FREE eBook

LEARNING

rest

Free unaffiliated eBook created from
Stack Overflow contributors.

#rest

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: rest

It is an unofficial and free rest ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official rest.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with rest

Remarks

This section provides an overview of what rest is, and why a developer might want to use it.

It should also mention any large subjects within rest, and link out to the related topics. Since the Documentation for rest is new, you may need to create initial versions of those related topics.

Examples

REST Overview

REST stands for **RE**presentational **S**tate **T**ransfer and was coined by **Roy Fielding** in his doctoral thesis Architectural Styles and the Design of Network-based Software Architectures. In it he identifies specific architectural principles like:

- Addressable Resources: the key abstraction of information and data in REST is a resource and each resource must be addressable via a URI.
- A uniform, constrained interface: use of small set of well-defined methods to manipulate our resources.
- Representation-oriented: A resource referenced by one URI can have different formats and different platforms need different formats, for example browsers need HTML, JavaScript needs JSON and Java applications may need XML, JSON, CSV, text, etc. So we interact with services using representation of that service.
- Communicate statelessly: stateless applications are easier to scale.
- Hypermedia As The Engine Of Application State: let our data formats drive state transitions in our applications.

The set of these architectural principles is called REST. The concepts of REST are inspired by that of HTTP. Roy Fielding who gave REST to us is also one of the authors of HTTP specifications.

Web Services and RESTful Web Services are services which are exposed to internet for programmatic access. They are online api which we can call from our code. There are two types of "Big" web services SOAP and REST web services.

RESTful Web Services: Web services which are written by applying the REST architectural concepts are called RESTful Web Services, which focus on system resources and how state of a resource can be transferred over HTTP protocol to different clients.

This document is solely focused on RESTful web services so we will not get into the details of SOAP WS.

There are no strict rules while designing RESTful web services like

- No protocol standard
- No communication channel standard
- No service definition standard

But SOAP has strict rules for all these. All SOAP web services follow SOAP specification which dictates what every SOAP web services should be. This specification is developed and managed by a committee and if SOAP WS does not follow even a single rule then by definition it is not SOAP.

Concepts of RESTful Web Services

There are few guidelines which needs to be considered while designing/developing RESTful api:

1. Resource based locations/URI
2. Proper use of HTTP methods
3. HATEOAS (Hypermedia As The Engine Of Application State)

The main approach while developing RESTful APIs should be to make the API “as RESTful as possible”.

REST over HTTP

REST is a **protocol-agnostic** architecture proposed by Roy Fielding in his dissertation (chapter 5 being the presentation of REST), that generalizes the proven concept of web browsers as clients in order to decouple clients in a distributed system from servers.

In order for a service or API to be RESTful, it must adhere to given constraints like:

- Client-server
- Stateless
- Cacheable
- Layered system
- Uniform interface
    - Resources identification
    - Resources representation
    - Self-descriptive messages
    - Hypermedia

Besides the constraints mentioned in Fielding's dissertation, in his blog post *REST APIs must be hypertext-driven*, Fielding clarified that **just invoking a service via HTTP does not make it RESTful**. A service should therefore also respect further rules which are summarized as follow:

- The API should adhere to and not violate the underlying protocol. Although REST is used via HTTP most of the time, it is not restricted to this protocol.

- Strong focus on resources and their presentation via media-types.

- Clients should not have initial knowledge or assumptions on the available resources or their returned state ("typed" resource) in an API but learn them on the fly via issued requests and analyzed responses. This gives the server the opportunity to move around or rename resources easily without breaking a client implementation.

Richardson Maturity Model

The Richardson Maturity Model is a way to apply REST constraints over HTTP in order to obtain RESTful web services.

Leonard Richardson divided applications into these 4 layers:

- Level 0: use of HTTP for the transport
- Level 1: use of URL to identify resources
- Level 2: use of HTTP verbs and statuses for the interactions
- Level 3: use of HATEOAS

As the focus is on the representation of the state of a resource, supporting multiple representations for the same resource is encouraged. A representation could therefore present an overview of the resource state while an other returns the full details of the same resource.

Note also that given Fielding constraints, **an API is effectively RESTful only once the 3rd level of the RMM is implemented**.

HTTP requests and responses

An HTTP request is:

- A verb (aka method), most of the time one of GET, POST, PUT, DELETE or PATCH
- A URL
- Headers (key-value pairs)
- Optionally a body (aka payload, data)

An HTTP response is:

- A status, most of the time one of 2xx (successful), 4xx (client error) or 5xx (server error)
- Headers (key-value pairs)
- A body (aka payload, data)

HTTP verbs characteristics:

- Verbs that have a body: `POST`, `PUT`, `PATCH`
- Verbs that must be safe (i.e. that mustn't modify resources): `GET`
- Verbs that must be idempotent (i.e. that mustn't affect resources again when run multiple times): `GET` (nullipotent), `PUT`, `DELETE`

```
        body  safe  idempotent
GET      ✗  ✔  ✔
POST  ✔  ✗  ✗
PUT   ✔  ✗  ✔
DELETE ✗  ✗  ✔
PATCH  ✔  ✗  ✗
```

Consequently, HTTP verbs can be compared to the CRUD functions:

- **C**reate: `POST`
- **R**EAD: `GET`
- **U**PDATE: `PUT`, `PATCH`
- **D**ELETE: `DELETE`

Note that **a `PUT` request asks clients to send the entire resource** with the updated values. To partially update a resource, a `PATCH` verb could be used (see *How to partially update a resource?*).

Usual HTTP response statuses

Success

- 201 (CREATED): resource has been created
- 202 (ACCEPTED): request accepted, but process still in progress
- 204 (NO CONTENT): request fulfilled, and no additional content
- Otherwise: 200 (OK)

Redirection

- 304 (NOT MODIFIED): client can use the cached version it has of the requested resource

Client errors

- 401 (UNAUTHORIZED): an anonymous request accesses a protected API
- 403 (FORBIDDEN): an authenticated request hasn't enough rights to access a protected API
- 404 (NOT FOUND): resource not found
- 409 (CONFLICT): resource state in conflict (e.g. a user trying to create an account with an already registered email)
- 410 (GONE): same as 404, but the resource existed
- 412 (PRECONDITION FAILED): request tries to modify a resource that is in an unexpected state
- 422 (UNPROCESSABLE ENTITY): request payload is syntactically valid, but semantically erroneous (e.g. a required field that has not been valued)
- 423 (LOCKED): resource is locked
- 424 (FAILED DEPENDENCY): requested action depended on another action that failed
- 429 (TOO MANY REQUESTS): user sent too many requests in a given amount of time
- Otherwise: 400 (BAD REQUEST)

Server errors

- 501 (NOT IMPLEMENTED): server does not support the functionality required to fulfill the request
- 503 (SERVICE UNAVAILABLE): server is currently unable to handle the request due to a temporary overload or scheduled maintenance
- 507 (INSUFFICIENT STORAGE): server is unable to store the representation needed to successfully complete the request
- Otherwise: 500 (INTERNAL SERVER ERROR)

Notes

Nothing stops you from adding a body to erroneous responses, to make the rejection clearer for clients. For example, the 422 (UNPROCESSABLE ENTITY) is a bit vague: response body should provide the reason why the entity could not be processed.

HATEOAS

Each resource must provide hypermedia to the resources it is linked to. A link is *at least* composed by:

- A `rel` (for **rel**ation, aka name): describes the relation between the main resource and the linked one(s)
- A `href`: the URL targeting the linked resource(s)

Additional attributes can be used as well to help with deprecation, content negotiation, etc.

Cormac Mulhall explains that *the client should decide what HTTP verb to use based on what it is trying to do*. When in doubt, the API documentation should anyway help you understanding the available interactions with all hypermedia.

Media types

Media types help having self-descriptive messages. They play the part of the contract between clients and servers, so that they can exchange resources and hypermedias.

Although `application/json` and `application/xml` are quite popular media-types, they do not contain much semantics. They just describe the overall syntax used in the document. More specialized media-types that support the HATEOAS requirements should be used (or extended through vendor media types), such as:

- Atom
- RSS 2.0
- HAL
- collection+json
- JSON-LD

- Siren

A client tells a server which media types it understands by adding the `Accept` header to his request, for example:

```
Accept: application/hal+json
```

If the server isn't able to produce requested resource in such a representation, it returns a 406 (NOT ACCEPTABLE). Otherwise, it adds the media type in the `Content-Type` header of the response holding the represented resource, for example:

```
Content-Type: application/hal+json
```

Stateless > stateful

Why?

A stateful server implies that the clients sessions are stored in a server-instance-local storage (almost always in web server sessions). This starts to be an issue when trying to scale horizontally : if you hide several server instances behind a load balancer, if one client is first dispatched to *instance #1* when signing in, but afterwards to *instance #2* when fetching a protected resource for example, then the second instance will handle the request as an anonymous one, as **the client session has been stored locally in *instance #1***.

Solutions have been found to tackle this issue (e.g. by configuring session replication and/or sticky session), but the REST architecture proposes another approach: just don't make you server stateful, *make it stateless*. According to Fielding:

> 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.**

In other words, a request must be handled exactly the same way, regardless of whether it is dispatched to *instance #1* or *instance #2*. This is why stateless applications are considered *easier to scale*.

How?

A common approach is a token-based authentication, especially with the trendy JSON Web Tokens. Note that JWT still have some issues though, particularly concerning invalidation and automatic prolongation of expiration (i.e. the *remember me* feature).

Side notes

Using cookies or headers (or anything else) has nothing to do with whether the server is stateful or stateless: these are just media that are here used to transport tokens (session identifier for stateful servers, JWT, etc.), nothing more.

When a RESTful API is only used by browsers, (HttpOnly and secure) cookies can be quite convenient as browsers will automatically attach them to outgoing requests. It's worth mentioning that **if you opt for cookies, be aware of CSRF** (a nice way of preventing it is to have the clients generate and send the same unique secret value in both a cookie and a custom HTTP header).

Cacheable API with conditional requests

With the `Last-Modified` header

The server can provide a `Last-Modified` date header to the responses holding resources that are cacheable. Clients should then store this date together with the resource.

Now, each time clients request the API to read the resource, they can add to their requests an `If-Modified-Since` header containing the latest `Last-Modified` date they received and stored. The server has then to compare the request's header and the actual *last modified* date of the resource. If they are equal, the server returns a 304 (NOT MODIFIED) with an empty body: the requesting client should use the currently cached resource it has.

Also, when clients request the API to update the resource (i.e. with an unsafe verb), they can add an `If-Unmodified-Since` header. This helps dealing with race conditions: if the header and the actual *last modified* date are different, the server returns a 412 (PRECONDITION FAILED). The client should then read the new state of the resource before retrying to modify the resource.

With the `ETag` header

An ETag (entity tag) is an identifier for a specific state of a resource. It can be a MD5 hash of the resource for a strong validation, or a domain-specific identifier for a weak validation.

Basically, the process is the same as with the `Last-Modified` header: the server provides an `ETag` header to the responses holding resources that are cacheable, and clients should then store this identifier together with the resource.

Then, clients provide an `If-None-Match` header when they want to read the resource, containing the latest ETag they received and stored. The server can now return a 304 (NOT MODIFIED) if the header matches the actual ETag of the resource.

Again, clients can provide an `If-Match` header when they want to modify the resource, and the server has to return a 412 (PRECONDITION FAILED) if the provided ETag doesn't match the actual one.

Additional notes

ETag > date

If clients provide both date and ETag in their requests, the date must be ignored. From RFC 7232 (here and here):

> A recipient MUST ignore `If-Modified-Since`/`If-Unmodified-Since` if the request contains an `If-None-Match`/`If-Match` header field; the condition in `If-None-Match`/`If-Match` is considered to be a more accurate replacement for the condition in `If-Modified-Since`/`If-Unmodified-Since`, and the two are only combined for the sake of interoperating with older intermediaries that might not implement `If-None-Match`/`If-Match`.

Shallow ETags

Also, while it's quite obvious that the *last modified dates* are persisted along with the resources server-side, several approaches are available with ETag.

A usual approach is to implement shallow ETags: the server processes the request as if no conditional headers were given, but at the very end only, it generates the ETag of the response it is about to return (e.g. by hashing it), and compares it with the provided one. This is relatively easy to implement as only an HTTP interceptor is needed (and many implementations already exist depending on the server). That being said, it's worth mentioning that this approach will **save bandwidth but not server performance**:

> A **deeper implementation** of the ETag mechanism could potentially provide much greater benefits – such as serving some requests from the cache and not having to perform the computation at all – but the implementation would most definitely not be as simple, nor as pluggable as the shallow approach described here.

Common pitfalls

Why shouldn't I put verbs in a URLs?

HTTP is not RPC: *what makes HTTP significantly different from RPC is that **the requests are directed to resources***. After all, URL stands for Uniform *Resource* Locator, and a URL is a URI: a Uniform *Resource* Idenfitier. **The URL targets *the resource you want to deal with*, the HTTP method indicates *what you want to do with it*.** HTTP methods are also known as *verbs*: verbs in URLs makes then no sense. Note that HATEOAS relations shouldn't contain verbs neither, as links are targeting resources as well.

How to partially update a resource?

As `PUT` requests ask clients to send **the entire resource** with the updated values, `PUT /users/123`

cannot be used to simply update a user's email for example. As explained by William Durand in Please. Don't Patch Like An Idiot., several REST-compliant solutions are available:

- Expose the resource's properties and use the `PUT` method to send an updated value, as the `PUT` specification states that *partial content updates are possible by targeting a separately identified resource with state that overlaps a portion of the larger resource*:

```
PUT https://example.com/api/v1.2/users/123/email
body:
  new.email@example.com
```

- Use a `PATCH` request that contains a set of instructions describing how the resource must be modified (e.g. following JSON Patch):

```
PATCH https://example.com/api/v1.2/users/123
body:
  [
    { "op": "replace", "path": "/email", "value": "new.email@example.com" }
  ]
```

- Use a `PATCH` request containing a partial representation of the resource, as proposed in Matt Chapman's comment:

```
PATCH https://example.com/api/v1.2/users/123
body:
  {
    "email": "new.email@example.com"
  }
```

What about actions that don't fit into the world of CRUD operations?

Quoting Vinay Sahni in Best Practices for Designing a Pragmatic RESTful API:

> This is where things can get fuzzy. There are a number of approaches:
>
> 1. Restructure the action to appear like a field of a resource. This works if the action doesn't take parameters. For example an *activate* action could be mapped to a boolean `activated` field and updated via a PATCH to the resource.
>
> 2. Treat it like a sub-resource with RESTful principles. For example, GitHub's API lets you star a gist with `PUT /gists/:id/star` and unstar with `DELETE /gists/:id/star`.
>
> 3. Sometimes you really have no way to map the action to a sensible RESTful structure. For example, a multi-resource search doesn't really make sense to be applied to a specific resource's endpoint. In this case, `/search` would make the most sense even though it isn't a resource. This is OK - just do what's right from the perspective of the API consumer and make sure it's documented clearly to

avoid confusion.

Common practices

- API is documented. Tools are available to help you building your documentation, e.g. Swagger or Spring REST Docs.
- API is versioned, either via headers or through the URL:

```
https://example.com/api/v1.2/blogs/123/articles
                        ^^^^
```

- Resources have plural names:

```
https://example.com/api/v1.2/blogs/123/articles
                            ^^^^^     ^^^^^^^^
```

- URLs use kebab-case (words are lowercased and dash-separated):

```
https://example.com/api/v1.2/quotation-requests
                            ^^^^^^^^^^^^^^^^^^
```

- HATEOAS provides a "self" link to resources, targeting themselves:

```
{
  ...,
  _links: {
    ...,
    self: { href: "https://example.com/api/v1.2/blogs/123/articles/789" }
    ^^^^
  }
}
```

- HATEOAS relations use lowerCamelCase (words are lowercased, then capitalized except the first one, and spaces are omitted), to allow JavaScript clients to use the dot notation while respecting the JavaScript naming conventions when accessing the links:

```
{
  ...,
  _links: {
    ...,
    firstPage: { "href":
"https://example.com/api/v1.2/blogs/123/articles?pageIndex=1&pageSize=25" }
    ^^^^^^^^^
  }
}
```

Blogs management through a RESTful HTTP API

The following examples use HAL to express HATEOAS, and make use of:

- CURIE (Compact URI): used to provide links to API documentation
- URI templates: URI that includes parameters that must be substituted before the URI is resolved

Get blog 123

Request

```
GET https://example.com/api/v1.2/blogs/123
headers:
  Accept: application/hal+json
```

Response

```
status: 200 (OK)
headers:
  Content-Type: application/hal+json
body:
  {
    "id": 123,
    "title": "The blog title",
    "description": "The blog description",
    "_links": {
      "curies": [{ "name": "doc", "href": "https://example.com/docs/{rel}", "templated": true
}],
      "self": { "href": "https://example.com/api/v1.2/blogs/123" },
      "doc:articles": { "href":
"https://example.com/api/v1.2/blogs/123/articles{?pageIndex,pageSize}", "templated": true }
    }
  }
```

Create a new article in blog 123

Request

```
POST https://example.com/api/v1.2/blogs/123/articles
headers:
  Content-Type: application/json
  Accept: application/hal+json
  X-Access-Token: XYZ
body:
  {
    "title": "The title 2",
    "content": "The content 2"
  }
```

Response

```
status: 201 (CREATED)
headers:
  Content-Type: application/hal+json
body:
  {
    "id": 789,
    "title": "The title 2",
    "content": "The content 2",
    "_links": {
      "curies": [{ "name": "doc", "href": "https://example.com/docs/{rel}", "templated": true
}],
      "self": { "href": "https://example.com/api/v1.2/blogs/123/articles/789" },
      "doc:blog": { "href": "https://example.com/api/v1.2/blogs/123", "title": "The blog
title" },
      "doc:comments": { "href":
"https://example.com/api/v1.2/blogs/123/articles/789/comments{?pageIndex,pageSize}",
"templated": true }
    }
  }
```

Get article 789 of blog 123

Request

```
GET https://example.com/api/v1.2/blogs/123/articles/789
headers:
  Accept: application/hal+json
```

Response

```
status: 200 (OK)
headers:
  Content-Type: application/hal+json
body:
  {
    "id": 789,
    "title": "The title 2",
    "content": "The content 2",
    "_links": {
      "curies": [{ "name": "doc", "href": "https://example.com/docs/{rel}", "templated": true
}],
      "self": { "href": "https://example.com/api/v1.2/blogs/123/articles/789" },
      "doc:blog": { "href": "https://example.com/api/v1.2/blogs/123", "title": "The blog
title" },
      "doc:comments": { "href":
"https://example.com/api/v1.2/blogs/123/articles/789/comments{?pageIndex,pageSize}",
"templated": true }
    }
  }
```

Get the 4th page of 25 articles of blog 123

Request

```
GET https://example.com/api/v1.2/blogs/123/articles?pageIndex=4&pageSize=25
headers:
  Accept: application/hal+json
```

Response

```
status: 200 (OK)
headers:
  Content-Type: application/hal+json
body:
  {
    "pageIndex": 4,
    "pageSize": 25,
    "totalPages": 26,
    "totalArticles": 648,
    "_link": {
      "firstPage": { "href":
"https://example.com/api/v1.2/blogs/123/articles?pageIndex=1&pageSize=25" },
      "previousPage": { "href":
"https://example.com/api/v1.2/blogs/123/articles?pageIndex=3&pageSize=25" },
      "self": { "href":
"https://example.com/api/v1.2/blogs/123/articles?pageIndex=4&pageSize=25" },
      "nextPage": { "href":
"https://example.com/api/v1.2/blogs/123/articles?pageIndex=5&pageSize=25" },
      "lastPage": { "href":
"https://example.com/api/v1.2/blogs/123/articles?pageIndex=26&pageSize=25" }
    },
    "_embedded": [
      {
        ...
      }, {
        "id": 456,
        "title": "The title 1",
        "content": "The content 1",
        "_links": {
          "curies": [{ "name": "doc", "href": "https://example.com/docs/{rel}", "templated":
true }],
          "self": { "href": "https://example.com/api/v1.2/blogs/123/articles/456" },
          "doc:blog": { "href": "https://example.com/api/v1.2/blogs/123", "title": "The blog
title" },
          "doc:comments": { "href":
"https://example.com/api/v1.2/blogs/123/articles/456/comments{?pageIndex,pageSize}",
"templated": true }
        }
      }, {
        "id": 789,
        "title": "The title 2",
        "content": "The content 2",
        "_links": {
          "curies": [{ "name": "doc", "href": "https://example.com/docs/{rel}", "templated":
true }],
```

```
          "self": { "href": "https://example.com/api/v1.2/blogs/123/articles/789" },
          "doc:blog": { "href": "https://example.com/api/v1.2/blogs/123", "title": "The blog
title" },
          "doc:comments": { "href":
"https://example.com/api/v1.2/blogs/123/articles/789/comments{?pageIndex,pageSize}",
"templated": true }
        }
      }, {
        ...
      }
    ]
  }
```

Update article 789 of blog 123

Request

```
PUT https://example.com/api/v1.2/blogs/123/articles/789
headers:
  Content-Type: application/json
  Accept: application/hal+json
  X-Access-Token: XYZ
body:
  {
    "id": 789,
    "title": "The title 2 updated",
    "content": "The content 2 updated"
  }
```

Response

```
status: 200 (OK)
headers:
  Content-Type: application/hal+json
body:
  {
    "id": 789,
    "title": "The title 2 updated",
    "content": "The content 2 updated",
    "_links": {
      "curies": [{ "name": "doc", "href": "https://example.com/docs/{rel}", "templated": true
}],
      "self": { "href": "https://example.com/api/v1.2/blogs/123/articles/789" },
      "doc:blog": { "href": "https://example.com/api/v1.2/blogs/123", "title": "The blog
title" },
      "doc:comments": { "href":
"https://example.com/api/v1.2/blogs/123/articles/789/comments{?pageIndex,pageSize}",
"templated": true }
    }
  }
```

Notes

- The identifier that is used to identify the resource to update is **the one in the URL**: the one in the body (if any) must be silently ignored.
- As a `PUT` request updates the whole resource, if no `content` would have been sent, it should have been removed from the persisted resource.

Delete article 789 of blog 123

Request

```
DELETE https://example.com/api/v1.2/blogs/123/articles/789
headers:
  Accept: application/hal+json
  X-Access-Token: XYZ
```

Response

```
status: 204 (NO CONTENT)
headers:
  Content-Type: application/hal+json
body:
  {
    "_links": {
      "curies": [{ "name": "doc", "href": "https://example.com/docs/{rel}", "templated": true
}],
      "doc:blog": { "href": "https://example.com/api/v1.2/blogs/123", "title": "The blog
title" }
    }
  }
```

Violating REST

```
<stock>
    <add>
        <item>
            <name>Milk</name>
            <quantity>2</quantity>
        </item>
    </add>
</stock>
```

Putting this body to an resource like `/stocks/123` violates the idea behind REST. While this body is `put` and it contains all informations neccessary, it also comes along with an method call to `add` somewhere when the body is processed. Following REST one would post the `item` to `/stocks/123/items/`.

Read Getting started with rest online: https://riptutorial.com/rest/topic/1664/getting-started-with-rest

Credits

S. No	Chapters	Contributors
1	Getting started with rest	Community, Jason, Najeeb Arif, Roman Vottner, Shog9, slartidan, sp00m, ssanrao, sschrass, zretscen