RESTwell

version 0.8. Please respect the work or original author:

Table of contents

...

INTRODUCTION

RESTwell is a set of rules for designing REST APIs that are extensible, easy to work with, and support solid design choices. The goal is to simplify API design in a more pragmatic way than previous attempts like HAL.

Status

This document is open for comments and additions but basic foundation is pretty solid.

Concepts & keywords

Concept	Description
REQUIRED	A practice that must be followed
RECOMMENDED	What it says..
DRAFT	Might be dropped or changed to any of above

OVERVIEW

A public announcement

Not all REST are alike. Some are meant for public consumption, some are for integrations between legacy enterprise systems where design choices are based on the consumer

requirements. The RESTwell priniples described in this document focuses around APIs designed for *public consumption* where:

- Consumer can rely on standards and solid patterns in design,
- Supplier of API can extend the API without running into walls of limitations

Authentication & Authorization

The process of authentication and authorization is not implemented in RESTwell. Authentication can be carried out in vastly different ways depending on scenarios. Authorization *may* be included in the specificaion. With this said:

- Bearer tokens (OAuth) are suitable for authentication towards 3:rd part services.
- Authorization (based on RBAC / PBAC etc) *may* be provided from 3:rd part. But to prevent tight coupling between systems, these roles etc should normally be translated to roles owned by the API system-domain.
- Session-based authorization may be utilized. Although it introduces state-awareness to the REST-server, the advantages of session-based authorization may overweigh the downsides of not being fully stateless. A well known fact is that the server can't invalidate an OAUTH-token (without implementing a black-list), this is not a problem when using session-based authorization.

Rule of thumb: OAUTH is the *passport*, the session is the *visa*, issued and revoked by the API.

Versioning

The often mentioned two design patterns for versioning is:

- Implementing version in the path resource (api.mydomain.com/v2.0/whoami)
- Implementing a header for sending the requsted version.

Both these alternatives have the obvious downside that a new API version cannot easily be hosted on another host or container. The solution to this is to put the versioning in the hostname itself, and pass the resolving of default API to the DNS.

- *api-1v4.mydomain.com*
- *api-stable.mydomain.com*

The versioning-format "api-1v4" etc is **RECOMMENDED** but may change to **REQUIRED** for simplicity. Any extra api aliases as *api-latest.mydomain.com* has no recommendations.

Services that uses this naming standard is for example *letsencrypt*.

GENERAL

Name conventions / casing

API vs data layers

One of the core features of RESTwell is using different types of CASING to separate data-arguments from api-arguments. This solves problems that may creep upon you over time, as for example when an entity contains the field *count or offset* - as below:

api-1v4.mydomain.com/measures?offset=10

By using casing *API* vs *data* it's easy to separate and the API becomes more solid for future expansions.

- api-1v4.mydomain.com/measures?OFFSET=10 (Limit sized of returned collection)
- api-1v4.mydomain.com/measures?offset=10 (Apply filter on field offset)

Casing for resources

Based on above - data (collections & fields), uses lower-case identifiers and the standard in RESTwell is "snake_casing". Some standards suggests "kebab-casing". The use of minus-sign in names makes it hard to read and also hard to map to variable names in some languages. So: snake_case is the **REQUIRED** casing for resources and fields.

Plurals for collections

The rule for plurals around collections is **recommended.** The problem with anomalies around plural entities (mouse vs mice, foot vs feet) aren't that big it justifies the advantages of *easily identify entities from fields*

Exception from this rule is for global or static resources such as:

api-dev.mydomain.com/session/status or

api-dev.mydomain.com/api/ping

Pascal casing for intents and views

Two central terms in RESTwell are *intents* and *views*. These will be further explained in separate chapters but in summary:

- An intent performs a non CRUD action on a resource (using POST-verb), that may return a 200 or 202 status on success, for example: *mydomain.com/api/doRestart*
- A view performs a server-side query with arbitrary arguments (using GET-verb):

  mydomain.com/customers/getNearby?gps_long=64&gps_lat=10&radius_km=100

They both use *camelCasing* to signal that custom code is executed on server.

Route-based semantics

The *intent of a route* may be evaluated by its format + its verb. Each segment of the route must be identified as either a *resource name* (string) or a *key*.

Key formats

The easiest form of keys are plain integers but other formats are supported:

Key-type	Example
Integer	myapi.com/customers/52
String/UUID	Must be base64url-encoded and surrounded by tildes ~...~
Composite	A base64url-encoded JSON object, also surrounded by tildes ~...~ *(See example below)*

Examples

myapi.com/customers/72e8b54c-37ce-11eb-adc1-0242ac120002

=> expressed as =>

myapi.com/customers/ ~NzJlOGI1NGMtMzdjZS0xMWViLWFkYzEtMDI0MmFjMTIwMDAy~

myapi.com/customers/{"country":"BR","customer_code":"X53Y"}

=> expressed as =>

myapi.com/customers/~eyJjb3VudHJ5IjoiQlIiLCJjdXN0b21lcl9jb2RlIjoiWDUzWSJ9~

Routing evaluation

Based on the structure of the request, the intent is evaluated as follows:

- String part: **s**
- Key part: **k**

Path	Method	Description	Remark
/s	GET	List collection of resources	
/s	POST	Create new resource	
/s/s	GET	Custom view with custom arguments	
/s/s	POST	Request intent on collection	1)
/s/k	GET	View resource	
/s/k	PATCH	Update resource	2)
/s/k	DELETE	Delete resource	R
/s/k	POST	Update/create resource with HTML form	3)
/s1/k/s2	POST	Request intent s2 on resource s1/k	S
/s1/k/s2	PUT	Binary upload of s2 on resource s1/k	S

1. Intents are *normally* performed on resources but may be applied to a resource-collection, for example: *mydomain.com/users/doImport*.
2. Patch is used for updating a resource - not PUT. Put is for replacing *complete* resource and API may lack knowledge of complete entity. Thus: **PUT IS ONLY USED FOR ADDING OR REPLACING BINARY CONTENT**
3. This endpoint type is *optional* and ment to simplify mocking or integrations to web-forms.

Sub-routes

Sub-routes may be used to make requests more semantic. Example:

api-dev.mydomain.com/customers/123/orders/456/order_rows/789 (DELETE)

is re-formatted so it's executing the query towards *order_rows* with some filters:

api-dev.mydomain.com/order_rows/789?customer_id=123&order_id=456 (DELETE)

The names of the keys are resolved at server which is one benefit of using sub-routes.

REQUIRED: Authorization of the request is evaluated by last resource only (order_rows in example above).

Status codes

- Any successful GET-request should reply 200-OK.
- Missing resource (resource or id): 404
- Malformed request: 400
- Intents may reply 200 or 202 based on if they are synchronous or asynchronous.

COLLECTION REQUESTS

Collection requests are mostly used for retrieving and filtering data based on common rules using GET-verb. POST-verb is used, according to REST-standard to create resources.

GET-requests

Three types of rules are applied:

1. Row filtering
2. Sorting
3. Field filtering
4. Pagination
5. HATEOAS-links
6. Format

1 - Row filtering

- Any field may be used for filtering data, *field=filterVal*
- Operator for filter is by default "EQU" (equal)

- Operator for filter may be changed using: *OPER_field=LTE*
- For between-filter, end-argument is passed using *OPEREND_field=endVal*

Operators

_OP	Meaning	Comment
EQU	=	default
NOT	<>	Not equal to
GT	>	Greater than
GTE	>=	Greater or equal
LT	<	Less than
LTE	<=	Less or equal than
LIKE	*	Limits set by API
IN	a,b,c	Comma-separated list of values
BTW	range	(field) sets from value
NULL	is null	Value set
NNULL	is not null	Value not set

Examples of row filtering

/books?title=fish&OPER_title=LIKE&year_of_pub=2002&OPER_year_of_pub=GTE

/books?year_of_pub=2012&OPER_year_of_pub=BTW&OPEREND_year_of_pub=2016

2 - Sorting

Sorting may be carried out in two ways:

- Using separate fields - appropriate for interacting with frontend-forms or similar
- Using *compound field* with a format suitable for interacting with an API client.

Separate fields

- Fields are prefixed with SORT_ , *books?SORT_title=asc*

- May be combined, books?SORT_title=asc&SORT_year_of_publication=desc *
- Priority of sorting is specified using SORT_BY, ?SORT_BY=year_of_publication,title

* **DRAFT:** Arguments *asc* and *desc* should possibly be UPPER_CASE, addressing the API.

Compound field

The SORT_BY may be used to provide a more compact and API<->API friendly syntax. The compound mode is activated when the SORT_BY-argument starts with either a "+" or "-" sign. Activating the compound mode disables any sorting-requests using separate fields. Example:

/books?SORT_BY=-year_of_publication,+title

3 - Field filtering

To reduce bandwidth usage, any collection request may use the FIELDS attribute, specifying the orders to fields:

*/books?SORT_BY=-year_of_publication,+title&FIELDS=title,author,year_of_publication

DRAFT: When using field-filtering, id fields needs to be added when so needed.

4 - Pagination

Pagination may be used by using the API-parameters COUNT and PAGE

/customers?COUNT=50&PAGE=10

5 - HATEOAS (DRAFT)

Returning the collection does not have to include all fields of the individual record. Links (HATEOAS) should be provided to collection and to collection-items **when requested**

The point of HATEOAS is for debate since each endpoint is so flexible it's not realistic to make links for all.

- The return-format are inspired by the JSON:API standard, but simplified.
- The 'links' are returned by default, with the types of
  - first, prev, next, last for page-navigation
  - HINT for new-record entry.
- The 'LINKS' on item could be requested with LINKS=1

Calculating the appropriate grants is disabled by default since it may require CPU to be trustworthy.

6 - Format

The argument FORMAT may be used to specify a non-JSON return format. The use of this format is **NOT SPECIFIED** and the API implement custom formats. Here's an example of returning a collection *without* HATEOAS-links etc, just the collection - for better integrations with Microsoft Power Query:

/clients?year_2012&FORMAT=msq

POST-requests

In accordance to REST-standards, invoking a POST-request to a collection endpoint should create a new resource (record). To support this and also support having natural or composite keys.

POST /books

- ID values are never provided in URL but in data-body.
- Data is sent as json-object and not as http-encoded form.

Return codes

If the resource is created the response should provide a link to where the new resource can be found

---

RESOURCE REQUESTS

Resource requests are applied to a single resource identified by a key, with support for surrogate-keys, natural keys and composite keys.

The benefit of having a strict set of rules on how to map routes to actions is that manual routing maps doesn't have to be maintained. The parsing can understand the intent of the request just "by looking at it".

Request verbs

- GET - retrieve data for a resource
- PATCH - updates data for a resource
- DELETE - deletes a resource
- POST - updates a resource according to HTML standards.

REQUIRED: PUT-method is *not* used for record manipulation.

Resource blob requests

Additional requests are defined to get, update and delete blobs associated with resource, see end of this chapter.

GET

When requesting a resource, additional HATEOAS-links are providied in accordance to authourized actions. Links to intents and views are also provided, prefixed with *intent_* and *view_* as in example below:

```
{
  'links': {
    'self': 'https://api-1v4.api.com/orders/.535-535.',
    'delete': '.',
    'intent_ship': './doShip',
    'view_orderStatus': './orderStatus'
  },
  'data': {
    'order_id': '535-535',
    'adress': '221B Baker Street',
    'status_code': 'open'
  },
  'enums': {
    'status_code': {
      'shipped': 'Is shipped',
      'open': 'Open for modification'
    }
  }
}
```

DRAFT: The API provides enums with their explainations to the frontend.

E-Tag

DRAFT: The API should provide an auto-generated ETag with both:

- ETag being sent in header response.
- optional ETAG being provided in data-collection of body.

Response codes

- 200 OK
- 404 Resource not found
- 400 Bad request format

PATCH

The patch-request submits fields to be updated in body. Fields not provided in body is by default not affected.

For simplicity, the PATCH-method may be replaced with the POST-method when working with html-forms.

REQUIRED: Authorization of request, fields and sanitation of field values must be carried out by server on each request.

Response codes

(Following recommendations from developer.mozilla.org). 204 No content + new ETag-header. No body.

```
HTTP/1.1 204 No Content
Content-Location: /users/123
ETag: "e0023aa4f"
```

DELETE

The DELETE-method - well you guessed it, deletes the resource.

Response codes

```
HTTP/1.1 204 No Content
```

POST

The POST-method is OPTIONAL and designed to support mocking or easier integrations to web-forms. It uses HTML standards to provide the data fields.

Response codes

The response-codes should be designed for the html-client to be as easy as possible to understand. TBA.

Binary data of resources

Additional requests to get, upload and delete binary data associated with resource

GET binary resource

The get-resource retrives the binary resource and by default sets the return mime-type based on the original file name of the blob.

/records/525?BLOB=front_cover

GET meta information about a blob

/records/525?BLOB=front_cover&BLOB_META

The json returned should contain:

```
{
    "file_name": "name of file when uploaded, including suffix",
    "file_size": "size of file in bytes"
}
```

and may contain extra attributes as:

```
{
    "img_width": 1024,
    "img_height": 800
}
```

PUT binary resource

To create or update binary resource, the put-method is used. (PATCH-method is NOT used for updating binary resources).

- Content shoud be sent RAW (not FORM-ENCODED since this is related to HTML and not HTTP).
- Original file_name may be sent in URI using FILE_NAME.
- The endpoint is NOT to the resource, but requires the addional query-string PUT /records/525?BLOB=front_cover
- The original filename may be supplied. PUT /records/525?BLOB=front_cover?FILE_NAME=myLogo.png

The file-name should be url-encoded and in same character set as API (normally UTF-8)

DELETE binary resource

The binary resource may be deleted using the endpoint

DELETE /records/525?BLOB=front_cover

Partial upload (DRAFT)

TBA

Response codes

If complete blob-upload: 200 OK

If partial blob-upload: 308 Resume Incomplete

Form-supported POST-methods for binary resources

To support applications that use HTML forms, the POST-method maybe used on *the resource* with the additional fields to update or delete blobs.

Updating blobs

By providing the blob in a *multipart/form-data* request the data may be provided as:

POST /records/525 {BLOB_front_cover=data}

The blob may also be deleted using the special argument *BLOB_DELETE*

POST /records/525 {BLOB_DELETE_front_cover=1}

SUB-RESOURCE COLLECTION REQUESTS

By providing deeper tree-like end-points, the API automatically returns relevant resources based as it parses the resource. Thus, an endpoint like:

GET /invoice_rows?row=10&invoice=5252&customer=123

may be requested by the more intuitive

GET /customers/123/invoices/5252/invoice_rows/10

When using sub-routes the following logic is applied:

- ID-values provided are identified and mapped to their name in the final table
- Authorization is *not* traversed, but only applied to the final request.

Managing sub-resources

Currently, only collection GET-method is supported for sub-resources. The returned array should provide HATEOAS-links to manage the individual items. It's the provided endpoint that's responsible to authorize any mutation-action on the resource.

VIEWS & INTENTS

By providing views and intents, any "special treatment" for both data retrieval and data mutations are supported. The standard for both views and intents are quite elementary as they should cover the "remaining 20%" of a complete API and thus should support things like:

- Downloading a PDF-report (view)
- Archive a conversation (intent)
- Fire a space rocket (intent)
- Find closest indian restaurant based on client gps-location (view)

Naming conventions

To avoid future naming collisions between views / intents and sub-resources use the following rule of thumb:

- By using a camelCase-identifier for the view or intent, the risk of future collisions is avoided, as sub-resources uses snake_case.

APPENDIX 1 - The Petstore

Given the rules in the RESTwell, let's look at the endpoints in the commonly known "petstore" from OpenApi aka petstore.swagger.io.

petstore.swagger.io

By comparing what they suggest to the RESTwell rules, we get some kind of stress-test of thRESTwellST principles. Many of their endpoints match RESTwell protocol, for example POST /user/createWidthList matches the format RESTwell *collection-intent*

However, there are some differences that might be worth looking in to. Decide for yourself what you prefer.

Uploading pet image

Swagger: POST /pet/{petID}/uploadImage

Providing petId, an image can be uploaded to the resource. Uses form-data for binary transfer.

RESTwell: PUT /pet/{petId}?BLOB=image

Provides better support for multiple images and uses raw binary content. Uses put to indicate the action is idempotent.

Fallback for form-based frontend

Create / Update: **POST /pet/{petId} {BLOB_front_cover=data}**

Delete: **POST /pet/{petId} {BLOB_DELETE_front_cover=1}**

Update existing pet

Swagger: PUT /pet

RESTwell: PATCH /pet

The PUT method is idempotent and may completely replaces earlier resource. This is suitable for documents and blobs, but not for data where:

- Complete resource information may be hidden to API
- There may be side-effects of the update making the action non idempotent.

Place order

Swagger: POST /store/order

Places an order of one or many pets of different quantities.

RESTwell: POST /store/placeOrder

Allthough not required, using camelCasing makes it obvious that this is an intent.

User management

The user-resource uses natural key and it's not clear how this key is escaped

Swagger: POST /user, GET, DELETE, PUT /user/{username}

RESTwell: POST /user, GET, DELETE, *PATCH* /user/~~{user%20name}~~

Patch used for update, url-encoding used for string-based keys.

Session management

The session-management in the swagger-implementation is mapped to the user, and uses GET-methods(!) As in:

Swagger: GET /user/login, GET /user/logout

This verb is problematic as requests may get stuck in proxies etc, also mapping the actions to the user is questionable. Here's the RESTwell suggested solution:

RESTwell: POST /session/sessionStart vs sessionEnd

with info for starting session via json-body, email+password, o-auth token etc.