* A REST API (also known as RESTful API) is an application programming interface (API or web API) that conforms to the constraints of REST architectural style and allows for interaction with RESTful web services. REST stands for representational state transfer and was created by computer scientist Roy Fielding.
* An API is a set of definitions and protocols for building and integrating application software. It’s sometimes referred to as a contract between an information provider and an information user—establishing the content required from the consumer (the call) and the content required by the producer (the response). For example, the API design for a weather service could specify that the user supply a zip code and that the producer reply with a 2-part answer, the first being the high temperature, and the second being the low
* You can think of an API as a mediator between the users or clients and the resources or web services they want to get. It’s also a way for an organization to share resources and information while maintaining security, control, and authentication—determining who gets access to what

REST is a set of architectural constraints, not a protocol or a standard. API developers can implement REST in a variety of ways.

When a client request is made via a RESTful API, it transfers a representation of the state of the resource to the requester or endpoint. This information, or representation, is delivered in one of several formats via HTTP: JSON (Javascript Object Notation), HTML, XLT, Python, PHP, or plain text. JSON is the most generally popular file format to use because, despite its name, it’s language-agnostic, as well as readable by both humans and machine

In order for an API to be considered RESTful, it has to conform to these criteria:

* A client-server architecture made up of clients, servers, and resources, with requests managed through HTTP.
* [Stateless](https://www.redhat.com/en/topics/cloud-native-apps/stateful-vs-stateless) client-server communication, meaning no client information is stored between get requests and each request is separate and unconnected.
* Cacheable data that streamlines client-server interactions.
* A uniform interface between components so that information is transferred in a standard form. This requires that:
  + resources requested are identifiable and separate from the representations sent to the client.
  + resources can be manipulated by the client via the representation they receive because the representation contains enough information to do so.
  + self-descriptive messages returned to the client have enough information to describe how the client should process it.
  + hypertext/hypermedia is available, meaning that after accessing a resource the client should be able to use hyperlinks to find all other currently available actions they can take.
* A layered system that organizes each type of server (those responsible for security, load-balancing, etc.) involved the retrieval of requested information into hierarchies, invisible to the client.
* Code-on-demand (optional): the ability to send executable code from the server to the client when requested, extending client functionality.

Though the REST API has these criteria to conform to, it is still considered easier to use than a prescribed protocol like SOAP (Simple Object Access Protocol), which has specific requirements like XML messaging, and built-in security and transaction compliance that make it slower and heavier.

* RESTful systems follow the client-server model.
* Standard HTTP proxy servers can be used in RESTful architecture.
* REST services can interact with non-REST services, and vice versa.

### Design principles in REST

REST is more of a style and less of a standard, so there are not many design principles to consider. In general, this is what you should follow:

* GET requests should not cause a change in state or alter data. If you wish to modify the state or data, use POST requests.
* Pagination is always a good practice; if your GET query reads entries, let it read the first N number of entries (for example, 20) and then use links to read more entries.
* Physical URLs are considered a bad practice, and logical URLs should be preferred.
* If the REST response is in XML, consider using a schema.

The **RESTful functionality is provided by AngularJS in the ngResource module**, which is distributed separately from the core AngularJS framework. Since we are using npm to install client-side dependencies, this step updates the package

Diffenrence between webservice and api

* Web service is a collection of open source protocols and standards used for exchanging data between systems or applications whereas API is a software interface that allows two applications to interact with each other without any user involvement.
* Web service is used for REST, SOAP and XML-RPC for communication while API is used for any style of communication.
* Web service supports only HTTP protocol whereas API supports HTTP/HTTPS protocol.
* Web service supports XML while API supports XML and JSON.
* All Web services are APIs but all APIs are not web services.

## Difference between API and Web Services

Here are important differences between Web services and API.

| **Web Serviced** | **API** |
| --- | --- |
| All web services are APIs. | All APIs are not web services. |
| It supports XML. | Responses are formatted using Web API’s MediaTypeFormatter into XML, JSON, or any other given format. |
| You need a SOAP protocol to send or receive and data over the network. Therefore it does not have light-weight architecture. | API has a light-weight architecture. |
| It can be used by any client who understands XML. | It can be used by a client who understands JSON or XML. |
| Web service uses three styles: REST, SOAP, and XML-RPC for communication. | API can be used for any style of communication. |
| It provides supports only for the HTTP protocol. | It provides support for the HTTP/s protocol: URL Request/Response Headers, etc. |

To summarize, APIs and **web** services are two technologies for transferring data between separate software applications. API is an interface that exposes data of an application to outside software, whereas web applications are one type of API with stricter requirements.

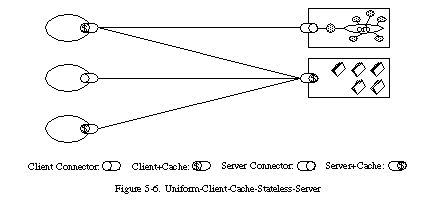
There are two common perspectives on the process of architectural design, whether it be for buildings or for software. The first is that a designer starts with nothing--a blank slate, whiteboard, or drawing board--and builds-up an architecture from familiar components until it satisfies the needs of the intended system. The second is that a designer starts with the system needs as a whole, without constraints, and then incrementally identifies and applies constraints to elements of the system in order to differentiate the design space and allow the forces that influence system behavior to flow naturally, in harmony with the system. Where the first emphasizes creativity and unbounded vision, the second emphasizes restraint and understanding of the system context. REST has been developed using the latter process.

### Stateless

We next add a constraint to the client-server interaction: communication must be stateless in nature, as in the client-stateless-server (CSS) style of [Section 3.4.3](https://www.ics.uci.edu/~fielding/pubs/dissertation/net_arch_styles.htm#sec_3_4_3) ([Figure 5-3](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#fig_5_3)), such that each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client.

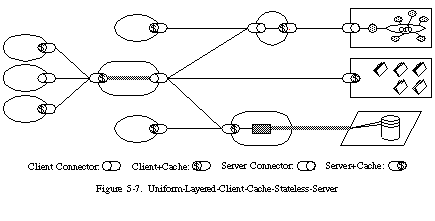
### Uniform Interface

The central feature that distinguishes the REST architectural style from other network-based styles is its emphasis on a uniform interface between components ([Figure 5-6](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#fig_5_6)). By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. Implementations are decoupled from the services they provide, which encourages independent evolvability. The trade-off, though, is that a uniform interface degrades efficiency, since information is transferred in a standardized form rather than one which is specific to an application's needs. The REST interface is designed to be efficient for large-grain hypermedia data transfer, optimizing for the common case of the Web, but resulting in an interface that is not optimal for other forms of architectural interaction.

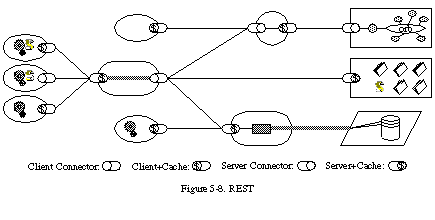


### Layered System

In order to further improve behavior for Internet-scale requirements, we add layered system constraints ([Figure 5-7](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#fig_5_7)). As described in [Section 3.4.2](https://www.ics.uci.edu/~fielding/pubs/dissertation/net_arch_styles.htm#sec_3_4_2), the layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond the immediate layer with which they are interacting. By restricting knowledge of the system to a single layer, we place a bound on the overall system complexity and promote substrate independence. Layers can be used to encapsulate legacy services and to protect new services from legacy clients, simplifying components by moving infrequently used functionality to a shared intermediary. Intermediaries can also be used to improve system scalability by enabling load balancing of services across multiple networks and processors.



The final addition to our constraint set for REST comes from the code-on-demand style of [Section 3.5.3](https://www.ics.uci.edu/~fielding/pubs/dissertation/net_arch_styles.htm#sec_3_5_3) ([Figure 5-8](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#fig_5_8)). REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented. Allowing features to be downloaded after deployment improves system extensibility. However, it also reduces visibility, and thus is only an optional constraint within REST.



The key abstraction of information in REST is a resource. Any information that can be named can be a resource: a document or image, a temporal service (e.g. "today's weather in Los Angeles"), a collection of other resources, a non-virtual object (e.g. a person), and so on. In other words, any concept that might be the target of an author's hypertext reference must fit within the definition of a resource. A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time.

The key abstraction of information in REST is a resource. Any information that can be named can be a resource: a document or image, a temporal service (e.g. "today's weather in Los Angeles"), a collection of other resources, a non-virtual object (e.g. a person), and so on. In other words, any concept that might be the target of an author's hypertext reference must fit within the definition of a resource. A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time.

EST uses various connector types, summarized in [Table 5-2](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#tab_5_2), to encapsulate the activities of accessing resources and transferring resource representations. The connectors present an abstract interface for component communication, enhancing simplicity by providing a clean separation of concerns and hiding the underlying implementation of resources and communication mechanisms. The generality of the interface also enables substitutability: if the users' only access to the system is via an abstract interface, the implementation can be replaced without impacting the users. Since a connector manages network communication for a component, information can be shared across multiple interactions in order to improve efficiency and responsiveness.

|  |  |
| --- | --- |
| **Table 5-2: REST Connectors** | |
| **Connector** | **Modern Web Examples** |
| client | libwww, libwww-perl |
| server | libwww, Apache API, NSAPI |
| cache | browser cache, Akamai cache network |
| resolver | bind (DNS lookup library) |
| tunnel | SOCKS, SSL after HTTP CONNECT |

REST components, summarized in [Table 5-3](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#tab_5_3), are typed by their roles in an overall application action.

|  |  |
| --- | --- |
| **Table 5-3: REST Components** | |
| **Component** | **Modern Web Examples** |
| origin server | Apache httpd, Microsoft IIS |
| gateway | Squid, CGI, Reverse Proxy |
| proxy | CERN Proxy, Netscape Proxy, Gauntlet |
| user agent | Netscape Navigator, Lynx, MOMspider |

REST architectural views

### Process View

A process view of an architecture is primarily effective at eliciting the interaction relationships among components by revealing the path of data as it flows through the system. Unfortunately, the interaction of a real system usually involves an extensive number of components, resulting in an overall view that is obscured by the details. [Figure 5-10](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#fig_5_10) provides a sample of the process view from a REST-based architecture at a particular instance during the processing of three parallel requests.

### Connector View

A connector view of an architecture concentrates on the mechanics of the communication between components. For a REST-based architecture, we are particularly interested in the constraints that define the generic resource interface.

### Data View

A data view of an architecture reveals the application state as information flows through the components. Since REST is specifically targeted at distributed information systems, it views an application as a cohesive structure of information and control alternatives through which a user can perform a desired task. For example, looking-up a word in an on-line dictionary is one application, as is touring through a virtual museum, or reviewing a set of class notes to study for an exam. Each application defines goals for the underlying system, against which the system's performance can be measured.

Rest usage in different programming languages:

Ruby

In Ruby, you can send HTTP requests using the Net::HTTP class. Thus, for GET requests, to look up the record number 1191 from the database of example.com, this is how you should do it:

require 'net/http'

url = 'http://www.example.com/database/1191'

resp = Net::HTTP.get\_response(URI.parse(url))

resp\_text = resp.body

In the preceding example, we are using an object to handle the HTTP response code.

Similarly, for POST requests:

require 'net/http'

url = 'http://www.example.com/database/user'

params = {

firstName =>'Sample',

lastName =>'User'

}

resp = Net::HTTP.post\_form(url, params)

resp\_text = resp.body

Python

In Python, we already have the urllib2 module, so for RESTful actions, we just need to pass the GET request and then handle the response.

For example:

import urllib2

url = 'http://www.example.com/database/1191'

response = urllib2.urlopen(url).read()

And for POST requests, we will once again rely on the urllib2 module:

import urllib

import urllib2

url = 'http://www.example.com/database/user'

params = urllib.urlencode({

'firstName': 'Sample',

'lastName': 'User'

})

response = urllib2.urlopen(url, params).read()

Perl

Personally, I have always relied on LWP, the library for **WWW** in Perl, for REST requests via HTTP.

For example, a GET request would look something like the following:

**use LWP::Simple;**

**my $url = 'http://www.example.com/database/1191';**

**# sample request**

**my $response = get $url;**

**die 'Error getting $url' unless defined $response;**

The preceding code is sufficient for a GET request without additional headers. For something more complex, you should consider creating a browser object in Perl and then handling it accordingly as follows:

**use LWP;**

**my $browser = LWP::UserAgent->new;**

**my $url = 'http://www.example.com/database/1191';**

**my $response = $browser->get $url;**

**die 'Error getting $url' unless $response->is\_success;**

**print 'Content type is ', $response->content\_type;**

**print 'Content is:';**

**print $response->content;**

Now, if you need to issue a POST request, you can follow the preceding approach again, and create a browser object and then pass the POST request as follows:

**my $browser = LWP::UserAgent->new;**

**my $url = 'http://www.example.com/database/1191';**

**my $response = $browser->post($url,**

**[**

**'firstName' =>'Sample',**

**'lastName' =>'User'**

**];**

**);**

**die 'Error getting $url' unless $response->is\_success;**

**print 'Content type is ', $response->content\_type;**

**print 'Content is:';**

**print $response->content;**

In the preceding example, we are using the browser object for issuing the POST request and then mapping the field names directly to the values.

C#

C# as a programming language has structures and concepts of its own. For all practical purposes, you will need to use the .NET classes HttpWebRequest and HttpWebResponse for handling REST requests sent via HTTP.

For example, the following is what a typical GET request in C# would look like:

static string HttpGet(string url) {

HttpWebRequest req = WebRequest.Create(url)

as HttpWebRequest;

string result = null;

using (HttpWebResponse resp = req.GetResponse()

as HttpWebResponse)

{

StreamReader reader =

new StreamReader(resp.GetResponseStream());

result = reader.ReadToEnd();

}

return result;

}

What does the preceding code do? It simply passes a request and then returns the entire response as one long string. For backward compatibility, I would suggest that if you are passing parameters with your requests, it is advisable to properly encode them. You can use any of the native C# classes or methods for such encoding.

For passing POST requests, the method is similar to GETing, as shown in the following:

static string HttpPost(string url,

string[] prName, string[] prVal)

{

HttpWebRequest req = WebRequest.Create(new Uri(url))

as HttpWebRequest;

req.Method = "POST";

req.ContentType = "application/x-www-form-urlencoded";

// Creating a string, encoded and with all parameters

// Assuming that the arrays prName and prVal are of equal length

StringBuilder przz = new StringBuilder();

for (int i = 0; i < prName.Length; i++) {

przz.Append(prName[i]);

przz.Append("=");

przz.Append(HttpUtility.UrlEncode(prVal[i]));

przz.Append("&");

}

// Encoding the parameters

byte[] frDat =

UTF8Encoding.UTF8.GetBytes(przz.ToString());

req.ContentLength = frDat.Length;

// Sending the request

using (Stream post = req.GetRequestStream())

{

post.Write(frDat, 0, frDat.Length);

}

// Getting the response

string result = null;

using (HttpWebResponse resp = req.GetResponse()

as HttpWebResponse)

{

StreamReader reader =

new StreamReader(resp.GetResponseStream());

result = reader.ReadToEnd();

}

return result;

}

Java

When using REST requests in Java, the concept is similar to that of C#, and an experience Java coder can easily pick up the ropes. Basically, you use the HttpURLConnection class and invoke its object type. Following is an example for a GET request:

public static String httpGet(String urlStr) throws IOException {

URL url = new URL(urlStr);

HttpURLConnection conn =

(HttpURLConnection) url.openConnection();

if (conn.getResponseCode() != 200) {

throw new IOException(conn.getResponseMessage());

}

// Buffering the result into a string

BufferedReader drdr = new BufferedReader(

new InputStreamReader(conn.getInputStream()));

StringBuilder sb = new StringBuilder();

String line;

while ((line = drdr.readLine()) != null) {

sb.append(line);

}

drdr.close();

conn.disconnect();

return sb.toString();

}

In the preceding code, we are issuing a GET request and then accepting the response as one long string. If you wish to use it in your projects, you might wish to tweak it a bit, probably with the help of try or catch. Plus, note that for backward compatibility, it is advisable to encode the parameters that are passed with the request URL.

Now, for POST requests, this is how we will work:

public static String httpPost(String urlStr, String[] prName,

String[] prVal) throws Exception {

URL url = new URL(urlStr);

HttpURLConnection conn =

(HttpURLConnection) url.openConnection();

conn.setRequestMethod("POST");

conn.setDoOutput(true);

conn.setDoInput(true);

conn.setUseCaches(false);

conn.setAllowUserInteraction(false);

conn.setRequestProperty("Content-Type",

"application/x-www-form-urlencoded");

// Creating form content

OutputStream out = conn.getOutputStream();

Writer writer = new OutputStreamWriter(out, "UTF-8");

for (int i = 0; i < prName.length; i++) {

writer.write(prName[i]);

writer.write("=");

writer.write(URLEncoder.encode(prVal[i], "UTF-8"));

writer.write("&");

}

writer.close();

out.close();

if (conn.getResponseCode() != 200) {

throw new IOException(conn.getResponseMessage());

}

// Buffering the result into a string

BufferedReader drdr = new BufferedReader(

new InputStreamReader(conn.getInputStream()));

StringBuilder bsbs = new StringBuilder();

String line;

while ((line = drdr.readLine()) != null) {

bsbs.append(line);

}

drdr.close();

conn.disconnect();

return bsbs.toString();

}

Once again, we are accepting a POST request with a parameter and then passing the response accordingly.

Note

You will need to supplement this code with try/catch structures before inserting it within your projects.

Also, an experienced Java coder will be aware that Java is not the most popular language for web development and that its support for handlers for web connections is not at the top of its league. It is, therefore, a good idea to make use of packages and handlers from the Apache library for this purpose. However, we will evade this discussion now since it is beyond the scope of this book, and Java code is of little merit for someone whose primary focus might be on using RESTful services with WordPress.

PHP

Now, finally, we come to the language in which WordPress has been coded. Using REST in PHP is very easy because even the most basic PHP functions with a file-access model can work seamlessly with HTTP requests and URLs.

Therefore, for GET requests, virtually any file-reading function of PHP can do the job, such as fopen, for example:

**$url = "http://www.example.com/database/1191";**

**$response = file\_get\_contents($url);**

**echo $response;**

If you are passing parameters with GET requests, it might be a good idea to encode them.

However, while GET requests are pretty easy to handle, POST requests require a bit of work because you need to open a connection to the target server and then send the HTTP header information. For example, consider the following code:

function httpRequest($host, $port, $method, $path, $prms){

// prms is to map from name to value

$prmstr = "";

foreach ($prms as $name, $val){

$prmstr .= $name . "=";

$prmstr .= urlencode($val);

$prmstr .= "&";

}

// Assign defaults to $method and $port

if (empty($method)) {

$method = 'GET';

}

$method = strtoupper($method);

if (empty($port)) {

$port = 80; // Default HTTP port

}

// Create the connection

$sock = fsockopen($host, $port);

if ($method == "GET") {

$path .= "?" . $prmstr;

}

fputs($sock, "$method $path HTTP/1.1\r\n");

fputs($sock, "Host: $host\r\n");

fputs($sock, "Content-type: " .

"application/x-www-form-urlencoded\r\n");

if ($method == "POST") {

fputs($sock, "Content-length: " .

strlen($prmstr) . "\r\n");

}

fputs($sock, "Connection: close\r\n\r\n");

if ($method == "POST") {

fputs($sock, $prmstr);

}

// Buffer the result

$result = "";

while (!feof($sock)) {

$result .= fgets($sock,1024);

}

fclose($sock);

return $result;

}

Now, using the preceding sample function, we can issue a POST request as follows:

$resp = httpRequest("www.example.com",

80, "POST", "/Database",

array("firstName" =>"Sample", "lastName" =>"User"));

We can also use the **client URL request library** (**cURL**) when working with RESTful requests in PHP.

JavaScript

Having covered all of that, let us finally discuss REST implementation in JavaScript. We will be saving the JSON issue for detailed discussion during the course of this book, so let's just focus on the traditional route now.

REST requests can be sent from client-side or in-browser JavaScript. If you have ever worked with an AJAX application, you have followed the REST design principles to a great extent, with the response being in JSON.

HTTP requests in JavaScript require the XMLHttpRequest object. The following function is a simple way to create the object:

function createRequest() {

var result = null;

if (window.XMLHttpRequest) {

result = new XMLHttpRequest();

if (typeof xmlhttp.overrideMimeType != 'undefined') {

result.overrideMimeType('text/xml'); // Or anything else

}

}

else if (window.ActiveXObject) {

result = new ActiveXObject("Microsoft.XMLHTTP");

}

return result;

}

Now that you have created the object, you are ready to send HTTP requests. However, the XMLHttpRequest object, while it can send requests, cannot return values by default. So it is better to have a callback function that can be invoked when your request is completed.

Thereafter, you are ready to send the request. For a GET request, the approach is fairly simple:

req.open("GET", url, true);

req.send();

And for POST requests:

req.open("POST", url, true);

req.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");

req.send(form-encoded request body);

As you can see, sending HTTP requests in JavaScript is pretty easy and you just need to call the appropriate function.