Introduction to REST

AXXES_

Introduction and overview

REST is an architectural style, or design pattern, for APIs.

- REST: the theory
  - Where does REST come from?
  - Explaining the rules
  - Zooming in on a resource
  - Resource methods
- REST over HTTP
- REST with Spring
  - A quick Spring recap
  - Creating a REST API
  - Consuming a REST API
- Schemas

REST: The Theory

- Introduced in 2000 by Roy Fielding
- Stands for REpresentational State Transfer.
- Is an architectural style for designing loosely coupled applications.

Explaining the rules

Before we dive into what makes an API RESTful and what constraints and rules you should follow if you want to create RESTful APIs, let’s explain two key terms:

- Client
- Resource

Explaining the rules

A RESTful web application exposes information about itself in the form of information about its resources. It also enables the client to take actions on those resources, such as create new resources (i.e. create a new user) or change existing resources (i.e. edit a post).

There are six constraints that make an APi RESTful:

- Uniform interface
- Client <-> server separation
- Stateless
- Layered system
- Cacheable
- Code-on-demand

AXXES_

Uniform Interface

- **Identification of resources**: The interface must uniquely identify each resource involved in the interaction between the client and the server.
- **Manipulation of resources through representations**: The resources should have uniform representations in the server response. API consumers should use these representations to modify the resources state in the server.
- **Self-descriptive messages**: Each resource representation should carry enough information to describe how to process the message. It should also provide information of the additional actions that the client can perform on the resource.
- **Hypermedia as the engine of application state**: The client should have only the initial URI of the application. The client application should dynamically drive all other resources and interactions with the use of hyperlinks.

Client <-> Server Separation

- Separation of concerns: By separating the user interface concerns (client) from the data storage concerns (server), we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components.
- Requires interface/contract between the client and the server
- Client and the server act independently.

Stateless

- Each request from the client to the server must contain all of the information necessary to understand and complete the request.
- The server does not remember anything about the user who uses the API after it responded to a request.
- The server cannot take advantage of any previously stored context information on the server.
- The client application must entirely keep the session state.

AXXES_

Layered System

Allows an architecture to be composed of hierarchical layers by constraining component behavior

For example, in a layered system, each component cannot see beyond the immediate layer they are interacting with. Between the client who requests a representation of a resource’s state, and the server who sends the response back, there might be a number of servers in the middle. These servers might provide a security layer, a caching layer, a load-balancing layer, or other functionality. Those layers should not affect the request or the response. The client is agnostic as to how many layers, if any, there are between the client and the actual server responding to the request.

AXXES_

Cacheable

The cacheable constraint requires that a response should implicitly or explicitly label itself as cacheable or non-cacheable. If the response is cacheable, the client application gets the right to reuse the response data later for equivalent requests and a specified period.

Code on demand

- Optional
- Allows client functionality to extend by downloading and executing code in the form of applets or scripts

Zooming in on a Resource

The key **abstraction of information** in REST is a resource. Any information that we can name can be a resource. For example, a REST resource can be a document or image, a temporal service, a collection of other resources, or a non-virtual object (e.g., a person). The **state of the resource**, at any particular time, is known as the **resource representation**.

A REST API consists of an assembly of interlinked resources. This set of resources is known as the REST API's resource model.

The resource representation should consist of:

- **Data**
- **Metadata**, describing the data
- **Hypermedialinks**, that can help in transition to the next state.

Resource Identiefiers

REST uses resource identifiers to identify each resource involved in the interactions between the client and the server components.

Hypermedia

- Known as a **media type**.
- Media type identifies a specification that defines how a representation is to be processed
- Every addressable unit of information carries an address, either explicitly (e.g., link and id attributes) or implicitly (e.g., derived from the media type definition and representation structure).

AXXES_

Metadata or Self-Descriptiveness

Further, resource representations shall be self-descriptive: the client does not need to know if a resource is an employee or a device. It should act based on the media type associated with the resource. So in practice, we will create lots of custom media types, usually one media type associated with one resource.

Every media type defines a default processing model. For example, HTML defines a rendering process for hypertext and the browser behavior around each element.

AXXES_

Resource Methods

Another important thing associated with REST is **resource methods**. These resource methods are used to perform the desired transition between two states of any resource. There have never been any recommendations in the original REST proposal around which method to use in which condition. All that is emphasized is that is should be a **uniform interface**.

An example: A large number of people wrongly relate resource methods to HTTP methods (i.e., GET/PUT/POST/DELETE). if we decide that the application APIs will use HTTP POST for updating a resource – rather than most people recommend HTTP PUT – it’s all right. Still, the application interface will be RESTful.

Ideally, everything needed to transition the resource state shall be part of the resource representation – including all the supported methods and what form they will leave the representation.

Resource Methods

‘’We should enter a REST API with no prior knowledge beyond the initial URI (a bookmark) and a set of standardized media types 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 the client selection of server-provided choices present in the received representations or implied by the user’s manipulation of those representations.”

AXXES_

REST over HTTP

Though REST also intends to make the web (internet) more streamlined and standard. The REST standard has nowhere mentioned any implementation direction, including any protocol preference or even HTTP. Despite that, many people prefer to compare and implement REST using the HTTP protocol but they are not the same.

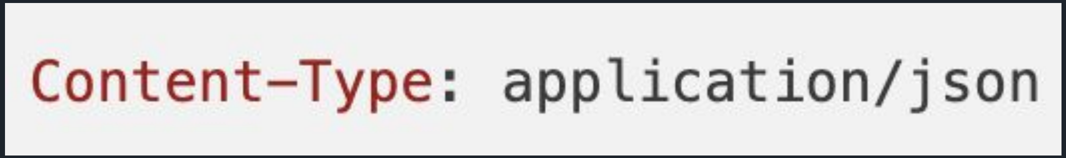
Identifying a Resource

When implementing REST using HTTP, one can use the URL as a unique identifier for a resource.

```
http://www.my-rest-api.com/api/resource-type/1
```

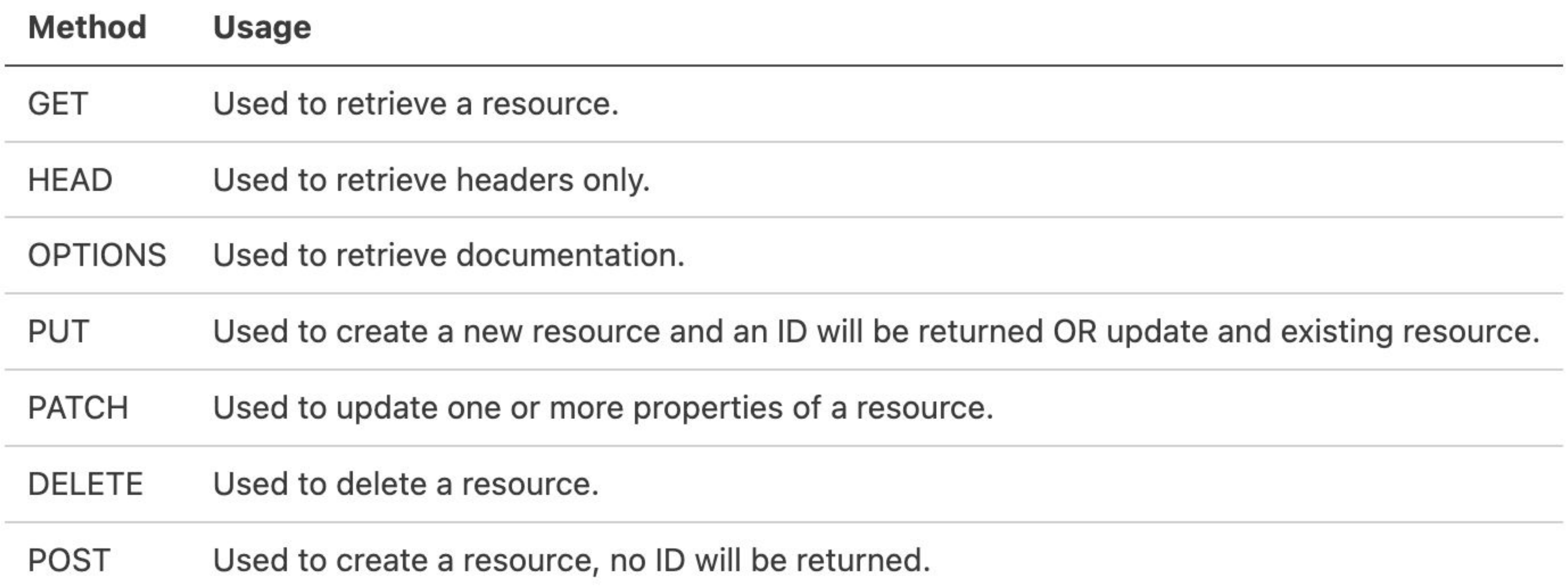
Media Types

Resource media types can be defined using headers.



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

Resource Methods



Method	Usage
GET	Used to retrieve a resource.
HEAD	Used to retrieve headers only.
OPTIONS	Used to retrieve documentation.
PUT	Used to create a new resource and an ID will be returned OR update and existing resource.
PATCH	Used to update one or more properties of a resource.
DELETE	Used to delete a resource.
POST	Used to create a resource, no ID will be returned.

GET

- GET requests to retrieve resource representation/information only – and not modify it in any way
- GET requests do not change the resource’s state
- Making multiple identical requests must produce the same result every time until another API (POST or PUT) has changed the state of the resource on the server.

```
GET all of a specific resource-type:
    http://www.my-rest-api.com/api/resource-type/

GET one of a specific resource-type by id:
    http://www.my-rest-api.com/api/resource-type/1

GET property of a specific resource:
    http://www.my-rest-api.com/api/resource-type/1/property
```

AXXES_

PUT

- update an existing resource (if the resource does not exist, then API may decide to create a new resource or not)
- Responses to PUT method are not cacheable.

```
PUT addition of a resource-type:
    http://www.my-rest-api.com/api/resource-type/

PUT addition of a resource-type's property:
    http://www.my-rest-api.com/api/resource-type/1/property/456
```

The difference between the POST and PUT APIs can be observed in request URIs. POST requests are made on resource collections, whereas PUT requests are made on a single resource.

PATCH

- Are to make a **partial update on a resource**

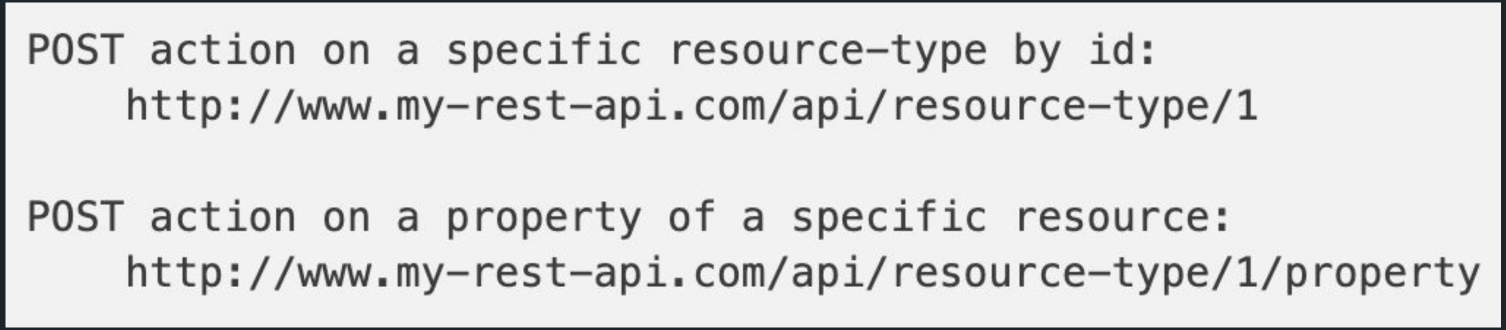
DELETE

- delete the resources (identified by the Request-URI)
- are idempotent, meaning you can delete a resource multiple times, it won’t change the final result.
- If you DELETE a resource, it’s removed from the collection of resources.

```
DELETE of a specific resource-type by id:
    http://www.my-rest-api.com/api/resource-type/1

DELETE of a resource-type's property:
    http://www.my-rest-api.com/api/resource-type/1/property/456
```

POST

- create new subordinate resources, e.g., a file is subordinate to a directory containing it or a row is subordinate to a database table
- used to create a new resource into the collection of resources.
- Responses to this method are not cacheable by default
- POST is not idempotent, and invoking two identical POST requests will result in two different resources containing the same information (except resource ids).



```
POST action on a specific resource-type by id:
    http://www.my-rest-api.com/api/resource-type/1

POST action on a property of a specific resource:
    http://www.my-rest-api.com/api/resource-type/1/property
```

Status Codes

Status code range	Usage	Simple Words
1XX	Informational	Hold on...
2XX	Successful	Here you go!
3XX	Redirection	Go Away
4XX	Client Error	You screwed up
5XX	Server Error	I screwed up

AXXES_

Query Parameters

For certain HTTP methods we can enrich our request by using additional parameters. They are a defined set of parameters attached to the end of a URL. They are an extensions of the URL and are used to help define specific content or actions based on the data being passed.

```
http://www.my-rest-api.com/api?search=keyword

http://www.my-rest-api.com/api/resource?limit=5&type=property-type
```

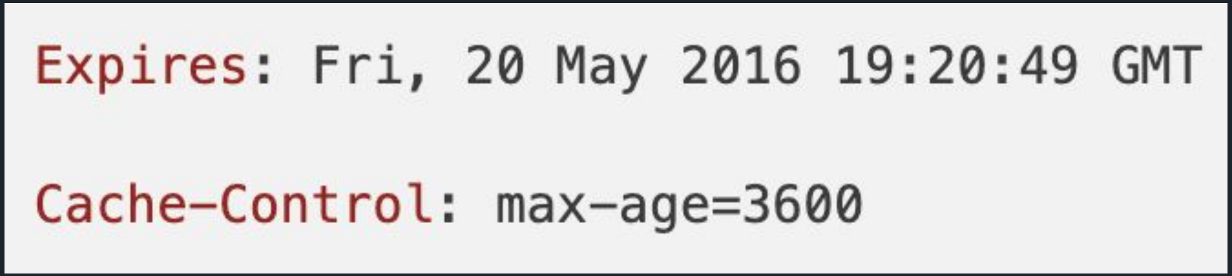
Caching

Adding an E-Tag to a response coming from the server. An E-Tag can be seen as a digital fingerprint of your response. Every time the client performs the same request, he can ask if the fingerprint has changed. And instead of the server sending the full response back to the client, the server can respond with a *304-Not modified* response code. Which tells the client nothing has changed and he can used his own caches response from the previous request.

```
GET
> http://www.my-rest-api.com/api/resource

RESPONSE
< HTTP/2 200
< Cache-Control: max-age=0, must-revalidate
< Content-Length: 524653
< ETag: "ebeb4dbc1362d124452335a71286c21d"

GET
> http://www.my-rest-api.com/api/resource
> If-None-Match: "ebeb4dbc1362d124452335a71286c21d"

RESPONSE
< HTTP/2 304
```

Caching

Additionally the sever can add an *Expires* or *Cach-Control* header to the response. The first header defines an absolute expiry time for the cached resources. Beyond that time, a cached representation is considered stale and must be re-validated with the origin server. The second header defines if a response is cachable and for how long.



```
Expires: Fri, 20 May 2016 19:20:49 GMT

Cache-Control: max-age=3600
```

AXXES_

Versioning

APIs only need to be up-versioned when a breaking change is made. Breaking changes include:

- A change in the format of the response data for one or more calls
- A change in the request or response type (i.e. changing an integer to a float)
- removing any part of the API.

Versioning Strategies: URI Versioning

Using the URI is the most straightforward approach (and most commonly used as well) though it does violate the principle that a URI should refer to a unique resource. You are also guaranteed to break client integration when a version is updated.

```
http://www.my-rest-api.com/api/v1/resource

http://www.v1.my-rest-api.com/api/resource
```

Versioning Strategies: Custom Header

A custom header (e.g. Accept-version) allows you to preserve your URIs between versions though it is effectively a duplicate of the content negotiation behavior implemented by the existing Accept header.

```
Accept-version: v1
Accept-version: v2
```

Versioning Strategies: Accept Header

Content negotiation may let you preserve a clean set of URLs, but you still have to deal with the complexity of serving different versions of content somewhere. This burden tends to be moved up the stack to your API controllers which become responsible for figuring out which version of a resource to send.

The result tends to be a more complex API as clients have to know which headers to specify before requesting a resource.

```
Accept: application/version.my-rest-api+json
Accept: application/version.my-rest-api+json;version=1.0
```

REST using Spring

For this section we jump to the readme on git