¿Qué es un webservice?

Un **servicio web** (en inglés, *web service* o *web services*) es una tecnología que utiliza un conjunto de protocolos y estándares que sirven para intercambiar datos entre aplicaciones. En las aplicaciones que estamos habituados a usar es el programa el que se comunica con nosotros mediante interfaces de usuario. Los webservices permite que dos aplicaciones se comuniquen entre sí sin ninguna intervención de los usuarios.

Permiten definir estándares en comunicaciones proporcionando funciones públicas. Si yo quiero obtener el tiempo que hace en una determinada ciudad un webservice que proporcione este servicio puede publicar los protocolos de acceso y el formato de los datos que proporciona, haciendo que su uso sea común.

Ocultan el funcionamiento interno del software creando una capa entre los consumidores del servicio y las aplicaciones. Lo que ven las aplicaciones son los servicios proporcionados, no cómo se generan. Esto hace que la implementación no esté accesible y que, además, pueda cambiarse totalmente sin necesidad de modificar los accesos a la misma.

Ejemplos de webservice: SOAP y REST

Aunque existen (y han existido) muchos tipos de webservice los que dominan actualmente el mercado (y no parece que vaya a cambiar) son SOAP y RESTful. Cada uno tiene sus ventajas e inconvenientes. SOAP es más pesado pero más potente y REST es más sencillo, útil para intercambiar información, generalmente en formato JSON.
Ventajas de SOAP respecto a REST:

- Idioma, plataforma y transporte independientes (REST requiere el uso de HTTP)
- Funciona bien en entornos empresariales distribuidos (REST asume la comunicación directa punto a punto)
- Estandarizado
- Proporciona una amplia extensibilidad previa a la construcción en forma de los estándares WS *
- Manejo de errores incorporado
- Automatización cuando se utiliza con ciertos lenguajes.

REST es más fácil de usar en su mayor parte y es más flexible. Tiene las siguientes ventajas en comparación con SOAP:

- Utiliza estándares fáciles de entender como Swagger y OpenAPI Specification 3.0
- Curva de aprendizaje más pequeña
- Eficiente (SOAP usa XML para todos los mensajes, REST generalmente usa formatos de mensajes más pequeños como JSON)
- Rápido (no requiere procesamiento extenso)

- Más cerca de otras tecnologías web en la filosofía del diseño.

XML

XML, siglas en inglés de *eXtensible Markup Language*, traducido como "Lenguaje de Marcado Extensible" o "Lenguaje de Marcas Extensible", es un meta-lenguaje que permite definir lenguajes de marcas desarrollado por el *World Wide Web Consortium* (W3C) utilizado para almacenar datos en forma legible. Proviene del lenguaje SGML y permite definir la gramática de lenguajes específicos (de la misma manera que HTML es a su vez un lenguaje definido por SGML) para estructurar documentos grandes.
XML no ha nacido únicamente para su aplicación en Internet, sino que se propone como un estándar para el intercambio de información estructurada entre diferentes plataformas. Se puede usar en bases de datos, editores de texto, hojas de cálculo y casi cualquier cosa imaginable.
XML es una tecnología sencilla que permite la compatibilidad entre sistemas para compartir la información de una manera segura, fiable y fácil.
En los sistemas antiguos, cuando se trataba de intercambiar información entre aplicaciones, normalmente se pasaban en archivos de texto en los que la información estaba en unas posiciones concretas (por ejemplo, el nombre del cliente entre la posición 0 y la 40, la dirección entre la 41 y la 90…). Esto tiene muchos problemas: por un lado la información no es legible ni decodificable y es muy complicado ampliar o modificar el formato de los datos.
Si utilizamos el formato XML, por el contrario, tenemos un documento más legible, fácilmente interpretable por las aplicaciones y cuyo formato se puede especificar en un documento DTD.
Veamos un ejemplo. Imaginemos que tengo que pasar al banco mis facturas que, por simplificar, tienen un número, una fecha, un nombre de cliente y un nif. Un archivo de texto podría tener el siguiente formato:

```
0012018-12-12Tornillería martínez             B1234567
0022018-12-11Pescadería González              B2389823
0032018-12-10Construcciones Viuda de García MoB8129398
```

A pesar de tener pocos datos ya se ve que el formato no es muy legible, e impone restricciones a los tamaños de los datos. Esto mismo en XML podría quedar como sigue:

```
<envio>
<factura>
      <numero>001</numero>
      <fecha>2018-12-12</fecha>
      <nombre>Tornillería Martínez</nombre>
      <nif>B1234567</nif>
</factura>
<factura>
      <numero>002</numero>
```

```
      <fecha>2018-12-11</fecha>
      <nombre>Pescadería González</nombre>
      <nif>B123B23898234567</nif>
</factura>
<factura>
      <numero>003</numero>
      <fecha>2018-12-10</fecha>
      <nombre>Construcciones Viuda de García Montañez</nombre>
      <nif>B8129398</nif>
</factura>
```

Como puede verse el segundo modelo es más legible, está más organizado, y permitiría con facilidad añadir campos o permitir cambios de tamaño más fácilmente.

XML en PHP

Es fácil manejar cadenas o ficheros XML desde PHP con la librería SimpleXML. Nos proporciona elementos para parsear el XML o incluso para generar uno. Veamos primero como interpretar XML. Para ello tenemos dos funciones:

simplexml_load_string Nos interpreta un xml desde una cadena
simplexml_load_file — Interpreta un fichero XML

Veamos como funcionan:

```php
<?php
$string = <<<XML
<?xml version='1.0'?>
<document>
 <title>¿Cuarenta qué?</title>
 <from>Joe</from>
 <to>Jane</to>
 <body>
  Sé que esa es la respuesta pero, ¿cuál es la pregunta?
 </body>
</document>
XML;

$xml = simplexml_load_string($string);

print_r($xml);
?>
```

```php
if (file_exists('test.xml')) {
    $xml = simplexml_load_file('test.xml');

    print_r($xml);
} else {
    exit('Error abriendo test.xml.');
}
```

En caso de que el xml tenga errores nos saldrán por pantalla. Si queremos evitarlo para gestionarlos nosotros después debemos usar :

libxml_use_internal_errors — Deshabilita errores libxml y permite al usuario extraer información de errores según sea necesario

De esta manera:

```php
libxml_use_internal_errors(true);

// carga el documento
$xml = simplexml_load_string($myXMLData);
if ($xml === false) {
        echo "Error XML: ";
        foreach(libxml_get_errors() as $error) {
        echo "<br>", $error->message;
        }
} else {
        print_r($xml);
}
```

Al aplicar estas funciones tendremos un objeto con tantos elementos como hijos. Veamos un ejemplo un poco más complejo.
Si yo tengo el siguiente XML:

```
$myXMLData =
"<?xml version='1.0' encoding='UTF-8'?>
<empresa>
<empleado>
        <nombre>Juan</nombre>
        <departamento>Ventas</departamento>
        <departamento>Marketing</departamento>
        <docs>
        <informe>AAAA</informe>
        <contrato>BBBB</contrato>
        </docs>
</empleado>
<empleado>
        <nombre>Ana</nombre>
```

```
        <departamento>Contabilidad</departamento>
        <docs>
        <informe>CCCC</informe>
        <contrato>DDDD</contrato>
        <curriculum>EEE</curriculum>
        </docs>
</empleado>
</empresa>";
```

Obtendré el siguiente objeto:

```
SimpleXMLElement Object
(
   [empleado] => Array
      (
         [0] => SimpleXMLElement Object
            (
               [nombre] => Juan
               [departamento] => Array
                  (
                     [0] => Ventas
                     [1] => Marketing
                  )

               [docs] => SimpleXMLElement Object
                  (
                     [informe] => AAAA
                     [contrato] => BBBB
                  )

            )

         [1] => SimpleXMLElement Object
            (
               [nombre] => Ana
               [departamento] => Contabilidad
               [docs] => SimpleXMLElement Object
                  (
                     [informe] => CCCC
                     [contrato] => DDDD
                     [curriculum] => EEE
                  )

            )

      )
```

)

Cada ‘nodo’ es un objeto del tipo SimpleXMLElementObject, lo que permite obtener acceso a los elementos e incluso añadir nuevos.

children() Devuelve los hijos de un nodo
count() Devuelve el número de hijos de un nodo
addChild() Añade un hijo a un nodo
Ejemplo:

```
$a = new SimpleXMLElement("<?xml version='1.0' encoding='UTF-8'?><raiz/>");
$a->addChild("nombre","juan");
$a->addChild("nif","1234");
//El objeto
print_r($a);
//Como XML
print_r($a->asXML());
```

```
SimpleXMLElement Object
(
   [nombre] => juan
   [nif] => 1234
)
<?xml version="1.0" encoding="UTF-8"?>
<raiz><nombre>juan</nombre><nif>1234</nif></raiz>
```

SOAP

SOAP (originalmente las siglas de *Simple Object Access Protocol*) es un protocolo estándar que define cómo dos objetos en diferentes procesos pueden comunicarse por medio de intercambio de datos XML.

Características:

Protocolo de web services para intercambiar datos mediante XML.
Un mensaje es un documento XML compuesto de Envelope, header, body y fault.
Utiliza WDSL para especificar las funciones disponibles.
Es más lento que otros protocolos (REST) pero adecuado para grandes aplicaciones.

Soap forma parte del núcleo del PHP, pero hay que activar la extensión. Se debe editar el archivo php.ini y descomentar la siguiente línea:

```
extension=php_soap.dll
```

Estructura de un mensaje SOAP

Veamos como es la estructura básica del protocolo y la correspondiente explicación:

```
<?xml version=”1.0″?>
<soap:Envelope xmlns:soap=”http://www.w3.org/2001/12/soap-envelope”
Soap:encodingStyle=”http://www.w3.org/2001/12/soap-encoding”>
<soap:Header>
…
</soap:Header>
<soap:Body>
…
<soap:Fault>
…
</soap:Fault>
</soap:Body>
</soap:Envelope>
```

Explicación del código anterior:

<?xml version=”1.0″?>

Como podemos ver en esta linea SOAP es un documento XML, y como tal, debe comenzar con el tag <?xml….?> y la versión correspondiente.

<soap:Envelope

Aquí se indica que comienza el envelope (sobre) del mensaje

xmlns:soap = “http://www.w3.org/2001/12/soap-envelope”

Un mensaje SOAP debe contener siempre un elemento envelope asociado con el namespace (espacio de nombres)http://www.w3.org/2001/12/soap-envelope

Soap:encodingStyle=”http://www.w3.org/2001/12/soap-encoding”>

En esta línea lo que se hace es indicar donde se encuentran definidos los tipos de datos utilizados en el documento.

<soap:Header>

Esta línea indica el comienzo del Header (encabezado). En esta sección se incluye información específica del mensaje, como puede ser la autenticación. Es opcional.

</soap:Header>

Como todo documento XML los tags que son abiertos deben ser cerrados, esta línea indica la finalización del Header(encabezado).

<soap:Body>

Aquí comienza el cuerpo del mensaje, en esta sección se incorpora toda la información necesaria para el nodo final. Esta sección es obligatoria, siempre debe estar.
Por ejemplo, los parámetros para la ejecución, o la respuesta a una petición.

<soap:Fault>

Cualquier tipo de fallo que se produzca será notificado en esta sección. La cual esta contenida dentro del cuerpo del mensaje.

</soap:Fault>

Cierre de la sección Fault.

</soap:Body>

Indica el final del cuerpo del mensaje.

</soap:Envelope>
Fin del mensaje SOAP.

Consumir servicios SOAP

Para poder consumir un servicio web utilizamos el objeto SoapClient. Necesitamos saber los parámetros del servicio web y los métodos a los que podemos acceder. Esto lo hacemos con el WDSL (definición de servicios). Obtenemos un objeto que encapsula el xml de la respuesta.

Veamos un ejemplo. El siguiente servicio nos proporciona la dirección de una ip determinada. Necesitamos saber la url de la definición de servicios, que es la siguiente:

http://ws.cdyne.com/ip2geo/ip2geo.asmx?wsdl

SI la abrimos en el navegador vemos que requiere dos parámetros:

<s:element minOccurs="0" maxOccurs="1" name="ipAddress" type="s:string"/>
<s:element minOccurs="0" maxOccurs="1" name="licenseKey" type="s:string"/>

Los tendremos que pasar al crear el cliente SOAP. Todo junto quedaría así:

```
$client = new SoapClient('http://ws.cdyne.com/ip2geo/ip2geo.asmx?wsdl');

$param = array(
 'ipAddress' => '95.23.148.203',
 'licenseKey' => '0',
);

$result = $client->ResolveIP($param);

print_r($result);
echo $result->ResolveIPResult->City;
```

$result es como sigue:

```
stdClass Object
(
   [ResolveIPResult] => stdClass Object
      (
         [City] => Mataró
         [StateProvince] => 56
         [Country] => Spain
         [Organization] =>
         [Latitude] => 41.5421
         [Longitude] => 2.444504
         [AreaCode] => 0
         [TimeZone] =>
         [HasDaylightSavings] =>
         [Certainty] => 90
         [RegionName] =>
         [CountryCode] => ES
      )

)
```

Podemos ver los métodos que tiene con las siguientes funciones:

```
var_dump($client->__getFunctions());
var_dump($client->__getTypes());
```

Más ejemplos:

```
$client = new
SoapClient('http://www.dataaccess.com/webservicesserver/numberconversion.wso?WSDL');
```

```
$param = array(
 'ubiNum' => '5293',
);

$result = $client->NumberToWords($param);

print_r($result);

$client = new SoapClient('http://www.dneonline.com/calculator.asmx?WSDL');

$param = array(
 'intA' => '5293',
 'intB' => '5293',
);

$result = $client->Add($param);

print_r($result);
```

JSON

JSON (JavaScript Object Notation - Notación de Objetos de JavaScript) es un formato ligero de intercambio de datos. Leerlo y escribirlo es simple para humanos, mientras que para las máquinas es simple interpretarlo y generarlo. Está basado en un subconjunto del Lenguaje de Programación JavaScript, Standard ECMA-262 3rd Edition - Diciembre 1999. JSON es un formato de texto que es completamente independiente del lenguaje pero utiliza convenciones que son ampliamente conocidos por los programadores de la familia de lenguajes C, incluyendo C, C++, C#, Java, JavaScript, Perl, Python, y muchos otros. Estas propiedades hacen que JSON sea un lenguaje ideal para el intercambio de datos.
JSON está constituído por dos estructuras:

- Una colección de pares de nombre/valor. En varios lenguajes esto es conocido como un *objeto*, registro, estructura, diccionario, tabla hash, lista de claves o un arreglo asociativo.
- Una lista ordenada de valores. En la mayoría de los lenguajes, esto se implementa como arreglos, vectores, listas o sequencias.

Estas son estructuras universales; virtualmente todos los lenguajes de programación las soportan de una forma u otra. Es razonable que un formato de intercambio de datos que es independiente del lenguaje de programación se base en estas estructuras.

Ejemplo:

```
var jason = {
        "edad" : "24",
```

```
        "ciudad" : "Barcelona",
        "nombre" : "Ana"
};
```

Un JSON también puede incluir arrays:

```
var jason = [{
        "edad" : "24",
        "ciudad" : "Barcelona",
        "nombre" : "Ana"
},
{
        "edad" : "34",
        "ciudad" : "Barcelona",
        "nombre" : "Juan"
}];
```

En PHP para crear un JSOn tenemos la instrucción json_encode:

```
$myObj->name = "Ana";
$myObj->age = 30;
$myObj->city = "Barcelona";

$myJSON = json_encode($myObj);

echo $myJSON;
```

Y para decodificar json_decode:

```
$cadena='{"name":"Juan","age":30,"city":"Barcelona"}';

$obj=json_decode($cadena);

print_r($obj);
```

Imprime:

```
stdClass Object ( [name] => Juan [age] => 30 [city] => Barcelona )
```

REST

REST, o Representación Estatal de Transferencia, es un estilo arquitectónico para proporcionar estándares entre los sistemas informáticos en la web, lo que facilita la comunicación entre los sistemas. Los sistemas compatibles con REST, a menudo llamados

sistemas RESTful, se caracterizan por no tener estado y separan las arquitecturas del cliente y el servidor. Veremos qué significan estos términos y por qué son características beneficiosas para los servicios en la Web.

Separación de cliente y servidor

En el estilo arquitectónico de REST, la implementación del cliente y la implementación del servidor se pueden realizar de manera independiente sin que cada uno conozca al otro. Esto significa que el código en el lado del cliente se puede cambiar en cualquier momento sin afectar la operación del servidor, y el código en el lado del servidor se puede cambiar sin afectar la operación del cliente.

Mientras cada lado sepa qué formato de mensajes enviar al otro, se pueden mantener modulares y separados. Al separar las preocupaciones de la interfaz de usuario de las preocupaciones de almacenamiento de datos, mejoramos la flexibilidad de la interfaz en todas las plataformas y mejoramos la escalabilidad simplificando los componentes del servidor. Además, la separación permite a cada componente la capacidad de evolucionar de forma independiente.

Al utilizar una interfaz REST, diferentes clientes llegan a los mismos puntos finales REST, realizan las mismas acciones y reciben las mismas respuestas.

En la actualidad no existe proyecto o aplicación que no disponga de una API REST para la creación de servicios profesionales a partir de ese software. Twitter, YouTube, los sistemas de identificación con Facebook… hay cientos de empresas que generan negocio gracias a REST y las APIs REST. Sin ellas, todo el crecimiento en horizontal sería prácticamente imposible. Esto es así porque REST es el estándar más lógico, eficiente y habitual en la creación de APIs para servicios de Internet.

Características de REST

- **Protocolo cliente/servidor sin estado**: cada petición HTTP contiene toda la información necesaria para ejecutarla, lo que permite que ni cliente ni servidor necesiten recordar ningún estado previo para satisfacerla. Aunque esto es así, algunas aplicaciones HTTP incorporan memoria caché. Se configura lo que se conoce como **protocolo cliente-caché-servidor sin estado**: existe la posibilidad de definir algunas respuestas a peticiones HTTP concretas como cacheables, con el objetivo de que el cliente pueda ejecutar en un futuro **la misma respuesta para peticiones idénticas**. De todas formas, que exista la posibilidad no significa que sea lo más recomendable.
- Las operaciones más importantes relacionadas con los datos en cualquier sistema REST y la especificación HTTP son cuatro: **POST** (crear), **GET** (leer y consultar), **PUT** (editar) y **DELETE** (eliminar).
- **Los objetos en REST siempre se manipulan a partir de la URI**. Es la URI y ningún otro elemento el identificador único de cada recurso de ese sistema REST. La URI

nos facilita acceder a la información para su modificación o borrado, o, por ejemplo, para compartir su ubicación exacta con terceros.

- **Interfaz uniforme**: para la transferencia de datos en un sistema REST, este aplica acciones concretas (POST, GET, PUT y DELETE) sobre los recursos, siempre y cuando estén identificados con una URI. Esto facilita la existencia de una interfaz uniforme que sistematiza el proceso con la información.
- **Sistema de capas**: arquitectura jerárquica entre los componentes. Cada una de estas capas lleva a cabo una funcionalidad dentro del sistema REST.
- **Uso de hipermedios**: hipermedia es un término acuñado por **Ted Nelson** en 1965 y que es una extensión del concepto de hipertexto. Ese concepto llevado al desarrollo de páginas web es lo que permite que el usuario puede navegar por el conjunto de objetos a través de enlaces HTML. En el caso de una API REST, el concepto de hipermedia explica la capacidad de una interfaz de desarrollo de aplicaciones de proporcionar al cliente y al usuario los enlaces adecuados para ejecutar acciones concretas sobre los datos.

¿Cómo se usa REST?

Veamos los verbos (POST, GET, PUT y DELETE) y un ejemplo de lo que significan.

Supongamos que tenemos un servicio web RESTful definido en la ubicación. http://demo.guru99.com/employee. Cuando el cliente realiza una solicitud a este servicio web, puede especificar cualquiera de los verbos HTTP normales de GET, POST, DELETE y PUT. A continuación se muestra lo que sucedería si los respectivos verbos fueran enviados por el cliente.

POST: se usaría para crear un nuevo empleado utilizando el servicio web RESTful
GET: se usaría para obtener una lista de todos los empleados que utilizan el servicio web RESTful
PUT: se usaría para actualizar a todos los empleados que utilizan el servicio web RESTful
ELIMINAR: se usaría para eliminar a todos los empleados que utilizan el servicio web RESTful

Echemos un vistazo desde la perspectiva de un solo registro. Digamos que hubo un registro de empleado con el número de empleado de 1.

Las siguientes acciones tendrían sus respectivos significados.

POST: esto no sería aplicable ya que estamos obteniendo datos del empleado 1 que ya está creado.
GET: se usaría para obtener los detalles del empleado con Employee no como 1 utilizando el servicio web RESTful
PUT - Esto se usaría para actualizar los detalles del empleado con Employee no como 1 usando el servicio web RESTful

ELIMINAR: se utiliza para eliminar los detalles del empleado con el número de empleado como 1

Consumir servicios REST

Al ser los servicios rest urls a las que se accede con diferentes verbos http, en PHP lo único que tenemos que hacer es realizar llamadas a esas direcciones con el verbo correcto, enviar los datos en formato JSON si es necesario y convertir los datos obtenidos a objetos PHP.

Un ejemplo muy sencillo. En la url https://blockchain.info/ticker nos devuelven un JSON con cotizaciones de criptodivisas. Como sólo tenemos que recuperar información, basta llamar a la url y decodificar los datos:

```
$datos= file_get_contents("https://blockchain.info/ticker");

$obj= json_decode($datos);

print_r($obj);
```

Devuelve:

```
stdClass Object
(
    [USD] => stdClass Object
        (
            [15m] => 3802.63
            [last] => 3802.63
            [buy] => 3802.63
            [sell] => 3802.63
            [symbol] => $
        )

    [AUD] => stdClass Object
        (
            [15m] => 5297.79
            [last] => 5297.79
            [buy] => 5297.79
            [sell] => 5297.79
            [symbol] => $
        )

    [BRL] => stdClass Object
        (
            [15m] => 13996.24
            [last] => 13996.24
```

```
        [buy] => 13996.24
        [sell] => 13996.24
        [symbol] => R$
    )
…
```

Para ver como funciona vamos a usar esta url: https://reqres.in/ que nos proporciona una api fake para probar nuestro código. En el caso de peticiones get no tenemos que hacer nada especial, basta con abrir directamente la url. Ejemplos:
Obtener el usuario 2:

```
$datos= file_get_contents("https://reqres.in/api/users/2");

$obj= json_decode($datos);
```

O usuarios paginados:

```
$datos= file_get_contents("https://reqres.in/api/users");

$obj= json_decode($datos);
```

Si necesitamos lanzar otros verbos que no sean get, tenemos que utilizar CURL:

```
//Inicializar curl
$ch = curl_init();
//Url petición
curl_setopt($ch, CURLOPT_URL, "https://reqres.in/api/users");
//Indicamos que la respuesta se devuelva en la variable y no se mande al navegador
curl_setopt($ch, CURLOPT_RETURNTRANSFER,true);
//El verbo será POST
curl_setopt($ch, CURLOPT_POST, 1);
//LOs campos a enviar
curl_setopt($ch, CURLOPT_POSTFIELDS, '{"name": "Ana", "job": "Developer"}');
$result = curl_exec($ch);

$obj= json_decode($result);
```

SI queremos actualizar usaremos el verbo PUT:

```
$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, "https://reqres.in/api/users/2");
curl_setopt($ch, CURLOPT_RETURNTRANSFER,true);
curl_setopt($ch, CURLOPT_PUT, 1);
curl_setopt($ch, CURLOPT_POSTFIELDS, '{"name": "Ana", "job": "Developer"}');
$result = curl_exec($ch);
```

```
$obj= json_decode($result);
```

Para borrar:

```
$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, "https://reqres.in/api/users/2");
curl_setopt($ch, CURLOPT_RETURNTRANSFER,true);
curl_setopt($ch,CURLOPT_CUSTOMREQUEST,"DELETE");

$result = curl_exec($ch);
```

Implementar servicios REST en PHP

Hay muchas maneras de implementar un servicio REST en PHP, utilizando librerías, frameworks e incluso desde cero, que es lo que vamos a ver en este ejemplo.
Nosotros desde nuestra página PHP podemos saber qué verbo nos han lanzado utilizando lo siguiente:

```
$_SERVER['REQUEST_METHOD']
```

Podemos acceder a la base de datos utilizando las clases definidas en lecciones anteriores.

Podemos acceder a los datos que nos envían utilizando filter_input para get y post, o la instrucción siguiente para obtener el cuerpo en bruto (raw) de lo que nos han enviado:

```
file_get_contents("php://input")
```

EL programa, entonces, sólo tiene que averiguar qué es lo que se está pidiendo, qué verbo se ha enviado, los datos que se envían y realizar la acción necesaria.

Recordemos los verbos típicos de una petición RESTful:

GET -> Para obtener elementos, bien todos o individuales
POST -> Para añadir un elemento a la base de datos
PUT -> Para modificar un elemento de la base de datos
DELETE -> Para eliminar un elemento.

Normalmente se utiliza algún tipo de formato en la URL. Un esquema típico puede ser:

```
dominio/controlador/[id]
```

Esto lo gestionaríamos en el .htaccess con redirecciones.

Ejemplo: Restful Alumno

Primero necesitamos la clase tabla y la clase Alumno que hemos creado en versiones anteriores, para conectarnos a la base de datos. En la clase Alumno vamos a crear una función loadAll para cargar todos los alumnos de la base de datos:

```
 function loadAll(){
        $alumnos=$this->getAll();
        return $alumnos;
  }
```

Vamos a crear un par de clases para dar las respuestas. Una clase HTTP que nos escriba las cabeceras:

```
class HTTP {

        private $httpVersion = "HTTP/1.1";

        public function setHttpHeaders($statusCode,$response="") {

        $statusMessage = $this->getHttpStatusMessage($statusCode);

        header($this->httpVersion . " " . $statusCode . " " . $statusMessage);
        echo $response;
        }

        public function getHttpStatusMessage($statusCode) {
        $httpStatus = array(
        100 => 'Continue',
        101 => 'Switching Protocols',
        200 => 'OK',
        201 => 'Created',
        202 => 'Accepted',
        203 => 'Non-Authoritative Information',
        204 => 'No Content',
        205 => 'Reset Content',
        206 => 'Partial Content',
        300 => 'Multiple Choices',
        301 => 'Moved Permanently',
        302 => 'Found',
        303 => 'See Other',
        304 => 'Not Modified',
        305 => 'Use Proxy',
        306 => '(Unused)',
```

```
        307 => 'Temporary Redirect',
        400 => 'Bad Request',
        401 => 'Unauthorized',
        402 => 'Payment Required',
        403 => 'Forbidden',
        404 => 'Not Found',
        405 => 'Method Not Allowed',
        406 => 'Not Acceptable',
        407 => 'Proxy Authentication Required',
        408 => 'Request Timeout',
        409 => 'Conflict',
        410 => 'Gone',
        411 => 'Length Required',
        412 => 'Precondition Failed',
        413 => 'Request Entity Too Large',
        414 => 'Request-URI Too Long',
        415 => 'Unsupported Media Type',
        416 => 'Requested Range Not Satisfiable',
        417 => 'Expectation Failed',
        500 => 'Internal Server Error',
        501 => 'Not Implemented',
        502 => 'Bad Gateway',
        503 => 'Service Unavailable',
        504 => 'Gateway Timeout',
        505 => 'HTTP Version Not Supported');
        return ($httpStatus[$statusCode]) ? $httpStatus[$statusCode] : $status[500];
        }

}
```

Y una clase respuesta para devolver los datos:

```
class Response {

        public $message;
        public $data;

        function __construct($message, $data = "") {
        $this->message = $message;
        $this->data = $data;
        }

        function __toString() {
        return json_encode($this);
        }
```

}

Vamos a crear una página index.php donde manejaremos todas las peticiones. Le pasaremos dos parámetros: controller para indicar el controlador (alumno, centro,...) y un id opcional si tenemos que recuperar/modificar un usuario. Una url típica será como sigue:

dominio/index.php?controller=alumno&id=1

Si queremos que la url sea más amigable tipo:

dominio/alumno/1

Deberemos redireccionar en el .htaccess, algo como lo siguiente:

```
RewriteEngine On
RewriteRule ^(.*)/([0-9]*)$ index.php?controller=$1&id=$2
```

Pero en el ejemplo no lo vamos a hacer.

Vemos el código del index.php

Lo primero es obtener el controlador y el id. Si el controlador está vacío, o no existe, devolvemos un badrequest:

```
$controller= filter_input(INPUT_GET, "controller");
$id= filter_input(INPUT_GET, "id");
$verb=$_SERVER['REQUEST_METHOD'];
$http = new HTTP();

if (empty($controller) || !file_exists($controller.".php")){
        $http=new HTTP();
        $http->setHttpHeaders(400,new Response("Bad request"));
        die();
}
```

Una vez tenemos un controlador válido creamos un objeto:

```
require $controller . ".php";

$objeto = new $controller;
```

Y ya podemos empezar con los verbos. El GET es sencillo, si no tenemos id cargamos todos los elementos. Si tenemos un id, sólo el mismo.

```
if ($verb == "GET") {
        if (empty($id)) {
        $datos = $objeto->loadAll();
        $http->setHttpHeaders(200, new Response("Lista $controller",$datos));
        } else {
        $objeto->load($id);

        $http->setHttpHeaders(200, new Response("Lista $controller",$objeto->serialize()));
        }
}
```

He creado una función serialize en el objeto que no es más que la función valores:

```
 function serialize() {
        return $this->valores();
  }
```

Si lanzamos la URL obtendremos los valores:

http://localhost/test/index.php?controller=alumno

{"message":"Lista alumno","data":[{"idalumno":"1","nombre":"Juan","mail":"pepe@pepe.com"},{"idalumno":"2","nombre":"asd","mail":"asd"},{"idalumno":"3","nombre":"asd","mail":"pepe@pepe.com"},{"idalumno":"4","nombre":"asd","mail":"pepe@pepe.com"},{"idalumno":"5","nombre":"Juan","mail":"pepe@pepe.com"},{"idalumno":"6","nombre":"Juan","mail":"pepe@pepe.com"},{"idalumno":"7","nombre":"Juan","mail":"pepe@pepe.com"},{"idalumno":"8","nombre":"Ana","mail":"pwpw@ww.com"},{"idalumno":"10","nombre":"Ana","mail":"pwpw@ww.com"}]}

http://localhost/test/index.php?controller=alumno&id=1

{"message":"Lista alumno","data":{"idalumno":"1","nombre":"Juan","mail":"pepe@pepe.com"}}

Para añadir usaremos el verbo POST:

```
if ($verb == "POST") {
        $raw=file_get_contents("php://input");
        $datos=json_decode($raw);
        foreach($datos as $c=>$v){
        $objeto->$c=$v;
        }
        $objeto->save();
}
```

Para modificar el PUT:

```
if ($verb == "PUT") {
        if (empty($id)) {
        $http->setHttpHeaders(400, new Response("Bad request"));
```

```
        die();
        }
        $objeto->load($id);
        $raw = file_get_contents("php://input");
        $datos = json_decode($raw);
        foreach ($datos as $c => $v) {
        $objeto->$c = $v;
        }
        $objeto->save();
}
```

Y para borrar el DELETE:

```
if ($verb == "DELETE") {
        if (empty($id)) {
        $http->setHttpHeaders(400, new Response("Bad request"));
        die();
        }
        $objeto->load($id);
        $objeto->delete();
}
```