

# What exactly is RESTful programming?

Asked 15 years, 9 months ago    Modified 1 year, 5 months ago

Viewed 1.7m times



What exactly is RESTful programming?

4195

rest

http

architecture

definition



Share



Improve this question



Follow

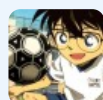
edited May 31, 2023 at 16:02



TylerH

21.2k ● 76 ● 79 ● 110

asked Mar 22, 2009 at 14:45



hasen

166k ● 66 ● 196 ● 233

35 Answers

Sorted by:

Highest score (default)



1

2

Next



3031



*REST* is the underlying architectural principle of the web. The amazing thing about the web is the fact that clients (browsers) and servers can interact in complex ways without the client knowing anything beforehand about the server and the resources it hosts. The key constraint is that the server and client must both agree on the *media* used, which in the case of the web is *HTML*.

An API that adheres to the principles of *REST* does not require the client to know anything about the structure of the API. Rather, the server needs to provide whatever information the client needs to interact with the service. An *HTML form* is an example of this: The server specifies the location of the resource and the required fields. **The browser doesn't know in advance where to submit the information, and it doesn't know in advance what information to submit. Both forms of information are entirely supplied by the server.** (This principle is called [\*HATEOAS: Hypermedia As The Engine Of Application State\*](#).)

**So, how does this apply to *HTTP*, and how can it be implemented in practice?** HTTP is oriented around verbs and resources. The two verbs in mainstream usage are `GET` and `POST`, which I think everyone will recognize. However, the HTTP standard defines several others such as `PUT` and `DELETE`. These verbs are then applied to resources, according to the instructions provided by the server.

For example, Let's imagine that we have a user database that is managed by a web service. Our service uses a custom hypermedia based on JSON, for which we assign the mimetype `application/json+userdb` (There might also be an `application/xml+userdb` and `application/whatever+userdb` - many media types may be supported). The client and the server have both been programmed to understand this format, but they don't know anything about each other. As [Roy Fielding](#) points out:

A REST API should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state, or in defining extended relation names and/or hypertext-enabled mark-up for existing standard media types.

A request for the base resource `/` might return something like this:

## Request

```
GET /  
Accept: application/json+userdb
```

## Response

```
200 OK  
Content-Type: application/json+userdb
```

```
{
  "version": "1.0",
  "links": [
    {
      "href": "/user",
      "rel": "list",
      "method": "GET"
    },
    {
      "href": "/user",
      "rel": "create",
      "method": "POST"
    }
  ]
}
```

We know from the description of our media that we can find information about related resources from sections called "links". This is called *Hypermedia controls*. In this case, we can tell from such a section that we can find a user list by making another request for `/user`:

## Request

```
GET /user
Accept: application/json+userdb
```

## Response

```
200 OK
Content-Type: application/json+userdb

{
  "users": [
    {
      "id": 1,
      "name": "Emil",
      "country": "Sweden",
```

```

        "links": [
            {
                "href": "/user/1",
                "rel": "self",
                "method": "GET"
            },
            {
                "href": "/user/1",
                "rel": "edit",
                "method": "PUT"
            },
            {
                "href": "/user/1",
                "rel": "delete",
                "method": "DELETE"
            }
        ]
    },
    {
        "id": 2,
        "name": "Adam",
        "country": "Scotland",
        "links": [
            {
                "href": "/user/2",
                "rel": "self",
                "method": "GET"
            },
            {
                "href": "/user/2",
                "rel": "edit",
                "method": "PUT"
            },
            {
                "href": "/user/2",
                "rel": "delete",
                "method": "DELETE"
            }
        ]
    }
]

```

We can tell a lot from this response. For instance, we now know we can create a new user by **POST** ing to **/user**:

## Request

```

POST /user
Accept: application/json+userdb
Content-Type: application/json+userdb

{
    "name": "Karl",

```

```
    "country": "Austria"
  }
```

## Response

```
201 Created
Content-Type: application/json+userdb

{
  "user": {
    "id": 3,
    "name": "Karl",
    "country": "Austria",
    "links": [
      {
        "href": "/user/3",
        "rel": "self",
        "method": "GET"
      },
      {
        "href": "/user/3",
        "rel": "edit",
        "method": "PUT"
      },
      {
        "href": "/user/3",
        "rel": "delete",
        "method": "DELETE"
      }
    ]
  },
  "links": {
    "href": "/user",
    "rel": "list",
    "method": "GET"
  }
}
```

We also know that we can change existing data:

## Request

```
PUT /user/1
Accept: application/json+userdb
Content-Type: application/json+userdb

{
  "name": "Emil",
  "country": "Bhutan"
}
```

## Response

```
200 OK
Content-Type: application/json+userdb

{
  "user": {
    "id": 1,
    "name": "Emil",
    "country": "Bhutan",
    "links": [
      {
        "href": "/user/1",
        "rel": "self",
        "method": "GET"
      },
      {
        "href": "/user/1",
        "rel": "edit",
        "method": "PUT"
      },
      {
        "href": "/user/1",
        "rel": "delete",
        "method": "DELETE"
      }
    ]
  },
  "links": {
    "href": "/user",

```

```
        "rel": "list",  
        "method": "GET"  
    }  
}
```

Notice that we are using different HTTP verbs ( `GET` , `PUT` , `POST` , `DELETE` etc.) to manipulate these resources, and that the only knowledge we presume on the client's part is our media definition.

Further reading:

- The many much better answers on this very page.
- ~~[How I explained REST to my wife.](#)~~
- [How I explained REST to my wife.](#)
- [Martin Fowler's thoughts](#)
- [PayPal's API has hypermedia controls](#)

(This answer has been the subject of a fair amount of criticism for missing the point. For the most part, that has been a fair critique. What I originally described was more in line with how REST was usually implemented a few years ago when I first wrote this, rather than its true meaning. I've revised the answer to better represent the real meaning.)

Share Improve this answer

edited Apr 10, 2019 at 15:31

Follow

community wiki



- 
- 184 No. REST didn't just pop up as another buzzword. It came about as a means of describing an alternative to SOAP-based data exchange. The term REST helps frame the discussion about how to transfer and access data.  
– [tvanfosson](#) Mar 22, 2009 at 15:11
- 
- 117 Nonetheless, the heart of REST (in practical application) is "don't use GET to make changes, use POST/PUT/DELETE", which is advice I've been hearing (and following) since long before SOAP appeared. REST *has* always been there, it just didn't get a name beyond "the way to do it" until recently. – [Dave Sherohman](#) Mar 22, 2009 at 15:16
- 
- 40 Don't forget "Hypertext as the engine of application state".  
– [Hank Gay](#) Mar 22, 2009 at 15:54
- 
- 48 This answer misses the point. HTTP is barely mentioned in Fielding's thesis. – [user359996](#) Nov 18, 2010 at 2:22
- 
- 20 This answer doesn't mention the purpose of REST, and makes it seem like it's all about clean URIs. While this might be the popular perception of REST, D.Shawley's and oluies answers are more accurate - it's about being able to take advantage of features built into the architecture, like caching, by working with it instead of against it. Prettier URIs are mostly a common side effect. – [T.R.](#) Dec 20, 2010 at 20:49
- 



823

An **architectural style** called [REST \(Representational State Transfer\)](#) advocates that web applications should use HTTP as it was **originally envisioned**. Lookups should use [GET](#) requests. [PUT](#), [POST](#), and [DELETE](#)



[requests](#) should be used for **mutation**, **creation**, and **deletion** respectively.



REST proponents tend to favor URLs, such as

```
http://myserver.com/catalog/item/1729
```



but the REST architecture does not require these "pretty URLs". A GET request with a parameter

```
http://myserver.com/catalog?item=1729
```

is every bit as RESTful.

Keep in mind that GET requests should never be used for updating information. For example, a GET request for adding an item to a cart

```
http://myserver.com/addToCart?  
cart=314159&item=1729
```

would not be appropriate. GET requests should be [idempotent](#). That is, issuing a request twice should be no different from issuing it once. That's what makes the requests cacheable. An "add to cart" request is not idempotent—issuing it twice adds two copies of the item to the cart. A POST request is clearly appropriate in this context. Thus, even a **RESTful web application** needs its share of POST requests.

This is taken from the excellent book *Core JavaServer Faces* book by David M. Geary.

Share Improve this answer

Follow

edited Jul 4, 2021 at 12:27



Zulkifil

337 ● 4 ● 4

answered Apr 15, 2015 at 11:26



Farhan stands with  
Palestine

14.1k ● 14 ● 66 ● 108

- 
- 2    Listing Available Idempotent Operations: GET(Safe), PUT & DELETE (Exception is mentioned in this link [restapitutorial.com/lessons/idempotency.html](http://restapitutorial.com/lessons/idempotency.html)). Additional Reference for Safe & Idempotent Methods [w3.org/Protocols/rfc2616/rfc2616-sec9.html](http://w3.org/Protocols/rfc2616/rfc2616-sec9.html) – [Abhijeet](#) Jul 21, 2015 at 4:00
- 
- 6    a) the important point about GET is safeness, not idempotence, b) @Abhijeet: RFC 2616 has been obsoleted in 2014; see RF 7230ff. – [Julian Reschke](#) May 6, 2016 at 6:16 ✎
- 
- 24    This is wrong. Read this for correct interpretation of REST [roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven](http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven) or this [stackoverflow.com/questions/19843480/...](http://stackoverflow.com/questions/19843480/...) – [HalfWebDev](#) Aug 25, 2017 at 7:09 ✎
- 
- 7    @kushalvm That academic definition of REST is not used in practice. – [Elliott Beach](#) Aug 27, 2017 at 15:31
- 
- 6    Effectively we can wonder if a concept is operational since we fail to simply give it a stable and understandable definition for all – [HoCo\\_](#) May 3, 2018 at 23:03 ✎
-



558



RESTful programming is about:

- resources being identified by a persistent identifier: URIs are the ubiquitous choice of identifier these days
- resources being manipulated using a common set of verbs: HTTP methods are the commonly seen case - the venerable `Create` , `Retrieve` , `Update` , `Delete` becomes `POST` , `GET` , `PUT` , and `DELETE` . But REST is not limited to HTTP, it is just the most commonly used transport right now.
- the actual representation retrieved for a resource is dependent on the request and not the identifier: use Accept headers to control whether you want XML, HTTP, or even a Java Object representing the resource
- maintaining the state in the object and representing the state in the representation
- representing the relationships between resources in the representation of the resource: the links between objects are embedded directly in the representation
- resource representations describe how the representation can be used and under what circumstances it should be discarded/refetched in a consistent manner: usage of HTTP Cache-Control headers

The last one is probably the most important in terms of consequences and overall effectiveness of REST.

Overall, most of the RESTful discussions seem to center on HTTP and its usage from a browser and what not. I understand that R. Fielding coined the term when he described the architecture and decisions that lead to HTTP. His thesis is more about the architecture and cache-ability of resources than it is about HTTP.

If you are really interested in what a RESTful architecture is and why it works, read [his thesis](#) a few times and read the **whole thing** not just Chapter 5! Next look into [why DNS works](#). Read about the hierarchical organization of DNS and how referrals work. Then read and consider how DNS caching works. Finally, read the HTTP specifications ([RFC2616](#) and [RFC3040](#) in particular) and consider how and why the caching works the way that it does. Eventually, it will just click. The final revelation for me was when I saw the similarity between DNS and HTTP. After this, understanding why SOA and Message Passing Interfaces are scalable starts to click.

I think that the most important trick to understanding the architectural importance and performance implications of a RESTful and [Shared Nothing](#) architectures is to avoid getting hung up on the technology and implementation details. Concentrate on who owns resources, who is responsible for creating/maintaining them, etc. Then think about the representations, protocols, and technologies.

Share Improve this answer

edited Oct 7, 2021 at 5:46

Follow



Community Bot

1 • 1

answered Mar 22, 2009 at 19:37



D.Shawley

59.5k ● 10 ● 105 ● 116

- 
- 40 An answer providing a reading list is very appropriate for this question. – [ellisbben](#) Feb 1, 2012 at 19:50
- 
- 27 Thanks for the update. `PUT` and `POST` don't really map one-to-one with update and create. `PUT` can be used to create if the client is dictating what the URI will be. `POST` creates if the server is assigning the new URI. – [D.Shawley](#) Feb 1, 2012 at 23:30
- 
- 4 A URN is a URI that uses the `urn:` scheme. Conceptually there is no difference; however, a URN does require that you have a separately defined method to "locate" the resource identified (named) by the URN. Care must be taken to ensure that you don't introduce implicit coupling when relating named resources and their location. – [D.Shawley](#) Mar 30, 2014 at 14:45
- 
- 2 @ellisbben Agreed. If I understand correctly this is the dissertation that gave rise to REST:  
[ics.uci.edu/~fielding/pubs/dissertation/rest\\_arch\\_style.htm](http://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm)  
– [Philip Couling](#) Sep 10, 2014 at 12:56
- 
- 2 One doubt on this point: "the actual representation retrieved for a resource is dependent on the request and not the identifier: use HTTP Accept headers to control whether you want XML, HTTP, or even a Java Object representing the resource" --Here, should it be "...whether you want XML, HTML, or even a Java Object representing the resource..." I am thinking that HTTP Accept headers tells the format for data exchanged. And HTTP is the protocol used in RESTel web APIs – [Krishna Shetty](#) Nov 27, 2014 at 9:34
-



This is what it might look like.

417

Create a user with three properties:



```
POST /user
fname=John&lname=Doe&age=25
```



The server responds:

```
200 OK
Location: /user/123
```

In the future, you can then retrieve the user information:

```
GET /user/123
```

The server responds:

```
200 OK
<fname>John</fname><lname>Doe</lname><age>25</age>
```

To modify the record ( `lname` and `age` will remain unchanged):

```
PATCH /user/123
fname=Johnny
```

To update the record (and consequently `lname` and `age` will be NULL):

```
PUT /user/123
fname=Johnny
```

Share Improve this answer

Follow

edited Feb 6, 2017 at 20:23



Kyle Baker

3,704 ● 2 ● 27 ● 35

answered Jul 4, 2009 at 5:47



pbreitenbach

11.3k ● 3 ● 35 ● 24

- 
- 42 For me this answer captured the essence of the desired answer. Simple and pragmatic. Granted there are lots of other criteria, but the example provided is a great launch pad. – [CyberFonic](#) Feb 1, 2012 at 22:09
- 
- 94 In the last example, @pbreitenbach uses `PUT` `fname=Jonny`. This would set `lname` and `age` to default values (probably NULL or the empty string, and integer 0), because a `PUT` **overwrites the whole resource** with data from the representation provided. This is not what is implied by "update", **to do a real update, use the `PATCH` method** as this does not alter fields which are not specified in the representation. – [Nicholas Shanks](#) Jan 31, 2013 at 9:43 ✎
- 
- 25 Nicholas is right. Also, the URI for the first POST creating a user should be called `users` because `/user/1` makes no sense and there should be a listing at `/users`. The response should be a `201 Created` and not just OK in that case. – [DanMan](#) Feb 16, 2013 at 19:58 ✎
- 
- 1 This is just an example of an API not necessarily a RESTful api. A RESTful has constraints it adheres to. Client-Server Architecture, Stateless, Cache-ability, Layered System, Uniform Interface. – [Radmation](#) Jun 26, 2018 at 22:11
-



Thats a very compact answer that covered all http servlet access methods – [Himanshu](#) Jan 3, 2019 at 19:45



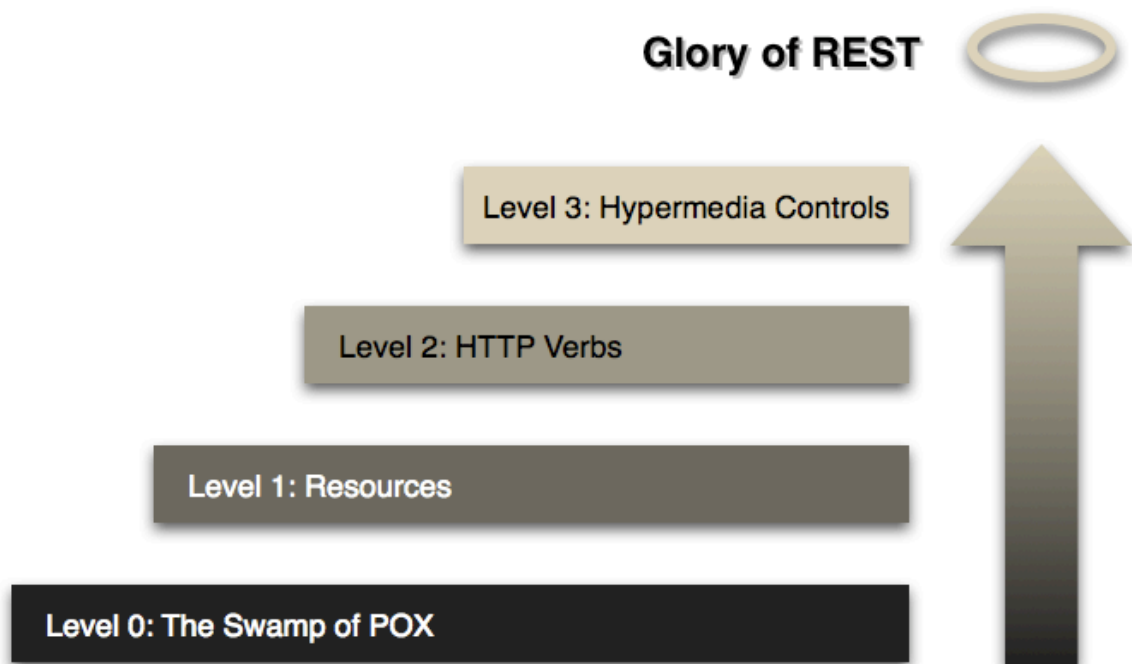
187



A great book on REST is [REST in Practice](#).

Must reads are [Representational State Transfer \(REST\)](#) and [REST APIs must be hypertext-driven](#)

See Martin Fowlers article the [Richardson Maturity Model \(RMM\)](#) for an explanation on what an RESTful service is.



To be RESTful a Service needs to fulfill the [Hypermedia as the Engine of Application State. \(HATEOAS\)](#), that is, it needs to reach level 3 in the RMM, [read the article](#) for details or the [slides from the qcon talk](#).

The HATEOAS constraint is an acronym for Hypermedia as the Engine of Application State. This principle is the key differentiator between a

REST and most other forms of client server system.

...

A client of a RESTful application need only know a single fixed URL to access it. All future actions should be discoverable dynamically from hypermedia links included in the representations of the resources that are returned from that URL. Standardized media types are also expected to be understood by any client that might use a RESTful API. (From Wikipedia, the free encyclopedia)

[REST Litmus Test for Web Frameworks](#) is a similar maturity test for web frameworks.

[Approaching pure REST: Learning to love HATEOAS](#) is a good collection of links.



[REST versus SOAP for the Public Cloud](#) discusses the current levels of REST usage.

[REST and versioning](#) discusses Extensibility, Versioning, Evolvability, etc. through Modifiability

Share Improve this answer

edited Sep 8, 2013 at 17:43

Follow

- 
- 7 I think this answer touches the key point of understanding REST: what does the word **representational** mean. Level 1 - Resources says about *state*. Level 2 - HTTP Verbs says about *transfer* (read *change*). Level 3 - HATEOAS says driving future transfers via representation (JSON/XML/HTML returned), which means you've got known how to say the next round of talk with the information returned. So REST reads: "(representational (state transfer))", instead of "((representational state) transfer)". – [lcn](#) Dec 9, 2014 at 19:49 
- 
- 1 [Difference between REST and POX](#) – [Brent Bradburn](#) Feb 8, 2018 at 22:41 
- 



139



## What is REST?

REST stands for Representational State Transfer. (It is sometimes spelled "ReST".) It relies on a stateless, client-server, cacheable communications protocol -- and in virtually all cases, the HTTP protocol is used.

REST is an architecture style for designing networked applications. The idea is that, rather than using complex mechanisms such as CORBA, RPC or SOAP to connect between machines, simple HTTP is used to make calls between machines.

In many ways, the World Wide Web itself, based on HTTP, can be viewed as a REST-based architecture. RESTful applications use HTTP requests to post data (create and/or update), read data (e.g., make queries), and delete data. Thus, REST uses HTTP for all four CRUD (Create/Read/Update/Delete) operations.

REST is a lightweight alternative to mechanisms like RPC (Remote Procedure Calls) and Web Services (SOAP, WSDL, et al.). Later, we will see how much more simple REST is.

Despite being simple, REST is fully-featured; there's basically nothing you can do in Web Services that can't be done with a RESTful architecture. REST is not a "standard". There will never be a W3C recommendation for REST, for example. And while there are REST programming frameworks, working with REST is so simple that you can often "roll your own" with standard library features in languages like Perl, Java, or C#.

One of the best reference I found when I try to find the simple real meaning of rest.

<http://rest.elkstein.org/>

Share Improve this answer

edited Oct 28, 2013 at 23:07

Follow



jball

25k ● 9 ● 72 ● 92

answered Nov 18, 2012 at 20:46



Ravi

3,202 ● 5 ● 27 ● 36

---

This is a really concise answer. Can you also describe why the REST is called stateless? – [Arefe](#) Feb 12, 2019 at 17:15

---



REST is using the various HTTP methods (mainly GET/PUT/DELETE) to manipulate data.

93



Rather than using a specific URL to delete a method (say, `/user/123/delete`), you would send a DELETE request to the `/user/[id]` URL, to edit a user, to retrieve info on a user you send a GET request to `/user/[id]`



For example, instead a set of URLs which might look like some of the following..

```
GET /delete_user.x?id=123
GET /user/delete
GET /new_user.x
GET /user/new
GET /user?id=1
GET /user/id/1
```

You use the HTTP "verbs" and have..

```
GET /user/2
DELETE /user/2
PUT /user
```

Share Improve this answer

answered Mar 22, 2009 at 15:20

Follow



dbr

169k ● 69 ● 283 ● 347

---

19 That's "using HTTP properly", which is not the same as "restful" (although it's related to it) – [Julian Reschke](#) Mar 22, 2009 at 15:56

---

2 You could also use /user/del/2 and /user/remove/2 or... GET/DELETE/PUT/POST are just the standardised, "proper" way to do such things (and as Julian says, that's not all there is to REST) – [dbr](#) Jun 2, 2009 at 21:32

---

1 Sure, but that's no reason to avoid them.. REST just saves you reinventing the wheel each time. For an API, REST is great (consistency!), but for structuring a random website it doesn't really matter I'd say (it can be more hassle than it's worth) – [dbr](#) Jun 3, 2009 at 22:34

---

14 Vadim, that would be simply RPC. It's also dangerous to use GET for modifying data since (among other reasons) a search engine may spider your deletion links and visit them all. – [aehlke](#) Jul 20, 2009 at 17:35

---

7 @aehlke - I think the real question there would be "Why does an anonymous user have the ability to delete records from your system?" – [Spencer Ruport](#) Dec 3, 2014 at 5:28

---



72



It's programming where the architecture of your system fits the [REST style](#) laid out by Roy Fielding in [his thesis](#).

Since this is the architectural style that describes the web (more or less), lots of people are interested in it.

Bonus answer: No. Unless you're studying software architecture as an academic or designing web services,



there's really no reason to have heard the term.



Share Improve this answer

edited Mar 22, 2009 at 15:56

Follow

answered Mar 22, 2009 at 14:53



Hank Gay

71.8k ● 36 ● 161 ● 222

- 
- 2 but not straight-forward .. makes it more complicated that it needs to be. – [hasen](#) Mar 22, 2009 at 15:38
- 
- 4 Also, even though the terms REST and RESTful are used almost exclusively in the realm of web applications right now, technically there's nothing tying REST to HTTP. – [Hank Gay](#) Mar 22, 2009 at 15:52
- 
- 3 Fielding's blog has some good, easier to comprehend articles on REST and common misconceptions:  
[roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven](http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven) – [aehlke](#) Jul 20, 2009 at 17:32
- 
- 3 @HankGay I think the reason it's not more esoteric is that most web service developers see REST as a wonderful simplification over alternatives like SOAP. They don't necessarily stick to getting all the REST technicalities correct - and that probably drives the REST fanatics mad - but in most cases they probably don't need to worry about things like making sure their results are "hypermedia-enabled".  
– [moodboom](#) Jul 4, 2013 at 11:56
- 



I would say RESTful programming would be about creating systems (API) that follow the REST architectural style.

50



I found this fantastic, short, and easy to understand tutorial about REST by Dr. M. Elkstein and quoting the essential part that would answer your question for the most part:

[Learn REST: A Tutorial](#)

REST is an *architecture style* for designing networked applications. The idea is that, rather than using complex mechanisms such as CORBA, RPC or SOAP to connect between machines, simple HTTP is used to make calls between machines.

- In many ways, the World Wide Web itself, based on HTTP, can be viewed as a REST-based architecture.

RESTful applications use HTTP requests to post data (create and/or update), read data (e.g., make queries), and delete data. Thus, REST uses HTTP for all four CRUD (Create/Read/Update/Delete) operations.

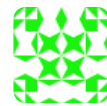
I don't think you should feel stupid for not hearing about REST outside Stack Overflow..., I would be in the same situation!; answers to this other SO question on [Why is REST getting big now](#) could ease some feelings.

Share Improve this answer

edited Feb 14, 2019 at 16:55

Follow





nsandersen

956 ● 2 ● 19 ● 45

answered Jul 12, 2013 at 16:33



Only You

2,072 ● 1 ● 21 ● 36

---

This article explains the relationship between HTTP and REST [freecodecamp.org/news/...](https://freecodecamp.org/news/...) – Only You Sep 21, 2019 at 21:32

---



47



I apologize if I'm not answering the question directly, but it's easier to understand all this with more detailed examples. Fielding is not easy to understand due to all the abstraction and terminology.

There's a fairly good example here:



[Explaining REST and Hypertext: Spam-E the Spam Cleaning Robot](#)

And even better, there's a clean explanation with simple examples here (the powerpoint is more comprehensive, but you can get most of it in the html version):

<http://www.xfront.com/REST.ppt> or  
<http://www.xfront.com/REST.html>

After reading the examples, I could see why Ken is saying that REST is hypertext-driven. I'm not actually sure that he's right though, because that `/user/123` is a URI that points to a resource, and it's not clear to me that

it's unRESTful just because the client knows about it "out-of-band."

That xfront document explains the difference between REST and SOAP, and this is really helpful too. When Fielding says, "[That is RPC. It screams RPC.](#)", it's clear that RPC is not RESTful, so it's useful to see the exact reasons for this. (SOAP is a type of RPC.)

Share Improve this answer

edited Sep 6, 2017 at 20:26

Follow



womp

117k ● 26 ● 238 ● 271

answered Mar 23, 2009 at 17:11



tompark

638 ● 4 ● 8

---

12 cool links, thanks. I'm tired of these REST guys that say some example is not "REST-ful", but then refuse to say how to change the example to be REST-ful. – [coder\\_tim](#) Feb 1, 2012 at 19:19

---



What is REST?

39



REST in official words, REST is an architectural style built on certain principles using the current “Web” fundamentals. There are 5 basic fundamentals of web which are leveraged to create REST services.



- Principle 1: Everything is a Resource In the REST architectural style, data and functionality are considered resources and are accessed using

Uniform Resource Identifiers (URIs), typically links on the Web.

- Principle 2: Every Resource is Identified by a Unique Identifier (URI)
- Principle 3: Use Simple and Uniform Interfaces
- Principle 4: Communication is Done by Representation
- Principle 5: Be Stateless

Share Improve this answer

Follow

edited Jul 31, 2014 at 17:11



Aliaksandr Belik

12.9k ● 6 ● 68 ● 92

answered Jul 25, 2013 at 9:05



Suresh Gupta

603 ● 7 ● 4

---

2 What does Communication is Done by Representation mean? – [mendez7](#) Mar 10, 2019 at 21:59

---



I see a bunch of answers that say putting everything about user 123 at resource `"/user/123"` is RESTful.

34



Roy Fielding, who coined the term, says [REST APIs must be hypertext-driven](#). In particular, "A REST API must not define fixed resource names or hierarchies".



So if your `"/user/123"` path is hardcoded on the client, it's not really RESTful. A good use of HTTP, maybe, maybe not. But not RESTful. It has to come from hypertext.

Share Improve this answer

answered Mar 22, 2009 at 16:36

Follow



Ken

521 ● 3 ● 4

---

7 so .... how would that example be restful? how would you change the url to make it restful? – [hasen](#) Mar 22, 2009 at 16:49

---

1 hasen: Using one resource for all operations might be *necessary* for RESTfulness, but isn't *sufficient*. – [Ken](#) Mar 22, 2009 at 17:20

---

18 ok well .. could you explain further? What's the point of saying "no these guys are wrong .. I know what's right" without saying what you know (or think) to be right? – [hasen](#) Mar 22, 2009 at 20:55

---

5 I gave the link to Fielding's description. I thought I said exactly the relevant diff to the other responses: needs to be driven by hypertext. If "/user/123" comes from some out-of-band API documentation, then it's not RESTful. If it comes from a resource identifier in your hypertext, then it is. – [Ken](#) Mar 23, 2009 at 2:08

---

2 @Andy: A **client** stops being RESTful when you put a hardcoded URL in it. In particular, the RESTful service may decide to renumber users on a whim which breaks that non-RESTful client. The service stops being RESTful when there's no way to discover `/user/123/` from a documented entry point, which indeed means all clients have to hardcode that URL. – [MSalters](#) Oct 20, 2016 at 15:19

---



The answer is very simple, there is a [dissertation](#) written by Roy Fielding.][1](#) In that dissertation he defines the

29

REST principles. If an application fulfills all of those principles, then that is a REST application.



[The term RESTful was created because ppl exhausted the word REST by calling their non-REST application as REST.](#) After that the term RESTful was exhausted as well. [Nowadays we are talking about Web APIs and Hypermedia APIs](#), because the most of the so called REST applications did not fulfill the HATEOAS part of the uniform interface constraint.

The REST constraints are the following:

1. client-server architecture

So it does not work with for example PUB/SUB sockets, it is based on REQ/REP.

2. stateless communication

So the server does not maintain the states of the clients. This means that you cannot use server side session storage and you have to authenticate every request. Your clients possibly send basic auth headers through an encrypted connection. (By large applications it is hard to maintain many sessions.)

3. usage of cache if you can

So you don't have to serve the same requests again and again.

4. uniform interface as common contract between client and server

The contract between the client and the server is not maintained by the server. In other words the client must be decoupled from the implementation of the service. You can reach this state by using standard solutions, like the IRI (URI) standard to identify resources, the HTTP standard to exchange messages, standard MIME types to describe the body serialization format, metadata (possibly RDF vocabs, microformats, etc.) to describe the semantics of different parts of the message body. To decouple the IRI structure from the client, you have to send hyperlinks to the clients in hypermedia formats like (HTML, JSON-LD, HAL, etc.). So a client can use the metadata (possibly link relations, RDF vocabs) assigned to the hyperlinks to navigate the state machine of the application through the proper state transitions in order to achieve its current goal.

For example when a client wants to send an order to a webshop, then it have to check the hyperlinks in the responses sent by the webshop. By checking the links it finds one described with the <http://schema.org/OrderAction>. The client know the schema.org vocab, so it understands that by activating this hyperlink it will send the order. So it activates the hyperlink and sends a `POST` `https://example.com/api/v1/order` message with the proper body. After that the service processes the message and responds with the result having the proper HTTP status header, for example `201 - created` by success. To annotate messages with

detailed metadata the standard solution to use an RDF format, for example [JSON-LD](#) with a REST vocab, for example [Hydra](#) and domain specific vocabs like [schema.org](#) or any other [linked data vocab](#) and maybe a custom application specific vocab if needed. Now this is not easy, that's why most ppl use HAL and other simple formats which usually provide only a REST vocab, but no linked data support.

#### 5. build a layered system to increase scalability

The REST system is composed of hierarchical layers. Each layer contains components which use the services of components which are in the next layer below. So you can add new layers and components effortless.

For example there is a client layer which contains the clients and below that there is a service layer which contains a single service. Now you can add a client side cache between them. After that you can add another service instance and a load balancer, and so on... The client code and the service code won't change.

#### 6. code on demand to extend client functionality

This constraint is optional. For example you can send a parser for a specific media type to the client, and so on... In order to do this you might need a standard plugin loader system in the client, or your client will be coupled to the plugin loader solution.

REST constraints result a highly scalable system in where the clients are decoupled from the implementations of the services. So the clients can be reusable, general just like the browsers on the web. The clients and the services share the same standards and vocabs, so they can understand each other despite the fact that the client does not know the implementation details of the service. This makes possible to create automated clients which can find and utilize REST services to achieve their goals. In long term these clients can communicate to each other and trust each other with tasks, just like humans do. If we add learning patterns to such clients, then the result will be one or more AI using the web of machines instead of a single server park. So at the end the dream of Berners Lee: the semantic web and the artificial intelligence will be reality. So in 2030 we end up terminated by the Skynet. Until then ... ;-)

Share Improve this answer

edited Sep 19, 2014 at 1:30

Follow

answered Nov 22, 2013 at 22:49



inf3rno

26.1k ● 12 ● 119 ● 205



23

[RESTful](#) (Representational state transfer) API programming is writing web applications in any programming language by following 5 basic software [architectural style](#) principles:





1. Resource (data, information).
2. [Unique global identifier](#) (all resources are unique identified by [URI](#)).
3. [Uniform interface](#) - use simple and standard interface (HTTP).
4. Representation - all communication is done by representation (e.g. [XML/JSON](#))
5. [Stateless](#) (every request happens in complete isolation, it's easier to cache and load-balance),

In other words you're writing simple point-to-point network applications over HTTP which uses verbs such as GET, POST, PUT or DELETE by implementing RESTful architecture which proposes standardization of the interface each “resource” exposes. It is nothing that using current features of the web in a simple and effective way (highly successful, proven and distributed architecture). It is an alternative to more complex mechanisms like [SOAP](#), [CORBA](#) and [RPC](#).

RESTful programming conforms to Web architecture design and, if properly implemented, it allows you to take the full advantage of scalable Web infrastructure.

Share Improve this answer

edited Jun 15, 2014 at 19:08

Follow

answered Jun 15, 2014 at 19:02



[kenorb](#)

166k ● 94 ● 706 ● 773



22



Here is my basic outline of REST. I tried to demonstrate the thinking behind each of the components in a RESTful architecture so that understanding the concept is more intuitive. Hopefully this helps demystify REST for some people!

REST (Representational State Transfer) is a design architecture that outlines how networked resources (i.e. nodes that share information) are designed and addressed. In general, a RESTful architecture makes it so that the client (the requesting machine) and the server (the responding machine) can request to read, write, and update data without the client having to know how the server operates and the server can pass it back without needing to know anything about the client. Okay, cool...but how do we do this in practice?

- The most obvious requirement is that there needs to be a universal language of some sort so that the server can tell the client what it is trying to do with the request and for the server to respond.
- But to find any given resource and then tell the client where that resource lives, there needs to be a universal way of pointing at resources. This is where Universal Resource Identifiers (URIs) come in; they are basically unique addresses to find the resources.

But the REST architecture doesn't end there! While the above fulfills the basic needs of what we want, we also

want to have an architecture that supports high volume traffic since any given server usually handles responses from a number of clients. Thus, we don't want to overwhelm the server by having it remember information about previous requests.

- Therefore, we impose the restriction that each request-response pair between the client and the server is independent, meaning that the server doesn't have to remember anything about previous requests (previous states of the client-server interaction) to respond to a new request. This means that we want our interactions to be stateless.
- To further ease the strain on our server from redoing computations that have already been recently done for a given client, REST also allows caching. Basically, caching means to take a snapshot of the initial response provided to the client. If the client makes the same request again, the server can provide the client with the snapshot rather than redo all of the computations that were necessary to create the initial response. However, since it is a snapshot, if the snapshot has not expired--the server sets an expiration time in advance--and the response has been updated since the initial cache (i.e. the request would give a different answer than the cached response), the client will not see the updates until the cache expires (or the cache is cleared) and the response is rendered from scratch again.

- The last thing that you'll often hear about RESTful architectures is that they are layered. We have actually already been implicitly discussing this requirement in our discussion of the interaction between the client and server. Basically, this means that each layer in our system interacts only with adjacent layers. So in our discussion, the client layer interacts with our server layer (and vice versa), but there might be other server layers that help the primary server process a request that the client does not directly communicate with. Rather, the server passes on the request as necessary.

Now, if all of this sounds familiar, then great. The Hypertext Transfer Protocol (HTTP), which defines the communication protocol via the World Wide Web is an implementation of the abstract notion of RESTful architecture (or an implementation of the abstract REST class if you're an OOP fanatic like me). In this implementation of REST, the client and server interact via GET, POST, PUT, DELETE, etc., which are part of the universal language and the resources can be pointed to using URLs.

[Share](#) [Improve this answer](#)

[edited Nov 17, 2020 at 21:56](#)

[Follow](#)

answered Mar 31, 2017 at 3:12



[Kal](#)

1,196 ● 20 ● 24



19

If I had to reduce the original dissertation on REST to just 3 short sentences, I think the following captures its essence:



1. Resources are requested via URLs.
2. Protocols are limited to what you can communicate by using URLs.
3. Metadata is passed as name-value pairs (post data and query string parameters).



After that, it's easy to fall into debates about adaptations, coding conventions, and best practices.

Interestingly, there is no mention of HTTP POST, GET, DELETE, or PUT operations in the dissertation. That must be someone's later interpretation of a "best practice" for a "uniform interface".

When it comes to web services, it seems that we need some way of distinguishing WSDL and SOAP based architectures which add considerable overhead and arguably much unnecessary complexity to the interface. They also require additional frameworks and developer tools in order to implement. I'm not sure if REST is the best term to distinguish between common-sense

interfaces and overly engineered interfaces such as WSDL and SOAP. But we need something.

Share Improve this answer

answered Feb 1, 2012 at 17:23

Follow



Nathan Andelin

199 ● 1 ● 2



18

REST is an architectural pattern and style of writing distributed applications. It is not a programming style in the narrow sense.



Saying you use the REST style is similar to saying that you built a house in a particular style: for example Tudor or Victorian. Both REST as an software style and Tudor or Victorian as a home style can be defined by the qualities and constraints that make them up. For example REST must have Client Server separation where messages are self-describing. Tudor style homes have Overlapping gables and Roofs that are steeply pitched with front facing gables. You can read Roy's dissertation to learn more about the constraints and qualities that make up REST.

REST unlike home styles has had a tough time being consistently and practically applied. This may have been intentional. Leaving its actual implementation up to the designer. So you are free to do what you want so as long as you meet the constraints set out in the dissertation you are creating REST Systems.

Bonus:

The entire web is based on REST (or REST was based on the web). Therefore as a web developer you might want aware of that although it's not necessary to write good web apps.

Share Improve this answer

answered Feb 1, 2012 at 21:20

Follow



suing

2,878 ● 2 ● 17 ● 18



17



I think the point of restful is the **separation of the statefulness into a higher layer** while making use of the internet (protocol) as a **stateless transport layer**. Most other approaches mix things up.

It's been the best practical approach to handle the fundamental changes of programming in internet era.

Regarding the fundamental changes, Erik Meijer has a discussion on show here:

[http://www.infoq.com/interviews/erik-meijer-programming-language-design-effects-purity#view\\_93197](http://www.infoq.com/interviews/erik-meijer-programming-language-design-effects-purity#view_93197) . He

summarizes it as the five effects, and presents a solution by designing the solution into a programming language. The solution, could also be achieved in the platform or system level, regardless of the language. The restful could be seen as one of the solutions that has been very successful in the current practice.

With restful style, you get and manipulate the state of the application across an unreliable internet. If it fails the current operation to get the correct and current state, it

needs the zero-validation principal to help the application to continue. If it fails to manipulate the state, it usually uses multiple stages of confirmation to keep things correct. In this sense, rest is not itself a whole solution, it needs the functions in other part of the web application stack to support its working.

Given this view point, the rest style is not really tied to internet or web application. It's a fundamental solution to many of the programming situations. It is not simple either, it just makes the interface really simple, and copes with other technologies amazingly well.

Just my 2c.

Edit: Two more important aspects:

- **Statelessness** is misleading. It is about the restful API, not the application or system. The system needs to be stateful. Restful design is about designing a stateful system based on a stateless API. Some [quotes from another QA](#):
  - REST, operates on resource representations, each one identified by an URL. These are typically not data objects, but **complex objects abstractions**.
  - REST stands for "representational state transfer", which means it's all about communicating and modifying **the state** of some resource in a system.



- **Idempotence:** [An often-overlooked part of REST is the idempotency of most verbs. That leads to robust systems and less interdependency of exact interpretations of the semantics.](#)

Share Improve this answer

edited Apr 30, 2018 at 16:18

Follow

answered Jul 3, 2014 at 17:30



minghua

6,561 ● 7 ● 47 ● 78

- 
- 1 A **MVC** viewpoint: The blog [Rest Worst Practices](#) suggested not to **conflating models and resources**. The book [Two Scoops of django](#) suggests that the Rest API is the view, and not to mix business logic into the view. The code for the app should remain in the app. – minghua Jun 25, 2015 at 6:20 ✎
- 
- 1 Another good article: [Wikipedia about Resource-Oriented Architecture](#) – minghua Jun 25, 2015 at 15:14 ✎
- 



REST defines 6 architectural constraints which make any web service – a **true RESTful API**.

15



1. Uniform interface
2. Client–server
3. Stateless
4. Cacheable
5. Layered system



## 6. Code on demand (optional)

<https://restfulapi.net/rest-architectural-constraints/>

Share Improve this answer

answered Oct 2, 2017 at 21:23

Follow



Jaider

14.9k ● 5 ● 80 ● 85

---

Fielding added [some further rules](#) RESTful APIs/clients have to adhere – [Roman Vottner](#) Oct 2, 2017 at 22:09

---



15

Old question, newish way of answering. There's a lot of misconception out there about this concept. I always try to remember:



1. Structured URLs and Http Methods/Verbs are not the definition of restful programming.



2. JSON is not restful programming



3. RESTful programming is not for APIs

I define restful programming as

An application is restful if it provides resources (being the combination of data + state transitions controls) in a media type the client understands

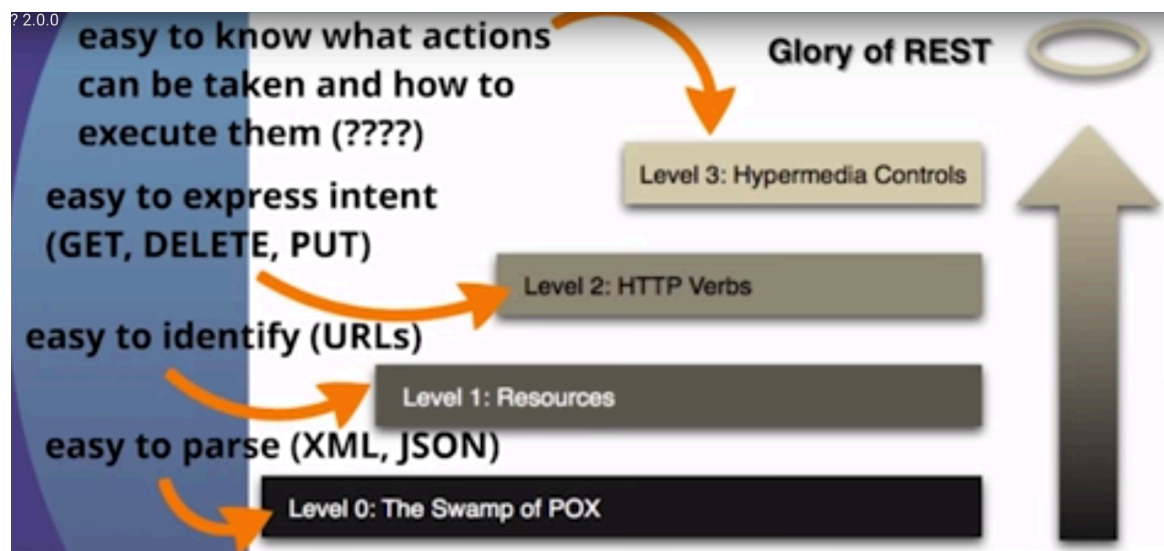
To be a restful programmer you must be trying to build applications that allow actors to do things. Not just exposing the database.

State transition controls only make sense if the client and server agree upon a media type representation of the resource. Otherwise there's no way to know what's a control and what isn't and how to execute a control. IE if browsers didn't know `<form>` tags in html then there'd be nothing for you to submit to transition state in your browser.

I'm not looking to self promote, but i expand on these ideas to great depth in my talk

<http://techblog.bodybuilding.com/2016/01/video-what-is-restful-200.html> .

An excerpt from my talk is about the often referred to richardson maturity model, i don't believe in the levels, you either are RESTful (level 3) or you are not, but what i like to call out about it is what each level does for you on your way to RESTful



Share Improve this answer

edited Feb 28, 2019 at 9:03

Follow

answered Nov 30, 2016 at 20:05



Chris DaMour

3,980 ● 33 ● 37

---

I am working my way through the content about APIs specifically, REST or RESTful APIs. And, although there are many upvoted answers above, I find yours particularly interesting not because of the immediate content within this SO answer (although it is interesting as well) but, because of the video in your shared blog post. It is outstanding. Your insights on Level 3 / Hypermedia API after an actual implementation was invaluable. It certainly provides points for considerations while authoring APIs. – [Talha Akbar](#) Aug 29, 2021 at 15:20 ✎

---



This answer is for absolute beginners, let's know about most used API architecture today.

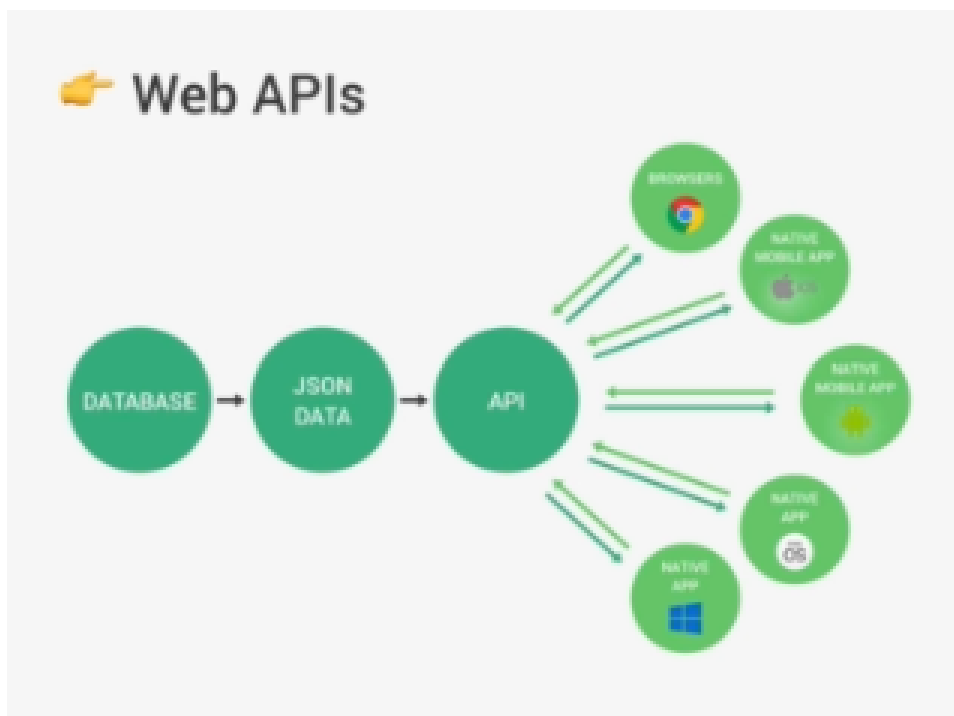
14



To understand Restful programming or Restful API. First, you have to understand what API is, on a very high-level API stands for Application Programming Interface, it's basically a piece of software that can be used by another piece of software in order to allow applications to talk to each other.



The most widely used type of API in the globe is web APIs while an app that sends data to a client whenever a request comes in.



In fact, APIs aren't only used to send data and aren't always related to web development or javascript or python or any programming language or framework.

The application in API can actually mean many different things as long as the piece of software is relatively stand-alone. Take for example, the File System or the HTTP Modules we can say that they are small pieces of software and we can use them, we can interact with them by using their API. For example when we use the read file function for a file system module of any programming language, we are actually using the `file_system_reading` API. Or when we do DOM manipulation in the browser, we're not really using the JavaScript language itself, but rather, the DOM API that browser exposes to us, so it gives us access to it. Or even another example let's say we create a class in any programming language like Java and then add some public methods or properties to it, these methods will then be the API of each object created from that class because we are giving other pieces of

software the possibility of interacting with our initial piece of software, the objects in this case. So, API has actually a broader meaning than just building web APIs.

## **Now let's take a look at the REST Architecture to build APIs.**


**REST** which stands for Representational State Transfer is basically a way of building web APIs in a logical way, making them easy to consume for ourselves or for others.

To build Restful APIs following the REST Architecture, we just need to follow a couple of principles. 1. We need to separate our API into logical resources. 2. These resources should then be exposed by using resource-based URLs. 3. To perform different actions on data like reading, creating, or deleting data the API should use the right HTTP methods and not the URL. 4. Now the data that we actually send back to the client or that we received from the client should usually use the JSON data format, where some formatting standard applied to it. 5. Finally, another important principle of REST APIs is that they must be stateless.

- 1 Separate API into logical **resources**
- 2 Expose structured, **resource-based URLs**
- 3 Use **HTTP methods** (verbs)
- 4 Send data as **JSON** (usually)
- 5 Be **stateless**

**Separate APIs into logical resources:** The key abstraction of information in REST is a resource, and therefore all the data that we wanna share in the API should be divided into logical resources. What actually is a resource? Well, in the context of REST it is an object or a representation of something which has some data associated to it. For example, applications like tour-guide tours, or users, places, or reviews are of the example of logical resources. So basically any information that can be named can be a resource. Just has to name, though,

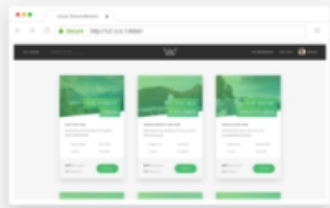
not a verb.

**Resource:** Object or representation of something, which has data associated to it. Any information that can be **named** can be a resource.

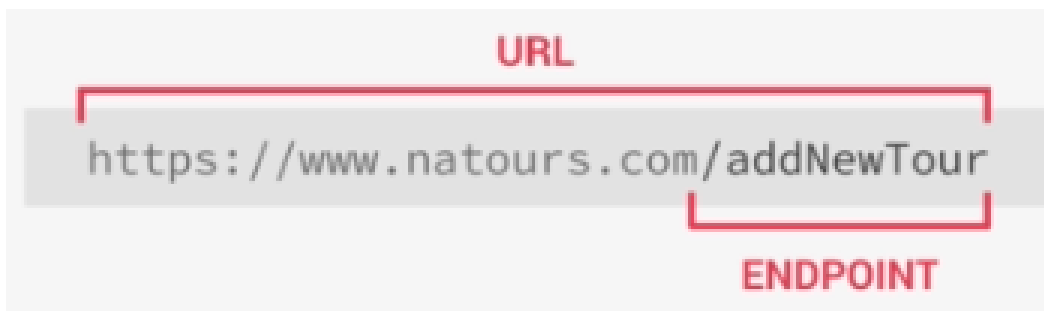
tours

users

reviews



**Expose Structure:** Now we need to expose, which means to make available, the data using some structured URLs, that the client can send a request to. For example something like this entire address is called the URL. and this / addNewTour is called and API Endpoint.



Our API will have many different endpoints just like bellow

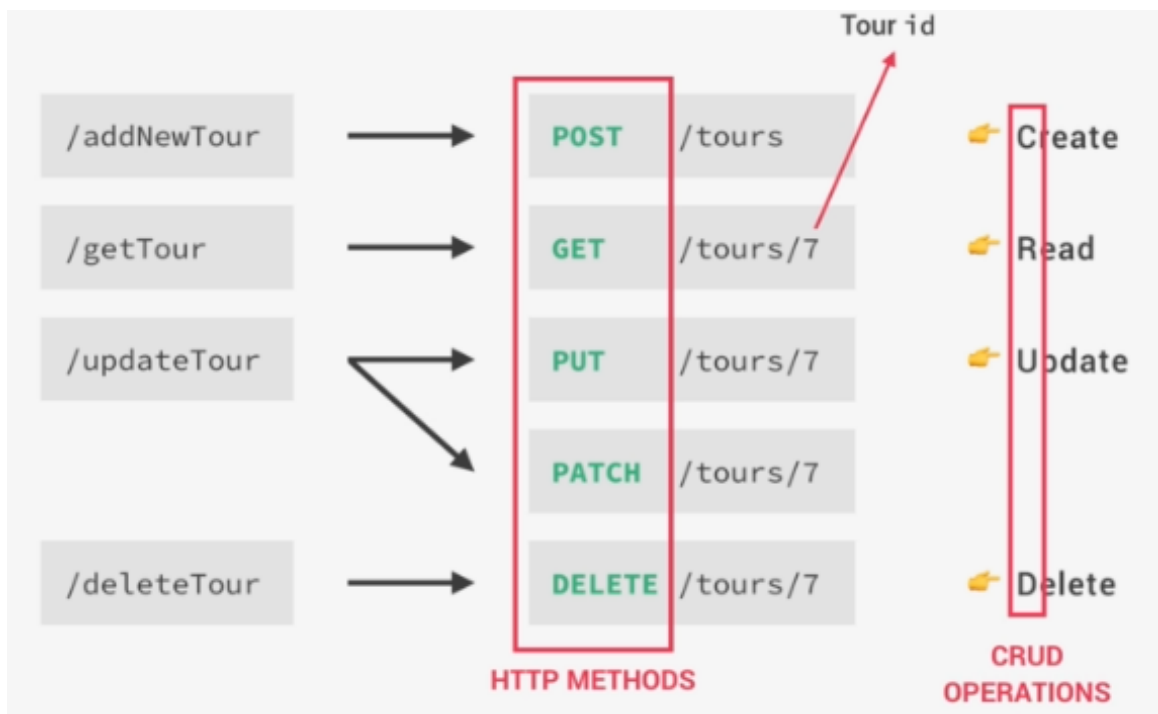
```
https://www.tourguide.com/addNewTour
https://www.tourguide.com/getTour
https://www.tourguide.com/updateTour
https://www.tourguide.com/deleteTour
https://www.tourguide.com/getRoursByUser
https://www.tourguide.com/deleteToursByUser
```

Each of these API will send different data back to the client on also perform different actions. Now there is **something very wrong** with these endpoints here because they really don't follow the third rule which says that we should only use the HTTP methods in order to perform actions on data. So endpoints should only contain our resources and not the actions that we are



performed on them because they will quickly become a nightmare to maintain.

How should we use these HTTP methods in practice? Well let's see how these endpoints should actually look like starting with /getTour. So this getTour endpoint is to get data about a tour and so we should simply name the endpoint /tours and send the data whenever a get request is made to this endpoint. So in other words, when a client uses a GET HTTP method to access the endpoint,



(we only have resources in the endpoint or in the URL and no verbs because the verb is now in the HTTP method, right? The common practice to always use the resource name in the plural which is why I wrote /tours nor /tour.) The convention is that when calling endpoint /tours will get back all the tours that are in a database, but if we only want the tour with one ID, let's say seven, we add that seven after another slash(/tours/7) or in a search

query (/tours?id=7), And of course, it could also be the name of a tour instead of the ID.

HTTP Methods: What's really important here is how the endpoint name is the exact same name for all.

```
GET: (for requesting data from the server.)
```

```
https://www.tourguide.com/tours/7
```

```
POST: (for sending data to the server.)
```

```
https://www.tourguide.com/tours
```

```
PUT/PATCH: (for updating requests for data to the server.) https://www.tourguide.com/tours/7
```

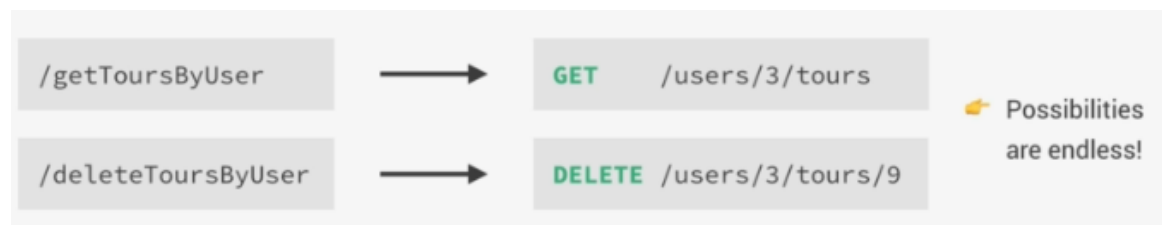
```
DELETE: (for deleting request for data to the server.)
```

```
https://www.tourguide.com/tours/7
```

The difference between PUT and PATCH-> By using PUT, the client is supposed to send the entire updated object, while with PATCH it is supposed to send only the part of the object that has been changed.

By using HTTP methods users can perform basic four CRUD operations, CRUD stands for Create, Read, Update, and Delete.

Now there could be a situation like a bellow:



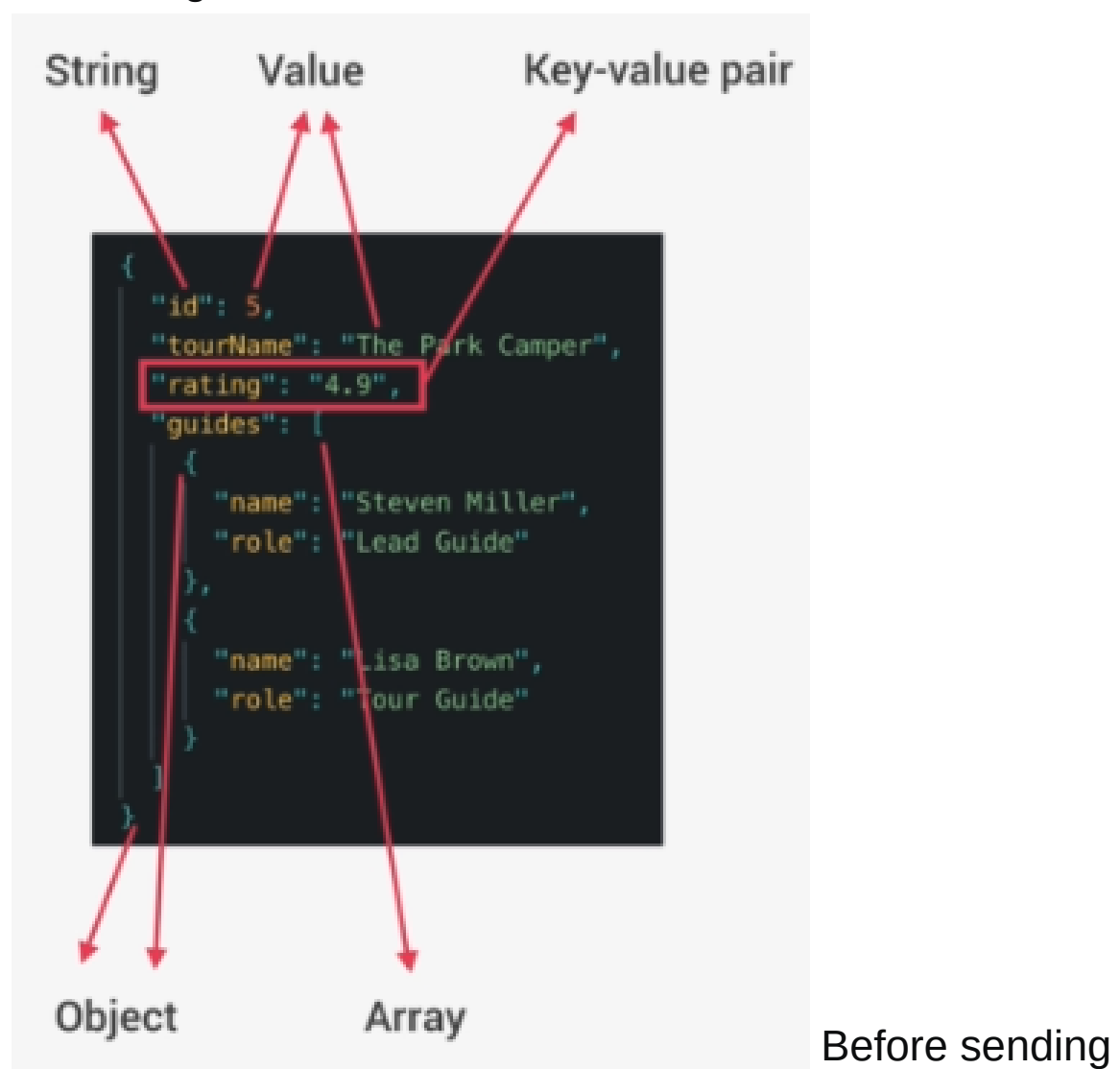
So, /getToursByUser can simply be translated to /users/tours, for user number 3 end point will be like

/users/3/tours.

if we want to delete a particular tour of a particular user then the URL should be like /users/3/tours/7, here user id:3 and tour id: 7.

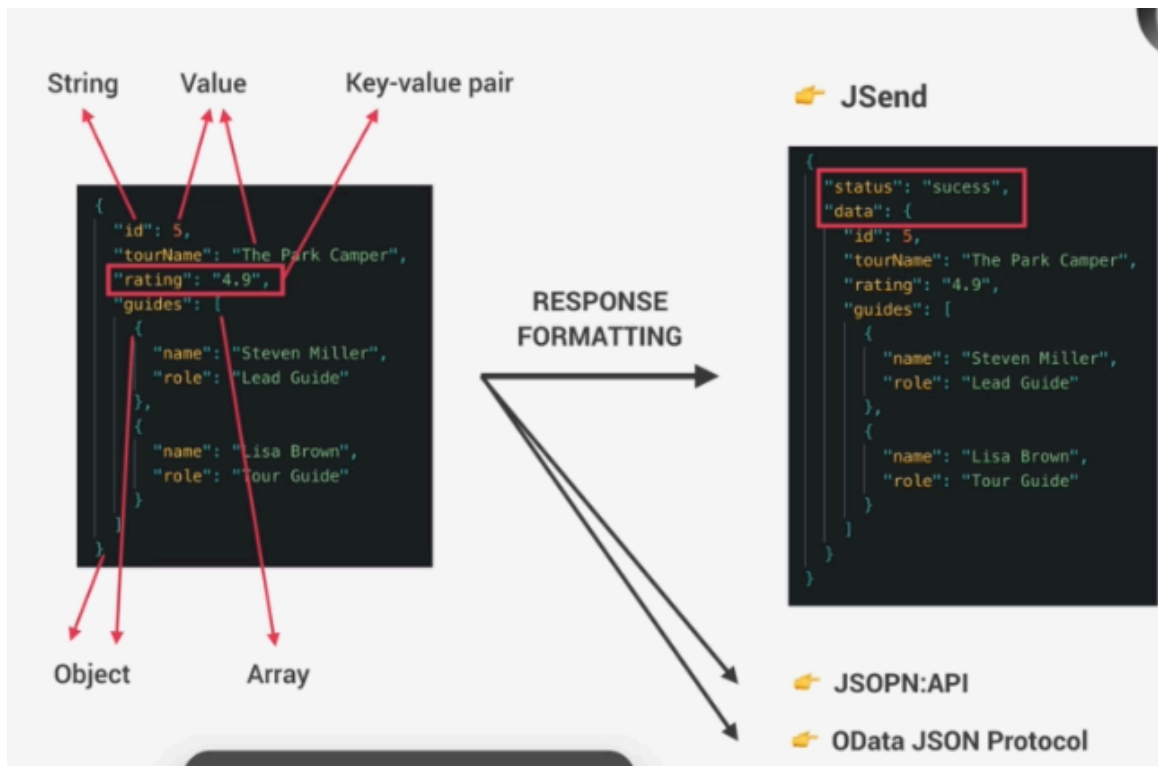
So there really are tons of possibilities of combining resources like this.

**Send data as JSON:** Now about data that the client actually receives, or that the server receives from the client, usually we use the JSON Data Format. A typical JSON might look like below:



JSON Data we usually do some simple response formatting, there are a couple of standards for this, but

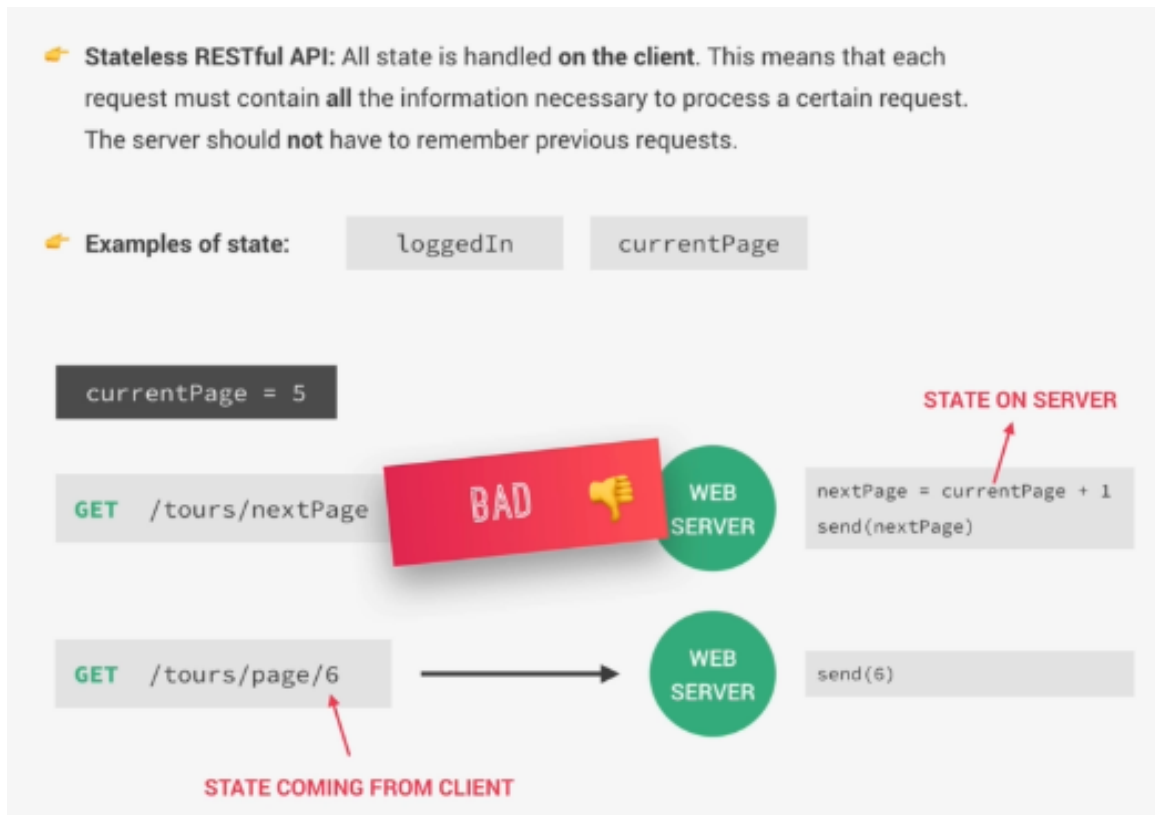
one of the very simple ones called Jsend. We simply create a new object, then add a status message to it in order to inform the client whether the request was a success, fail, or error. And then we put our original data into a new object called Data.



Wrapping the data into an additional object like we did here is called Enveloping, and it's a common practice to mitigate some security issues and other problems.

**Restful API should always be stateless:** Finally a RESTful API should always be stateless meaning that, in a stateless RESTful API all state is handled on the client side no on the server. And state simply refers to a piece of data in the application that might change over time. For example, whether a certain user is logged in or on a page with a list with several pages what the current page is? Now the fact that the state should be handled on the client means that each request must contain all the

information that is necessary to process a certain request on the server. So the server should never ever have to remember the previous request in order to process the current request.



Let's say that currently we are on page five and we want to move forward to page six. So we could have a simple endpoint called `/tours/nextPage` and submit a request to server, but the server would then have to figure out what the current page is, and based on that server will send the next page to the client. In other words, the server would have to remember the previous request. This is what exactly we want to avoid in RESTful APIs.

Instead of this case, we should create a `/tours/page` endpoint and paste the number six to it in order to request page number six `/tours/page/6`. So the server

doesn't have to remember anything in, all it has to do is to send back data for page number six as we requested.

Statelessness and Statefulness which is the opposite are very important concepts in computer science and applications in general

Share Improve this answer

answered May 21, 2020 at 11:09

Follow



Rafiq

11.3k ● 5 ● 42 ● 44



This is amazingly long "discussion" and yet quite confusing to say the least.

13

IMO:



1) There is no such a thing as restful programing, without a big joint and lots of beer :)



2) **Representational State Transfer (REST)** is an architectural style specified in [the dissertation of Roy Fielding](#). It has a number of constraints. If your Service/Client respect those then it is **RESTful**. This is it.

You can summarize(significantly) the constraints to :

- stateless communication
- respect HTTP specs (if HTTP is used)
- clearly communicates the content formats transmitted

- use hypermedia as the engine of application state

There is another [very good post](#) which explains things nicely.

A lot of answers copy/pasted valid information mixing it and adding some confusion. People talk here about levels, about RESTful URIs(there is not such a thing!), apply HTTP methods GET,POST,PUT ... REST is not about that or not only about that.

For example links - it is nice to have a beautifully looking API but at the end the client/server does not really care of the links you get/send it is the content that matters.

In the end **any RESTful client should be able to consume to any RESTful service as long as the content format is known.**

Share Improve this answer

edited Jan 24, 2017 at 10:54

Follow

answered Jan 13, 2017 at 14:02



kalin

3,566 ● 2 ● 26 ● 31



REST === HTTP analogy is not correct until you do not stress to the fact that it "MUST" be [HATEOAS](#) driven.

11

Roy himself cleared it [here](#).



A REST API should be entered with no prior knowledge beyond the initial URI (bookmark) and set of standardized media types that are appropriate for the intended audience (i.e., expected to be understood by any client that might use the API). From that point on, all application state transitions must be driven by client selection of server-provided choices that are present in the received representations or implied by the user's manipulation of those representations. The transitions may be determined (or limited by) the client's knowledge of media types and resource communication mechanisms, both of which may be improved on-the-fly (e.g., code-on-demand).

[Failure here implies that out-of-band information is driving interaction instead of hypertext.]

Share Improve this answer

Follow

edited Aug 23, 2017 at 10:00



Priyantha

5,063 ● 6 ● 29 ● 46

answered Jun 3, 2016 at 11:35



Lokesh Gupta

472 ● 5 ● 8

---

doesn't answer the question as well as the others, but +1 for information that is relevant! – [CybeX](#) Oct 2, 2017 at 19:06

---

I think this answers the question too, but for example statelessness is missing from it. Every constraint is important... The standard media type part is not always true. I mean there are layers of understanding. For example if you use RDF, then the media type can be understood, but the meaning of the content not. So the client needs to know the vocabulary as well. Some people are experimenting with this





10



**REST** stands for **Representational state transfer**.

It relies on a stateless, client-server, cacheable communications protocol -- and in virtually all cases, the HTTP protocol is used.

REST is often used in mobile applications, social networking Web sites, mashup tools and automated business processes. The REST style emphasizes that interactions between clients and services is enhanced by having a limited number of operations (verbs). Flexibility is provided by assigning resources (nouns) their own unique universal resource indicators (URIs).

[Introduction about Rest](#)

Share Improve this answer

answered Nov 10, 2014 at 13:37

Follow



**GowriShankar**

1,654 ● 18 ● 31



9



The point of rest is that if we agree to use a common language for basic operations (the http verbs), the infrastructure can be configured to understand them and optimize them properly, for example, by making use of caching headers to implement caching at all levels.



With a properly implemented restful GET operation, it shouldn't matter if the information comes from your server's DB, your server's memcache, a CDN, a proxy's cache, your browser's cache or your browser's local storage. The fastest, most readily available up to date source can be used.

Saying that Rest is just a syntactic change from using GET requests with an action parameter to using the available http verbs makes it look like it has no benefits and is purely cosmetic. The point is to use a language that can be understood and optimized by every part of the chain. If your GET operation has an action with side effects, you have to skip all HTTP caching or you'll end up with inconsistent results.

Share Improve this answer

answered Feb 1, 2012 at 23:52

Follow



[Benoit Essiambre](#)

328 ● 3 ● 7

---

5 "Saying that Rest is just a syntactic change... makes it look like it has no benefits and is purely cosmetic" --- that's exactly why I am reading answers here on SO. Note that you did not explain, why REST is not purely cosmetic.

– [Sergey Orshanskiy](#) Oct 8, 2013 at 17:14

---



9

*Talking* is more than simply *exchanging information*. A Protocol is actually designed so that no talking has to occur. Each party knows what their particular job is because it is specified in the protocol. Protocols allow for pure information exchange at the expense of having any



changes in the possible actions. Talking, on the other hand, allows for one party to ask what further actions can be taken from the other party. They can even ask the same question twice and get two different answers, since the State of the other party may have changed in the interim. **Talking is RESTful architecture.** Fielding's thesis specifies the architecture that one would have to follow if one wanted to allow machines to *talk* to one another rather than simply *communicate*.

Share Improve this answer

answered Dec 6, 2014 at 6:54

Follow



qmckinsey

305 ● 1 ● 3 ● 12



9

There is not such notion as "RESTful programming" per se. It would be better called RESTful paradigm or even better RESTful architecture. It is not a programming language. It is a paradigm.



[From Wikipedia:](#)



In computing, representational state transfer (REST) is an architectural style used for web development.

Share Improve this answer

edited Aug 26, 2016 at 20:35

Follow

answered Aug 24, 2016 at 17:57



ACV

10.5k ● 5 ● 81 ● 89



5



This is very less mentioned everywhere but the **Richardson's Maturity Model** is one of the best methods to actually judge how Restful is one's API. More about it here:

[Richardson's Maturity Model](#)

Share Improve this answer

answered Aug 29, 2017 at 11:55

Follow



Krishna Ganeriwal

1,979 ● 20 ● 17

If you look at the constraints Fielding put on REST you will clearly see that an API needs to have reached Layer 3 of the RMM in order to be viewed as RESTful, though this is simply not enough actually as there are still enough possibilities to fail the REST idea - the decoupling of clients from server APIs. Sure, Layer 3 fulfills the HATEOAS constraint but it is still easy to break the requirements and to couple clients tightly to a server (i.e. by using typed resources)

– [Roman Vottner](#) Oct 2, 2017 at 22:21



3



What is [API Testing](#)?

API testing utilizes programming to send calls to the API and get the yield. It testing regards the segment under test as a black box. The objective of API testing is to confirm right execution and blunder treatment of the part preceding its coordination into an application.



## REST API

REST: Representational State Transfer.

- It's an arrangement of functions on which the testers performs requests and receive responses. In REST API interactions are made via HTTP protocol.
- REST also permits communication between computers with each other over a network.
- For sending and receiving messages, it involves using HTTP methods, and it does not require a strict message definition, unlike Web services.
- REST messages often accepts the form either in form of XML, or JavaScript Object Notation (JSON).

### 4 Commonly Used API Methods:-

1. GET: – It provides read only access to a resource.
2. POST: – It is used to create or update a new resource.
3. PUT: – It is used to update or replace an existing resource or create a new resource.
4. DELETE: – It is used to remove a resource.

### Steps to Test API Manually:-

To use API manually, we can use browser based REST API plugins.

1. Install POSTMAN(Chrome) / REST(Firefox) plugin

2. Enter the API URL
3. Select the REST method
4. Select content-Header
5. Enter Request JSON (POST)
6. Click on send
7. It will return output response

### [Steps to Automate REST API](#)

Share Improve this answer

Follow

edited Aug 1, 2016 at 12:18



Krrishnaaaa

689 ● 11 ● 23

answered Aug 1, 2016 at 6:42



kkashyap1707

531 ● 2 ● 8 ● 16

---

5 this is not even a proper answer – [therealprashant](#) Aug 5, 2016 at 7:17

---



2

I would say that an important building block in understanding REST lies in the endpoints or mappings, such as `/customers/{id}/balance`.



You can imagine such an endpoint as being the connecting pipeline from the website (front-end) to your database/server (back-end). Using them, the front-end can perform back-end operations which are defined in the



corresponding methods of any REST mapping in your application.

Share Improve this answer

answered Mar 27, 2019 at 10:11

Follow



Kürsat Aydınli

121 ● 6

1

2

Next



**Highly active question.** Earn 10 reputation (not counting the [association bonus](#)) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.