Representational State Transfer (REST) is a increasingly popular style of interface design for services exposed via HTTP. But what is the best practice for creating a concrete REST implementation? In this paper, I will present a reusable pattern for any REST-compliant interface.

JAREST: a recommended REST implementation pattern

Contents

In this paper, I will introduce a pattern for the implementation of RESTful interfaces. I call this pattern "JAREST". It consists of the following elements:

1. The URI format
2. The format of the bodies of the messages
3. Partially embedded resources
4. Creating new resources by template
5. Versioning policy
6. How to apply the principle of 'Hypermedia as the engine of application state' (HATEOAS)

I have limited the amount of text, at the expense of thorough justifications and analysis of alternatives. The reader is assumed to know HTTP, JSON and the REST principles.

1. The URI format

The REST interface is defined by an API schema. I recommend to use Swagger (see swagger.io/specification). The URI of the schema has the following format:

http(s)://≪env≫.≪2ld≫.≪tld≫/v≪major≫/swagger.json

For example: "http://api.demo.org/v1/swagger.json". The elements of this format are as follows:

- ≪env≫, the third-level domain name, is the environment, e.g. "api" for the production environment and "api-uat" for the user acceptance test environment.
- ≪2ld≫ is the second-level domain name, e.g. a company name.
- ≪tld≫ is the top-level domain name, e.g. "com" or a country code.
- v≪major≫ is the major version number. See section 5 for more information.

Some people argue that having an API schema breaks the REST requirement of 'Hypermedia as the engine of application state'. I don't agree. I will discuss this topic at the end of section 6.

All resource URIs have the following format:

http(s)://«env».«2ld».«tld»/v«major»/«path»

or http(s)://«env».«2ld».«tld»/v«major»/«path»?«query-string»

Example: "http://api.demo.org/v1/project-a/planning/calendar?year=2016&month=02". In its simplest form, the «path» is the name of the resource, which is typically a noun, e.g. "project-a". If a resource is accessed via other resources, these resources are separated by slashes. For example, "project-a/planning/calendar" would refer to the calendar included in the planning of project-a.

If the resource is a collection, the noun is plural and is optionally followed by an ID to select one item in the collection. For example, "persons/12345" refers to the person with ID=12345.

If the resource is not a JSON resource, then its name has an extension, e.g. "photo.jpg".

Every path usually allows multiple HTTP methods; the method specifies what action to undertake.

```
GET    api.demo.org/v1/persons/12345
HEAD   api.demo.org/v1/persons/12345
PATCH  api.demo.org/v1/persons/12345
```

2. The HTTP body format

The format of the body of the HTTP message depends on the HTTP method, as summarized in the following table.

Method	Purpose	Request body format	Success response body format
GET	Get resource	empty	complete resource
HEAD	Check existance	empty	empty
POST	Create resource	incomplete resource	complete resource *)
PUT	Replace resource	complete resource	empty
PATCH	Modify resource	incomplete resource	empty
DELETE	Delete resource	empty	empty

*) In individual POST cases, it may be decided to return a large resource only partially.

For HTTP methods not mentioned in the table, JAREST does not specify a particular body format.

A ‘complete resource’ is any document, completely representing the resource. This is typically, but not necessarily, a JSON document, media type “application/x-jarest”. This type is not officially registered; if your infrastructure rejects it, you may use “application/json” instead.

An ‘incomplete resource’ is a JSON document in which not all properties and links of the resource have to be mentioned. In case of POST, the resource is created; all properties not mentioned get their default values. In case of PATCH, the resource is updated according to IETF RFC 7396 (tools.ietf.org/html/rfc7396). If a non-JSON resource, like a PDF or HTML document, is POSTed or PATCHed, then it must always be sent completely.

A ‘link’ is a piece of JSON that refers to a resource, optionally in combination with a particular HTTP method and a particular media type. In its most extended form, a link has the following format:

```
"linkname" : {
     "href" : "URI-template",
     "method" : "HTTP-method",
     "mediaType" : "media-type"
}
```

The “linkname” is an identifier in camelCase format, specifying what the link is intended for, e.g. “findPerson”. The “URI-template” is a relative URI, optionally containing parameters, e.g. “/v1/persons?name={name}”. Parameters are identifiers in camelCase, enclosed in braces. The “HTTP-method” is GET, HEAD, POST, PUT, PATCH or DELETE. It may be omitted, in which case GET is assumed. The “media-type” is the media type of the response, e.g. “application/pdf” or “text/html”. This is optional as well. The default value is “application/x-jarest”.

A link is just a property of a JSON resource. A link can be distinguished from a regular property, by the fact that its value is an object with at least property “**href**”. Property name “href” is reserved exclusively for this purpose. Example:

```
{
    "id": "12345",
    "name": "John Brown",
    "address" : {
        "href" : "/v1/persons/12345/address"
    },
    "telephoneNumbers" : {
        "office" : "+31201234567",
        "mobile" : "+31612345678"
    },
    "photo" : {
        "href" : "/v1/persons/12345/photo.jpg",
        "mediaType" : "image/jpeg"
    },
    "update" : {
        "href" : "/v1/persons/12345",
        "method" : "PATCH"
    },
    "delete" : {
        "href" : "/v1/persons/12345",
        "method" : "DELETE"
    }
}
```

In this example, address, photo, update and delete are links. As a REST interface designer, you may choose whether to send a complex resource in one go (as one big JSON document with nested objects), or to split it up and regard some of these nested objects as resources in their own right. In the example, the REST designer has chosen to regard “address” as a separate resource. I have tried to minimize the impact of such design choices; that’s why in JAREST, links are not located in a separate “links” property (see also section 6, paragraph f).

Reserved property names

As mentioned above, I have reserved property name “**href**” in order to distinguish links from other properties. On the other hand, “method” and “mediaType” are not reserved, i.e. an object without property “href” may have properties called “method” and “mediaType” with an other meaning.

Apart from “href”, there are a few more reserved property names. Links for updating and deleting the current resource are consistently named “**update**” and “**delete**”, respectively and shouldn’t be used for anything else. If there are multiple ways to update a resource, extra links with other names may be added.

Creating resources is discussed in section 4, but I’ll tell you that “**new**” and “**create**” are reserved property names for this purpose.

Finally, property name “**self**” is reserved in order to standardize the way our REST interface deals with partial resource embedding, as explained in the next section.

3. Partial resource embedding

What would happen if we would get the full list of persons (GET /v1/persons) ? Maybe we would get an array of links to the actual person resources. Maybe we would get an array containing all person data fully embedded. But most likely, we will get an array of partial persons, in which only the main person properties are present. In order to get more person details, every person has a link to get those details. By convention, the name of that link is “**self**”. In general, if a resource is partially embedded in the representation of another resource, then it is accompanied by a link called “self”, which can be used to get the complete representation of the embedded resource.

Here is a different example:

```
{
    "id": "12345",
    "name": "John Brown",
    "address" : {
        "country" : "Ireland",
        "self" : {
            "href" : "/v1/persons/12345/address"
        }
    },
    ...
```

This person representation shows “address” as a partially embedded resource. To get any address property other than “country”, the client application should follow the “self” link.

4. Creating new resources by template

To create a new JSON resource, the REST interface demands that the REST clients first get a **template**, then fill in the template and then post it.

To get a template, a link called “**new**” is available in the parent resource. For example, to create a person, a “new” link is available in the parent resource “persons”. Here is the first part of that parent resource:

```
{
    "new" : {
        "href" : "/v1/persons/new"
    },
    "persons" : [ {
        "ID" : "12345",
        "name" : "John Brown",
        ...
```

The template resource contains all properties we are allowed to post, plus one link called “**create**”. The values of the properties are the default values, or, for properties without a default value, the property values are empty strings. For instance, if we do GET /v1/persons/new, we get:

```
{
    "name" : "",
    "telephoneNumbers" : {
        "office" : "",
        "mobile" : ""
    }
    "create": {
        "href": "/v1/persons",
        "method": "POST"
    }
}
```

Properties “ID”, “address” and “photo” are not present, because we’re not allowed to post these. The ID can’t be posted, because it will be assigned a value automatically. Address and photo can’t be posted, because these are separate resources. When the person is created, the server will also create an address resource, with empty properties and link it to the new person. The photo property will get value null, or it may be a link to a default image (e.g. a silhouette image).

This is what we can post to /v1/persons:

```
{
    "name": "John Brown",
    "telephoneNumbers" : {
        "office" : "+31201234567",
        "mobile" : "+31612345678"
    }
}
```

In this example, we didn’t use default values, but you can imagine that the template could contain a property “isVIP”, which has value “false” and that we may post value “true” if we want John Brown to be a VIP. Properties with default values don’t *have* to be posted. If omitted, they will get their default values.

This pattern for creating resources in two steps has two benefits. The most important one is, that it gives the server the opportunity to specify context-sensitive default values. Fixed default values can be specified in the schema, but it frequently occurs that default values depend on the user’s profile or the state of the business process. Secondly, the template provides the client with an object ready to fill in, which might be easier to use for developers than the schema.

For creating resources with media types other than JSON, this pattern does not apply. There will be links with appropriate names for doing this. Example:

```
{
    "id": "12345",
    "name": "John Brown",
    "address" : {
        "href" : "/v1/persons/12345/address"
    },
    "telephoneNumbers" : {
        "office" : "+31201234567",
        "mobile" : "+31612345678"
    },
    "photo" : null,
    "uploadPhoto" : {
        "href" : "/v1/persons/12345/photo.jpg",
        "method" : "POST",
        "mediaType" : "image/jpeg"
    },
    ...
}
```

5. Versioning policy

Semantic versioning (in accordance with semver.org) is applied to the schema and to each resource. Multiple operations on the same resource do not have separate version numbers. The major part of the version number is included in the URI, as specified in section 1. The full version number of the schema and of each resource is specified in the schema.

When a new major version of a resource is created, a new major version of the schema is also created. The old versions are either removed, or remain available, if there are still clients that make use of them.

As explained in section 1, an application starts communicating with a REST interface by getting a particular version of the schema. The schema URI is specified in the application’s configuration file. If a new major version of the schema is created (and the old version is not removed), the client applications will still use the old version and will not be impacted. Applications that want to benefit from the new version, will have to update their configurations. Client applications should not bother about individual resource version numbers. They should follow the links provided in the schema and in the resources and they will automatically get the appropriate resource versions.

When a new patch or minor version of a resource is created, the old version is replaced, because the URI does not change. The schema is replaced as well and its patch or minor version number is adjusted. This means that clients silently move to the new version. For instance, if a property was added, or if a property that was mandatory has now become optional, then only the minor version is increased. If we demand that our clients are very flexible, we could even say that replacing a "GET" link by an embedded or partially embedded resource, or vice versa, is a minor change, but I think such requirements on clients would be too ambitious, as I will explain in section 6f.

One resource R1, version $x_1.y_1.z_1$, may have a link to another resource R2, version $x_2.y_2.z_2$.

- When a new patch or minor version of R2 becomes available (y_2 or z_2 is increased), then R1, version $x_1.y_1.z_1$, will point to that new version. Although the link to R2, as present in R1, changes, there is no need to change the version number of R1.
- When a new major version of R2 becomes available, version $(x_2+1).0.0$, then R1, version $x_1.y_1.z_1$ will keep pointing to R2, version $x_2.y_2.z_2$, but a new version of R1, $(x_1+1).0.0$ will be created, which points to R2, version $(x_2+1).0.0$.
- When a new version of R1 becomes available, R2 does not change.

Whenever an arbitrary resource gets a new major version number, this will trigger a chain of version increments of resources that directly or indirectly refer to that resource.

6. How to apply the HATEOAS principle

By definition, a REST-compliant interface applies the principle of 'Hypermedia as the engine of application state' (HATEOAS). Strictly speaking, a REST client should not assume that a resource has a specific structure beyond the structure that is defined by the media type. In other words, if the media type is "application/x-jarest", the client should be able to accept any document that follows the JAREST pattern, like a Web browser accepts any HTML document. If all REST clients would be that generic, the user experience would not be very good. But still, an interface can only be called RESTful if, *in theory*, it would be possible to reach all its functionality using a **generic client application**.

Let's take the example of person 12345, John Brown, as shown in section 2. A generic client application will show the ID and the name, as well as one button for every link (address, photo, update and delete). If the user presses the "address" button, the application will get that resource and will again show its properties as data and its links as buttons. On pressing the "photo" button, the client will show the photo image. Link names "update" and "delete" are reserved for updating and deleting the current resource, respectively, and the generic client application knows that. If a link called "update" is present, it will make all properties editable. When the user presses the "update" button, it will first check the validity of the property values, using the schema, and then send the updated properties to the server using the URI and the HTTP method specified in the "update" link. Similarly, the generic client knows about our pattern for creating resources (see section 4) and can supply that functionality to the user as well.

In practice, we don't build generic, but *dedicated* client applications, which have knowledge of the specific properties and links of the resources, in order to arrange the data, the buttons etc. in a user-friendly manner on the screen. We may offer only a subset of the available functionality (i.e. ignore some of the links), we may combine functionality of multiple resources on the same page (not only the functionality of the resource most recently fetched), we may offer breadcrumbs functionality, etc. Especially when applying a microservices architecture, applications may use many independent REST interfaces and REST interfaces may be used by multiple applications, in which case it becomes clear that a single REST interface will not be able to dictate the full state of the presentation layer.

But even though we don't have a generic client application, we should design the REST interface *as if* we had. This will make the interface more self-explanatory. More importantly, it offers clients the option to choose how generic they want to be, i.e. how resilient to server-side changes they want to be. Clients which need to provide a great user experience may still be resilient to URI changes, to changes in default values and more. Clients doing batch jobs without user interaction may be resilient to other kinds of changes. We can easily make dedicated clients a lot more flexible now – not as flexible as a generic client, but still gaining some important benefits, most notably that business rules are maintained by the server and not duplicated in the client. Client applications should...

a) allow the server to change the URIs;
b) allow the server to hide or delete properties;
c) allow the server to disable or discontinue actions;
d) allow the server to change default values of properties;
e) allow the server to change validation rules;
f) allow the server to change even more?

a) Allow the server to change the URIs

Client applications know (hard-coded) the names of the links a resource may have, but not the corresponding URIs. To get a URI, it should look at the 'href' property of a link. For example, if a client has already got "/v1/persons/12345" and now, it wants to get the address of this person, then it should not concatenate "/v1/persons/12345" and "/address", but it should take the "address" link of the person resource. To update person 12345, it should not just do a PATCH using the same URI used for getting the resource, but the client should use the "update" link of the person resource and use the URI and the HTTP method specified by that link. This form of indirection enables the server to change URIs without breaking the client.

b) Allow the server to hide or delete properties

Client applications know (hard-coded) the names and the types of the properties a resource may have, because usually they have specific look-and-feel requirements for specific properties. Additional properties may be added in the REST interface without breaking the clients – well, that's not a big deal. But clients should also allow the REST interface to omit some of the properties they expect. Properties may have been deleted, because the product owner has decided that they have no business value anymore. Properties may also be present in the database, but not returned to the client application, because the user is not authorized to view them, or there is another business rule that demanded the properties to be hidden in the current context. Client applications should not break if some properties are absent. This makes them more resilient to change. Okay, you may make an exception for properties that are inherently mandatory, like IDs.

c) Allow the server to disable or discontinue actions

Client applications should never assume that a particular link is always present. For example, if person 12345 does not have link "address", then the client application should not break, but deal with the fact that the person does not have an address. If person 12345 does not have link "update", then the option to update the person should be disabled in the application. Particular links will not be returned by the REST interface in case the target resource does not exist, or if the corresponding action on the resource is prohibited by a business rule or a permission restriction.

d) Allow the server to change default values of properties

See section 4. Default values should be obtained from a resource template.

e) Allow the server to change validation rules

Validation rules should be obtained from the schema instead of being hard-coded in the client. Complex validation rules, that cannot be expressed in the schema, should not be validated by the client at all, but only server-side. Clients should be prepared to receive HTTP response codes other than 200 and act accordingly (display an appropriate error message etc.).

f) Allow the server to change even more?

It always good to check for more opportunities to make the client applications more flexible and generic, but there should be a decent business case for it. For example, it might be a good idea to allow the server to replace links with method "GET" by embedded resources or partially embedded resources, as defined in section 3. This could be implemented in a reusable JAREST access layer, through which the client applications performs all REST requests. I haven't tried this myself, but I assume it is very well doable. The other way around is not feasible, I think: allowing the server to replace any embedded resource by a link without impacting the client application would lead to an unacceptable increase in client complexity – but again, I have never made an attempt to implement this.

Note that the flexibility requirements you demand from your clients also influence the version numbering. For example, if all clients allow the server to replace links by embedded resources, then this kind of change is a minor change, but otherwise, a new major version must be created, because it introduces an incompatibility.

Can an interface with a schema still be called RESTful?

Every JAREST interface publishes a schema, called "swagger.json" (see section 1). Some people say that it is better not to provide a schema or any other documentation, in order to force clients to be generic and have no out of band information. As explained above, it is not feasible to build user-friendly client applications without hard-coding some metadata, so a schema is valuable information for client developers. But another important reason to have the schema is actually to allow clients to be *more* generic. This is because the schema contains information about data types and validation rules (e.g. regular expressions), which can be discovered dynamically by client applications instead of being hard-coded. In that case, when the validation rules change, the clients don't have to be modified. This supports the notion of HATEOAS rather than jeopardizing it.

7. Your feedback

I'm very interested in your ideas to make JAREST clients more flexible. In general, any comment on the JAREST pattern is very much appreciated.

Hans Admiraal, (freelance IT architect)
admiraal@aol.nl
www.admiraalit.nl