Introducción al uso de Docker

Docker desglosado

Este apartado es un **tutorial opcional** para aprender a dominar Docker en su totalidad. Algunos de los conceptos que se verán ya se han tratado en el apartado anterior. Esto se debe a que este documento trabaja tomando como referencia el siguiente vídeo de youtube:

Este es el vídeo que vamos a tomar como referencia

> **Actividad**
> Realiza una memoria de todo el proceso, hasta configurar tu servidor MySQL en Docker.

Contenedor

Los contenedores son una forma de empaquetar una aplicación y todas sus dependencias.
De esta forma, se consigue una mayor portabilidad y se mejora la posibilidad de compartir la aplicación en diferentes equipos, facilitando el desarrollo y despliegue de aplicaciones.

Los contenedores de Docker se almacenan en repositorios, ya sean privados o públicos, como dockerhub, que es el repositorio oficial.

Para unificar las herramientas de desarrollo de un equipo, un contenedor almacena todas las dependecias necesarias. Se construyen sobre una imagen de Linux. De esta forma, se evitan errores relacionados con compatibilidad y dependencias, al tenerlas almacenadas en imágenes. Por lo tanto, los contenedores son un tipo de virtualización.

Para entender la diferencia entre una máquina virtual y un contenedor, debemos saber que un ordenador se divide en diferentes capas: la capa de hardware, el kernel y la capa de aplicaciones. Con una máquina virtual, se virtualizan tanto el kernel como la capa de aplicaciones. El contenedor, por su parte, solo virtualiza la capa de aplicaciones, empleando el kernel del sistema operativo anfitrión (Linux o Mac OSX) o, en el caso de windows, WSL2 (Windows Subsystem for Linux).

Al virtualizar solamente la capa de aplicaciones, los contenedores son más rápidos y ocupan menos que las máquinas virtuales, por lo que son más apropiados para el desarrollo.

Instalar Docker

Para instalar Docker debes ir a su página web y descargar Docker Desktop. Docker Desktop incluye herramientas como Docker-compose y CLI (la interfaz línea de comandos).

Manejo de imágenes

Para manejar Docker, emplearemos la línea de comandos. Para que funcione, debemos tener ejecutándose Docker Desktop, programa que habilita el uso de CLI.

```
$ docker images
```

Este comando muestra todas las imágenes descargadas. Nos muestra los siguientes campos:

- **Repository**: Nombre de la imagen.
- **Tag**: Etiqueta de la imagen.
- **Image Id**: Identificador de la imagen.
- **Created**: Información sobre cuándo fue creada la imagen.
- **Size**: Tamaño de ocupa la imagen.

```
$ docker pull node
$ docker pull node:lts-alpine3.19
```

En el primer caso, descarga la imagen de `node` etiquetada como `latest`. En el segundo caso, descarga la imagen de `node` etiquetada como `lts-alpine3.19`. En ambos casos, descargará el contenido del repositorio en caso de que no haya sido descargado previamente. El contenido de las imágenes está dividido en módulos para hacer más eficiente la descarga de imágenes, ya que diferentes imágenes pueden compartir algunos módulos, o la misma imagen podría estar etiquetada de diferentes formas (cosa que suele ocurrir con aquella etiquetada con `latest`).

En ocasiones, es necesario especificar la plataforma (especialmente si no se corresponde con la nuestra). Por ejemplo, si estamos usando un Macbook Air con el procesador M1 y queremos descargar una imagen de `mysql` debemos emplear el siguiente comando:

```
$ docker pull --platform linux/x86_64 mysql
```

Para borrar una imagen que hayamos descargado, empleamos el siguiente comando:

```
$ docker image rm node:lts-alpine3.19
```

En este caso, borraremos la imagen de `node` etiquetada con `18` que habíamos descargado antes.

Manejo de contenedores

Los contenedores son el elemento principal de Docker y se crean empleando una imagen como base.

```
$ docker pull mongo
$ docker create mongo
```

En este caso, primero descargamos la imagen de mongo con el comando `pull` y, después, creamos un contenedor con el comando `create`. Este comando nos devuelve un id, que debemos copiar y añadir al siguiente comando:

```
$ docker start [id]
```

En este caso, sustituimos `[id]` por la id que habíamos copiado y ejecuta el contenedor, devolviéndonos la propia id de nuevo. Si ejecutamos el siguiente comando:

```
$ docker ps
```

Veremos todos los contenedores en ejecución. Si añadimos el argumento `-a` veremos todos los contenedores en la máquina, tanto en ejecución como no.

Este comando nos devuelve una tabla similar a la que obteníamos con `$ docker images`, que nos revela la siguiente información:

- **Container ID**: La id, en formato reducido, del contenedor.
- **Image Command**: La imagen base del contenedor.
- **Created**: La fecha de creación del contenedor.
- **Status**: El estado (activo o inactivo) del contenedor.
- **Ports**: Los puertos que usa el contenedor.
- **Name**: El nombre del contenedor. Docker asigna nombres aleatorios al contenedor, pero podemos asignarlo nosotros de forma manual con el siguiente comando:

```
$ docker create --name mimongo mongo
```

En este caso, creará una imagen basada en mongo y llamada `mimongo`. Para ejecutarla, podemos usar `$ docker start` y añadir al final el nombre o la id del contenedor.

```
$ docker stop mimongo
$ docker rm mimongo
```

El primer comando sirve para detener la ejecución del contenedor llamado `mimongo`. En lugar del nombre, también se puede usar la id. El segundo comando sirve para borrar el contenedor `mimongo` y, de la misma forma que antes, el nombre se puede sustituir por la id.

Es recomendable usar nombres personalizados, ya que facilita el uso de los contenedores.

Port Mapping

En telemática y redes informáticas, los puertos son como ventanas virtuales que permiten que una computadora se conecte con varias otras al mismo tiempo. Cada ventana está numerada del 0 al 65,535 y sirve para dirigir la información entrante al programa correcto que está esperándola.

El puerto que aparecía cuando ejecutábamos el comando `$ docker ps` y teníamos un contenedor corriendo era un puerto interno del contenedor, pero cerrado y no accesible desde nuestro equipo. Para poder acceder a un puerto de un contenedor, debemos mapearlo a uno de nuestro equipo.

```
$ docker create -p27017:27017 mongo
```

Con este comando, mapeamos el puerto 27017 del contenedor a nuestro puerto 27017. Es una práctica común mapear los puertos internos del contenedor al mismo número en nuestro equipo y, en caso de no poderse, a un número similar.

En la instrucción `-p27017:27017`, el primer puerto es el de nuestra máquina y el segundo el del contenedor. El nombre extendido del puerto de nuestra máquina es, en realidad, `0.0.0.0:27017` o `localhost:27017`.

Además del comando anterior, también podemos no especificar un puerto de nuestro equipo y que Docker lo asigne arbitrariamente usando `-p27017`. Sin embargo, es preferible asignar los puertos de manera manual.

De esta forma, podremos acceder al contenedor a través de su puerto. Para ello, abrimos un navegador y nos dirigimos a la dirección de nuestro equipo asociada al contenedor, como por ejemplo http://localhost:27017. Mientras tanto, en el terminal, podemos acceder a los logs internos del contenedor de la siguiente forma:

```
$ docker logs mimongo
$ docker logs --follow mimongo
```

En el primer caso, vemos los logs del contenedor `mimongo` en el momento dado en el que ejecutamos el comando. En el segundo caso, el terminal se queda escuchando los logs internos del contenedor `mimongo`.

Docker Run

Docker Run es un comando que sirve para crear y ejecutar un contenedor en un solo paso. Es decir, es una combinación de los comandos `pull`, `create` y `start`. Todos los parámetros anteriores son compatibles.

```
$ docker run mongo
```

Descarga la imagen de mongo si no está presente en el sistema, crea el contenedor y lo ejecuta. Muestra los logs del contenedor, por lo que si cerramos el terminal cerraremos el contenedor.

```
$ docker run -d mongo
```

El argumento `-d` indica que se va a ejecutar en modo "detached", que significa despegado. Esto quiere decir que vamos a poder seguir introduciendo comandos, ya que no se verán los logs.

```
$ docker run --name mimongo -p27017 -d mongo
```

Por último, con este comando estamos creando y ejecutando un contenedor de nombre `mimongo` basado en la imagen `mongo` y, además, estamos mapeando el puerto interno 27017 del contenedor a nuestro puerto 27017.

El comando `$ docker run` siempre crea un contenedor nuevo cuando se ejecuta. Si queremos ejecutar un contenedor ya creado, debemos usar `$ docker start` y el nombre o la id del contenedor.

Variables de entorno y cómo conectarse a un contenedor

En node.js existe una librería llamada mongoose que nos permite conectarnos a bases de datos mongo. Imagina la siguiente instrucción de javascript:

```
mongoose.connect('mongodb://nombre:password@localhost:27017/miapp?authSource=admin');
```

En esta instrucción, le estamos diciendo al objeto mongoose que se conecte a `mongoddb` usando como nombre `nombre` y como password `password`. La @ sirve para separar esos datos de dónde vamos a conectarnos, en este caso a `localhost:27017' que es nuestro equipo en su puerto 27017. Finalmente, el URI nos indica a qué parte del programa nos conectamos, en e` miapp `es la aplicación y después de ? se colocan los parámetros de configuración,` authSource=admin`.

Para poder realizar correctamente esta conexión, debemos haber configurado las **variables de entorno** del contenedor. Cada imagen tiene sus propias variables de entorno, que debemos consultar en el mismo repositorio. En el caso de la imagen de mongo, solamente necesitamos configurar las siguientes variables de entorno para que funcione todo:

```
MONGO_INITDB_ROOT_USERNAME
MONGO_INITDB_ROOT_PASSWORD
```

Para ello, empleamos el siguiente comando:

```
$ docker create -p27017:27017 --name mimongo
-e MONGO_INITDB_ROOT_USERNAME=nombre
-e MONGO_INITDB_ROOT_PASSWORD=password
mongo
```

El argumento `-e` indica que lo que viene a continuación es una variable de entorno. Para inicializarlas, colocamos su nombre, el signo = y el valor que queremos asignarle. Recuerda que las variables de entorno se colocan antes de la imagen. Aunque en el ejemplo estén en diferentes líneas, se trata de un solo comando.

Dockerfile, o cómo crear imágenes personalizadas

Los archivos dockerfile sirven para construir imágenes personalizadas. Dentro de él se escriben instrucciones para crear imágenes que ejecutarán nuestro código, basadas generalmente en otras imágenes.

```
FROM node:lts-alpine3.19
RUN mkdir -p /home/app
COPY . /home/app
EXPOSE 3000
CMD ["node", "/home/app/index.js"]
```

Suponiendo que index.js es un archivo que hemos creado nosotros, el anterior archivo dockerfile hace lo siguiente:

- `FROM node:lts-alpine3.19` : Le indica a Docker cuál es la imagen sobre la que creamos nuestra imagen propia.
- `RUN mkdir -p /home/app` : Crea directorio llamado "app" dentro del directorio "/home". La opción -p le indica al comando mkdir que cree el directorio y cualquier directorio padre que aún no exista.
- `COPY . /home/app` : Copia el contenido de la dirección `.` de nuestro ordenador en la dirección `/home/app` de la imagen. `.` representa el directorio en el que se encuentra el archivo dockerfile.
- `EXPOSE 3000` : Expone el puerto 3000, de forma similar a como si hiciéramos `-p3000:3000` en los comandos vistos en los apartados anteriores.
- `["node", "/home/app/index.js"]` : Ejecuta dentro de la imagen el comando `node /home/app/index.js` . Esto permite que nuestro código se ejecute en la imagen.
  Con estas instrucciones, puedes probar a ejecutar tu propio código javascript en un contenedor de node, en lugar de tener que descargarte una versión de node en tu ordenador.

Para montar la imagen, empleamos el siguiente comando:

```
$ docker build -t miapp:1 .
```

El comando `$ docker build` construye la imagen usando el archivo dockerfile como referencia. El argumento `-t` sirve para etiquetar la imagen. En este caso, `miapp` es el nombre que le damos a la imagen y `1` la etiqueta que le asignamos. Finalmente, el '.' sirve para indicarle dónde está el archivo dockerfile, en este caso, indica el directorio actual.

Para probar que esto funciona bien, vamos a usar el siguiente código (archivo index.js):

```
const express = require('express');

var app = express();

app.get('/', function(req, res) {
  res.send('hello world');
});
app.listen(3000);
```

En este caso, necesitamos el módulo `express` que instalaremos con la orden `$ npm install express` en el directorio raíz.

Para ejecutar el código a través de la imagen de docker, debemos introducir la siguiente instrucción:

```
$ docker run -p 3000:3000 miapp:1
```

De esta forma, comunicaremos el puerto expuesto dentro de docker usando `EXPOSE 3000` con el puerto 3000 de nuestra máquina. Al ir al navegador y conectarnos a `http://localhost:3000/` obtendremos como respuesta `hello world` .

Redes internas de docker

Para que un contenedor pueda conectarse a otro contenedor, necesitamos definir una red interna. Docker genera algunas de forma automática, aunque si no usamos `docker-compose` deberemos especificarlas de forma manual.

```
$ docker network ls
```

Este comando lista todas las redes de docker.

```
$ docker network create mired
```

Crea una red interna de docker con el nombre `mired` .

```
$ docker network rm mired
```

Borra la red interna de docker llamada `mired` .

Para conectarse entre sí, los contenedores de una misma red se comunican a través de su nombre. Para ello, debemos añadir el argumento `--network` a la creación de cada contenedor. Supongamos que tenemos nuestra imagen personalizada `miapp` creada del paso anterior:

```
$ docker create -p27017:27017 --name mimongo --network mired
-e MONGO_INITDB_ROOT_USERNAME=nombre
-e MONGO_INITDB_ROOT_PASSWORD=password
mongo

$ docker create -p3000:3000 --name app --network mired miapp:1
```

En este caso, al crear los dos contenedores, estamos asignándoles a cada uno la red que hemos creado anteriormente.

A continuación, la línea de javascript para conectarse a mongo desde nuestro contenedor con imagen personalizada:

```
mongoose.connect('mongodb://nombre:password@mimongo:27017/miapp?authSource=admin');
```

Como se puede observar, se sustituye `localhost` (nombre de nuestra máquina) por `mimongo` (nombre del contenedor de MongoDB).

El archivo completo en javascript, `index.js` es como sigue:

```
const express = require('express');
const mongoose = require('mongoose');

var app = express();
mongoose.connect('mongodb://nombre:password@mimongo:27017/miapp?authSource=admin');

const kittySchema = new mongoose.Schema({
    name: String
});
const Kitten = mongoose.model('Kitten', kittySchema);
const silence = new Kitten({ name: 'Silence' });
silence.save();

app.get('/', (req, res) => {
    Kitten.find().then(result => {
        res.send(result);
    }).catch(error=>{
        let data = errorMsg(error);
        res.send(data);
    });
});
app.listen(3000);
```

Se trata de una aplicación `javascript` que usa los módulos `express` y `mongoose` , este último para conectarse a `mimongo` .

Para usar `mongo` , primero se crea un esquema para la base de datos y después se crea el modelo. A través del modelo se hacen todas las peticiones, desde la creación y guardado con `save()` en la base de datos hasta la consulta con `find()` . En este caso, dicha consulta ocurrirá cuando accedamos a `http://localhost:3000/` .

Para lanzarlo bien, debemos ejecutar, aparte del los comandos para lanzar el contenedor de mongo, los siguientes cada vez que hagamos un cambio en nuestro archivo `index.js` :

```
$ docker stop app
$ docker rm app

$ docker build -t miapp:1 .
$ docker create -p3000:3000 --name app --network mired miapp:1
$ docker start app
```

Docker Compose

Como se ha podido observar en las líneas de terminal empleadas en el ejemplo anterior, hasta ahora, para poder emplear un contenedor de docker de forma efectiva hemos tenido que realizar una gran cantidad de pasos. Sin embargo, todo ese proceso se puede simplicar empleando el comando docker-compose y un archivo de configuración de tipo `.yml` que llamaremos 'docker-compose.yml`.

El lenguaje **yml**(pronunciado yámel) se emplea para crear archivos de configuración. En este lenguaje, como ocurre en otros como **python** una indentación adecuada es necesaria para que se ejecute correctamente. Vigila, por lo tanto, los espacios.

```
version: "3.9"
services:
    app:
        build: .
        ports:
            - "3000:3000"
        links:
            - mimongo
    mimongo:
        image: mongo
        ports:
            - "27017:27017"
        environment:
            - MONGO_INITDB_ROOT_USERNAME=nombre
            - MONGO_INITDB_ROOT_PASSWORD=password
```

A continuación, pasamos a explicar las líneas de código.

```
version: "3.9"
```

Esta línea especifica la versión de Docker Compose que se utilizará. En este caso, se está utilizando la versión 3.9.

Esta línea es opcional. Si se omite la declaración de la versión en el archivo docker-compose.yml, Docker Compose intentará inferir automáticamente la versión correcta basándose en las características y sintaxis utilizadas en el archivo.

```
services:
```

Esta línea comienza un bloque que define los servicios que se ejecutarán como contenedores.

```
    app:
        build: .
```

Con estas líneas, se define el servicio `app` , que se construirá a partir del Dockerfile presente en el directorio actual (`.`). Este servicio se ejecutará en un contenedor y representará la aplicación.

```
        ports:
            - "3000:3000"
```

Esta sección especifica el mapeo de puertos para el servicio `app` . El puerto 3000 del contenedor se mapeará al puerto 3000 de la máquina host, lo que significa que la aplicación en el contenedor estará disponible en `localhost:3000` en la máquina host.

```
        links:
            - mimongo
```

Esta sección especifica a qué otros servicios se enlaza el servicio `app`. De esta manera, docker gestiona de forma automática las redes internas.

```
    mimongo:
        image: mongo
```

Se define el servicio `mimongo`, que se creará a partir de la imagen `mongo` disponible en Docker Hub. Este servicio se ejecutará en un contenedor y representará una instancia de MongoDB.

```
        ports:
            - "27017:27017"
```

Esta sección especifica el mapeo de puertos para el servicio `mimongo`.

```
        environment:
            - MONGO_INITDB_ROOT_USERNAME=nombre
            - MONGO_INITDB_ROOT_PASSWORD=password
```

Aquí se definen las variables de entorno para el servicio `mimongo`. Estas variables se utilizan para establecer el nombre de usuario y la contraseña de la base de datos MongoDB. En este caso, el nombre de usuario se establece como `nombre` y la contraseña como `password`.

Para que docker ejecute todos los comandos especificados en el archivo docker-compose.yml debemos usar el siguiente comando:

```
$ docker compose up
```

Se le puede añadir el argumento `-d` para que se ejecute en modo *detached*. Si no lo hacemos, cuando pulsemos `ctrl + C`, se terminará la ejecución de los contenedores, aunque seguirán creados.

Para detener y/o eliminar lo creado con `docker compose up`, se emplea el siguiente comando:

```
$ docker compose down
```

Es importante borrar todo lo creado cuando se realizen cambios, antes de volver a subirlo.

De esta manera se integran todos los comandos anteriores en solamente dos, lo que facilita el trabajo con contenedores de docker. Sin embargo, la información interna de estos contenedores no es persistente, lo que significa que lo que guardemos en la base de datos `mimongo` se borrará al detener la ejecución del contenedor. Además, para poder hacer pruebas en el archivo `index.js` tenemos que estar alternando entre `down` y `up` constantemente. Para solucionar estos dos inconvenientes, tenemos a nuestra disposición la herramienta `volumes`.

Volúmenes

Los volúmenes en Docker son un mecanismo para persistir datos fuera del sistema de archivos de un contenedor. Esto es especialmente útil para bases de datos, archivos de configuración o cualquier dato que deba sobrevivir a la eliminación o reinicio de un contenedor. Los volúmenes permiten compartir datos entre el host y los contenedores, o entre múltiples contenedores.

Tipos de volúmenes

1. **Volúmenes con nombre**:
   - Son gestionados por Docker y se almacenan en un directorio específico del host (generalmente en `/var/lib/docker/volumes` en Linux).
   - Son ideales para persistir datos de manera independiente al ciclo de vida de los contenedores.
2. **Bind mounts**:
   - Permiten mapear un directorio o archivo específico del host a un directorio o archivo dentro del contenedor.
   - Útiles cuando necesitas acceso directo a archivos del host o para desarrollo.
3. **Tmpfs mounts**:
   - Almacenan datos en la memoria RAM del host.

- Son temporales y se eliminan cuando el contenedor se detiene.

Creación y uso de volúmenes

Crear un volumen con nombre

Para crear un volumen con nombre, usa el siguiente comando:

```
docker volume create mi_volumen
```

Esto creará un volumen llamado `mi_volumen` que podrás usar en tus contenedores.

Usar un volumen en un contenedor

Para usar un volumen en un contenedor, utiliza el argumento `-v` o `--mount` en el comando `docker run`. Por ejemplo:

```
docker run -d --name mi_contenedor -v mi_volumen:/ruta/en/contenedor nginx
```

En este caso:

- `mi_volumen` es el nombre del volumen.
- `/ruta/en/contenedor` es la ruta dentro del contenedor donde se montará el volumen.

Usar bind mounts

Para mapear un directorio del host a un contenedor, usa:

```
docker run -d --name mi_contenedor -v /ruta/en/host:/ruta/en/contenedor nginx
```

Aquí:

- `/ruta/en/host` es la ruta absoluta en el host.
- `/ruta/en/contenedor` es la ruta dentro del contenedor.

Gestión de volúmenes a través de Docker CLI

Listar volúmenes

Para ver todos los volúmenes creados en tu sistema, usa:

```
docker volume ls
```

Inspeccionar un volumen

Para obtener detalles sobre un volumen específico, usa:

```
docker volume inspect mi_volumen
```

Esto mostrará información como la ubicación del volumen en el host y su configuración.

Eliminar un volumen

Para eliminar un volumen que ya no necesitas, usa:

```
docker volume rm mi_volumen
```

Si el volumen está en uso por un contenedor, primero debes detener y eliminar el contenedor.

Ejemplo práctico: Persistencia de datos en MySQL

Supongamos que quieres ejecutar un contenedor de MySQL y persistir los datos de la base de datos en un volumen.

1. Crea un volumen para MySQL:

   ```
   docker volume create mysql_data
   ```

2. Ejecuta el contenedor de MySQL usando el volumen:

   ```
   docker run -d --name mysql_db -e MYSQL_ROOT_PASSWORD=contraseña -v mysql_data:/var/lib/mysql -p 3306:3306 mysql
   ```

   Aquí:
   - `mysql_data` es el volumen que almacenará los datos de MySQL.
   - `/var/lib/mysql` es la ruta dentro del contenedor donde MySQL guarda sus datos.
3. Verifica que los datos persisten:
   - Detén y elimina el contenedor:

     ```
     docker stop mysql_db
     docker rm mysql_db
     ```

   - Vuelve a crear el contenedor con el mismo volumen:

     ```
     docker run -d --name mysql_db -e MYSQL_ROOT_PASSWORD=contraseña -v mysql_data:/var/lib/mysql -p 3306:3306 mysql
     ```

   - Los datos de la base de datos seguirán intactos.

Algunas imágenes oficiales ya llevan incluida la creación de volúmenes anónimos (sin nombre) cuando ejecutamos su archivo Dockerfile, y esto es una práctica cada vez más frecuente. Si no estamos seguros, cuando ejecutemos por primera vez el Docker Run podemos comprobar si se ha creado un volumen nuevo (y qué nombre tiene) o bien a través de la línea de comandos o a través de la aplicación Docker Desktop.

> **Actividad**
> Crea una conexión a tu servidor corriendo en Docker desde DBeaver u otro editor SQL. Puedes descargar DBeaver aquí.

Anexo: Express y API REST

`Express` es un marco de aplicación web rápido, minimalista y flexible para `Node.js`. Se utiliza para crear aplicaciones web y API REST de manera sencilla y eficiente.

Una API REST (Representational State Transfer) es un estilo arquitectónico para diseñar sistemas distribuidos y servicios web que se basa en los principios del protocolo HTTP. Aquí hay una descripción detallada de los conceptos clave asociados con una API REST:

1. **Recursos:** En una API REST, los recursos son entidades de datos que pueden ser accedidas o manipuladas mediante el protocolo HTTP. Estos recursos pueden ser cualquier cosa, desde objetos de datos simples hasta entidades más complejas. Por ejemplo, en una aplicación de redes sociales, los recursos podrían ser usuarios, publicaciones, comentarios, etc.
2. **URIs (Identificadores de Recursos Uniformes):** Cada recurso en una API REST se identifica mediante un URI único. Los URIs son las rutas a través de las cuales se accede a los recursos en el servidor. Por ejemplo, `/users`, `/posts`, `/users/123`, etc.
3. **Métodos HTTP:** Los métodos HTTP son verbos que indican la acción que se debe realizar en un recurso dado. Los métodos HTTP comúnmente utilizados en una API REST son:
   - **GET:** Se utiliza para recuperar datos de un recurso.
   - **POST:** Se utiliza para crear un nuevo recurso.
   - **PUT:** Se utiliza para actualizar un recurso existente.
   - **DELETE:** Se utiliza para eliminar un recurso.
4. **Representaciones:** Los datos asociados con un recurso pueden ser representados en diferentes formatos, como JSON, XML, HTML, etc. En una API REST, los clientes pueden solicitar la representación de un recurso en un formato específico utilizando encabezados HTTP como `Accept`.
5. **Estado del Cliente y Estado del Servidor:** En una API REST, el servidor no mantiene ningún estado de cliente entre las solicitudes. Cada solicitud del cliente al servidor debe contener toda la información necesaria para que el servidor comprenda y procese la solicitud. El estado de la aplicación reside completamente en el servidor, y el cliente puede cambiar el estado del servidor enviando solicitudes HTTP.
6. **HATEOAS (Hypertext As The Engine Of Application State):** Este principio de diseño de API REST propone que las respuestas de la API deben incluir enlaces hipertextuales que permitan a los clientes descubrir de manera dinámica y navegar a través de los recursos disponibles. Esto promueve la independencia entre el cliente y el servidor, ya que el cliente puede navegar por la API sin necesidad de conocimiento previo de sus rutas.

En segundo curso trabajarás en profundidad esta materia, pero si despierta tu curiosidad, aquí tienes esta información introductoria.