

Servizos web

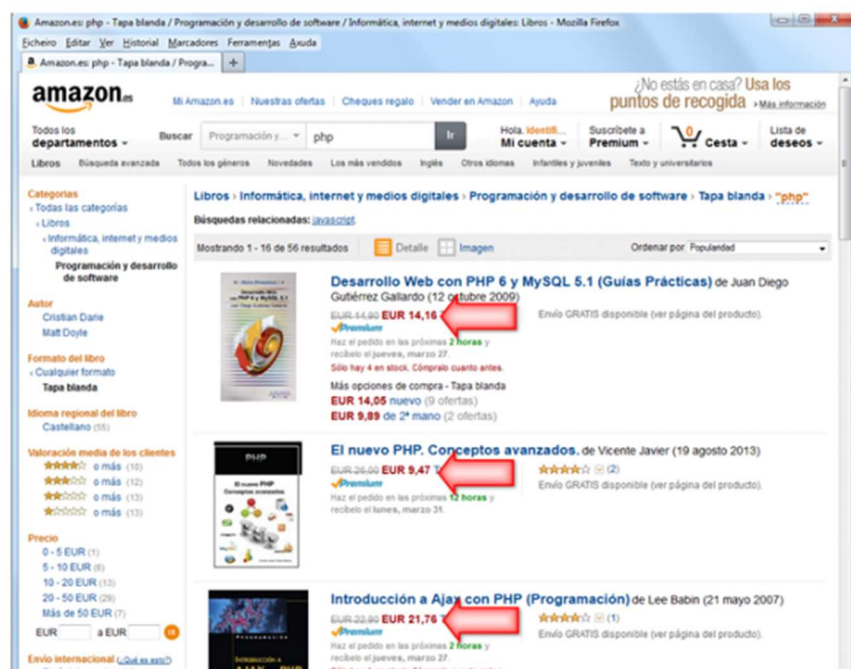
As aplicacións poden necesitar compartir información con outras aplicacións. Por exemplo, un blog pode publicar un listado cos títulos dos últimos artigos para que sexa empregado dentro de outras páxinas. Ou unha tenda online pode dar acceso aos prezos dos produtos que vende. Ou á inversa, pode suceder que se programe unha aplicación web que empregue información dispoñible noutras páxinas. Por exemplo, a tenda online pode acceder a información sobre o cambio de divisas para presentar os prezos dos artigos en moedas estranxeiras para os compradores internacionais. Ou engadir información sobre o coste dos seus envíos que estea dispoñible na páxina da compañía de paquetería coa que fai os seus envíos. Existen principalmente catro formas de compartir información entre aplicacións informáticas:

- **Compartir o acceso á base de datos.** Para compartir a información que xestiona unha aplicación, normalmente é suficiente con dar acceso á base de datos na que se almacena.



Pero esta xeralmente non é unha boa idea:

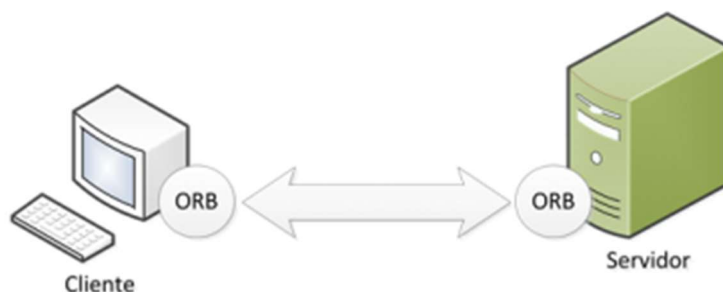
- Cantas máis aplicacións utilicen os mesmos datos, máis posibilidades hai de que se xeren erros nestes.
- Se xa hai unha aplicación funcionando, xa se programou a lóxica de negocio correspondente, e esta non poderá aproveitarse noutras aplicacións se utilizan directamente a información almacenada na base de datos.
- Se se quere poñer a base de datos a disposición de terceiros, estes necesitarán coñecer a súa estrutura. E ao dar acceso directo aos datos, será complicado manter o control sobre as modificacións que se produzan nestes.
- **Acceder á información publicada pola aplicación.** Gran parte da información que xestionan as aplicacións web xa está dispoñible para que outros a utilicen (deixando a un lado as consideracións relacionadas co control de acceso). Por exemplo, se alguén quere coñecer o prezo dun produto dentro dunha tenda web, abonda con buscar ese produto na páxina na que se listan todos os produtos.



Pero, para que un programa poda obter esa mesma información (o prezo dun produto), este tería que programar un procedemento para buscar o produto concreto dentro das etiquetas HTML da páxina e extraer o seu prezo.

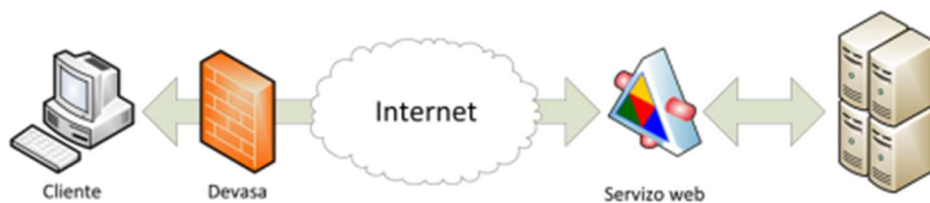
Esta técnica coñécese polo seu nome en inglés: *screen scraping*. É un procedemento custoso e pode requirir refacer o código cando se producen cambios no contido ou no aspecto da páxina que contén a información.

- **Empregar técnicas tradicionais de acceso a información compartida.** Os primeiros mecanismos para compartir información de xeito automatizado e estándar entre dúas ou máis aplicacións informáticas facían uso de tecnoloxías como DCOM e CORBA.



Estas tecnoloxías seguían o modelo de chamada a procedemento remoto ([RPC](#)), que é máis axeitado para contornos illados onde se poden controlar todos os parámetros relacionados coa comunicación. RPC é un protocolo de comunicación de procesos que permite a un programa realizar chamadas a procedementos que se executan noutros equipos. Ao basearse en RPC, DCOM e CORBA pretenden permitir ás aplicacións traballar de xeito semellante a como o farían dous procesos comunicándose entre eles dentro da mesma máquina.

- **Empregar servizos web.** Os servizos web creáronse para permitir o intercambio de información ao igual que RPC, pero sobre a base do protocolo HTTP (de aí o termo web). En lugar de definir o seu propio protocolo para transportar as peticións de información, utilizan principalmente HTTP para este fin (aínda que como xa veremos tamén é posible empregar outros protocolos de transporte como RPC, FTP ou SMTP). Polo tanto, calquera ordenador que poida consultar unha páxina web, poderá tamén solicitar información dun servizo web. Se existe algunha devasa na rede, tratará a petición de información igual que o faría coa solicitude dunha páxina web.



Servizos web

Un servizo web é unha tecnoloxía que comunica dúas aplicacións informáticas a través da Web, empregando un conxunto de protocolos e estándares que posibilitan o intercambio de información. Xurdiron ante a necesidade de establecer mecanismos estándar de comunicación entre distintas plataformas e linguaxes de programación. Os servizos web permiten publicar conxuntos de procedementos e funcións de programación (APIs) independentes da plataforma, de xeito que sexan accesibles dende aplicacións ou servizos remotos a través de Internet. Nos servizos web diferéncianse dous **actores**:

– **O provedor ou servidor**, que é o crea e publica o mecanismo para acceder á información.

– **O consumidor ou cliente**, que é o que establece a comunicación co provedor para acceder á información.

Tipos de servizos web

Hai distintos tipos de servizos web:

- **RPC (Remote Procedure Call)**
 - XML-RPC
 - JSON-RPC
 - SOAP
- **RESTful**

RPC (Remote Procedure Call)

- **XML-RPC.** É unha arquitectura sinxela. Basease no envío de mensaxes XML empregando peticións POST do protocolo HTTP, que é o que se emprega para o transporte.

Por exemplo, a mensaxe XML-RPC que chama a un método para obter o tipo de cambio de dúas moedas, podería ser:

```
<?xml version="1.0"?>
<methodCall>
  <methodName>moeda.getCambio</methodName>
  <params>
    <param>
      <value><int>3</int></value>
    </param>
    <param>
      <value><int>1</int></value>
    </param>
  </params>
</methodCall>
```

E a respectiva mensaxe de resposta:

```
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value><double>1.2416</double></value>
    </param>
  </params>
</methodResponse>
```

- **JSON-RPC.** [JSON](#) (JavaScript Object Notation) é un formato de ficheiro que usa texto lexible para transmitir obxectos de datos mediante pares atributo-valor e arrays. É independente da linguaxe e multitude de linguaxes poden xerar e analizar datos en formato JSON. O media type (MIME type ou content type) para JSON é **application/json**. Os ficheiros JSON usan a extensión **.json**.

Exemplo 1:

```
{"nome": "Xan", "apelido": "Pérez", "estaVivo": true, "idade": 27}
```

Exemplo 2:

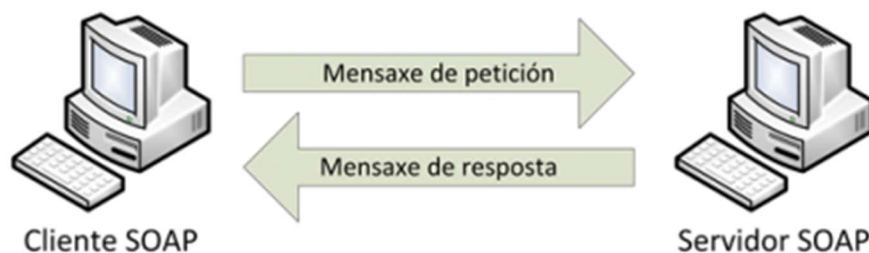
```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    },
    {
      "type": "mobile",
      "number": "123 456-7890"
    }
  ],
  "children": [],
  "spouse": null
}
```

JSON-RPC é un RPC codificado en JSON. Define só uns poucos tipos de datos e comandos. Funciona enviado unha

petición dende un cliente que tenta chamar a un método nun servidor, podendo enviarlle múltiples parámetros nun array ou obxecto e o método podendo enviar múltiples parámetros de saída.

- **SOAP.** O nome SOAP xurdiu como acrónimo de Simple Object Access Protocol, pero, a partir da versión 1.2 do protocolo, o nome SOAP xa non se refire a nada en concreto.

Ao igual que o seu antecesor, XML-RPC, SOAP utiliza XML para compoñer as mensaxes que se transmiten entre o cliente (que xera unha petición) e o servidor (que envía unha resposta) do servizo web, pero a súa arquitectura é moito máis complexa.



SOAP normalmente emprega HTTP como protocolo de transporte. Desta forma pode funcionar sobre calquera servidor web e, o que é aínda máis importante, utilizando o porto 80 reservado para este protocolo. Pero non é o único protocolo de transporte que pode utilizar; é posible empregar SOAP tamén sobre outros protocolos como FTP ou SMTP.

O seguinte é un exemplo dunha petición semellante á que rematamos de ver con XML-RPC, pero empregando neste caso SOAP:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope

xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:ns1="http://omeuservizoweb.com/tipos"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
  xmlns:soap-
enc="http://schemas.xmlsoap.org/soap/encoding/"

soap:encodingStyle="http://schemas.xmlsoap.org/soap/en
coding/">
  <soap:Body>
    <ns1:getCambio>
      <param0 xsi:type="xsd:int">3</param0>
      <param1 xsi:type="xsd:int">1</param1>
    </ns1:getCambio>
  </soap:Body>
</soap:Envelope>
```

E a mensaxe de resposta:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope

xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:ns1="http://omeuservizoweb.com/tipos"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
  xmlns:soap-
enc="http://schemas.xmlsoap.org/soap/encoding/"

soap:encodingStyle="http://schemas.xmlsoap.org/soap/en
coding/">
  <soap:Body>
    <ns1:getCambioResponse>
      <return xsi:type="xsd:float">1.2416</return>
    </ns1:getCambioResponse>
  </soap:Body>
</soap:Envelope>
```

Na seguinte URL <https://api2cart.com/api-technology/rest-vs-soap-feel-difference-infographic/> pode lerse unha análise comparativa entre SOAP e REST.

RESTful

As siglas REST veñen do termo inglés **RE**presentational **St**ate **T**ransfer. O termo fai referencia a un conxunto de regras para o deseño de arquitecturas distribuídas. Aos servizos web deseñados seguindo estas regras aplícaselles comunmente o termo **RESTful**.

A principal vantaxe dun servizo web RESTful fronte a un servizo web baseado en SOAP é a súa sinxeleza. Nun servizo web RESTful os recursos represéntanse por URIs e asóciase a un tipo concreto de funcionalidade (ex. crear, borrar, engadir ou actualizar nun CRUD).

Podemos ter recursos cunha representación multimedia (imaxes JPEG ou GIF) e recursos en outros formatos como JSON ou XML.

REST é unha forma simple de organizar interaccións entre sistemas independentes. REST permite traballar de forma sinxela con clientes con diferentes sistemas operativos e plataformas. REST utiliza directamente o protocolo **HTTP** para obter datos ou indicar a execución de operacións sobre os datos en calquera formato (XML, JSON, etc).

Introdución a HTTP

HTTP é un protocolo que permite o envío e recibo de documentos a través da web. Un protocolo é un conxunto de normas que determina o formato das mensaxes que se poden intercambiar.

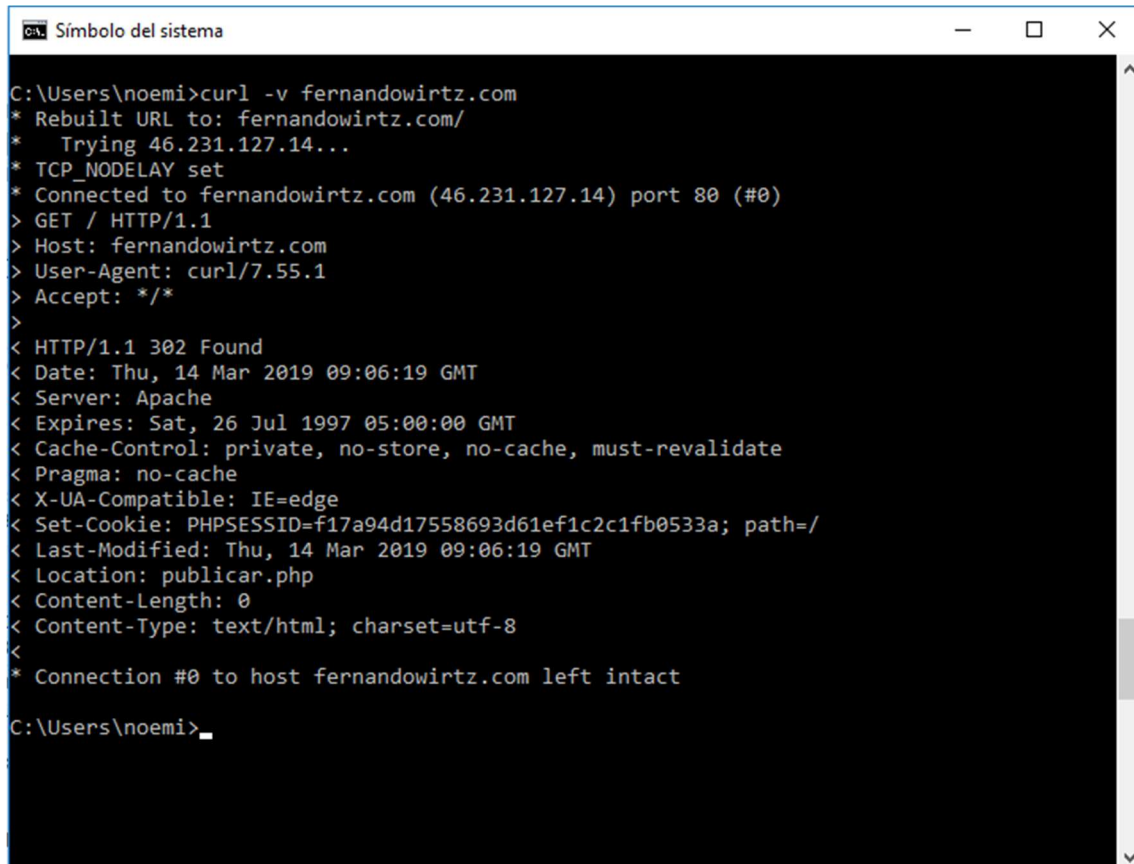
En HTTP hai dous roles diferentes: **servidor** e **cliente**. En xeral, o cliente sempre inicia a conversa e o servidor responde. As mensaxes HTTP son mensaxes de texto aínda que o corpo da mensaxe pode conter outros formatos.

As mensaxes HTTP teñen unha cabeceira (header) e un corpo (body). O corpo pode estar baleiro. As cabeceiras conteñen metadatos como información sobre a codificación das mensaxes.

En REST as cabeceiras son máis importantes que o corpo.

Con navegadores como Chrome (coas ferramentas de desenvolvemento) ou Firefox (coa extensión firebug), poden verse os detalles das mensaxes HTTP ao navegar.

Outra forma de familiarizarse con HTTP é utilizar un cliente dedicado como cURL. **cURL** é unha ferramenta de liña de comandos dispoñible na maioría dos sistemas operativos.



```
C:\Users\noemi>curl -v fernandowirtz.com
* Rebuilt URL to: fernandowirtz.com/
* Trying 46.231.127.14...
* TCP_NODELAY set
* Connected to fernandowirtz.com (46.231.127.14) port 80 (#0)
> GET / HTTP/1.1
> Host: fernandowirtz.com
> User-Agent: curl/7.55.1
> Accept: */*
>
< HTTP/1.1 302 Found
< Date: Thu, 14 Mar 2019 09:06:19 GMT
< Server: Apache
< Expires: Sat, 26 Jul 1997 05:00:00 GMT
< Cache-Control: private, no-store, no-cache, must-revalidate
< Pragma: no-cache
< X-UA-Compatible: IE=edge
< Set-Cookie: PHPSESSID=f17a94d17558693d61ef1c2c1fb0533a; path=/
< Last-Modified: Thu, 14 Mar 2019 09:06:19 GMT
< Location: publicar.php
< Content-Length: 0
< Content-Type: text/html; charset=utf-8
<
* Connection #0 to host fernandowirtz.com left intact

C:\Users\noemi>
```

URLs

As URLs son a forma de identificar os recursos sobre os que se quere operar. Cada URL identifica un recurso.

Por exemplo:

/clientes

Identificará unha lista de clientes, e

/clientes/zaira

Identificará unha cliente, chamada Zaira, supoñendo que é a única con ese nome na lista de clientes.

O host é importante para asegurar que o identificador do recurso é único en toda web. O host inclúese na cabeceira separado da ruta do recurso:

```
GET /clientes/zaira HTTP/1.1
```

```
Host: exemplo.com
```

Os recursos son entendidos como nomes. O seguinte non sería RESTful xa que utiliza unha URL para describir unha acción:

```
/clientes/anadir
```

As URLs deben ser tan precisas como sexa posible. Calquera cousa necesaria para identificar de forma única un recurso debería estar na URL.

Para especificar a acción, en lugar de empregar a URL, empréganse os **métodos HTTP**.

Métodos HTTP

Cada solicitude HTTP especifica un verbo ou método HTTP na cabeceira. Esta é a primeira palabra en maiúsculas que aparece en calquera cabeceira, a solicitude seguinte utiliza o método GET:

```
GET / HTTP/1.1
```

A seguinte solicitude utiliza o método DELETE:

```
DELETE /clientes/marcos HTTP/1.1
```

Os verbos HTTP dinlle ao servidor o que hai que facer cos datos identificados na URL. A solicitude pode, opcionalmente, conter información adicional no corpo, que podería requirir realizar algunha operación, como gardar datos no recurso. Poden proporcionarse estes datos en cURL coa opción -d.

Para construír unha API RESTful os verbos máis importantes son: GET, POST, PUT e DELETE. Hai outros métodos dispoñibles, como HEAD e OPTIONS, pero non son moi empregados.

Método GET

GET é o método máis simple de HTTP. É o que utilizan os navegadores cada vez que se fai clic nunha ligazón ou se escribe unha URL na barra de navegación. Dille ao servidor que transmita os datos identificados pola URL ao cliente. Os datos nunca deberían ser modificados non lado do servidor cunha solicitude GET. Unha solicitude GET é de só lectura, pero unha vez que o cliente recibe os datos, pode facer calquera operación, como darlle formato para mostralo.

Método PUT

Unha solicitude PUT emprégase cando se quere crear ou editar o recurso identificado pola URL:

```
PUT /clientes/maria
```

A solicitude anterior podería crear un cliente chamado maría no servidor. Non hai nada na solicitude que informe ao servidor como se deben crear os datos, só que debe crealos. Isto permite poder cambiar a tecnoloxía do backend no caso de ser necesario.

As solicitudes PUT conteñen os datos a usar á hora de actualizar ou crear un recurso no corpo da páxina. En cURL pódense engadir datos á solicitude con -d:

```
curl -v -X PUT -d "Texto"
```

Método DELETE

O método DELETE fai o contrario que PUT, úsase cando se quere eliminar un recurso identificado pola URL da solicitude.

```
curl -v -X DELETE /clientes/maria
```

Isto eliminará todos os datos asociados co recurso.

Método POST

POST emprégase cando o proceso que se quere que ocorra poda ser repetido se el POST se repite (isto é, que non é **idempotente**).

Ademais, as solicitudes de **POST** deben procesar o corpo da solicitude como subordinado da URL á que se está enviando:

```
POST /clientes/
```

Por exemplo, pode engadirse un novo cliente á lista, cun id xerado polo servidor:

```
/clientes/id-unico
```

As solicitudes de PUT poden usarse en lugar dos POST e viceversa. Algúns sistemas utilizan só un, e outros os dous, incluso para empregar POST para actualizar e PUT para crear.

Clasificación de métodos HTTP

Os métodos HTTP poden clasificarse en:

- **Métodos seguros e inseguros:** os métodos seguros son os que nunca modifican *recursos*. O único seguro dos vistos é GET.
- **Métodos idempotentes:** obteñen o mesmo resultado independentemente do número de veces que se repita o request: GET, PUT e DELETE. O único método non idempotente é POST. PUT e DELETE parece raro que a explicación é que repetir un método PUT con exactamente o mesmo body modifica un recurso de forma que permanece idéntico ao descrito na solicitude de PUT anterior, polo que nada cambia. Da mesma forma, non ten sentido con DELETE eliminar un recurso dúas veces. Non importa cantas veces se utilice PUT e DELETE, o resultado será o mesmo como se só se tivera feito unha vez.

É importante utilizar o método correcto na situación adecuada e iso depende totalmente do desenvolvedor.

Representación de recursos

O **cliente HTTP** e o **servidor HTTP** intercambian información sobre os recursos identificados polas URLs.

O ***request*** e o ***response*** conteñen unha **representación do recurso**. Por representación enténdese **información nun formato específico**, sobre o estado do recurso ou sobre como estará o estado non futuro. Tanto as cabeceiras como o corpo son parte de esa representación.

As cabeceiras **HTTP**, que conteñen **metadatos**, están definidos pola **HTTP spec**. Só poden conter texto e están formateadas dunha maneira específica.

O corpo pode conter datos en calquera formato, e aquí é onde se pode ver o poder de **HTTP**. Pode enviar textos, imaxes, HTML e XML en calquera idioma. A través dos **metadatos das solicitudes de diferentes URLs**, pode elixirse entre diferentes representacións do mesmo recurso. Por exemplo, pode enviarse unha páxina web aos navegadores e un JSON ás aplicacións.

A **resposta HTTP** debería especificar o ***content type* do *body***. Isto realízase no header, no campo Content-Type:

```
Content/Type: application/json
```

Librerías HTTP client

Para **experimentar cos diferentes métodos *request***, é necesario un cliente que permita especificar que método usar. Os formularios HTML só permiten facer solicitudes GET e POST, polo que non sirven para probar os métodos.

Unha librería HTTP client moi popular é cURL. Ademais de proporcionar unha ferramenta en liña de comandos, pode empregarse en diferentes linguaxes, entre les, PHP.

Códigos de resposta

As cabeceiras deben ser o que primeiro aparece nas respostas, non se debe escribir nada antes dunha cabeceira.

As cabeceiras conteñen todo tipo de metadatos, como a codificación de texto utilizada no corpo da mensaxe, ou o tipo MIME do contido do corpo. Neste caso estamos especificando explicitamente os códigos de resposta HTTP. Os código de resposta estandarizan unha forma de informar ao cliente acerca do resultado da súa solicitude. Por defecto, PHP devolve un código de resposta 200, que significa que foi satisfactoria.

O servidor debe enviar o **código de resposta HTTP mais apropiado**. Desta forma o cliente pode intentar reparar os seus erros, supoñendo que haxa algún.

O significado dun código de resposta HTTP non é moi preciso, polo que debe intentar usarse o que mais concorde coa situación.

Os códigos de resposta máis comunmente utilizados con REST son:

- **200 OK**. Satisfactoria.
- **201 Created**. Creouse un recurso. Resposta satisfactoria a unha solicitude POST ou PUT.
- **400 Bad Request**. A solicitude ten algún erro, por exemplo cando os datos proporcionados en POST ou PUT non pasan a validación.
- **401 Unauthorized**. É necesario identificarse primeiro.
- **404 Not Found**. Esta resposta indica que o recurso requirido non pode atoparse (A URL non se corresponde cun recurso).
- **405 Method Not Allowed**. O **método HTTP** utilizado non o soporta este recurso.
- **409 Conflict**. Conflito, por exemplo cando se usa unha solicitude PUT para crear o mesmo recurso dúas veces.

- **500 Internal Server Error.** Un erro 500 soe ser un erro inesperado no servidor.

Exemplo de aplicación en REST

Imos ver un exemplo de aplicación en **REST**, nun servidor con **PHP**. O arquivo *servidor.php* responderá a todas as solicitudes que veñan do servidor. Debemos colocalo en C:\xampp\htdocs\servidor.php.

Todas as solicitudes das URLs que comezan con /clientes/ deben ir enrutadas no arquivo *servidor.php*.

A aplicación no arquivo *servidor.php*, tan só é unha clase con 8 métodos e un array.

En httpd.conf (C:\xampp\apache\conf\httpd.conf) engadimos o seguinte para enviar todas as peticións a servidor.php.

```
<Location />
    <IfModule mod_rewrite.c>
        RewriteEngine On
        RewriteBase /
        RewriteOptions +FollowSymLinks
        RewriteCond %{REQUEST_FILENAME} !-f
        RewriteRule .* servidor.php/$0 [L]
    </IfModule>
</Location>
```

En primeiro lugar debemos iniciar un proceso diferente en función do **método HTTP**, incluso cando as URLs sexan as mesmas. En **PHP** está o array global **\$_SERVER**, que determina que método se ten utilizado para a solicitude.

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

Esta variable contén o nome do método nun string, xa sexa GET, PUT...

En segundo lugar, debemos saber que URL se ten solicitado. Para o que se utiliza outra variable:

```
$_SERVER['REQUEST_URI']
```

Esta variable contén a URL desde a primeira /. Se o host é 'exemplo.com', ['http://exemplo.com/'](http://exemplo.com/) devolverá '/', e ['http://exemplo.com/proba/'](http://exemplo.com/proba/) devolverá '/proba/'.

Primeiro averíguase a **URL solicitada**, considerando só as URL que comezan con *clientes*.

```
$resource = array_shift($paths);  
if($resource == 'clientes'){  
    $name = array_shift($paths);  
    if(empty($name)){  
        $this->handle_base($method);  
    } else {  
        $this->handle_name($method, $name);  
    }  
} else {  
    // Sólo se aceptan resources desde 'clientes'  
    header('HTTP/1.1 404 Not Found');  
}
```

Só hai dúas posibilidades:

1. Se o recurso é *clientes*, devólvese unha lista dos clientes.
2. Se se proporciona un identificador máis, suponse que é o **nome do cliente**, polo que se envía a unha función en concreto **dependendo do método HTTP**. Para facelo utilizamos un switch:

```
switch($method){  
    case 'PUT':  
        $this->create_contact($name);  
        break;  
    case 'DELETE':  
        $this->delete_contact($name);  
        break;  
    case 'GET':  
        $this->display_contact($name);  
        break;  
    default:  
        header('HTTP/1.1 405 Method not allowed');  
        header('Allow: GET, PUT, DELETE');  
        break;  
}
```

}

Empregamos **cURL** para todas as operacións.

Se queremos os **detalles do cliente** 'xoan', enviamos unha solicitude GET á URL deste *recurso*:

```
curl -v http://localhost:80/clientes/xoan
```

A opción -v permite visualizar as cabeceiras da resposta HTTP. A última liña da resposta será o body. Neste caso será en **JSON** que contén o enderezo de Xoan.

Para **obter a información de todos os clientes** á vez:

```
curl -v http://localhost:80/clientes/
```

Para **crear un novo cliente** Marta en Linux:

```
curl -v -X PUT http://localhost:80/clientes/marta -d '{"address":"Rua Hispanidade"}'
```

A opción -X permite especificar na solicitude un método para comunicarse co servidor HTTP. O método que se usa por defecto é o método GET.

Para **crear un novo cliente** Marta en Windows:

```
curl -v -X PUT http://localhost:80/clientes/marta -d "{\"address\":\"Rua Hispanidade\"}"
```

E finalmente, para **eliminar un cliente**:

```
curl -v -X DELETE http://localhost:80/clientes/zaira
```

No caso de solicitar un cliente que non existe, devolve un **erro 404**, en cambio, se se tenta crear un cliente que xa existe devolverá un **erro 409**.

Para que nos funcionen todas as aplicacións que temos aloxadas htdocs do XAMPP, imos facer algúns cambios na configuración.

Imos crear un cartafol en htdocs de nome servizoWeb1. Movemos a este cartafol o arquivo servidor.php.

Como non queremos reiniciar apache cada vez que facemos cambios nas regras de reescritura de URLs, imos borrar do arquivo httpd.conf

todo o que engadimos para facer a nosa primeira proba e reiniciamos o servidor Apache.

Engadimos un arquivo .htaccess en htdocs co seguinte contido:

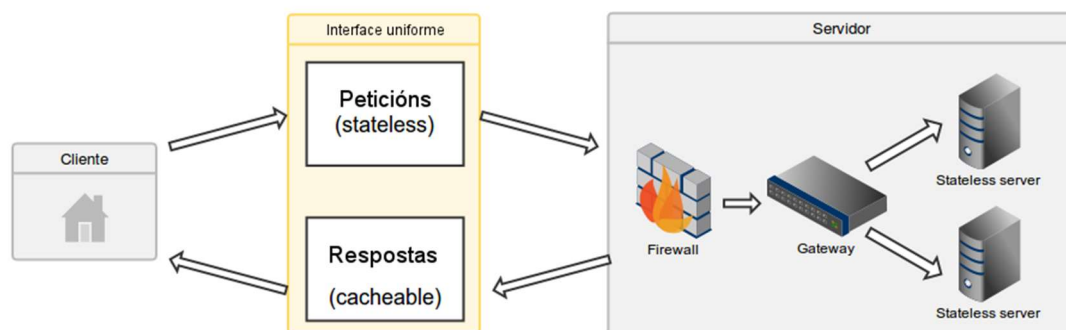
```
RewriteEngine On
RewriteBase /
options +FollowSymLinks
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^clientes(.*) /servizoWeb1/servidor.php/$0 [L]
```

Calquera modificación que se faga do arquivo .htaccess actualízase sen obrigar a reiniciar o servidor Apache.

Verificamos, con curl, que o servizo web segue funcionando tal e como funcionaba con anterioridade.

Regras da arquitectura REST

REST define unha serie de regras que toda aplicación que pretenda chamarse REST debe cumprir. Estas regras veñen impostas, en grande medida, polo uso do protocolo HTTP.



As regras son as seguintes:

- **Arquitectura cliente-servidor.** Existe unha separación clara entre os dous axentes básicos que realizan o intercambio de información: o cliente e o servidor. Estes dous axentes deben ser independentes entre si.

- **Sen estado (stateless):** O servidor non ten que almacenar datos do clientes para manter un estado do mesmo. HTTP tamén cumpre esta norma.
- **Cacheable:** Esta regra implica que o servidor que serve as petición debe definir algún modo de cachear as petición, para aumentar o rendemento, a escalabilidade, etc. HTTP implementa isto coa cabeceira “Cache-control”, que dispón de varios parámetros para controlar que as respostas se podan cachear. As directivas “Cache-control” determinan quen pode almacenar en caché a resposta, en que circunstancias e durante canto tempo.
- **Sistema por capas:** Un sistema non debe obrigar ao cliente a saber as capas polas que viaxa a información, o que permite o cliente conserve a súa independencia con respecto ás capas.
- **Interface uniforme:** Non queremos que a interface de comunicación entre un cliente e un servidor dependa nin do servidor ao que se lle fan petición nin do cliente que as realiza. Este regra garántenos que independentemente de quen faga as petición ou quen as reciba, haberá comunicación sempre que ambos cumbran coa interface definida de antemán.