UNIDAD 5: Tarea

Servicios web

SERVICIOS WEB

a) ¿Qué es un servicio web (diagrama)? Lee, investiga, mira todo este vídeo y explícalo con tus palabras.

b) Indica qué ventajas e inconvenientes presentan los servicios web y para qué se usan principalmente.

c) Enumera algunos de todos los protocolos y especificaciones existentes en los servicios web.

d) Describe la arquitectura (diagrama) de un servicio web y qué tres roles y tres operaciones intervienen.

e) Indica los dos tipos de servicios web más populares en la actualidad

SOAP

a) Lee, investiga y explica con tus propias palabras qué es el protocolo de mensajes SOAP (diagrama).

b) Indica qué ventajas e inconvenientes presenta SOAP y cuál es la estructura de un mensaje SOAP.

c) Muestra un ejemplo de petición SOAP así como una respuesta del servidor a dicha petición.

d) Explica qué es WSDL y UDDI y explica detalladamente el funcionamiento de un servicio web SOAP.

REST y RESTful

a) Lee, investiga y explica el estilo de arquitectura de software para sistemas hipermedia REST.

b) Esclarece la diferencia entre REST y RESTful.

c) Explica las seis restricciones de arquitectura REST que tiene que cumplir un servicio web RESTful.

d) Señala las diferencias entre REST y SOAP, cuándo es mejor uno que otro y qué retos afrontan sus API.

HTTP RESTful API

a) Define qué es una RESTful API

b) Estudia detenidamente e indica los tres aspectos que definen una RESTful API basada en HTTP.

c) Define qué es una URI y detalla cómo se usan para nombrar recursos en una HTTP RESTful API.

d) Define HTTP y explica el uso de los métodos HTTP GET, POST, PUT y DELETE en una HTTP RESTful API.

e) Define media type, enumera algunos e indica qué media type se usa en este y en este HTTP RESTful API.

f) Define JSON, cuáles son sus tipos de datos, qué diferencias hay con XML y pon tres ejemplos de JSON.

g) Explica con tus propias palabras y detalladamente los ejemplos de esta tabla para el recurso Order.

h) Explica con tus propias palabras y detalladamente todas las combinaciones de URI y métodos HTTP para el recurso User.

HTTP RESTful API en SERVICIOS WEB

a) Di en qué consiste el servicio web JSONPlaceholder, enumera sus recursos, y pon ejemplos de rutas.

b) Explica qué es Fetch API y ejecuta los ejemplos que vienen en la guía del servicio web JSONPlaceholder.

c) Usando Fetch API, haz una petición con método HTTP GET con URI /posts/10 a JSONPlaceholder.

d) Usando Fetch API, obtén todos los comentarios de la publicación 8 de JSONPlaceholder.

e) Usando Fetch API, obtén todas las publicaciones del usuario 1 de JSONPlaceholder.

f) Busca tres clientes REST online, lístalos y usa uno para obtener todos las TODO's de JSONPlaceholder.

g) Usando Fetch API o un cliente REST online, obtén todas los posts del usuario 1 de JSONPlaceholder.

h) Usando Fetch API o un cliente REST online, crea un nuevo comentario a un post de JSONPlaceholder.

i) Usando Fetch API o un cliente REST online, actualiza o modifica un post de JSONPlaceholder.

j) Usando Fetch API o un cliente REST online, borra el usuario 5 de JSONPlaceholder.

SERVICIOS WEB

a) ¿Qué es un servicio web (diagrama)? Lee, investiga, mira todo este vídeo y explícalo con tus palabras.

Es un método de comunicación entre dos aparatos electrónicos en una red mediante una colección de protocolos abiertos y estándares usados para intercambiar datos entre aplicaciones, independientemente del lenguaje o sistema operativo en el que están programados.

b) Indica qué ventajas e inconvenientes presentan los servicios web y para qué se usan principalmente.

Las ventajas principales son las siguientes:

- Permiten que dos o más aplicaciones se comuniquen, independientemente del sistema operativo o del lenguaje de programación empleado.
- La mayoría de los servicios web están basados en protocolos de texto, como HTTP o Java Message Service (JMS).
- Permite que servicios y aplicaciones alojadas en servidores repartidos por todo el mundo puedan acoplarse.

Los inconvenientes principales son:

- La estructura de las transacciones realizadas con los servicios web son sencillas, por lo que puede que no se adecúen a nuestro caso de uso.
- A consecuencia de estar basados en protocolos de texto, su rendimiento disminuye en comparación con otros modelos de computación distribuida como Java Remote Method Invocation (RMI), Common Object Request Broker Architecture (CORBA) o Distributed Component Object Model (DCOM).
- Al basarse en protocolos de texto, pueden esquivar medidas de seguridad basadas en firewall.

Se usan principalmente para:

- Se pueden utilizar con HTTP sobre Transmission Control Protocol (TCP) en el puerto de red 80 (por defecto), por lo que se evita el uso de más puertos y permite modificar los parámetros de red de forma más sencilla al tratarse de solo uno.
- Facilita la comunicación entre sistemas conectados a la red gracias al protocolo Simple Object Access Protocol (SOAP).
- El proceso de desarrollo del servicio web es independiente al proceso de desarrollo de servicios que usan el servicio web. Gracias a la modularidad que aportan, es más fácil construir servicios de mayor envergadura.

c) Enumera algunos de todos los protocolos y especificaciones existentes en los servicios web.

Los protocolos principales son los siguientes:

- Simple Object Access Protocol (**SOAP**) es un protocolo basado en XML para acceder a servicios web. Está recomendado por el W3C y, al estar basado en XML, es independiente del lenguaje y la plataforma.
- Universal Description, Discovery and Integration (**UDDI**) es un catálogo de negocios de Internet basado en XML cuyo objetivo es ser accedido por los mensajes SOAP y dar paso a documentos WSDL.
- Otros menos usados son **BEEP**, **Hessian**, **WPS** o **JSON-RPC.**

Las especificaciones principales son las siguientes:

- De normas de servicios web: IBM Developerworks
- XML: XML, XML Namespaces, XSD
- De mensajería: SOAP.
- De intercambio de metadatos: JSON-WSP, UDDI
- De seguridad: WS-Security, XML Signature, SAML
- De privacidad: P3P
- De recursos: Web Services Resource Framework
- De mensajería fiable: WS-ReliableMessaging, WS-Reliability
- De procesos de negocios: WSCL, WSCI
- De transacciones: WS-BusinessActivity, WS-CAF
- De gestión: WS-Management, WSDM
- Orientadas a la presentación: Web Services for Remote Portlets

d) Describe la arquitectura (diagrama) de un servicio web y qué tres roles y tres operaciones intervienen.

El servicio web está estructurado con tres roles y tres acciones para no establecer una sobrecarga de responsabilidades y tener una estructura común, independientemente de las funcionalidades de los servicios.

Los tres roles que intervienen son los siguientes:

- Proveedor: Crea el servicio web y lo hace accesible a las aplicaciones que quieren acceder a él.
- Solicitante: Es la aplicación que intenta contactar con el servicio web.
- Registrador (también llamado bróker): Es la aplicación que ofrece acceso al UDDI (Universal Description Discovery and Integration), el cual permite a la aplicación localizar el servicio web al que intenta acceder.

Las tres operaciones que se realizan entre los roles del servicio web son las siguientes:

- Publicar: El proveedor informa al registrador/bróker de la existencia del servicio web usando la interfaz de publicación del registrador/bróker para hacer al servicio web accesible a otros servicios/aplicaciones.
- Encontrar: El solicitante consulta al registrador/bróker para localizar a un servicio web publicado.
- Enlazar: Con la información obtenida del registrador/bróker sobre el servicio web, el solicitante puede enlazar o invocar al servicio web.

e) Indica los dos tipos de servicios web más populares en la actualidad

---

1. El tipo de servicio web **SOAP** consiste en un protocolo de mensajería independiente del transporte. Se encarga de transferir XMLs como mensajes SOAP. Cada mensaje es un documento XML. La información se transfiere a través del protocolo HTTP.
2. El tipo de servicio web **RESTful** es un tipo de arquitectura de software mantenible y escalable basada en la arquitectura REST. Fue creado para guiar el diseño y el desarrollo de la estructura de la World Wide Web.

SOAP

a) Lee, investiga y explica con tus propias palabras qué es el protocolo de mensajes SOAP (diagrama).

Simple Object Access Protocol (SOAP), como bien indica el enunciado, es un protocolo de mensajes para el intercambio de información estructurada en la implementación de servicios web a través de redes. Usa el formato XML para los mensajes, los cuales son transferidos a través del protocolo HTTP.

Dentro de la arquitectura de software de los servicios web, SOAP interpreta el rol de registrador/bróker. Es decir, es el intermediario entre el solicitante y el registrador. Realiza el proceso de encontrar el servicio web publicado por el proveedor (registrador) que una aplicación/servicio (solicitante) solicita.

b) Indica qué ventajas e inconvenientes presenta SOAP y cuál es la estructura de un mensaje SOAP.

Las principales ventajas de SOAP son las siguientes:

- SOAP es un protocolo neutro, por lo que permite que se use cualquier protocolo de transporte. El más común es HTTP, pero también se puede usar SMTP y JMS.
- SOAP, combinado con el protocolo HTTP, permite una comunicación a través de firewall y proxies existentes, por lo que no requiere que se modifiquen las comunicaciones existentes para procesar el texto recibido mediante HTTP.
- SOAP dispone del acceso a todas las herramientas XML, como los namespaces y la internacionalización del texto intercambiado.

Las principales desventajas de SOAP son las siguientes:

- Cuando se usa el enlazamiento SOAP - HTTP por defecto, el modelo de datos abstracto de XML se serializa como XML. Para mejorar el rendimiento de los XMLs con binario embebido, se introdujo el mecanismo de optimización de transmisión de mensajes.
- Cuando se usa HTTP como protocolo de transporte y no se usan WS-Addressing (Web Services Addressing) o ESB (Enterprise Service Bus, que consiste en un bus de servicios empresariales que implementa un sistema de comunicación entre aplicaciones de software que interactúan entre sí en una arquitectura orientada a servicios), los roles de las aplicaciones/servicios y el servicio web son fijos. Solo las aplicaciones/servicios pueden usar los servicios del servidor web.
- SOAP no puede aprovechar las características y optimizaciones específicas del protocolo, como la interfaz uniforme de REST o el almacenamiento en caché, sino que tiene que reimplementarlas (como ocurre con WS-Addressing).
- La cantidad de texto que se necesita para construir un XML y su lenta velocidad de parseo, junto con la falta de la estandarización de un modelo de interacción, son las principales causas que llevaron a SOAP ser reemplazado por servicios como REST.

Un mensaje SOAP se compone de las siguientes partes:

- Envelope: *Obligatorio*. Identifica el documento XML como un mensaje SOAP.
- Header: Contiene la información del header.
- Body: *Obligatorio*. Contiene información de la llamada y la request.
- Fault: Contiene información de los errores durante el procesamiento de los mensajes.

c) Muestra un ejemplo de petición SOAP así como una respuesta del servidor a dicha petición.

Petición:

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPrice>
    <m:StockName>IBM</m:StockName>
  </m:GetStockPrice>
</soap:Body>

</soap:Envelope>
```

Respuesta:

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPriceResponse>
    <m:Price>34.5</m:Price>
  </m:GetStockPriceResponse>
</soap:Body>

</soap:Envelope>
```

d) Explica qué es WSDL y UDDI y explica detalladamente el funcionamiento de un servicio web SOAP.

Web Services Description Language (WSDL) está escrito en XML y sirve para describir la funcionalidad ofrecida por los servicios web.

Universal Description, Discovery and Integration (UDDI) es un directorio para almacenar información sobre servicios web. Está descrito por WSDL y se comunica vía SOAP.

Ambos forman parte de SOAP y, por extensión, WSDL y UDDI también están recomendados por el W3.

El funcionamiento de SOAP es sencillo. El proveedor del servicio envía un archivo WSDL a UDDI. La aplicación/servicio solicitante contacta a UDDI para averiguar quién es el proveedor de los datos que está solicitando. Una vez que UDDI encuentra al proveedor, UDDI conecta al solicitante y al proveedor mediante el protocolo SOAP. El

El proveedor del servicio valida la petición de servicio y envía los datos estructurados en un archivo XML, el cual es validado por el solicitante mediante el uso de un archivo XSD.

REST y RESTful

a) Lee, investiga y explica el estilo de arquitectura de software para sistemas hipermedia REST.

Representational State Transfer (REST) es una arquitectura de software creada con el fin de establecer un flujo de diseño y el desarrollo para los sistemas hipermedia distribuidos por la World Wide Web.

---

b) Esclarece la diferencia entre REST y RESTful.

REST es un estilo de arquitectura software que usa tecnologías y protocolos existentes de la Web, mientras que RESTful hace referencia a lo servicios web que implementan la arquitectura REST.

c) Explica las seis restricciones de arquitectura REST que tiene que cumplir un servicio web RESTful.

---

1. **Interfaz uniforme (Uniform interface)**: Simplifica y desacopla la arquitectura para poder acceder a cada parte independientemente. Se divide en cuatro restricciones:

    a. Identificación de recursos en peticiones: El concepto de los recursos es dividido conceptualmente de la representación que ve el cliente del recurso. Por ejemplo, el servidor manda información de la base de datos como XML, cuyo formato no se corresponde con la representación interna del servidor (podría ser un archivo, una imagen, un vídeo…)

    b. Manipulación de recursos a través de representaciones: La representación del recurso que le llega al cliente tiene la suficiente información como para poder ser modificada o eliminada.

    c. Mensajes autodescriptivos: Cada mensaje incluye la información necesaria para saber cómo se tiene que procesar dicho mensaje.

    d. Hypermedia as the engine of application state (HATEOAS): Tras haber accedido a una URI inicial para la aplicación REST (como el archivo index.html de un sitio web), el cliente debería de ser capaz de usar de usar URIs generadas dinámicamente para poder acceder a los recursos disponibles. El server responde con texto que incluye hyperlinks a los recursos. No es necesario que el cliente tenga información sobre la estructura o la dinámica de la aplicación

2. **Arquitectura cliente-servidor (Client–server architecture)**: El patrón de diseño cliente-servidor refuerza el principio de separación de responsabilidades. En este caso, la portabilidad de la interfaz del usuario se puede adaptar a cualquier aplicación, independientemente de su estructura de datos. También se facilita la escalabilidad y el desarrollo independiente de ambos componentes.

3. **Sin estado (Statelessness):** Que REST sea *statelessness* implica que sea un protocolo de comunicaciones en el que el proveedor no almacena información de la sesión. En su lugar, la información de la sesión es enviada al proveedor por el solicitante, de forma que cada paquete de datos enviado pueda ser comprendido sin necesidad de paquetes previos de la sesión. Esto conlleva un mayor rendimiento al no retener la información de la sesión de cada usuario.

4. **Cacheabilidad (Cacheability):** Para mejorar la comunicación entre el cliente y el servidor, las respuestas del proveedor debe definir qué información puede cachearse o no, con el fin de eliminar interacciones cliente-servidor innecesarias con el fin de mejorar el rendimiento y la escalabilidad.

5. **Sistema de capas (Layered system):** Consiste en añadir un intermediario en la comunicación entre el cliente y el servidor. Puede ser mediante un proxy o un load balancer (balanceador de carga), los cuales no requieren de código ni actualizaciones necesarias en la parte del servidor ni en la del cliente. Esto permite mejoras en la escalabilidad al añadir cachés compartidas y balanceo de cargas, aparte de añadir una capa de seguridad al dividir la lógica de negocio de la lógica de seguridad.
6. **Código en demanda (Code on demand)** [Opcional]**:** Los servidores pueden suministrar código ejecutable, como Java applets.

d) Señala las diferencias entre REST y SOAP, cuándo es mejor uno que otro y qué retos afrontan sus API.

Estas son las principales diferencias entre REST y SOAP.

- SOAP significa Simple Object Access Protocol mientras que REST significa Representational State Transfer.
- SOAP es un protocolo mientras que REST es un patrón arquitectónico
- SOAP usa interfaces de servicios para prestar sus servicios mientras que REST usa URLs para acceder a sus recursos.
- SOAP requiere de mayor banda ancha para su uso, al contrario que REST.
- SOAP solo opera con archivos XML mientras que REST opera con texto plano, XML, HTML y JSON.
- SOAP no puede hacer uso de REST mientras que REST puede hacer uso de SOAP.

REST opera mejor en las siguientes situaciones:

- Cuando hay recursos y banda ancha limitada.
- Si no hay necesidad de mantener un estado de información de una petición a otra (statelessness). Por ejemplo, una página de ecommerce, por lo general, no podría adaptarse a REST ya que los items añadidos al carrito se transfieren a la página del pago para poder completar el pedido, por lo que se requiere conocer el estado del carrito.
- Para cachear las consultas más frecuentes y hacer la comunicación más óptima.
- Facilitar y agilizar el desarrollo de un servicio web.

SOAP opera mejor en las siguientes situaciones:

- Para entornos que requieren procesamiento asíncrono, alta seguridad y robustez.
- Cuando el servicio web sabe los parámetros que va a recibir, como cuando se manda una petición para comprar un artículo que cuenta con parámetros como el nombre, el precio o la cantidad.
- Cuando se requieren operaciones con información de estado (stateful).

Los retos que afronta REST son los siguientes:

- Falta de seguridad. Su naturaleza pública (lo cual es bueno para diseñar APIs) implica que los datos sensibles pasados entre el cliente y el servidor puedan ser interceptados fácilmente.

- Falta de estado. Muchas páginas requieren de información de estado, lo cual es algo que no ofrece REST. En todo caso, la responsabilidad de mantener el estado recae en el cliente, por lo que la aplicación resulta más pesada y difícil de mantener.

Los retos que afronta SOAP son los siguientes:

- Archivo WSDL. Si recordamos, el archivo WSDL es el que le comunica al cliente todas las operaciones que el servicio web puede realizar. Esto implica que el cliente y el servidor tengan una fuerte dependencia, ya que cualquier cambio realizado a un archivo WSDL debería de estar reflejado también en el código de la aplicación/servicio para conformar dicho cambio y evitar que se produzcan problemas.
- Tamaño del archivo. Debido al uso de archivos XML, la verbosidad de dicho formato conlleva un uso de banda ancha mayor en comparación con REST.

HTTP RESTful API

a) Define qué es una RESTful API

Es la API de un servicio web que acoge las reglas establecidas por el patrón arquitectónico REST. Hay que recordar que el sufijo ‘-ful’ se añade cuando un servicio web conforma las reglas del patrón REST.

---

b) Estudia detenidamente e indica los tres aspectos que definen una RESTful API basada en HTTP.

Las APIs RESTful basadas en el protocolo HTTP son definidas con los siguientes aspectos:

- Una URI base. Ejemplo: http://api.example.com/
- Métodos HTTP (GET, POST, PUT y DELETE).
- El *media type* (tipo de media) que define el estado de transición de los datos. Ejemplo: application/json

---

c) Define qué es una URI y detalla cómo se usan para nombrar recursos en una HTTP RESTful API.

Uniform Resource Identifier (URI) es un identificador de recursos uniforme que sirve para, como su propio nombre indica, identificar un recurso localizado en la red. Se usan también en el mundo real para identificar libros (ISBN) o personas (por ejemplo, DNI).

Ejemplo: https://horizon.mycompany.com/?domainName=finance&userName=fred

HTTP RESTful APIs usan URIs para localizar recursos. Es necesario que los recursos tengan nombres representativos para facilitar el uso de la API.

d) Define HTTP y explica el uso de los métodos HTTP GET, POST, PUT y DELETE en una HTTP RESTful API.

HyperText Transfer Protocol (HTTP) es un protocolo de la capa de aplicación enfocado a la comunicación de texto y archivos en la World Wide Web.

Los principales métodos HTTP son los siguientes:

- **GET**: Devuelve texto o la representación de un recurso. Se dice que es un método seguro ya que no modifica el estado del recurso/texto obtenido. Ejemplo: HTTP GET http://www.example.com/users
- **POST**: Aplicado a REST, crea un recurso dentro de una colección de recursos. Ejemplo: HTTP POST http://www.appdomain.com/users
- **PUT**: Actualiza un recurso existente o crea uno en caso contrario. Ejemplo: HTTP POST http://www.appdomain.com/users
- **DELETE**: Elimina un recurso identificado por su URI. Ejemplo: HTTP POST http://www.appdomain.com/users

---

e) Define *media type*, enumera algunos e indica qué media type se usa en este y en este HTTP RESTful API.

Media type es un identificador para formato de archivos y formato de contenidos transmitidos en Internet dividido en dos partes.

Ejemplo: application/pdf

En la primera imagen el media type es `application/json` mientras que en la segunda imagen el media type es `application/html`.

f) Define JSON, cuáles son sus tipos de datos, qué diferencias hay con XML y pon tres ejemplos de JSON.

JavaScript Object Notation (JSON) es un formato de archivo para el intercambio de datos que usa texto interpretable por humanos que ordena la información por pares de atributo-valor y arrays.

Los tipos de datos de JSON son los siguientes:

- string
- number
- boolean
- null/empty
- object
- array

Las principales diferencias con XML residen en los siguientes puntos:

- Los datos en un JSON tienen un tipo determinado mientras que en XML no.
- Los datos en un JSON son accesibles como objetos JSON, mientras que en XML los datos necesitan ser parseados.
- JSON está soportado por casi todos los navegadores mientras que el parseo *cross-browser* de los XML puede dar problemas.
- JSON es menos seguro que XML.
- JSON no soporta namespaces mientras que XML sí.
- JSON no soporta comentarios mientras que XML sí.
- JSON solo soporta codificación UTF-8 mientras que XML soporta varios formatos de codificación.
- JSON tiene soporte nativo de objetos mientras que los objetos en XML tienen que ser escritos por convención mediante atributos y elementos.
- JSON puede ser usado junto a AJAX mientras que XML no está completamente soportado.
- Partiendo de los mismos datos, JSON requiere de muchos menos caracteres que un XML para formar el archivo completo.

Ejemplos de JSON:

1:

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
```

2:

```
{"widget": {
    "debug": "on",
    "window": {
        "title": "Sample Konfabulator Widget",
       "name": "main_window",
       "width": 500,
       "height": 500
    },
   "image": {
        "src": "Images/Sun.png",
        "name": "sun1",
       "hOffset": 250,
       "vOffset": 250,
       "alignment": "center"
    },
   "text": {
        "data": "Click Here",
        "size": 36,
        "style": "bold",
       "name": "text1",
       "hOffset": 250,
       "vOffset": 100,
       "alignment": "center",
        "onMouseUp": "sun1.opacity = (sun1.opacity / 100) * 90;"
    }
}}
```

3:


```
{"menu": {
    "header": "SVG Viewer",
    "items": [
        {"id": "Open"},
        {"id": "OpenNew", "label": "Open New"},
        null,
        {"id": "ZoomIn", "label": "Zoom In"},
        {"id": "ZoomOut", "label": "Zoom Out"},
        {"id": "OriginalView", "label": "Original View"},
        null,
        {"id": "Quality"},
        {"id": "Pause"},
        {"id": "Mute"},
        null,
        {"id": "Find", "label": "Find..."},
        {"id": "FindAgain", "label": "Find Again"},
        {"id": "Copy"},
        {"id": "CopyAgain", "label": "Copy Again"},
        {"id": "CopySVG", "label": "Copy SVG"},
        {"id": "ViewSVG", "label": "View SVG"},
        {"id": "ViewSource", "label": "View Source"},
        {"id": "SaveAs", "label": "Save As"},
        null,
        {"id": "Help"},
        {"id": "About", "label": "About Adobe CVG Viewer..."}
    ]
}}
```

g) Explica con tus propias palabras y detalladamente los ejemplos de esta tabla para el recurso Order.

---

1. **GET** (http://example.com/api/orders): Devuelve los recursos que derivan del recurso “orders”
2. **GET** (http://example.com/api/orders/123): Devuelve el subrecurso “123” del recurso “orders”.
3. **POST** (http://example.com/api/orders): Crea un subrecurso dentro del recurso “orders”.
4. **PUT** (http://example.com/api/orders/123): Crea o actualiza (dependiendo de si existe o no el recurso) el subrecurso “123” del recurso “orders”.
5. **DELETE** (http://example.com/api/orders/123): Elimina el subrecurso “123” del recurso “orders” y los recursos que penden de él.

---

h) Explica con tus propias palabras y detalladamente todas las combinaciones de URI y métodos HTTP para el recurso User.

/users

1. **GET**: Devuelve el recurso “users”.
2. **POST**: Crea el recurso “users”.
3. **PUT**: Crea o actualiza (dependiendo de si existe o no el recurso) el recurso “users”.
4. **DELETE**: Elimina el recurso “users” y los subrecursos que penden de él.

/users/123

1. **GET**: Devuelve el subrecurso “123” del recurso “users”.
2. **POST**: Crea el subrecurso “123” dentro del recurso “users”.
3. **PUT**: Crea o actualiza (dependiendo de si existe o no) el subrecurso “123” del recurso “users”.
4. **DELETE**: Elimina el subrecurso “123” y los subrecursos que penden de él.

HTTP RESTful API en SERVICIOS WEB

a) Di en qué consiste el servicio web JSONPlaceholder, enumera sus recursos y pon ejemplos de rutas.

Es una API REST gratuita que se usa para hacer pruebas con la obtención/envío de JSONs. Los JSON que esta API devuelve contienen información “falsa” a modo de *placeholder*.

Los recursos con los que cuenta esta API REST son los siguientes:

- /posts (100 posts)
- /comments (500 comments)
- /albums (100 albums)
- /photos (5000 photos)
- /todos (200 todos)
- /users (10 users)

Ejemplos de rutas:

- Ruta al recurso “posts”: `/posts`
- Ruta al subrecurso “1” del recurso “posts”: `/posts/1`
- Ruta al subrecurso “comments” del subrecurso “1” del recurso “posts”: `/posts/1/comments`
- Ruta al subrecurso con id “1” del recurso “comments”: `/comments?=postId=1`
- Ruta al recurso “posts”: `/comments?=postId=1`

b) Explica qué es [Fetch API]() y ejecuta los ejemplos que vienen en la [guía]() del servicio web JSONPlaceholder.

Es una interfaz sencilla para obtener recursos de una API RESTful. La dirección del recurso al que queremos acceder es determinada como parámetro en la llamada a la función fetch().

EJEMPLOS:

Obteniendo un subrecurso de un recurso:

```
> fetch('https://jsonplaceholder.typicode.com/posts/1')
    .then((response) => response.json())
    .then((json) => console.log(json));
<- ▶ Promise {<pending>}
                                                                              VM49:3
   {userId: 1, id: 1, title: 'sunt aut facere repellat provident occaecati excepturi optio reprehen
 ▶ derit', body: 'quia et suscipit\nsuscipit recusandae consequuntur …strum rerum est autem sunt re
   m eveniet architecto'}
```

Listando todos los subrecursos de un recurso:

```
> fetch('https://jsonplaceholder.typicode.com/posts')
    .then((response) => response.json())
    .then((json) => console.log(json));
<- ▶ Promise {<pending>}
                                                                              VM137:3
   (100) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
   {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
 ▶ {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
   {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
   {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
   {…}, {…}, {…}, {…}, {…}, {…}, {…}]
```

Creando un recurso:

```
> fetch('https://jsonplaceholder.typicode.com/posts', {
    method: 'POST',
    body: JSON.stringify({
      title: 'foo',
      body: 'bar',
      userId: 1,
    }),
    headers: {
      'Content-type': 'application/json; charset=UTF-8',
    },
  })
    .then((response) => response.json())
    .then((json) => console.log(json));
<- ▶ Promise {<pending>}
   ▶ {title: 'foo', body: 'bar', userId: 1, id: 101}
```

Actualizando un recurso:

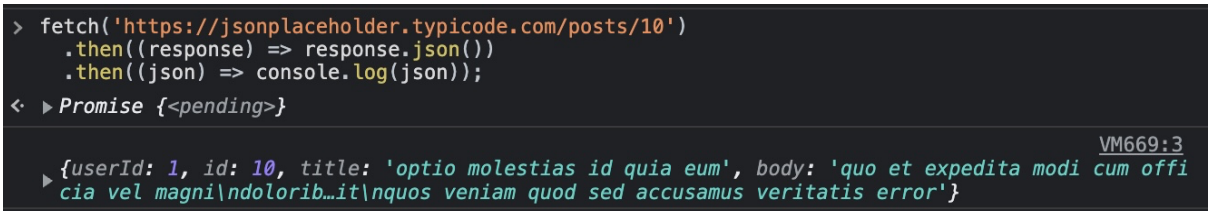
```
> fetch('https://jsonplaceholder.typicode.com/posts/1', {
    method: 'PUT',
    body: JSON.stringify({
      id: 1,
      title: 'foo',
      body: 'bar',
      userId: 1,
    }),
    headers: {
      'Content-type': 'application/json; charset=UTF-8',
    },
  })
    .then((response) => response.json())
    .then((json) => console.log(json));
<· ▶ Promise {<pending>}
  ▶ {id: 1, title: 'foo', body: 'bar', userId: 1}
```

Filtrando recursos:

```
> fetch('https://jsonplaceholder.typicode.com/posts?userId=1')
    .then((response) => response.json())
    .then((json) => console.log(json));
<· ▶ Promise {<pending>}
  ▶ (10) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}]
```

c) Usando Fetch API, haz una petición con método HTTP GET con URI `/posts/10` a JSONPlaceholder.

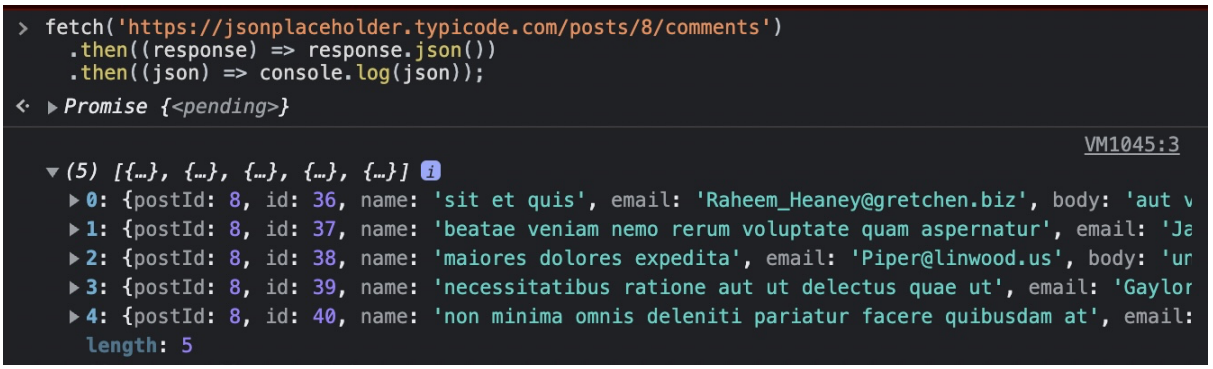
```
fetch('https://jsonplaceholder.typicode.com/posts/10')
  .then((response) => response.json())
  .then((json) => console.log(json));
```



---

d) Usando Fetch API, obtén todos los comentarios de la publicación 8 de JSONPlaceholder.

```
fetch('https://jsonplaceholder.typicode.com/posts/8/comments')
  .then((response) => response.json())
  .then((json) => console.log(json));
```



e) Usando Fetch API, obtén todas las publicaciones del usuario 1 de JSONPlaceholder.

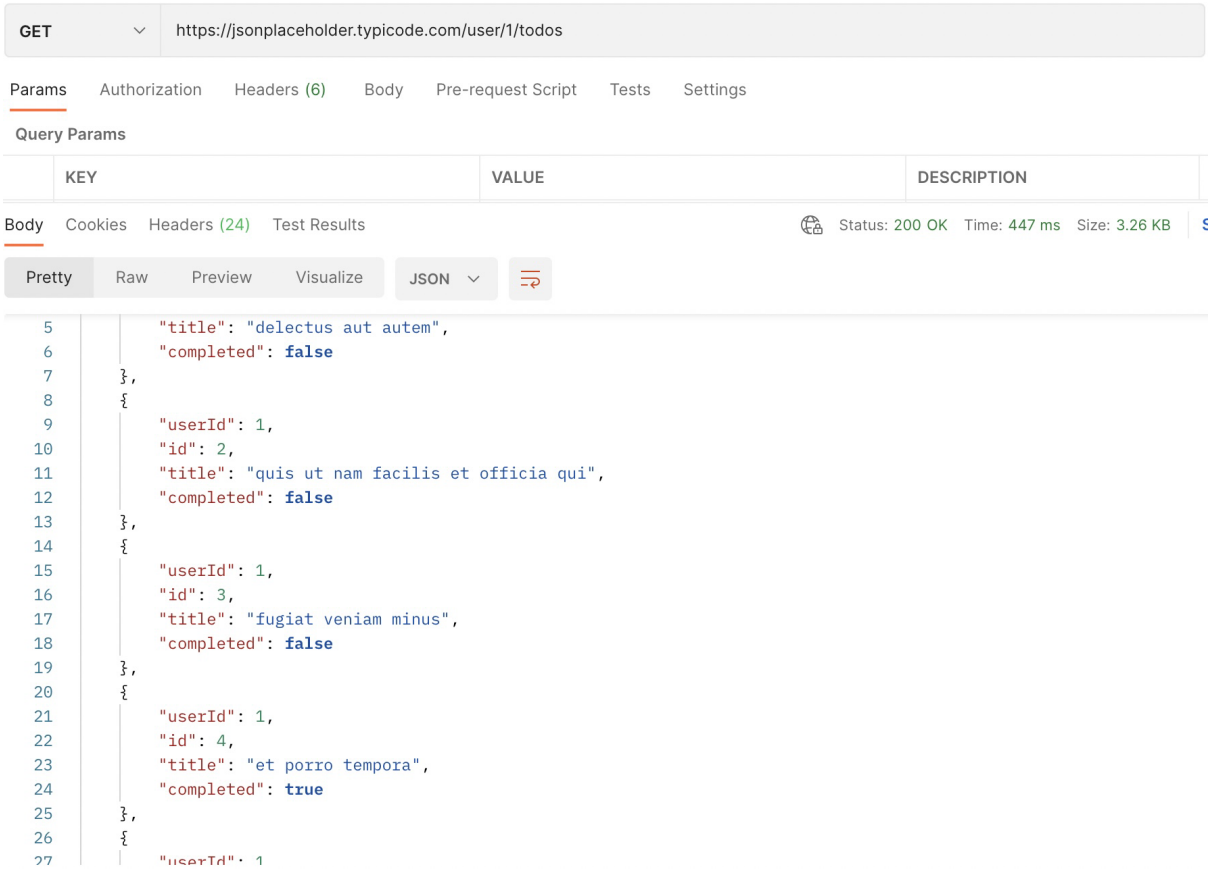
```
fetch('https://jsonplaceholder.typicode.com/user/1/posts')
  .then((response) => response.json())
  .then((json) => console.log(json));
```

f) Busca tres clientes REST online, lístalos y usa uno para obtener todos las TODO's de JSONPlaceholder.

Tres clientes REST online:

- Postman
- Testfully
- Insomnia

He usado Postman ya que es al que más habituado estoy y el que uso en el trabajo.



g) Usando Fetch API o un cliente REST online, obtén todas los `posts` del usuario 1 de JSONPlaceholder.

```
fetch('https://jsonplaceholder.typicode.com/user/1/posts')
  .then((response) => response.json())
  .then((json) => console.log(json));
```

h) Usando Fetch API o un cliente REST online, crea un nuevo comentario a un `post` de JSONPlaceholder.

```
fetch('https://jsonplaceholder.typicode.com/posts/1/comments', {
      method: 'POST',
      body: JSON.stringify({
            postId: 1,
            id: 100,
            name: 'Nombre',
            email: 'email@email.com',
            body: 'Cuerpo'
      }),
      headers: {
            'Content-type': 'application/json; charset=UTF-8',
      },
})
  .then((response) => response.json())
  .then((json) => console.log(json));
```

```
> fetch('https://jsonplaceholder.typicode.com/posts/1/comments', {
      method: 'POST',
      body: JSON.stringify({
  postId: 1,
  id: 100,
  name: 'Nombre',
  email: 'email@email.com',
  body: 'Cuerpo'
  }),
  headers: {
              'Content-type': 'application/json; charset=UTF-8',
  },
  })
    .then((response) => response.json())
    .then((json) => console.log(json));
<- ▶ Promise {<pending>}
                                                          VM3065:15
  ▼ {postId: '1', id: 501, name: 'Nombre', email: 'email@email.com', body: 'Cuerpo'} i
      body: "Cuerpo"
      email: "email@email.com"
      id: 501
      name: "Nombre"
      postId: "1"
```

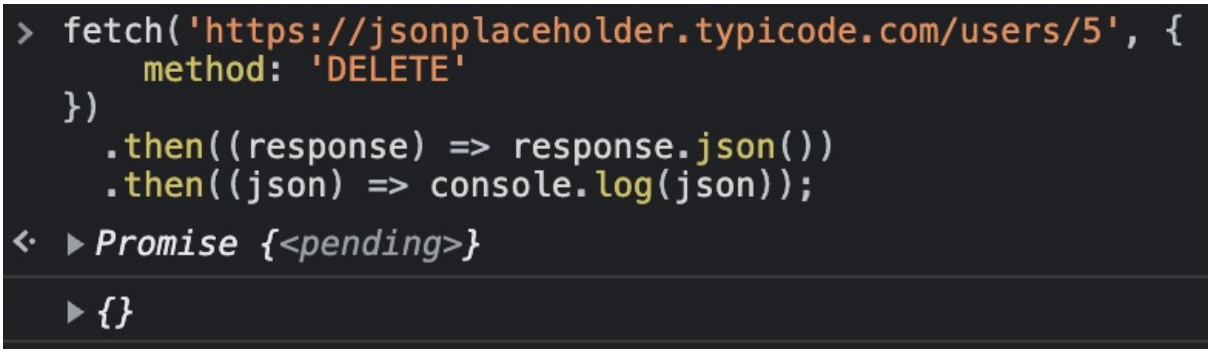
i) Usando Fetch API o un cliente REST online, actualiza o modifica un `post` de JSONPlaceholder.

```
fetch('https://jsonplaceholder.typicode.com/posts/10', {
      method: 'PUT',
      body: JSON.stringify({
            id: 1,
            userId: 2,
            title: 'Soy el título',
            body: 'Soy el cuerpo'
      }),
      headers: {
            'Content-type': 'application/json; charset=UTF-8',
      },
})
  .then((response) => response.json())
  .then((json) => console.log(json));
```

```
fetch('https://jsonplaceholder.typicode.com/posts/10', {
  method: 'PUT',
  body: JSON.stringify({
    id: 1,
    title: 'Soy el título',
    body: 'Soy el cuerpo',
    userId: 2,
  }),
  headers: {
    'Content-type': 'application/json; charset=UTF-8',
  },
})
  .then((response) => response.json())
  .then((json) => console.log(json));
▶ Promise {<pending>}
▶ {id: 10, title: 'Soy el título', body: 'Soy el cuerpo', userId: 2}
```

j) Usando Fetch API o un cliente REST online, borra el usuario 5 de JSONPlaceholder.

```
fetch('https://jsonplaceholder.typicode.com/users/5', {
      method: 'DELETE'
});
```



[ACTIVIDAD EXTRA]

> Realiza de principio a fin este tutorial, haz pruebas con él, explica cómo funciona y documéntalo todo en el PDF servicios_web.pdf.

Como primer paso, es importante definir la estructura del proyecto para dividir la explicación en la responsabilidad de cada paquete y sus respectivos archivos:

```
├── Controllers
│   └── Api
│       ├── BaseController.php
│       └── UserController.php
├── inc
│   ├── bootstrap.php
│   └── config.php
├── index.php
└── Models
    ├── Database.php
    └── User.php
```

Los archivos del directorio *Controllers/Api* son los que gestionan la lógica de la API. Cuando la aplicación reciba una petición, el controlador correspondiente se encargará de gestionarla y procesarla, con la ayuda del modelo que contiene los datos que el cliente está solicitando.

En este caso, la API devolverá usuarios, por lo que las peticiones serán procesadas por el controlador *UserController*, el cual accede al modelo de *User* que, a su vez, contiene la estructura y la información de los usuarios en la base de datos.

Es importante destacar que, atendiendo al paradigma de la programación orientada a objetos y al principio DRY (Don't Repeat Yourself), se crea el controlador *BaseController*, el cual contiene los métodos comunes a cualquier controlador. Esto ofrece también una alta escalabilidad del proyecto, ya que facilita la adición de nuevos controladores y modelos asociados.

Por otra parte, tenemos el directorio *inc*, el cual contiene información y configuración de la aplicación. El archivo *config.php* es muy similar al archivo *.env* de Laravel, el cual contiene constantes que definen credenciales y configuraciones generales de la aplicación. En este caso, utilizaremos el archivo para almacenar las credenciales de la BBDD con el fin de tenerlas en un solo archivo, para evitar tener que refactorizar el proyecto entero en caso de que se haga alguna modificación respectiva al usuario, contraseña, etc.

El archivo *bootstrap.php* realiza el proceso de *bootstrapping,* En este caso, se importa en el archivo *index.php* y su labor es asegurar que todos los recursos externos utilizados son importados al principio de la ejecución del código.

El archivo *index.php* es el punto de entrada de la API. En este caso, actúa de controlador de la petición del usuario y comprueba que los parámetros sean correctos. En caso de que no lo sean, se manda una respuesta HTTP informando al usuario del problema. De lo contrario, procesamos la solicitud del cliente y, si todo va bien, le devolvemos la lista de usuarios.

En el directorio de *Models* se encuentran dos archivos. Uno de ellos es *Database*, el cual contiene el código necesario para hacer consultas CRUD genéricas a la base de datos. Análogamente al caso visto con el controlador base, este modelo servirá para seguir el principio DRY y de aplicar los principios del paradigma de la programación orientada a objetos. Por otra parte, el archivo *User* hereda la clase *Database* para poder hacer operaciones CRUD con el modelo.

Explicada ya la estructura del proyecto, se puede proceder a poner un ejemplo y explicar el flujo de ejecución. Supongamos que el cliente quiere obtener el recurso *index.php/user/list*. Al realizar la petición, el servidor la recibe a través del archivo *index.php*. En caso de que la petición esté bien formulada, el archivo *UserController* es el que se encarga de gestionar la acción. En este caso, se solicita la lista de usuarios, por lo que se obtienen los usuarios de la BBDD, se codifican en formato JSON y se envían al cliente a través del *BaseController*, ya que es una acción común a todos los controladores que se creen en el proyecto.

Con ello, ya debería de aparecer en pantalla un JSON con la información de todos los usuarios del proyecto almacenados en la BBDD.

Hay que tener en cuenta que, atendiendo al proyecto, se contempla que el usuario introduzca una petición *index.php/user/list?limit=1*. En este caso, se limitará a 1 el número de usuarios devueltos por la API. Si no se pasa ese parámetro, se devuelven 10 registros como máximo.


```
{

    [
        {
            "id":1,
            "name":"Bob",
```

```
            "email":"bob@gmail.com",
            "status":0
        }
    ]

}
```