Bienvenidos

Universidad
Provincial del Sudoeste
Promoviendo el Desarrollo Armónico de la Región

Clase 07.
Aplicaciones Móviles y Cloud Computing

API RestFull

Universidad Provincial del Sudoeste
Promoviendo el Desarrollo Armónico de la Región

Temario

05

Desarrollo FullStack

- ✓ Conceptos de desarrollo FullSatak
- ✓ Stacks de programación
- ✓ SOLID - STUPID

07

Desarrollo de aplicaciones Restfull

- ✓ Arquitectura REST
- ✓ CRUD con NodeJS y Express
- ✓ Middlewares / Routers

08

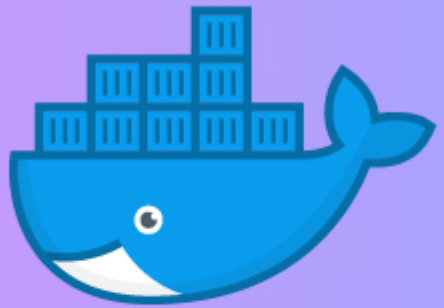
Desarrollo de aplicaciones Restfull

- ✓ WebSockets
- ✓ Motores de Plantillas
- ✓ Dependencias socket.io y express-handlebars

Universidad Provincial del Sudoeste
Promoviendo el Desarrollo Armónico de la Región

Objetivos de la clase

- Comprender los principios básicos de funcionamiento de una API-RESTFull
- Introducción a Node. Introducción a ExpressJS. Desarrollo de un CRUD con persistencia en memoria con ExpressJS
- Middlewares / Routers



docker

Universidad
Provincial del Sudoeste
Promoviendo el Desarrollo Armónico de la Región

Docker

Docker es una plataforma que permitirá crear, probar e implementar aplicativos en unidades de software estandarizadas llamadas **contenedores**.

Con docker, podremos “virtualizar” el sistema operativo de un servidor con el fin de realizar ejecución de aplicaciones con la máxima compatibilidad.

Gracias a tener nuestro aplicativo en un contenedor que corra un software con exactamente las especificaciones que necesita esta app, evitamos el típico problema del desarrollador **“en mi computadora sí funcionaba”**

Instalar Docker Desktop

¿Qué es WSL? - Proceso de instalación

Universidad Provincial del Sudoeste
Promoviendo el Desarrollo Armónico de la Región

Docker run

Docker run levanta imágenes (system images) en contenedores aislados.
Su forma más genérica es la siguiente:

```
docker run [OPTIONS] IMAGE
```

Documentación Oficial

Opciones útiles:

- **-- name**: permite especificar el nombre del contenedor a generar
- **-e**: permite definir variables de entorno; se utiliza el formato –e VARIABLE=Valor_de_la_variable
- **-p**: permite definir el puerto; recordar que la sintáxis es –p n:m, donde n es el puerto que se expone al exterior del contenedor, y m un puerto determinado característico de la imagen
- **-d**: quiere decir “detached”. Ejecuta el contenedor en segundo plano. Si no se coloca este flag, al ejecutar Docker run, la terminal queda “tomada” por el contenedor.

Ejemplo:

```
docker run --name server-mysql -e MYSQL_ROOT_PASSWORD=123 -p 3306:3306 -d mysql
```

Docker exec

Este comando se utiliza para ejecutar un comando dentro de un contenedor en ejecución.
Acompañado por el flag "**–it**", permite interactuar con el comando ejecutado dentro del contenedor.

- **-i** mantiene abierta la entrada estándar (stdin) del contenedor,
- **-t** asigna una pseudo terminal (tty) para la interacción.

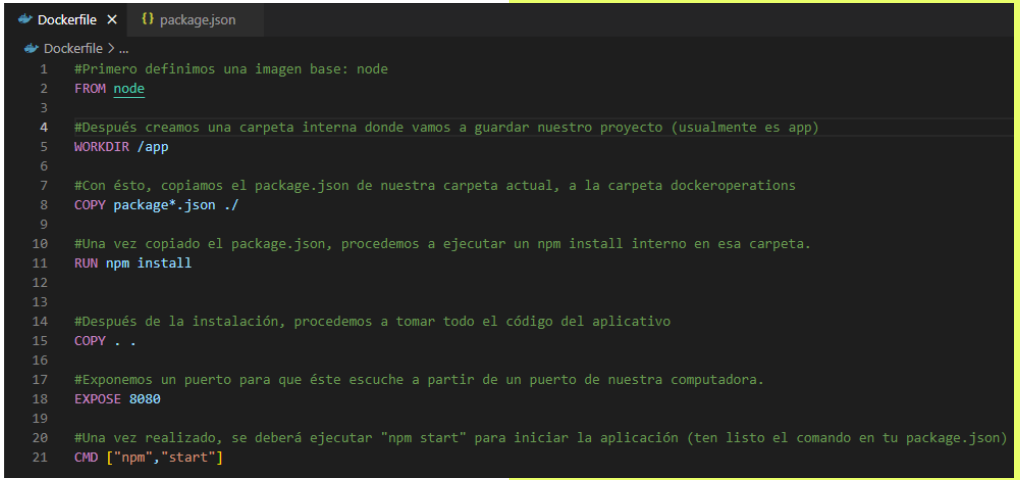
Por ejemplo, podemos ejecutar mysql en el contenedor recientemente creado de esta forma:

```
docker exec -it server-mysql mysql -uroot -p
```

Donde "**mysql –uroot –p**" es el comando que quiero ejecutar. Con mysql corro precisamente el gestor de base de datos. Con la opción **–uroot** indico que quiero ingresar como root. Con **–p** el sistema me va a solicitar la contraseña.
Ingresando 123 (la contraseña seteada en la variable de entorno al ejecutar Docker run), accedo al CLI de mysql, dentro del contenedor. Y puedo desde allí crear DB's, crear tablas, etc.

Para generar imágenes: Crear un archivo Dockerfile

FROM: define la imagen de la cual vamos a partir
WORKDIR: define el directorio de trabajo, dentro de la imagen
COPY: copia archivos desde el directorio actual, a la carpeta de la imagen
RUN nos permitirá ejecutar comandos. Al usar la imagen base **node**, significa que el entorno podrá correr comandos de node y npm sin problema.
EXPOSE: expone un puerto al exterior del contenedor, para que este pueda ser "tomado" por nuestra computadora
CMD al final es la ejecución del comando final que se utilizará al momento de echar a andar el servidor cuando hagamos **docker run**



Documentación Oficial

Universidad Provincial del Sudoeste
Promoviendo el Desarrollo Armónico de la Región

Ejecutamos el comando build

Una vez configurado nuestro respectivo dockerfile, podemos poner a prueba éste ejecutando el comando build.

```
docker build -t dockeroperations .
```

El comando build leerá el archivo y comenzará con la construcción de la imagen para nuestro aplicativo.
Una vez que tenga la imagen del aplicativo, necesita colocarle un nombre. El flag **-t** significa “tag” y es para nombrar la imagen.

El punto **.** sirve para indicarle que el **dockerfile** que necesitamos que lea está en la misma ubicación donde estamos corriendo el comando

```
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 32B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/node:latest
=> [1/5] FROM docker.io/library/node@sha256:743707dbaca64ff4ec8673a6e0c4d03d048e32b4e8ff3e89e
=> [internal] load build context
=> => transferring context: 27.69kB
=> CACHED [2/5] WORKDIR /app
=> CACHED [3/5] COPY package*.json ./
=> CACHED [4/5] RUN npm install
=> CACHED [5/5] COPY . .
=> exporting to image
=> => exporting layers
=> => writing image sha256:a13f7eb192efbdf0d43b55a2ab79d0032a5d10c6e8ec1d76ac6fba834e6ce32c
=> => naming to docker.io/library/dockeroperations
```

Volumenes en Docker

Docker permite definir estructuras denominadas Volúmenes, que se asocian por un lado a un path de nuestro sistema de archivos, y por otro a determinado contenedor. De esta forma, podemos mantener información del contenedor, en nuestros directorios. Accesibles desde cmd, powershell, explorador de windows, etc.
Nos sirve entre otras cosas para persistir de manera simple los datos sensibles que tengamos en el contenedor. Además, permite interactuar fácilmente con el mismo, incluyendo archivos desde nuestro file-system. Archivos que luego podremos tomar sin inconvenientes desde el contenedor.

¿Cómo generamos un volumen nuevo en Docker? Así:

```
docker volume create Volumen01 --opt type=none --opt o=bind --opt device="C:\data\Volumen01"
```

El flag **––opt type=none**, especifica el tipo de volumen como "none“. Significa que no se utilizará un controlador específico de Docker para el volumen. En otras palabras, el volumen no será gestionado internamente por Docker.
La opción **––opt o=bind** permite que los archivos y directorios dentro del volumen sean almacenados y accesibles en una ubicación específica del host

Docker run & Volumenes

Para levantar un contenedor, indicando un volumen, y garantizar de esta forma la persistencia de ciertos datos (comúnmente los sensibles dentro de la imagen con la que estamos trabajando), se utiliza Docker run, con el flag **–v**. La sintáxis del mismo es "**-v NOMBRE_VOLUMEN:/PATH_DENTRO_DEL_CONTENEDOR**". Completamos el ejemplo propuesto un par de pantallas atrás:

```
docker run --name server-mysql -e MYSQL_ROOT_PASSWORD=123 -p 3306:3306 -d -v Volumen01:/var/lib/mysql mysql
```

En este caso, la imagen con la que estamos trabajando es la de mysql (es el último argumento del comando, la palabra final). Y la carpeta donde almacena los archivos de base de datos es precisamente la que se indica en el comando: /var/lib/mysql

Docker run & MongoDB

De la misma forma en que configuramos con los comandos que se muestran en las pantallas anteriores un contenedor en donde corre un servidor de MySql, podemos configurar un contenedor donde trabaje un server de MongoDB. Sería de esta forma:

```
docker volume create Volumen02 --opt type=none --opt o=bind --opt device="C:\data\Volumen02"
```

para crear un volumen. Y luego:

```
docker run --name server-mongo -p 27017:27017 -d -v Volumen02:/data/db mongo
```

Esto último para levantar un contenedor con nombre server-mongo, que exponga al exterior el puerto 27017 (el que utiliza MongoDB), de manera “detached”, y asociando el volumen creado en el paso anterior. Parte de la imagen “mongo” (última palabra del comando).

Buenas prácticas
Clean Code

Universidad
Provincial del Sudoeste
Promoviendo el Desarrollo Armónico de la Región

La mayoría de los sistemas funcionan mejor si se mantienen simples en lugar de hacerlos complejos.

- ✓ No tiene sentido aumentar infinitamente el nivel de abstracción, hay que saber detenerse a tiempo
- ✓ No tiene sentido poner en el proyecto funciones redundantes «en reserva» que algún día alguien pueda necesitar.
- ✓ Para qué incluir una biblioteca enorme si solo se necesita un par de funciones de ella
- ✓ La descomposición de algo complejo en componentes simples es un enfoque arquitectónicamente correcto (aquí KISS se hace eco de DRY)
- ✓ No siempre se necesita precisión matemática absoluta o detalles extremos: Los datos pueden y deben procesarse con la precisión suficiente para una solución de alta calidad del problema, y los detalles se dan en la cantidad necesaria para el usuario, y no en el volumen máximo posible.

Universidad Provincial del Sudoeste
Promoviendo el Desarrollo Armónico de la Región

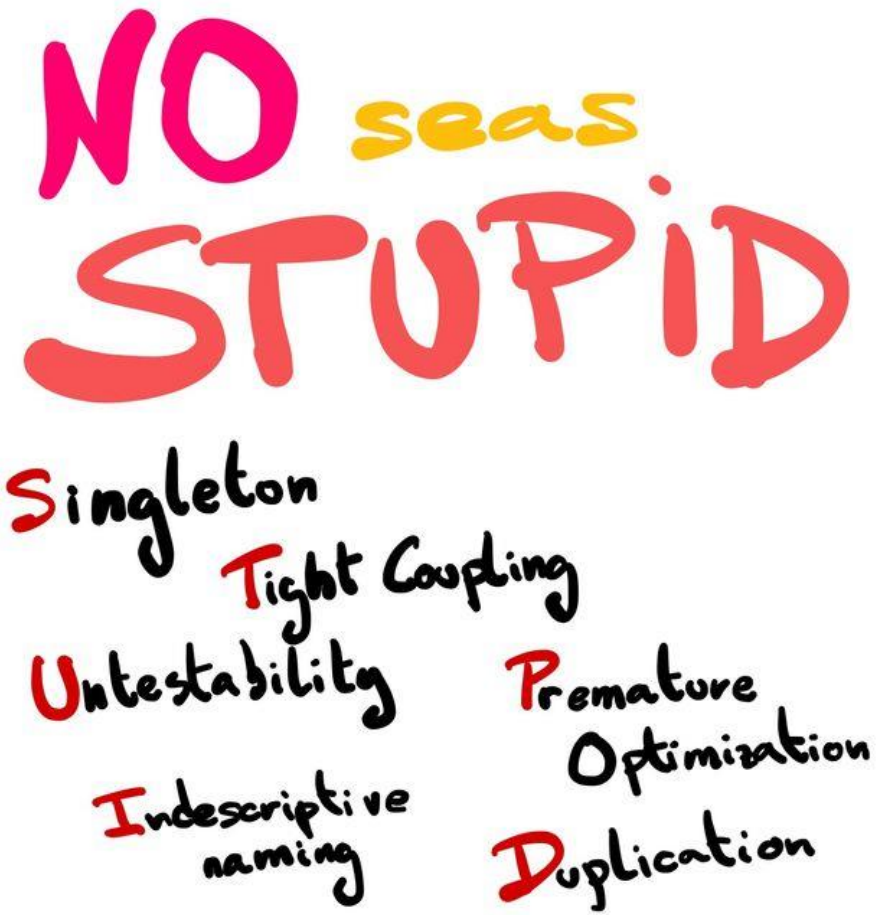
DRY nos recuerda que cada comportamiento repetible en el código debe estar aislado (por ejemplo, separado en una función) para su reutilización.

Si el código no está duplicado, para cambiar la lógica, basta con hacer correcciones en un solo lugar. Además, es más fácil probar una función puntual, aislada, en lugar de código suelto, perdido dentro de nuestro desarrollo. **Simplifica Testing**

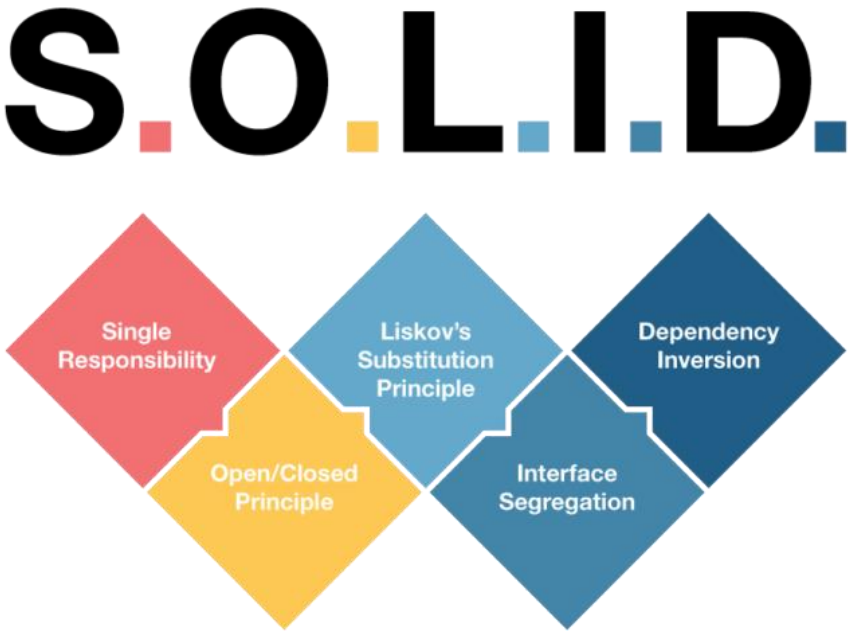
Seguir este principio siempre conduce a la descomposición de algoritmos complejos en funciones simples. Y la descomposición de operaciones complejas en operaciones más simples (y reutilizables) simplifica enormemente la comprensión del código del programa. Incrementa **Mantenibilidad**, mejora la **Calidad**

El acceso a una funcionalidad específica debe estar disponible en un solo lugar, unificado y agrupado de acuerdo con algún principio, y no «disperso» dentro del sistema.
Este enfoque se cruza con el principio de responsabilidad única,
de los cinco principios SOLID

Universidad Provincial del Sudoeste
Promoviendo el Desarrollo Armónico de la Región



- ✓ Implementación de patrón Singleton
- ✓ Código fuertemente acoplado
- ✓ Imposibilidad de realizar testing
- ✓ Optimización prematura
- ✓ Nombres poco descriptivos
- ✓ Duplicidad



- ✓ Responsabilidad única
- ✓ Principio Abierto Cerrado
- ✓ Principio de sustitución de Liskov
- ✓ Principio de segregación de interfaces
- ✓ Principio de inversión de dependencias

Universidad Provincial del Sudoeste
Promoviendo el Desarrollo Armónico de la Región

Clase 07.
Aplicaciones Móviles y Cloud Computing

API RestFull

Universidad Provincial del Sudoeste
Promoviendo el Desarrollo Armónico de la Región

Temario

05

Desarrollo FullStack

- ✓ Conceptos de desarrollo FullSatak
- ✓ Stacks de programación
- ✓ SOLID - STUPID

07

Desarrollo de aplicaciones Restfull

- ✓ Arquitectura REST
- ✓ CRUD con NodeJS y Express
- ✓ Middlewares / Routers

08

Desarrollo de aplicaciones Restfull

- ✓ WebSockets
- ✓ Motores de Plantillas
- ✓ Dependencias socket.io y express-handlebars

Universidad Provincial del Sudoeste
Promoviendo el Desarrollo Armónico de la Región

Objetivos de la clase

- Comprender los principios básicos de funcionamiento de una API-RESTFull
- Introducción a Node. Introducción a ExpressJS. Desarrollo de un CRUD con persistencia en memoria con ExpressJS
- Middlewares / Routers

Comprendiendo una API REST

Universidad Provincial del Sudoeste
Promoviendo el Desarrollo Armónico de la Región

API (Application Programming Interface)

Es un conjunto de definiciones y reglas que permiten que dos equipos puedan integrarse para trabajar juntos. La mejor analogía que hay para comprender ésto es que una API trabaja como un “contrato” entre el front y el back.

La API permite entonces que se respondan preguntas como:

- ✓ ¿A qué endpoint debo apuntar para la tarea que necesito?
- ✓ ¿Qué método debo utilizar para ese recurso?
- ✓ ¿Qué información debo enviar para realizar correctamente mi petición?

El cliente necesita algo del servidor, por lo que tiene que realizar una petición (request)

Petición

Para que la petición llegue correctamente al servidor, deberá apuntar al endpoint correcto, con el método correcto, con la información correcta

El servidor recibe la petición. Si se cumplieron todas las especificaciones de la API, el procesamiento se podrá llevar a cabo con éxito

Procesamiento

Respuesta

El cliente, al haber cumplido con lo que especificaba la API, podrá obtener su resultado satisfactoriamente y utilizarlo.

Resultado

Cumplir con el contrato de la API asegura (en la mayoría de los casos) que habrá un resultado satisfactorio

REST

Ya tenemos las reglas para comunicarse, ¿Pero qué tal la estructura del mensaje? Cuando hacemos una petición o cuando recibimos una respuesta, ésta debe tener un **formato**. REST (**RE**presentational **S**tate **T**ransfer) permite definir **la estructura** que deben tener los datos para poder transferirse.

La API respondía a preguntas sobre cómo comunicarse correctamente. En cambio, REST define cómo debe ser el cuerpo del mensaje a transmitir. Puedes llegar a hablar con el presidente si cumples con el protocolo (HTTP) y las reglas (API), pero ¿de qué nos servirá si la forma en que estructuramos nuestro mensaje (REST) no es correcta?

Los dos formatos más importantes son **JSON** y **XML**. La utilización de la estructura dependerá de las necesidades del proyecto. Nosotros utilizaremos **JSON**. Como notarás, ¡un JSON parece un objeto! así que es mucho más amigable la sintaxis.

✓ XML

```xml
<factura>
  <cliente>Gomez</cliente>
  <emisor>Perez S.A.</emisor>
  <tipo>A</tipo>
  <items>
    <item>Producto 1</item>
    <item>Producto 2</item>
    <item>Producto 3</item>
  </items>
</factura>
```

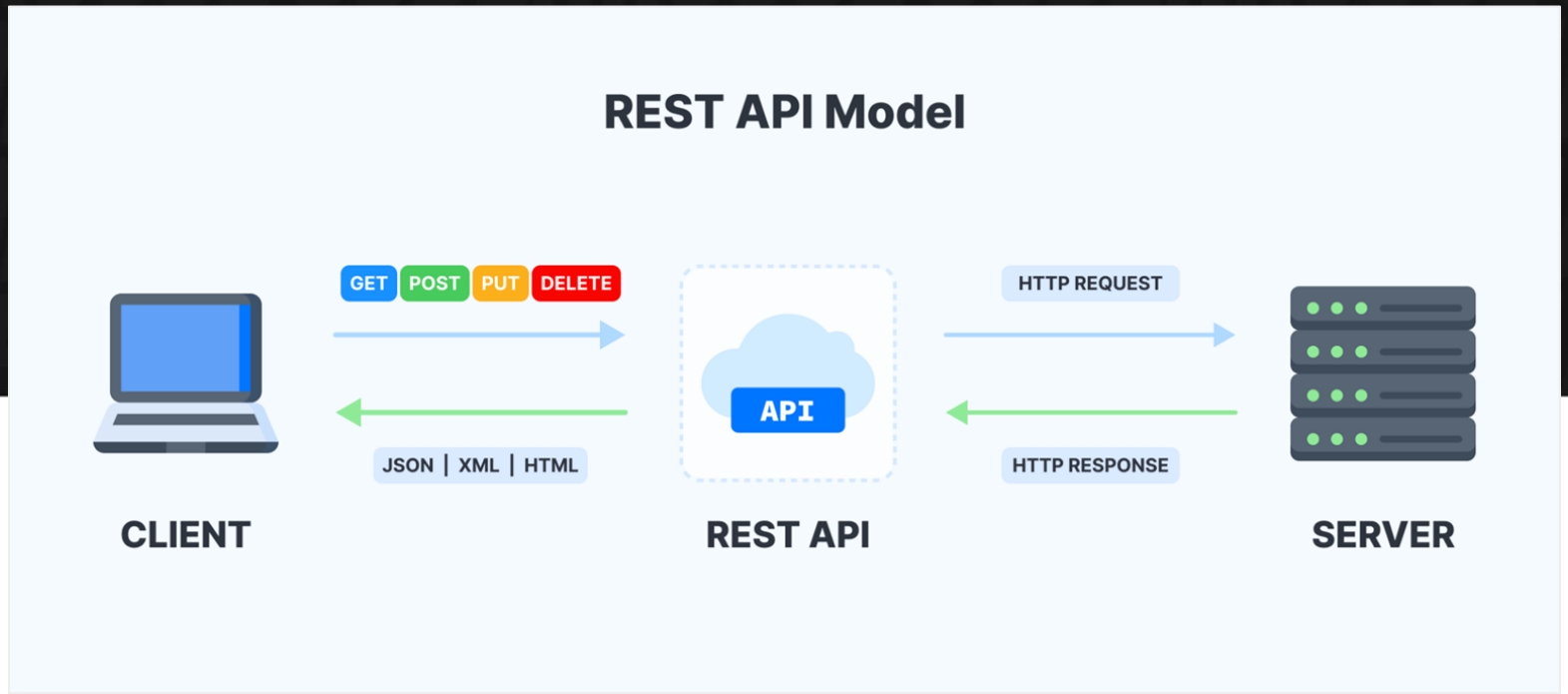
✓ JSON

```json
{
  "cliente":"Gomez",
  "emisor":"Perez S.A.",
  "tipo":"A",
  "items": [
    "Producto 1",
    "Producto 2",
    "Producto 3"
  ]
}
```

Entonces una API REST es...

Un modelo completo para tener perfectamente estipulados los protocolos, las reglas, e incluso la estructura de la información, con el fin de poder hacer un sistema de comunicación completo entre las computadoras.

Modelo de una API REST



¿Qué características debe tener una API REST?

Arquitectura Cliente–Servidor sin estado

- ✓ Cada mensaje HTTP contiene toda la información necesaria para comprender la petición.
- ✓ En consecuencia, ni el cliente ni el servidor necesitan recordar ningún estado de las comunicaciones entre mensajes.
- ✓ Esta restricción mantiene al **cliente** y al **servidor débilmente acoplados**: el cliente no necesita conocer los detalles de implementación del servidor y el servidor se “despreocupa” de cómo son usados los datos que envía al cliente.

Cacheable

- ✓ Debe admitir un sistema de almacenamiento en caché.
- ✓ La infraestructura de red debe soportar una caché de varios niveles.
- ✓ Este almacenamiento evita repetir varias conexiones entre el servidor y el cliente, en casos en que peticiones idénticas fueran a generar la misma respuesta.

Operaciones comunes

- ✓ Todos los recursos detrás de nuestra API deben poder ser consumidos mediante peticiones HTTP, preferentemente sus principales (POST, GET, PUT y DELETE).
- ✓ Con frecuencia estas operaciones se equiparan a las operaciones CRUD en bases de datos (en inglés: Create, Read, Update, Delete, en español: Alta, Lectura, Modificación, y Baja).
- ✓ Al tratarse de peticiones HTTP, éstas deberán devolver con sus respuestas los correspondientes códigos de estado, informando el resultado de las mismas.

Interfaz uniforme

- ✓ En un sistema REST, cada acción (más correctamente, cada recurso) debe contar con una URI (Uniform Resource Identifier), o identificador único.
- ✓ Ésta nos facilita el acceso a la información, tanto para consultarla, como para modificarla o eliminarla, pero también para compartir su ubicación exacta a terceros.

¿Preguntas?

Universidad
Provincial del Sudoeste
Promoviendo el Desarrollo Armónico de la Región

Muchas gracias.

Universidad
Provincial del Sudoeste
Promoviendo el Desarrollo Armónico de la Región