RFC API-3: LiveRamp API Standards

Summary

This document establishes the standards that all LiveRamp's REST[1] APIs should follow. This will help consistency with our APIs, and explain the philosophy of our approach to developers implementing them.

This document will also serve as a guide when approving APIs in our API review process.

Tags: [api, standards]
Categories: []
Owner: Andrew, Abhishek
Version: 0.4.0
History: [0.1.0] Started as part of the API work (Abhishek), [0.1.1] Updated codes returned for actions (Andrew + Hashir) [0.2.0] Added PATCH & described anti-patterns (Andrew) [0.3.0] Made it standard to return a single object for errors, and array as an alternative. (Andrew) [0.4.0] Added appendix B to describe the evolution and versioning approach more directly. (Abhishek) [0.5.0] Services should ignore extra fields to support future compatibility.
Key stakeholders: [architects, squad leads]
Affected teams: [architects, squad leads]

The Problem

We want to expose our APIs as REST resources. However, REST is an architectural style, and does not mandate URL conventions or best practices. Further, HATEOAS[2] and other hypermedia concepts can result in complexity we wish to avoid.

This RFC defines standards around our REST APIs, to ensure uniformity across areas. It also covers items that relate to keeping our API stable and dependable, including evolution and backwards compatibility.

[1] Please see Appendix for REST primer.
[2] HATEOAS stands for Hypermedia as the Engine of Application State. It's a foundational technique where links are always returned in response bodies for navigation purposes - the dream is to make a world where clients need no prior knowledge of how to interact with an application except via a generic understanding of hypermedia concepts. This approach usually adds complexity to APIs, precisely because no client would operate in such an unknown state. https://en.wikipedia.org/wiki/HATEOAS

Evolving These Standards

This represents v0.1 of our standards. As we implement MVP with one of the teams, we will present our observations and put out another RFC version for discussion.

Tooling

We will supply tools which can generate the Swagger specification from a resource model - this will be described in detail in a future RFC. This will ensure that our APIs adhere to these standards, removing the chance of accidentally breaking them.

Requirements

Here is a non-exhaustive set of areas we need to have answers for:

1. What is our general API philosophy?
2. What do URLs look like and what are the rules?
3. How do we expose sub-resources?
4. Do we return full representations from a multi-GET or just IDs?
5. How do we handle versioning and evolution?
6. How do we handle search requests with large parameter bodies?
7. How do we handle large request and response bodies? Do we paginate?
8. What do we do with synchronous requests and long running, async requests?

Proposal

Philosophy & Mechanics

1. **We will use a REST-based, resource oriented approach.**
    a. Resources are the nouns in our domain.
    b. Our API should be modeled around explicit resources and support some or all of the Uniform HTTP interface - GET, PUT, POST and DELETE, with their standard semantics of “read, update, create and delete resource”.

2. **We will use HTTP(s) and JSON as the primary MIME type.**

   a. JSON should be supported as a default, unless the data volume etc requires a different, more efficient representation.
   b. Must support non compressed and gzip compressed requests and responses.

3. **Standard data representations.**
   a. Datetimes, dates, times and durations should be formatted as ISO8601 compliant strings with timezone information or in UTC. E.g. "1994-11-05T08:15:30-05:00" or "1994-11-05T13:15:30Z". We do not want times that may be misinterpreted as being in our local time. All datetimes in our responses should be in UTC.
   b. Text will be encoded as UTF-8 for internationalization purposes. We have no position currently on internationalized error messages.

4. **We will use a standard naming scheme for URLs and parameters.**
   The URL scheme is: .../namespace/major-version/resource-name/resource-id

   a. **Namespace:** Used to indicate a collection of related APIs such as Identity, Data Management, Onboarding, etc., aligning 1:1 with the capability or product areas (to be defined). This segments our API surface into manageable parts, and will also be useful internally for routing. So Note that the major version refers to the resource version rather than the namespace. Namespaces could be hierarchical.
   b. **Versioning:** Only the major version of the resource is included in the URL, preceded by a “v”. We only increase this version if an endpoint cannot be kept backwards compatible with the older version. More on this below in the section on evolution. This scheme allows us to support multiple major versions simultaneously.
   c. **Resource Names:** These are the nouns of the system and we can implement all or some of GET, POST, PUT and DELETE operations. Resource names should be plural and multiple words are hyphenated. Resources can be nested to indicate subresources (see below), but as a rule of thumb, we shouldn’t have URLs with more than two levels.
   d. **Resource ID**: Identifies a specific instance of a resource. For e.g., in v1/cars/12, “v1” is the version of the “car” resource and “12” is the identifier of a particular car instance.
   e. **Parameters**: All parameters (in path, query or request body) should be camelCased. For e.g., v1/rental-requests/{rentalRequestID} where “rental-requests” is the resource name and “rentalRequestID” is the parameter name.

      c

5. **We will use Swagger / OpenAPI Specification to document the API and JSON payload / response.**
   The OpenAPI specification should include full model information describing input / output data structures for each API. We will supply tools which can generate the Swagger specification from a resource model - this will be described in detail in a future RFC.

   Swagger specifications will be made available through standard URLs. This will allow the API to be published with the API gateway / management portal. This will also enable third party developers to generate client libraries & our teams to generate server stubs from the specification.

Standard Operations & Parameters

1. **Use a GET with an ID to return a representation of a single resource.**
   E.g. GET vehicles/v1/cars/7

   The resource ID exposed should be alphanumeric (with possible hyphens), for e.g, ../v1/cars/1234, ../v2/users/john-doe etc.

   Note that if you expose primary DB IDs, clients may store that reference which would need to be maintained. With auto-incrementing numeric IDs, clients can discern the size of a given resource and even enumerate entries - this may not be an acceptable risk for specific domains.

2. **Use a GET with filter parameters or body for retrieving multiple resources.**
   Multiple parameters of the same type are separated by commas and represent an OR operation. Multiple parameters of different types represent an AND operation.

   E.g. GET vehicles/v1/cars?make=toyota,ford&type=compact returns an array of “compact” cars AND the make is either “toyota” OR “ford”.

   This data can be paginated - see section below. Path and body parameters are up to the individual resource. The full resource representation will be returned, not a list of IDs.

   GETs with large filter parameters will be implemented as POST queries to allow using request body -- more about search requests below.

3. **Use a POST with a body to create a resource.**
   E.g. POST vehicles/v1/cars with body {.. “type”: “sedan”, “color”: “black” ...} returns { “id”: 20} and status 201.

   Multi-POSTs to create multiple resources should be handled as asynchronous requests (described later).

4. **Use a PUT with an id in the URL, and a body, to update a single resource.**
   This should return 200 ok upon successful update.

   A PUT should always include all non-optional fields in each request. If you want a partial set of required fields, instead use PATCH.

   Similar to multi-POSTs, multi-PUTs to update multiple resources should be handled as asynchronous requests.

Updates which require asynchronous processing (e.g. deactivating an account) should be handled via asynchronous requests as described later.

5. **Use a PATCH with an id in the URL, and a body, to update any fields in a single resource.**
   This should return 200 ok upon successful update.

   Similar to multi-POSTs, multi-PUTs to update multiple resources should be handled as asynchronous requests.

   A PATCH differs from a PUT, in that even required fields are optional in the payload. It can be used to update any field. Prefer a PUT for its idempotency guarantee.

   NOTE that both PATCH and PUT should be used only to update a resource's state. It should never be used to trigger implicit commands or start processes - this is a massive antipattern - e.g. updating "status" triggers an asynchronous job. If you need to do that, use a request or action (see below).

6. **Use a DELETE with id in the URL to remove a resource.**

7. **Multi-DELETEs work off the resource name, and accept filter parameters.**
   E.g. DELETE vehicles/v1/cars?make=toyota&model=prius

Singleton Resources

1. **If there is truly only one instance of a resource, do not pluralize it.**
   In this case the resource name itself is the ID.
   E.g. GET vehicles/v1/car-registry returns the single car registry.

2. **Subresources can be singletons as well.**
   E.g. GET vehicles/v1/cars/23/steering-wheel

3. **Be very certain that you will only ever have one of these resources.**
   E.g. what happens if cars are made with two steering wheels.

Subresources & Other Associated Resources

1. **If a resource contains fully owned objects, that need to be treated as first class resources, then it is ok to explicitly expose these as subresources.**
   E.g. vehicles/v1/cars/23/wheels/12 is the wheel belonging to car 23.

   Subresource IDs may be global (can be identified without the parent resource ID) or scoped within the top-level resource (needs to be qualified by the parent resource ID), depending on the data model.

Note that typically GETting a car would not return the wheel subresources in the car representation, although it is reasonable to include IDs. Note that the subresource representation does not have its own version. Use subresources sparingly.

2. **Multi-GETs for subresources behave identically to top level resources.**
   E.g. GET vehicles/v1/cars/23/wheels will retrieve all wheels for car 23.

3. **If a resource representation has connections to other resources, it should specify the IDs in a field with a suffix of "{resource}ID" or "{resource}IDs" to indicate the type.**
   E.g. If a car refers to a garage the body might look like: {..., "homeGarageID": 75… }

Asynchronous & Synchronous Requests

1. **Asynchronous, long running actions should be turned into a resource with -requests as a suffix (aka resourcification).**
   If a job is long running with multiple state changes, turn the workflow into a top-level resource.

   E.g. POST vehicles/v1/service-car-requests will create a resource which can be inspected via GET, updated via PUT, and DELETEd.

2. **Actions on long-running asynchronous requests should be modeled as verbs, with an "/actions/" path prefix.**

   E.g. POST vehicles/v1/service-car-requests/{serviceCarRequestID}/actions/cancel to cancel a given ServiceCarRequest resource.

3. **State changes on asynchronous requests will be notified out via a notification system.**
   We have not chosen this notification system, but it will be akin to something like Google pub-sub with permissioned topics. The full resource representation will be sent out on the queue with selectors allowing clients to filter out unwanted state changes.

4. **If you issue an action, you may get a different response depending on the state of the resource, and whether the action is synchronously completely or async**
   a. 200 (success) if the operation has been completed synchronously
   b. 202 (accepted for processing, but not completed) if this is an async action and it hasn't fully been completed yet
   c. 204 (The server has successfully fulfilled the request already) if this action has already been submitted and we are currently doing it, or if it is redundant. Note that you will get no creation of a resource, or content returned but it is not an error.
      E.g. A Distribution request is already stopped or processing to stop, and a user makes an API call to stop it.

d. 409 if the current state does not allow for this action
   E.g. job is completed and we asked it to be paused.

Search Requests

1. **Multi-GETs with large parameter sets should be alternatively handled via a search-requests resource which implements POST.**
   E.g. POST vehicles/v1/car-search-requests creates a resource and returns an ID. A GET on vehicles/v1/car-search-requests/{id}/cars returns the results of the search.

2. **If the search process can run for a long time, it should be handled as an asynchronous request.**

Evolution and Backwards Compatibility

The scheme described below allows new versions of individual resources to be introduced incrementally.

1. **Major version for each resource must be in each URL.**
   Note the position of the v1 in the URLs above - this is for the major version of the resource, not the entire namespace. In this way, we allow versioning each resource within a namespace separately. This will allow us to evolve different parts of our APIs at different rates, rather than having to evolve every resource in a namespace at the same time. However, a given team can choose to upversion each resource in synchrony if they wish to evolve the entire namespace.

   There are 2 different ways to do versioning - one via the version number in the URL, another with it in the header. The issue with the latter is that it doesn't translate well into a discoverable API - you can't easily see which versions are supported.

   The rationale for placing only the major version is that we expect that minor version upgrades will be backwards compatible with previous minor versions. This also leads onto the next point.

2. **Clients implementations must handle additional fields.**
   Clients should ignore extra fields sent back from a GET. This allows us to add extra field representations in minor version updates, knowing we won't break the client.

3. **Service implementations must ignore extraneous fields to be forward compatible.**
   APIs must support forward compatibility by allowing clients to pass extraneous fields intended for a future version of the API. In such cases, a service implementation should ignore any fields that it does not understand.

4. **We can support as many major versions as required.**
   In a perfect API we would never have to remove a method that a client depends upon. The above scheme allows us to support any number of major versions for a given resource, allowing us to make a more dependable platform.

Error Handling

1. **We will use a consistent format for returning errors, and map onto standard HTTP status codes.**
   Errors should be returned as the following object:

```
{
    "httpStatus": the integer HTTP error status code for this problem,
    "errorCode": "service specific error code, more granular",
    "message: "general, human readable error message"
}
```

   For complex cases that require more than a single error code in a response, use an array keyed by `errors` in the top level response, with each error object formatted as above.

   Standard error codes (which could be added to, on a server specific basis) are:

   400 - bad request
   - BAD_HEADER_PARAMETER
   - BAD_QUERY_PARAMETER
   - BAD_PATH_PARAMETER
   - BAD_JSON_FORMAT

   401 - client unauthenticated
   - TOKEN_INVALID
   - TOKEN_EXPIRED

   403 - client unauthorized

   404 - not found
   - RESOURCE_NOT_FOUND
   - SERVICE_ENDPOINT_NOT_FOUND

   422 - input validation error

   429 - too many requests (for rate limiting)

Large Bodies and Responses

1. **Support for pagination to handle GET responses with large data sets.**

All multi-GETs should allow two query parameters `offset` and `limit` to support pagination.
The API response should also include an additional header `X-Total-Count` to indicate the total number of records in the data set.

Optionally, the API response can include pagination links in the Link header field if the service decides to do so. For e.g.,

```
Link: <https://api.lr.com/v1/cars?offset=10&limit=10>; rel="next",
<https://api.lr.com/v1/cars?offset=0&limit=10>; rel="prev",
<https://api.lr.com/v1/cars?offset=0&limit=10>; rel="first",
<https://api.lr.com/v1/cars?offset=150>; rel="last"
```

This would allow API consumers to navigate through the data sets without having to construct the URLs themselves. This is optional, nice to have if an API owner wants to provide this convenience to their API consumers, but in this case the service would be responsible to construct the URLs for the correct host (staging, prod, etc) which comes with its own cost.

2. **POST and PUT requests should support `Transfer-Encoding: chunked`**
   This allows large bodies to be transferred in multiple requests.

Authentication and Authorization

These topics will be covered in a subsequent RFC.

References

1. URL naming conventions: https://restfulapi.net/resource-naming
2. Swagger doesn’t support GET with request bodies: https://github.com/swagger-api/swagger-ui/issues/2136
3. Example of an API using OpenAPI specification - https://petstore.swagger.io
   (Note that this is just an example of OpenAPI spec and not the conventions described above.)

Appendix A: Quick REST Primer

REST stands for Representational State Transfer. Wikipedia describes it as “a software architectural style that defines a set of constraints to be used for creating Web services. RESTful Web services allow the requesting systems to access and manipulate textual representations of Web resources by using a uniform and predefined set of stateless operations”. We call this view of REST a resource-oriented approach, as it focuses on web resources (aka nouns) as the primary element, which can be created, read, updated and deleted.

As expected, when using HTTP, resources are identified by a URL - e.g. https://liveramp.com/dm/v1/segments/12

Although the application of the technique varies, it is essentially oriented around “resources” that are operated on using a collection of methods/operations termed the “uniform interface”. We can

1. Create a resource using a POST operation
   This creates the resource on the server and returns the id of the created resource.

2. Read resource(s) using a GET operation
   We pass in parameters identifying the resource(s) to retrieve.

3. Update a resource in-place using a PUT operation
   This is idempotent (aka can be repeated many times if needed due to concurrency or failure) because the client passes the id and full representation each time.

4. Delete resource(s) using a DELETE operation
   We pass in parameters identifying the resource(s) to be deleted.

The REST style has become a vitally important way to design APIs that operate at scale (due to statelessness), and do not require continual tinkering as new use cases are added (due to the uniform interface).

Even though this all sounds abstract, you actually have experienced this style more than any other - it is how the web works! A browser is GETting an HTML page resource which it then displays. A browser can upload to servers using POST and PUT techniques.

REST by itself is not able to handle the full complement of workflows required in a business system. To deal with this, we relax some of the constraints on this style, and turn these workflows themselves into first-class resources, with notifications telling us when they asynchronously change state.

Appendix B: Evolving & Versioning REST APIs

The purpose of this appendix is to clarify the way we wish to evolve and version APIs and resources. The standards above describe this approach, in an indirect way - this appendix attempts to make it much clearer.

For this example we will use the Reslang spec here, which created the Swagger spec here. The example is a bit nonsensical - consisting of ResourceA and ResourceB resources, but it will do for this discussion.

Firstly, note that we have 2 concept of versions in each API:

a. **The overall semver for the API (e.g. 1.1.2)**
   This information lives in the info/version field of each Swagger doc. No part of this API version information appears in URLs.

```
info:
     title: An API showing evolution
     description: This is the description of the API
     version: 1.1.2
```

   The same information can be seen in the Reslang namespace definition.

```
"This is the description of the API"
namespace {
     title "An API showing evolution"
     version 1.1.2
}
```

b. **The semver for each resource inside that API**
   Each resource has its own separate semver. We only put the major version in the URL because clients of the API only care to know about breaking changes. E.g.

```
/v1/resource-as/{id}
```

   Note that the /v1 applies specifically to ResourceA, not to the API as a whole. If we now add a field to the resource, but it is optional or only provided on output, then this is a minor change and we don't need to reflect that in the resource's URL.

   Note that in Reslang, unless specified, a resource is considered to be v1. If it is a different major version, you need to be explicit:

```
resource v2/ResourceFoo { … }
```

How does all this work together?

Firstly, remember that you release the entire API, not individual resources. So if you make a set of backwards compatible changes to resources, then bump the minor or patch version of the entire API when you release it.

Also, because you can support multiple major versions of the same resource in a single API, there are relatively easy ways to avoid having to break backwards compatibility of the entire API. If you need to make a breaking change of a single resource - you just support v1 and v2 (say) of the same resource in the same API. You only need to bump the major version of the entire API when you stop supporting v1.

Here’s a simple example:

a. **Released v1.0.0** v1/ResourceB, and v1/ResourceA which links to v1/ResourceB
b. **Released v1.0.2** added output only field to v1/ResourceB
c. Later, you decide you need to make a breaking change to ResourceB, but are happy to support both v1 and v2 of this resource
d. **Released v1.1.2** v1/ResourceB and v2/ResourceB, with v1/ResourceA having optional links to either v1/ResourceB or v2/ResourceB to avoid breaking clients

Note step (d) is the current state of the Reslang spec for this API.

```
"This is the description of the API"
namespace {
   title "An API showing evolution"
   version 1.1.2
}
resource ResourceA {
   id: int
   name: string
   bId: linked ResourceB
   newBId: linked v2/ResourceB

   /operations
       GET
}

resource ResourceB {
   id: int
   size: int
   /operations
       POST GET PUT
}

resource v2/ResourceB {
   id: int
   totalSize: int
```

```
        /operations
            POST GET
    }
```

Note that this approach doesn't shield you from the difficult bit of deciding how to support both v1 and v2 of ResourceB. Because the links in the above spec are not optional, the implication is that you can create a v1/ResourceB and GET it as v2/ResourceB. This is not always the case…

A breaking API change (e.g. 1.1.2 -> 2.0.0) is a serious situation. You will need to contact the clients and work with them to make sure that they can migrate. In some cases, you might even want to support both v1.1.2 of the API and 2.0.0 on separate URLs. Our approach allows for this also.