var/lib/postgresql/data

    # Variables de entorno para configurar el usuario, contraseña y nombre de la base de datos

    environment:

      POSTGRES\_USER: postgres

      POSTGRES\_PASSWORD: qwerty

      POSTGRES\_DB: postgres

  # Interfaz de administración de la base de datos

  adminer:

    # Nombre que tendrá el contenedor

    container\_name: adminer

    # Imagen de Docker que se usará (Adminer oficial)

    image: adminer:latest

    # El contenedor se reiniciará automáticamente si se detiene

    restart: always

    # Indica que este servicio depende de que el de la base de datos esté listo primero

    depends\_on:

      - postgres\_db

    # Mapea el puerto 8080 del contenedor al 9090 de tu máquina

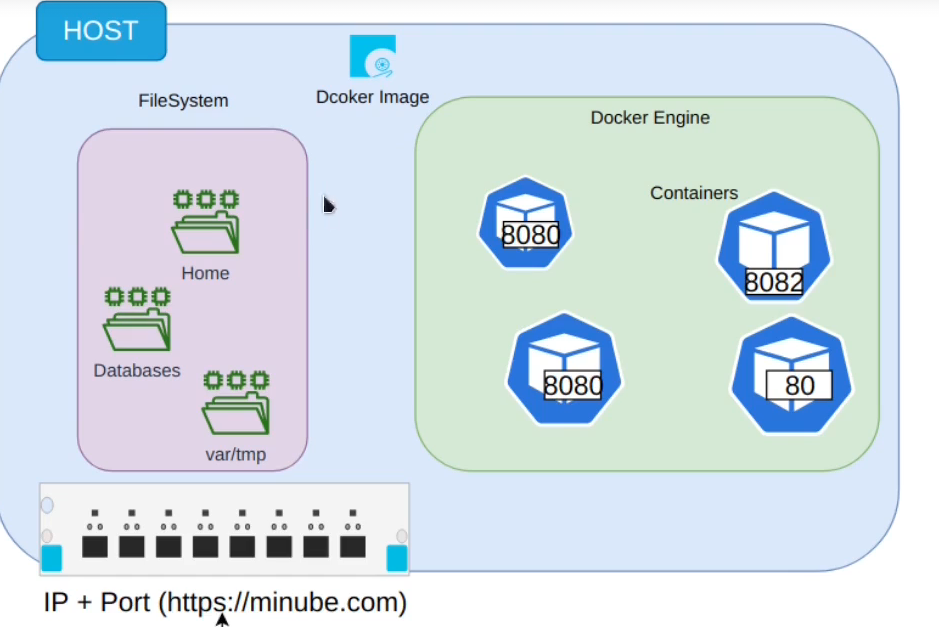
    ports:

      - 9090:8080

**En la terminal nos vamos a la dirección del archivo yml y lo ejecutamos con:** docker compose -f stack-billing.yml up -d

Listo, si ya ingresamos al localhost con los puertos que le asignamos debemos de ver el login de postgres ya esntado ahí ingresamos las credenciales y en “base de datos” tenemos que tener creada la base de datos billing-app\_db.

**Arquitectura:**



**Recordatorio:**Los volúmenes es información de los archivos.

Una buena practica es guardar la información de db en un volumen.

**Creamos el dockerfile para cada uno (back y front)**

**Front:**

FROM nginx:stable-alpine

 # use a volume is mor efficient and speed that filesystem

VOLUME /tmp

#Install web app on nginx server

RUN rm -rf /usr/share/nginx/html/\*

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

COPY billingApp /usr/share/nginx/html

COPY mime.types /etc/nginx/mime.types

#The ports 80 for web app

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]

**BACK END:**# Usa una imagen base con JRE 17 en Alpine

FROM eclipse-temurin:17-alpine

# Create a new group with specific uid and non-root user called "admin"

# Be sure that group id are not present on host. if already exist change by arbitrary other uid

RUN addgroup -g 1028 devopsc \

    && adduser -D -G devopsc admin

# Create a new mount point at /tmp on native host because a volume is more efficient and faster than filesystem

VOLUME /tmp

# Copiamos el jar a la imagen

ARG JAR\_FILE

# Establece una variable de entorno para la contraseña de la base de datos

ARG DB\_PASSWORD

# Establece la variable de entorno DB\_PASSWORD con el valor del argumento

ENV DB\_PASSWORD=$DB\_PASSWORD

# Copia el archivo JAR

COPY ${JAR\_FILE} /tmp/app.jar

# Change ownership of the /app directory to the "admin" user

RUN chown -R admin:devopsc /tmp

# Cambia al usuario 'admin'

USER admin

# Ejecutamos el jar al iniciar el contenedor

ENTRYPOINT ["java","-jar","/tmp/app.jar"]

**Modificamos el yml par aque construya y ejecute todo:**# Versión de la sintaxis de Docker Compose

version: '3.8'

# Sección donde se definen los servicios del proyecto

services:

  # Servicio para la base de datos PostgreSQL

  postgres\_db:

    # Nombre que tendrá el contenedor

    container\_name: postgres\_db

    # Imagen de Docker que se usará (PostgreSQL oficial)

    image: postgres:latest

    # El contenedor se reiniciará automáticamente si se detiene

    restart: always

    # Mapea el puerto 5432 del contenedor al 5432 de tu máquina

    ports:

      - "5432:5432"

    # Volúmenes para almacenar datos y archivos de inicialización

    volumes:

      # Carpeta local para scripts de inicialización

      - ./dbfiles:/docker-entrypoint-initdb.d

      # Carpeta local donde se guardan los datos de la base de datos

      - ./postgres\_data:/var/lib/postgresql/data

    # Variables de entorno para configurar el usuario, contraseña y nombre de la base de datos

    environment:

      POSTGRES\_USER: postgres

      POSTGRES\_PASSWORD: qwerty

      POSTGRES\_DB: postgres

  # Interfaz de administración de la base de datos

  adminer:

    # Nombre que tendrá el contenedor

    container\_name: adminer

    # Imagen de Docker que se usará (Adminer oficial)

    image: adminer:latest

    # El contenedor se reiniciará automáticamente si se detiene

    restart: always

    # Indica que este servicio depende de que el de la base de datos esté listo primero

    depends\_on:

      - postgres\_db

    # Mapea el puerto 8080 del contenedor al 9090 de tu máquina

    ports:

      - 9090:8080

#Billing app backend service

  billingapp\_back:

    build:

      context: ./java

      args:

        - JAR\_FILE=\*.jar

        - DB\_PASSWORD=qwerty

    container\_name: billingapp\_back

    environment:

      - JAVA\_OPTS=

      -Xms26M

      -Xmx512M

  depends\_on:

      - postgres\_db

  ports:

      - 8080:8080

#Billing app frontend service

  billingapp\_front:

    build:

      context: ./angular

    container\_name: billingapp\_front

    depends\_on:

      - billingapp\_back

    ports:

      - 80:80

**Ya con esto solo necesitamos unn comando en la terminal y se hará todo**docker compose -f “nombre.yml” up -d

**Puede servir.**

Ejecutamos para conectarnos a la terminal de ese contendeor:

docker exec -it “nombre del contenedor” sh

Pra checar una variable ejmplo:

echo $DB\_PASSWORD

**DOCKER SWARMS: ORQUESTADOR AVANZADO PARA ESCALAR CONTENEDORES.**

**Para ver los recursos que están usando es:** docker stats

**Para limitar los recursos es en yml**

**Ejemplo de las lines donde se limita el front:**

#Despliegue

deploy:

resources:

limits:

cpus: '0.15' #15% de un CPU

memory: 250M

reservations:

cpus: '0.1'

memory: 125M

**Redes virtuales o entornos virtualizados.**

Primero separamos pro el servio que trabajaremos(producción, preproducción etc)

Ejemplo en el de base de datos:

services:

  # Servicio para la base de datos PostgreSQL

  postgres\_db\_prod:

    # Nombre que tendrá el contenedor

    container\_name: postgres\_prod

    # Imagen de Docker que se usará (PostgreSQL oficial)

    image: postgres:latest

    # El contenedor se reiniciará automáticamente si se detiene

    restart: always

    # Mapea el puerto 5432 del contenedor al 5432 de tu máquina

    ports:

      - "5432:5432"

    # Volúmenes para almacenar datos y archivos de inicialización

    volumes:

      # Carpeta local para scripts de inicialización

      - ./dbfiles:/docker-entrypoint-initdb.d

      # Carpeta local donde se guardan los datos de la base de datos

      - ./postgres\_data\_prod:/var/lib/postgresql/data

    # Variables de entorno para configurar el usuario, contraseña y nombre de la base de datos

    environment:

      POSTGRES\_USER: postgres

      POSTGRES\_PASSWORD: qwerty

      POSTGRES\_DB: postgres

  # Servicio para la base de datos PostgreSQL

  postgres\_db\_prep:

    # Nombre que tendrá el contenedor

    container\_name: postgres\_prep

    # Imagen de Docker que se usará (PostgreSQL oficial)

    image: postgres:latest

    # El contenedor se reiniciará automáticamente si se detiene

    restart: always

    # Mapea el puerto 5432 del contenedor al 5432 de tu máquina

    ports:

      - "4432:4432"

    # Volúmenes para almacenar datos y archivos de inicialización

    volumes:

      # Carpeta local para scripts de inicialización

      - ./dbfiles:/docker-entrypoint-initdb.d

      # Carpeta local donde se guardan los datos de la base de datos

      - ./postgres\_data\_prep:/var/lib/postgresql/data

    # Variables de entorno para configurar el usuario, contraseña y nombre de la base de datos

    environment:

      POSTGRES\_USER: postgres

      POSTGRES\_PASSWORD: qwerty

      POSTGRES\_DB: postgres

**En el adminer le pasas ambas db:**

    depends\_on:

      - postgres\_db\_prep

      - postgres\_db\_prep

**Redes:**

networks:

env\_prod:

driver: bridge

#active ipv6

driver\_opts:

com.docker.network.bridge.enable\_ipv6: "true"

#IP ADRES MANAGER

ipam:

driver: default

config:

-

subnet: 172.16.232.0/24

gateway: 172.16.232.1

-

subnet: "2001:3974:3979::/64"

gateway: "2001:3974:3979::1"

env\_prep:

driver: bridge

#active ipv6

driver\_opts:

com.docker.network.bridge.enable\_ipv6: "true"

#IP ADRES MANAGER

ipam:

driver: default

config:

-

subnet: 172.16.235.0/24

gateway: 172.16.235.1

-

subnet: "2001:3984:3989::/64"

gateway: "2001:3984:3989::1"

Simplemente pasa el argumento en dode vaya de lo anterior.

networks:

      - env\_prod

**Docker swarm**

Usaremos el mismo script

Lo necitamos activar con:

docker swarm init

guardar el node addres.

Con docker swarm si es necesario definir la imagen.

No soporta container\_name ya que no lo soporta.

Se puede dar un rango de puertos.

Salir de docker swarm

docker stack rm billing

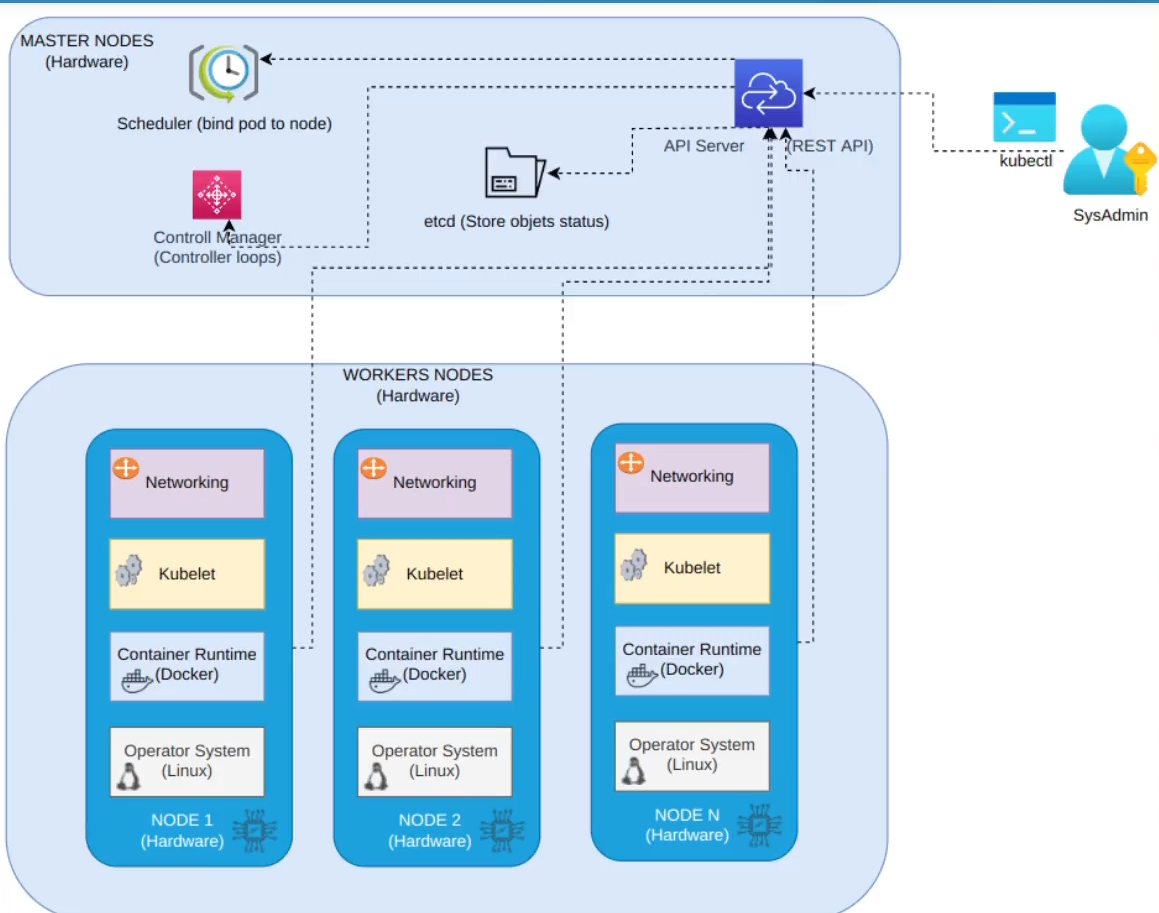
docker swarm leave --force

**kubernetes.**

Gestionar varias máquinas, muy similar a docker compose.

Escalado vertical: Mas recursos al mismo nodo.

Escalado horizontal: Agregar mas nodos.

Cluster de cubernetes:  


Los nodos maestros son las maquinas que destinan a tener todas las tareas que se encargan de la gestión del cluster.

Nodos worker: todas las tareas de la propia aplicación.

**Etcd:** Almacena toda la configuración y el estado del cluster.

**SCHEUDELER:** Selecciona los nodos para los pods y los ejecuta.

**Controller-manager:** Supervisa y gestiona toda la estructura.

**API SERVER:** interfaz con la cual interactuamos con el cluster. Accedemos mediante Kubectl.

**Nodo worker:**

4 capas.

**Networking:**

**KUBELET:** Se encarga de monotorizar para que corra en un pod.

**Contenedor:** Empaquetar una app.

**SO.**

**Tipos.**

**Gestionado:** Es un clúster (conjunto de servidores) administrado por un proveedor en la nube (como AWS, Azure o Google Cloud). Ellos se encargan del mantenimiento, escalado, seguridad, etc.

**On-premise:** Es cuando tú tienes y manejas tu propio clúster localmente, en tus instalaciones físicas. Tú te encargas de todo: hardware, red, seguridad, actualizaciones, etc.

* + **All un one:** se instala en un único nodo usando mini kube par propósitos educativos y pruebas.

**Objetos de kubernetes.**

Se define en yaml.

Se guardan y se ejecutan el cluster mediante API Server.

* + **Pods:** Unidad mas pequeña que se puede desplegar y gestionar kubernetes.
  + **Deployments:** Describe el estado deseado de una implementación, ejecuta múltiples replicas de la aplicación, remplaza las que están defectuosas o las que no responden.
  + **Services:** Definicion de como exponer una aplicación que es un conjunto de pods como un servicio de red.
  + **Config map:** Permite desacoplar la configuración para hacer las imágenes mas portables, almacenan variables de entorno, argumentos para línea de comandos, o configuración de volúmenes que pueden consumir los pods.
  + **Labels:** Par de claves valor para organizar, seleccionar, consultar y monitorear objetos de forma mas eficiente, ideales para UI y CLis.
  + **Selectores:** Mecanismo para hacer consultas a las etiquetas, kubectl get pods -l ‘environment in (production), tier in (frontend)’

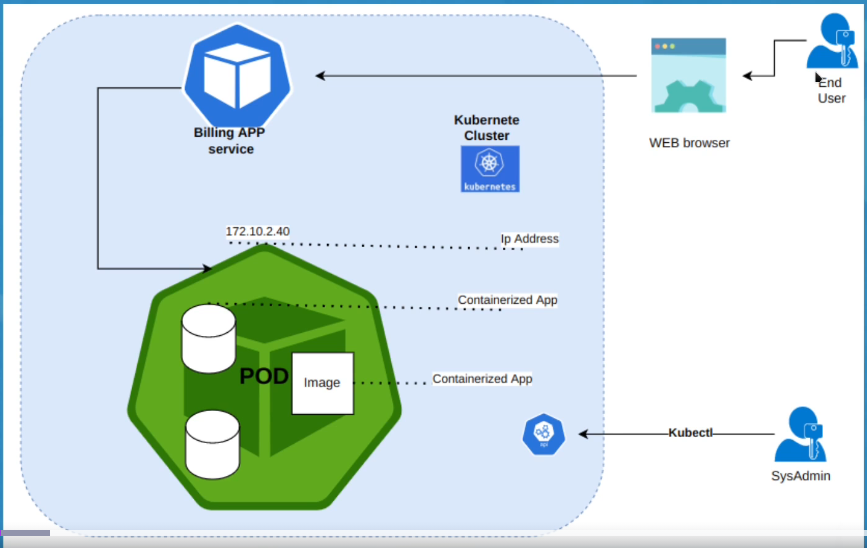
**Minikube start**

**Instalamos kubernetes:**

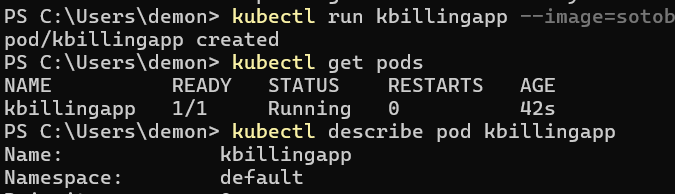
Checamos que tengamos la vitualizacion encendida.

**Instalamos Kubectl**

**Intalamas minikube**

**Vamos a desplegar la app de facturación en kubernetes:**

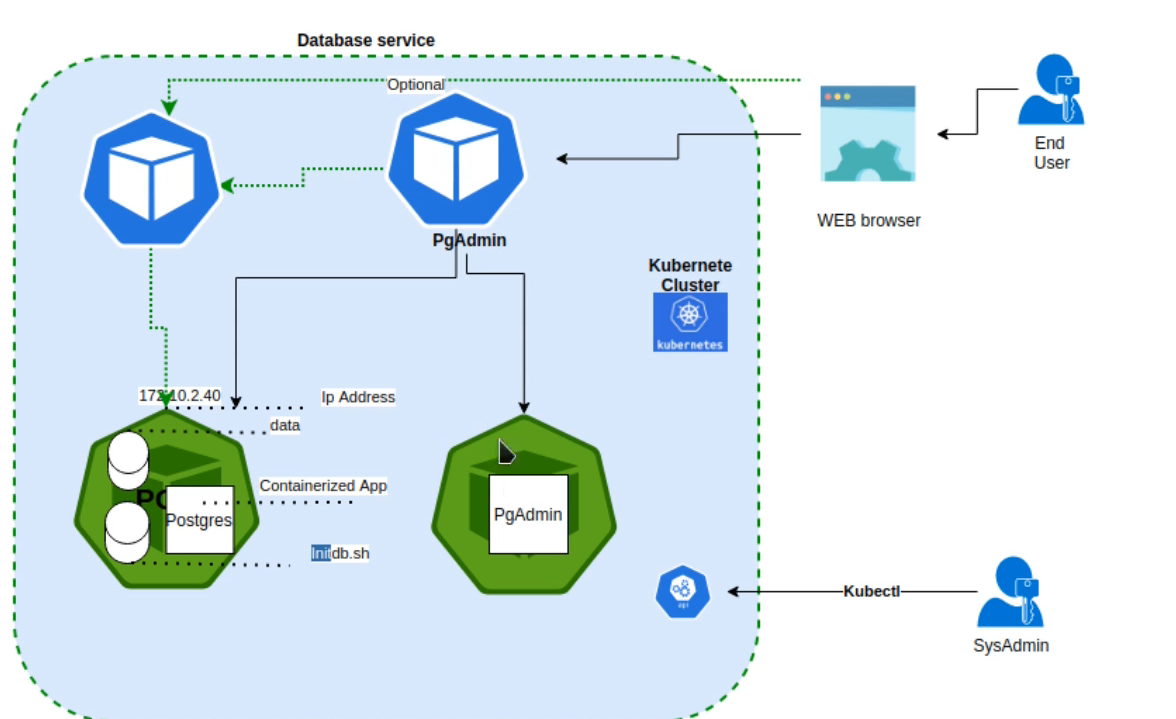
Para la practica descargamos la imagen:  
kubectl run kbillingapp --image=sotobotero/udemy-devops:0.0.1 --port=80 80



Ahora vamos a exponer el servicio

kubectl expose pod kbillingapp --type=LoadBalancer --port=8080 --target-port=80

**Montar un servicio mas Complejo:**

****

**Creamos los srvhivos yaml**

**Ahora a hacer los pods:**

**Definimos los servicios**

**Ahora nos vamos a conectar.**

Crear uno a uno: kubectl apply -f service-pgadmin.yaml  
**Abrir el pgadmin:**minikube service pgadmin-service