Microservice Architecture Blueprint
REST Misc Topics

Kim Horn

Version 0.1 **1 August 2017**

Plural Resources

The resource should always be **plural** in the API endpoint and if we want to access one instance of the resource, we pass the id in the URL.

- method GET path /companies should get the list of all companies
- method GET path /companies/34 should get the detail of company 34
- method DELETE path /companies/34 should delete company 34

If we have resources under a resource, e.g Employees of a Company, then example API endpoints would be:

- GET /companies/3/employees should get the list of all employees from company 3
- GET /companies/3/employees/45 should get the details of employee 45, which belongs to company 3
- DELETE /companies/3/employees/45 should delete employee 45, which belongs to company 3
- POST /companies should create a new company and return its details

Use Two URLs per Resource

One URL for:

- the collection and
- one for a certain element

- /employees #collection URL
- /employees/56 #element URL

- To add a new resource POST to the collection

Nouns and Verbs

Use Nouns for Resources

Use Verbs for Non-Resource Responses

- Sometimes a response to an API call doesn’t involve resources (like calculate, translate or convert).
- Example:
  - GET /translate?from=de_DE&to=en_US&text=Hallo
  - GET /calculate?para2=23&para2=432

- URI's identify a resource, they should not indicate *what* we're doing to that resource.
  - That’s where HTTP Operations come

Path Vs Query

A REST API can have parameters in at least two ways:

- **As part of the URL-path** (i.e. /api/resource/parametervalue)
- **As a query argument** (i.e. /api/resource?parameter=value)

The URI standard states the path is for hierarchical parameters and the query is for non-hierarchical parameters.

1. **Path variables** are used for the direct action on the resources, like a contact or a song ex..
   GET etc /api/resource/{songid} or
   GET etc /api/resource/{contactid} will return respective data.
2. **Query perms/argument** are used for the in-direct resources like metadata of a song eg
   GET /api/resource/{songid}?metadata=genres return the genres data for that particular song.

Searching, Sorting, Filtering – using the URL Query (?)

Sorting:
Similar to filtering, a generic parameter sort can be used to describe sorting rules. Accommodate complex sorting requirements by letting the sort parameter take in list of comma separated fields, each with a possible unary negative to imply descending sort order. For example,
GET /groups?sort=status,-name *– Returns list of groups in ascending order of status; Within the same status, groups returned will be sorted by name in descending order*

Filtering
Use a unique query parameter for each field that implements filtering.
GET /groups?status=active *– Returns a list of active groups*

Searching:
Sometimes basic filters aren't enough and you need the power of full text search. When full text search is used as a mechanism of retrieving resource instances for a specific type of resource, it can be exposed on the API as a query parameter on the resource's endpoint. Let's say q. Search queries should be passed straight to the search engine and API output should be in the same format as a normal list result.
GET /groups/?status=active&sort=-name&q=test *– Return list of all active groups sorted by name (desc) which contain 'test' in their names*

Search as a Resource

One way to implement a RESTful search (or filter) is to consider the search (or filter) itself to be a resource.

- Then you can use the POST verb because you are creating a search.
- You do not have to literally create something in a database in order to use a POST.

For example:

POST http://example.com/people/searches

```
{
      { "terms":  {
            ”id": "123456789"
      },
      "order":
      { ...
      },
... }
```

Get has no BODY

Complex queries could use the GET Body to include a JSON search representation. But it is recommended not to use the body on GET. Postman does not even allow it to be specified. However Elasticsearch API uses the Get body and argues:

The RFC for HTTP 1.1 RFC2616 says this:

> *A server SHOULD read and forward a message-body on any request; if the request method does not include defined semantics for an entity-body, then the message-body SHOULD be ignored when handling the request.*

The *recommendation* (note: "SHOULD") is that a GET body shouldn't have any effect on server processing.

But GET is a better semantic fit for search than POST:

> *In particular, the convention has been established that the GET and HEAD methods SHOULD NOT have the significance of taking an action other than retrieval. These methods ought to be considered "safe". This allows user agents to represent other methods, such as POST, PUT and DELETE, in a special way, so that the user is made aware of the fact that a possibly unsafe action is being requested.*

Also mentioned in the spec:

> *A cache or origin server receiving a conditional request, other than a full-body GET request, MUST use the strong comparison function to evaluate the condition.*

Googles' Model

While a simple search could be modeled as a resourceful API, eg., **dogs/?q=red**, a more complex search across multiple resources requires a different design.

Global search

/search?q=fluffy+fur

Here, search is the verb; **?q** represents the query.

Scoped search

To add scope to your search, you can prepend with the scope of the search. For example, search in dogs owned by resource ID 5678

/owners/5678/dogs/search?q=fluffy+fur

Formatted results

For search or for any of the action oriented (non-resource) responses, you can prepend with the format as follows:

/search.xml?q=fluffy+fur

Creating (and storing) actual Resources for Searching

Consider treating the set of possible queries as a collection resource, e.g. /jobs/filters, and create new entities as first class resources.

POST request entities to this resource, with the query parameters in the body, will either create a new resource or identify an existing equivalent filter and return a URL containing its ID:

/jobs/filters/12345.

The id can then be used in a GET request for jobs:

/jobs?filter=12345.

Subsequent GET requests on the filter resource will return the definition of the filter.

Advantage: it frees you from the query parameter format for filter definition, potentially providing you with more power to define complex filters. OR conditions are one example that I can think of that are difficult to accomplish with query strings.

Disadvantage: you lose readability of the URL (although this can be mitigating by retrieving the definition though a GET request for the filter resource).

Complex Query Parameters

Passing JSON Objects as a parameter. Complex search statements could be created in JSON, and passed as a parameter with GET , rather than in the body with POST.

- This gets messy and should use URL encoded string.
- Other coded string could be used. However it gets hard to interpret:
  - /cars;color-blue+doors-4+type-sedan
  - /garage[id=1-20,101-103,999,!5]/cars[color=red,blue,black;doors=3]
  - /cars[?;]color[=-:]blue[,;+&], *

ODATA:

OData queries operate as a GET call but allow you to restrict the properties which are returned and filter them.

Tokens such as $select= and $filter= are used to build ‘complex’ statements.

A URI would look something like this:

/users?$select=Id,Name$filter=endswith(Name, 'Smith')

HTTP SEARCH Method – The Draft

"Using a GET request with some combination of query parameters included within the request URI is arguably the most common mechanism for implementing search in web applications."

- Implementations are required to parse the request URI into distinct path (everything before the '?') and query elements (everything after the '?').
- The path identifies the resource processing the query (in this case 'http://example.org/ feed') while the query identifies the specific parameters of the search operation.

Issues:

- Query expressions included within a request URI must either be restricted to relatively simple key value pairs or encoded such that the query can be safely represented in the limited character- set allowed by URL standards. Such encoding can add significant complexity, introduce bugs, or otherwise reduce the overall visibility of the query being requested.
- While most modern browser and server implementations allow for long request URIs, there is no standardized minimum or maximum length for URIs in general.
  - Using the payload provides a way aroudn this but GET does not recommend using the payload.
- The client has not specified to the server that a SEARCH Operation is being requested.

HTTP SEARCH Method - Draft

"As an alternative to using GET, many implementations make use of the HTTP POST method to perform queries, as ……. In this case, the input parameters to the search operation are passed along within the request payload as opposed to using the request URI."

Issues:

- Not specified that a search Operation is being requested. Indeed POST is usually used to create a resource.
- An HTTP Search Operation has been suggested and defined by RFC5323 to get around these issues of GET and POST
- Messes with caching; as search is probably transient

"The SEARCH method is used to initiate a server-side search. Unlike the HTTP GET method, which requests that a server return a representation of the resource identified by the effective request URI (as defined by [RFC7230]), the SEARCH method is used by a client to ask the server to perform a query operation (described by the request payload) over some set of data scoped to the effective request URI. The payload returned in response to a SEARCH cannot be assumed to be a representation of the resource identified by the effective request URI."

Aliases and Field Selection

Aliases for common queries

To make the API experience more pleasant for the average consumer, consider packaging up sets of conditions into easily accessible RESTful paths. For example, when querying for mostactive,recommended groups etc, we can have endpoints like

GET /groups/mostactive – *Returns list of mostactive groups*

Default values can be used for the sort parameters.

Field selection

The API consumer doesn't always need the full representation of a resource (mobile clients for example). The ability select and chose returned fields goes a long way in letting the API consumer minimize network traffic and speed up their own usage of the API. Use a fields query parameter that takes a comma separated list of fields to include.

GET /groups/?fields=id,name,owner,status&status=active&sort=-name

Pagination

The right way to include pagination details today is using the Link header introduced by RFC 5988. An API that uses the Link header can return a set of ready-made links so the API consumer doesn't have to construct links themselves.

GET /groups?offset=20&limit=20

The response should include pagination information through links as below:

```
{
        "start": 1,
        "count": 20,
        "totalCount": 100,
        "totalPages": 5,
        "links": [
                {
                        "href": "https: //<url>/offset=40&limit=20",
                        "rel": "next"
                },
                {
                        "href": "https: //<url>/offset=0&limit=20",
                        "rel": "previous"
                }
        ]
}
```

Error Handling

Error Response

An API should provide a useful error message in a known consumable format in case of errors in API's.

Both successful and error response can be handled using the HTTP Status codes.

However, the representation of an error should be no different than the representation of any resource, just with its own set of fields. A JSON error body should provide a few things for the developer – a useful error message, a unique error code (that can be looked up for more details in the docs) and possibly a detailed description. JSON output representation for something like this would look like:

```
{
    "code" : 1234,
    "message" : "Something bad happened :(",
    "description" : "More details about the error here"
}
```

Rate Limiting

To prevent abuse, it is standard practice to add some sort of rate limiting to an API. RFC 6585 introduced a HTTP status code 429 Too Many Requests to accommodate this.

- The 429 status code indicates that the user has sent too many requests in a given amount of time ("rate limiting").

However, it can be very useful to notify the consumer of their limits before they actually hit it. This is an area that currently lacks standards but has a number of popular conventions using HTTP response headers.

At a minimum, include the following headers

- X-Rate-Limit-Limit- The number of allowed requests in the current period
- X-Rate-Limit-Remaining – The number of remaining requests in the current period
- X-Rate-Limit-Reset – The number of seconds left in the current period