DLI REST-style API Reference

Generated by Doxygen

20170928T215054Z

Contents

1 REST-style API Reference

1.1 REST-style API overview

REST is an architectural style where data are presented and manipulated as an hierarchy of resources, and the underlying communications protocol (commonly HTTP) is used to perform action signaling, content negotiation, etc [11]. In practice this means automated clients for it are easier to implement; they can send and receive state in different formats (representations of resources). The REST-style API follows the principles of REST principles where possible without sacrificing simplicity or adding unnecessary bloat.

1.1.1 Target audience

The REST-style API is designed to be used for automating access to the device, but can also be used manually with the browser. Due to the way device configuration is implemented, any new configuration variables are exposed to the REST API automatically, while designing would take time.

1.1.2 Design goals

- Simple uniform widely-implemented interfaces

  The REST-style API identifies resources with URIs [2]. Just about everything there is to know about the controller (states of the outlets and their names, different environment meters, AutoPing items, network interfaces and authorized users, etc.) has its place in an hierarchical URI namespace. Groups of similar resources (e.g. all locked outlets, or all voltage meters, or all enabled AutoPing items) have matrix URIs (see later) to reference them as a group.

  The API relies on HTTP, a widely implemented protocol, to perform:

  - action signaling (what to do with the specified resource);
  - content negotiation (what format you want to input data in and how you want to receive output);
  - limiting output verbosity or nesting (only get the data you need);
  - related resource discovery (with the Link headers).

- Feature compatibility with existing web UI and legacy API

  If you are reading this, you have probably used some scripts to access the DLI power controllers. It wasn't convenient, sometimes involved reverse engineering the web pages output by the devices, etc. But it allowed you to automate a lot of device configuration and control tasks. The REST-style API intends to allow access to the same, and more parameters, in a uniform fashion. The legacy APIs will remain in place, but now there's an alternative.

- Discoverability

  You can use your web browser to access the API, to discover the visible configuration and state resources, and manipulate them. That's actually what the web UI advises you to do.

  Discovering the API without a browser is also possible, just less convenient. For example, you can get a representation of the whole configuration by issuing a GET request to the API root (/restapi/) requesting output as JSON (Accept: application/json). However, without extra information it's not obvious what further actions you can take, and HTML output explains most of it to you (and provides forms for you).

- Extensibility

  The way the API is designed makes it easier for us to integrate new features into the controllers, and for you to find them. For example, if you have a script to cycle all outlets, it will really cycle them all, even if the controller has more outlets than you expected.

- Ease of use for common operations

  The underlying data model and primary interchange format are based on JSON, a compact widely supported notation [3]. For example, you can switch off all non-locked outlets, or retrieve the states of all outlets at once, with a single request (in the latter case you will just receive an array of states if you need, you won't have to parse HTML), and that without sacrificing the uniformity.

- Flexibility

  Requests with different HTTP headers will may accept different argument types and yield different representations of resources (e.g. plain text, HTML, JSON, etc.). If you are interested in a yet unsupported encoding, please let us know and we'll happily add it, without sacrificing compatibility.

- Consistency

  Different means of achieving seemingly same results should be as equivalent as possible. For example, you could change a property of an object by issuing a PUT request to that property, or by issuing an appropriate PATCH request to parent (object's) URL, and the behavior should be the same; same goes for DELETE.

1.2 REST architectural style introduction

REST exposes the manipulated objects as an hierarchical collection of resources. You may think of such collection as of a directory in a file system, an SNMP OID subtree, etc. Resources can be retrieved, modified, created and deleted; all of these operations are performed using standard HTTP methods, with well-defined semantics and behavior adjustable by standard HTTP headers.

1.2.1 HTTP verbs

In the REST architectural style, HTTP verbs are used to indicate what you want to do with the resource.

The following HTTP verbs are supported:

- GET

  GET is used to retrieve representation of the resource.

- HEAD

  HEAD is similar to GET, but returns only the headers for the resource.

- POST

  POST is used to create a new element in the collection resource, or perform the function call.

- PUT

  PUT is used to create or overwrite the resource.

- PATCH

  PATCH is used to update the resource in place, possibly conditionally.

- DELETE

  DELETE is used to delete the resource from its collection.

- OPTIONS

  OPTIONS is used to preflight cross-origin requests (it's usually sent internally by browsers).

1.2.2 HTTP request headers

Request headers inform the server about what and how the client wants the server to return. Here are the more important ones:

- Authorization

  Authorization contains access credentials; it's usually sent internally by clients.

- Accept

  Accept indicates client's preferences in the content type of representation to be returned by the server.

- Content-Type, Content-Length

  Content-Type and Content-Length in the request indicate the kind and size of payload the client sends to the server.

- Range

  Range allows to select only a portion of the resource to be retrieved. We define a specific kind of range that limits tree nesting.

- Prefer

Prefer allows to customize different aspects of server behaviour, e.g. whether the full representation of the resource should be returned, etc.

- X-Requested-With

  An X-Requested-With: XMLHttpRequest header indicates that XMLHttpRequest has been used to issue the request (it's used for cross-side request forgery prevention).

Additionally, the following nonstandard headers are supported:

- X-HTTP-Method

  X-HTTP-Method can be used to emulate a PUT/DELETE/PATCH request if the client doesn't support it (it is honored in POST requests only).

- X-CSRF

  An X-CSRF header with any value can be used to mark legitimate requests (used for cross-side request forgery prevention).

1.2.3 Content types

1.3 Basic examples

These examples do not intend to describe the API completely; rather, they show the ways in which it can be used to encourage further reading. They use 'curl', a popular HTTP command-line client, adn assume you have a unit accessible at

```
http://admin:1234@192.168.0.100/
```

Commands and responses are formatted and sometimes truncated for readability.

- Retrieve all top-level resources in the REST API:

```
curl -H "Range: dli-depth=1" -H "Accept: application/json" \
--digest 'http://admin:1234@192.168.0.100/restapi/'
```

Result:

```
200 OK
```

```
{
 "relay":    {"$ref":"relay\/",    "title":"Relay object"},
 "auth":     {"$ref":"auth\/",     "title":"Authentication server object"},
 "config":   {"$ref":"config\/",   "title":"Configuration server object"},
 "network":  {"$ref":"network\/",  "title":"Network configuration data"},
 "autoping": {"$ref":"autoping\/", "title":"Autoping server object"},
 "script":   {"$ref":"script\/",   "title":"Scripting server object"},
 "meter":    {"$ref":"meter\/",    "title":"Meter server object"},
 "renderer": {"$ref":"renderer\/", "title":"Renderer object"}
}
```

- Retrieve two levels of generic configuration-related resources in the REST API in freeform text:

```
curl -H "Range: dli-depth=2" -H "Accept: text/plain" \
--digest 'http://admin:1234@192.168.0.100/restapi/config/'
```

Result:

200 OK

object: allow_plaintext_logins = false protect_private_config = false hide_wifi_key = false custom_brand_↩ url = false syslog_ip_address = 192.168.0.1 protect_firmware = false is_registered = false hide_passwords = true protect_network = false custom_brand_name = false known_quantities = illuminance=(...Quantity...), current=(...Quantity...), [truncated] allow_jsonrpc = false https_port = 443 custom_brand_logo_height = false refresh↩ _enabled = true protect_maintenance = true custom_brand_logo_width = false http_port = 80 known_timezones = UTC-6=(...Timezone...), UTC+4=(...Timezone...), [truncated] protect_admin = false allow_restapi = true relax↩ _nonhtml_content_types = false lockout_delay = 60 plots = (...Plot...),(...Plot...), [truncated] ssh_enabled = true timezone = UTC ssh_port = 22 upgrade_notify_beep = false meters = (...Meter configuration...),(...Meter configuration...), [truncated] custom_brand_company_name = false refresh_delay_minutes = 1 links = (...Link...) version = (...Version...) hardware_id = (...) custom_brand_logo = false image_format = svg hostname = (...Hostname...) serial = (...Serial...) relax_nonhtml_methods = false upgrade_notify_blink = false

- Switch all outlets off:

```
curl -v -X PUT -H 'X-CSRF: x' -H "Accept: application/json" --data 'value=false' --digest
       'http://admin:1234@192.168.0.100/restapi/relay/outlets/all;/state/'
```

Result:

```
207 Responses from multiple resources follow
```

- Add a link to the web UI:

```
curl -H 'X-CSRF: x' --data 'href=http://www.digital-loggers.com/&description=Digital+Loggers' \
--digest 'http://admin:1234@192.168.0.100/restapi/config/links/'
```

Result:

```
201 created
```

- Get states of outlets #1,#2 and #5 in a single JSON array:

```
curl -H "Accept: application/json" --digest \
'http://admin:1234@192.168.0.100/restapi/relay/outlets/=0,1,4/state/'
```

Result: 207 Responses from multiple resources follow

```
[true,true,true]
```

- Get names and physical states of all locked outlets in a single JSON array:

```
curl -H "Accept: application/json" --digest \
'http://admin:1234@192.168.0.100/restapi/relay/outlets/all;locked=true/=name,physical_state/'
```

Result: 207 Responses from multiple resources follow

```
["a9999",true,"Outlet 4",false]
```

- Add a user, and also return the representation for the user object (when showing non-administrative user passwords is enabled):

```
curl -H "Accept: application/json" -H 'Prefer: return=representation' -H 'X-CSRF: x' \
--data
       'name=fred&password=foobar&is_allowed=true&outlet_access=false,false,false,false,false,false,false,false' --digest \
'http://admin:1234@192.168.0.100/restapi/auth/users/'
```

Result:

```
201 Created
```

```
{
 "name":"fred",
 "password":"foobar",
 "is_allowed":true,
 "is_admin":false,
 "outlet_access":[false,false,false,false,false,false,false,false]
}
```

- Get unit firmware version and serial number in a single JSON array:

```
curl -H "Accept: application/json" --digest \
'http://admin:1234@192.168.0.100/restapi/config/=version,serial/'
```

Result:

```
207 Responses from multiple resources follow
```

```
["(...Version...)","(...Serial...)"]
```

- Get names and values for all bus B meters in a single JSON array:

```
curl -H "Accept: application/json" --digest \
'http://admin:1234@192.168.0.100/restapi/meter/values/all;bus=1/=name,value/'
```

Result:

```
207 Responses from multiple resources follow
```

```
[
 "current",0,
 "voltage",167,
 "total energy",645501.064831
]
```

- Change the administrator password, and return the administrator configuration (the administrator password is never shown):

```
curl -H "Accept: application/json" -H 'Prefer: return=representation' -H 'X-CSRF: x' \
-X PATCH --data 'old_password=1234&new_password=4321' --digest \
'http://admin:1234@192.168.0.100/restapi/auth/users/is_admin=true/'
```

Result:

```
200 OK
```

```
{
 "name":"admin",
 "password":{"$ref":"password/","title":"Password"},
 "is_allowed":true,
 "is_admin":true,
 "outlet_access":[true,true,true,true,true,true,true,true]
}
```

1.4 Data model

The data model is based on the JSON data format, and borrows most type definitions from there. We will start with the types present in JSON.

1.4.1 Scalar types

Scalar data types (having no internal structure and corresponding to simple properties) are:

- a string literal;
- a number literal (following JSON rules, it must be a finite number, so no infinities or NaNs).

1.4.2 Singleton types

Singleton data types are a special type of scalar types. They are also known as unit types, and they have only a single value:

- the constant value true;
- the constant value false;
- the constant value null.

You may wonder why there's no separate boolean type. Surprisingly, the JSON specification doesn't mention it. true and false are just separate special values, like null. See below for how the boolean type is handled.

1.4.3 Container types

JSON defines two container types:

- arrays are ordered lists with integer keys like:

```
[1,"a",true]
```

- objects are unordered maps with string keys like:

```
{"name":"fred","is_admin":true}
```

In our data model, a container can either have an arbitrary number of keys, all with the same type (called the element type, and we will call such containers homogeneous), or a set of pre-specified keys (which we will call fields), each with its own type (which we will call heterogeneous). So, we distinguish the following four types, which are based on the JSON ones but additionally constrain their values:

container	object-based	array-based
heterogeneous	object	tuple
homogeneous	map	array

The most important difference is that you can only add or remove elements from homogeneous containers (maps and arrays).

Additionally, all JSON object-based values have an additional implicit constraint on the keys: they cannot be empty. For example,

```
{"":"I am empty!"}
```

is a valid JSON object, but is guaranteed to neither be accepted nor generated by the API. Otherwise such fields would have syntactically invalid URIs (with a double slash). The empty string "" can be given a special meaning when it appears in the place of an object key name.

There are also two more types of resources, which don't correspond to JSON types.

1.4.4 The sum type

The sum type is something that cannot be attributed to a JSON value, it's a type of the resource. A resource of a sum type can have any value of a specified set of types (if you are familiar with a 'restricted variant' type, or a 'union' type, this is similar). For example, a boolean is the sum type of constant true and constant false. An 'optional' string (string that can be said to have no value at all) can be encoded as the sum type of the string type and constant null (or constant false in some cases). With additional constraints (see below), the sum type can be used to model enumerations from a fixed list of values.

The type is not a 'tagged variant', that is, it doesn't store the information about which variant is stored, it only stores the data.

The sum type has the following additional restrictions:

- A sum type cannot be a variant of another sum type (that would be redundant and add unneeded complexity).
- The variant corresponding to a contained value must be uniquely determined by the value itself (no tagging);
- A sum type can only have at most one container variant; this is partly a consequence of the previous restriction, otherwise an empty map and an empty object would be indistinguishable.

1.4.5 The call type

The call type is a type of a function call. It has no value that can be represented in JSON, but has argument and result types. The argument type is usually a tuple containing the function's arguments. Resources with this type are accessed using POST, where the request content specifies the arguments, and return a value of the result type.

The call type may be seen as a violation of the REST principles, but in practice some actions (like cycling an outlet) are more naturally modeled with a function call than with a resource.

The call type cannot by a variant of a sum type; resources with this type are usually contained in object-type resources, which models traditional object method calls.

1.4.6 Constraints

Any non-call data type can have additional constraints on it. Input data which don't satisfy constraints are rejected.

For example, a number can be restricted to values greater than zero, or a string can be restricted to be equal to "".

Another common constraint is that one value (usually a scalar value) is an index into another value (usually of a container type). This can be seen as an alternative for a sum type for modeling enumerations, but in this case, the enumeration is 'open': the reference side doesn't enumerate what values exactly are; the referee does. This can also be seen as a 'foreign key' relation by people with a relational background.

For example, in miscellaneous configuration there is a tunable string value that represents the preferred image format for plots and meters in the web UI. It is located at

```
/restapi/config/image_format/
```

and has the constraint that is an index into

```
/restapi/renderer/known_image_formats/
```

If you go there, you will see a map

```
gif  => GIF
jpeg => JPEG
png  => PNG
svg  => SVG
```

that represents the image formats supported by the renderer. If you attempt to PUT any other value into /restapi/config/image_format/ , you will receive a 409 Conflict response.

1.5 Supported standard interfaces

1.5.1 URIs

In the REST architectural style, URIs are used to indicate what object (resource) you want to access.

All API URIs must end with a slash ('/'). This is not some hard limitation, but other URIs are reserved for future extenstion. At the moment, if you attempt a request to a non-slash-terminated URI, you will receive an appropriate 30x redirect to the correct (slash-terminated) URI. Correctly configured clients (e.g. browsers) will follow the redirect. You shouldn't rely on this behavior though.

URI query parameters (like ?a=b&c=d ...) are not currently used, and reserved for future extension. It should be noted that some browsers that still survive despite all reason (IE 6) have problems with query parameters and digest authentication (see below.

Instead of query parameters, URI matrix parameters, a less-widespread feature, is used. They enable parallel operations on similar resources, and are discussed below. For now, it suffices to say that you should make sure that if you send characters ';', '=' or ',' in a URI path segment, you should percent-encode them unless you know what you are doing.

1.5.2 HTTP verbs

All verbs described in HTTP verbs are supported, with some Deviations from standards and common practices deviations from standard semantics.

1.5.3 HTTP authentication

In the REST architectural style, standard authentication methods are preferred. This includes Basic and Digest authentication.

Because Basic auth sends the username and password in plain text, it is disabled in newer DLI controllers unless "Allow legacy plaintext login methods" (/restapi/config/allow_plaintext_logins/) is set.

Digest auth is a challenge-response mechanism which is similar to the legacy authentication method in which the client received a challenge from the server and hashed it combined with the username and password to prove their identity, but has additional security features, as well as the benefits of being standard and widely implemented. Digest auth supports different "quality of protection" (qop) levels. qop="auth" and qop="auth-int" are supported. See the related RFCS [8], [7] for details.

If you are really security-conscious, you should disable HTTP completely in the controller's web UI, and use HTTPS only, as this will provide a higher degree of security and make Basic and Digest authentication methods more or less equivalent.

1.5.4 Input MIME types

Data in the following formats can be specified as a representation of a resource (for PUT or POST) using the Content-Type header:

1.5.4.1 application/json

JSON is the main interchange format. It is accepted by all resources in all circumstances. Other content types can be thought of as if they were reduced to it.

JSON data can be any valid JSON literals, not only objects; see RFC [3] for details. All REST API requests must supply valid credentials, and implicit CORS credentials passing is prohibited; this rules out different "JSON hijacking" attacks, should they ever become possible again.

1.5.4.2 text/plain

The plain text format is limited in what resources it can represent. It supports all scalar types (constant null is encoded as an empty string), and also arrays of scalar types (values are separated by commas ','').

If the resource is of a sum type, and there is an ambiguity (e.g. "false" could mean the constant false, or a string literal "false"), the request is ill-formed and denied.

If you face ambiguities, or need to express values of other types, you need to use JSON.

1.5.4.3 application/x-www-form-urlencoded

The URL-encoded format is limited in what resources it can represent. Scalar values are represented by a 'value' key with a value corresponding to the textual representation, if they have one. For example,

```
value=1
```

```
value=false
```

```
value=Hello
```

Field-based types can be represented in the format

```
field1=value1&field2=value2&...
```

The values are again in textual representation.

1.5.5 Patch MIME types

Data in the following formats can be specified as a representation of a change to resource (for PATCH) using the Content-Type header:

1.5.5.1 application/json-patch+json

JSON patch is the main patch format. It is accepted by all resources in all circumstances. Other patch formats can be thought of as if they were reduced to it. See the RFC [4] for details and examples.

Note that paths in JSON patch documents are JSON pointers, as per the JSON patch RFC [5], and not URIs. They are normally either empty, or start with a slash (/), and never end with a slash. Their components are never percent-encoded (there are specific '~' escapes in JSON pointers [5]), and no matrix URI support is in place. Be sure not to confuse JSON pointers and relative URIs.

1.5.5.2 application/x-www-form-urlencoded

The URL-encoded format is limited in what resources it can represent. It is similar to the full resource representation, but must include only the values you want to change. The field names (or the 'value' keyword) may be preceded by 'new_' to emphasize that this is the new value you want to write. If you want to supply the previous value, and have it checked, you should prefix it with 'old_'. Note that in some cases, like changing the administrator username password, you may be forced to supply the old password value to be checked.

When such patches are handled, first all 'old_' values are checked against the resource values, and the patch is only applied if all of them match.

For example, the following patch:

```
old_name=admin&new_name=root&old_password=1234&new_password=4321
```

behaves as though you supplied the following JSON patch document instead:

```
[
 {"op":"test","path":"/name","value":"admin"},
 {"op":"test","path":"/password","value":"1234"},
 {"op":"replace","path":"/name","value":"root"},
 {"op":"replace","path":"/password","value":"4321"}
]
```

Requests to patch any objects with fields starting with old_ or new_ prefixes (currently no objects have such fields) are ill-formed. If you need to work with such objects, you need to supply the patch in the JSON patch format.

1.5.6 Output MIME types

Data in the following formats can be requested as a representation of a resource (usually for GET, but also other methods except DELETE) using the Accept header:

1.5.6.1 application/json

JSON is the main interchange format. It is retrievable for all resources in all circumstances. Objects which are not readable, or not reachable due to depth limitations, will be replaced to JSON references to them.

1.5.6.2 text/html

HTML is the browser output format, which enables you to interact with the API using the browser. Requests generated with the browser can serve as templates for your own requests, but you are by no means limited by them.

1.5.6.3 text/plain

Plain text is the simple human-readable format, not intended to be processed automatically. You should not rely on any particular format for it.

1.5.7 Preference indication

Some methods (e.g. PUT, PATCH and POST to create resources) can return a representation of the result, but it's optional. The HTTP Prefer header [10] can be used to indicate your preference. Requests with this header

```
Prefer: return=representation
```

will return the representation of the created or modified resource, while requests with the header

```
Prefer: return=minimal
```

will return only the status code and headers.

The server will respond with an appropriate Preference-Applied header if the preference has been understood and affected the output.

1.6 Extensions to the standard interfaces

1.6.1 Cross-site request forgery protection

The legacy API allowed to create "convenient" links for actions like "/outlet?a=OFF" or "/script.cgi?run100".

Such "convenience", however, comes at a cost: if, while being logged in to the controller, the user visits a malicious page with something like

```
<img src="http://192.168.0.100/outlet?a=OFF"/>
```

or

```
<script src="http://192.168.0.100/outlet?a=OFF"></script>
```

or similar, the corresponding action will be taken. More malicious page examples would include e.g. erasing some non-administrative users, etc. You could receive an HTML email containing a reference to such an "image", and it could work.

This is an extreme case of a vulnerability named CSRF (cross-site request forgery) [9]. It is not affected by authentication or encryption used. Using POST to send such requests would only be a bit more secure, as those can be imitated by an HTML form, and the user could be tricked into submitting it.

The legacy API will stay in place for the time being, but the new APi is designed to make it more resistant to such kind of attacks.

In modern browsers, XMLHttpRequest can be used to perform arbitrary requests, but if the sender's origin doesn't match the request target, a preflight request is sent, which is currently denied by the controller, so it can't by itself be used to forge a request. Some old Flash plugins didn't play by the rules and allowed arbitrary requests, but they should be fixed by now. So, the remaining sources of potentially-problematic requests are HTML forms, which are currently limited to GET and POST methods with Content-Type of application/x-www-form-urlencoded (among those supported by the REST API as input content types). However, any state-modifying operation by default requires additional request headers which indicate that the source of operation is not an HTML form:

```
X-CSRF: <any value>
```

or

```
X-Requested-With: XMLHttpRequest
```

The first option is supported because it's shorter and self-descriptive, and the second one because it is what JavaScript frameworks usually send.

More sophisticated CSRF protection methods, like passing a token around, are not much more effective and aren't easy to implement considering REST design restrictions.

Additionally, the requirement to add such headers may be lifted if the following options are set:

- Relax non-HTML method CSRF checks (/restapi/config/relax_nonhtml_content_types/)

  This setting allows HTTP clients to perform API requests with e.g. application/json or application/json-rpc without a CSRF protection header (such content currently cannot be sent via an HTML form).

- Relax non-HTML content type CSRF checks (/restapi/config/relax_nonhtml_methods/)

  This setting allows HTTP clients to perform PUT/PATCH/DELETE API requests without a CSRF protection header (such requests currently cannot be sent via an HTML form).

As of the time of this writing, these settings don't reduce security; however, more HTML form methods or content types may be standartized in the future, making the setup less secure.

1.6.2 HTTP verb tunneling

If you have problems sending PUT, PATCH or DELETE requests, or if you are sending non-idempotent PUT or DELETE requests (see PUT and DELETE methods not always idempotent below), you can wrap an arbitrary method request in a POST request using, e.g.

```
POST /restapi/auth/users/2/ HTTP/1.0
X-HTTP-Method: DELETE
```

1.6.3 Matrix URIs and working with multiple resources in parallel

Matrix URIs [1] are a way of encoding URI-segment-specific request parameters in the segment (URI part delimited by '/') itself. In the API, a matrix URI segment defines a selector which performs filtering of matching subresources. Consider the following 'collection' of resources, each of which is the current physical state of the appropriate outlet:

```
/restapi/relay/outlets/0/physical_state/
/restapi/relay/outlets/1/physical_state/
...
/restapi/relay/outlets/7/physical_state/
```

You can manipulate them, or any subset of them, as a whole, by replacing '0', '1', etc. (the outlet index) with a matrix URI selector. Here are some samples:

- Physical states of all locked outlets:

  ```
  /restapi/relay/outlets/all;locked=true/physical_state/
  ```

- Physical states of outlets #1, #3, #5:

  ```
  /restapi/relay/outlets/=0,2,4/physical_state/
  ```

- Physical state of the sole outlet whose name is 'lamp':

  ```
  /restapi/relay/outlets/name=lamp/physical_state/
  ```

- Physical states of all outlets whose name is 'lamp' or 'server':

  ```
  /restapi/relay/outlets/all;name=lamp,server/physical_state/
  ```

- Cycle all unlocked ON outlets (using POST):

  ```
  /restapi/relay/outlets/all;physical_state=true;locked=false/cycle/
  ```

Matrix URI segment parsing is activated by the presence of '=',';' or ','. If you need to include them in a regular path segment (you shouldn't need to in the current data model), you need to percent-encode them.

A matrix URI segment contains one or more parts, separated by an (unescaped) ';'. The first part may be a filter policy, and must be followed by a ';' even if no other parts are present. The supported filter policies are:

- all: the operation will be performed on all matching resources, and the response will be a 207 Multiple results response; even if no resources match, the result will be successful;
- one: the operation will expect exactly one resource to match; if no resources are found, 404 Not found is returned; if multiple resources are found, a 300 Multiple choices response, mentioning the options, is presented;

All other must parts contain key=value entries, the meaning of which depends on the key:

- if key is empty (like '=0,2,4'), the value is a comma-separated list of keys to match; additionally, if no filter policy is set, it defaults to 'all';
- if key is nonempty, it is treated as the name of the field of resources being matched; the value is a comma-separated list of values to match; additionally, if no filter policy is set, it defaults to 'one'.

If multiple key=value parts are specified, only the resources matching all of them are selected. For example,

```
/restapi/relay/outlets/=0,2,4;locked=true/physical_state/
```

will select physical states of locked outlets, but only among outlets #1, #3 and #5.

These seemingly complex rules allow for short URIs for common use cases.

1.6.4 Link relation 'templated'

1.6.5 Response depth limiting

In many cases, you don't want to see all of the output of a request. Range: bytes makes no sense for most API calls. You likely won't get a well-formed HTML or JSON document if you request a byte range. As resources are arranged in an hierarchy, the natural way to do output limiting is by referring to a depth level in this hierarchy. You can use the Range header with a dli-depth range unit to perform limiting. When a level of hierarchy is not descended into, the value is replaced by a JSON reference to it, e.g.:

```
... "relay": {"$ref":"relay\/", "title":"Relay object"}, ...
```

The $ref parameter is a relative URI (not to be confused with a JSON pointer) to the value.

For example,

```
Range: dli-depth=1
```

will give you the immediate child resources of this resource as references, and

```
Range: dli-depth=0
```

will give you

```
{"$ref":"","title"="..."}'
```

if the resource exists.

The default is:

```
Range: dli-depth=infinity
```

which means infinite depth.

The limiting is only supported for non-HTML output formats (JSON, plain text). HTML output depth is context-dependent, but can be thought of as fixed at 1 in most use cases.

1.7 Deviations from standards and common practices

1.7.1 Nonstandard 207 Multiple responses format

The WebDAV RFC [6], where HTTP 207 Multiple responses code is defined, is XML-centric, and requires that the response type be XML. The API generates this code when processing matrix URI requests, and produces output in the format requested by the client in the Accept header instead, because we believe it to be more useful. However, the spirit of the RFC is preserved as much as possible so as not to introduce complexity.

The HTML-formatted response follows the format defined in [6], but instead of custom elements, it creates div elements with corresponding classes.

The JSON-formatted responce contains an array of responses. The HTTP response will contain corresponding "Link: <...>, rel=item" headers which can be used to gain the information on what resource each response is returned from.

1.7.2 PUT and DELETE methods not always idempotent

PUT is normally an idempotent method, but it may interact with matrix URIs in a possibly unexpected manner. For example, consider

```
PUT /restapi/auth/users/name=fred/name/ HTTP/1.0
...
value=joe
```

This request will rename user 'fred' to 'joe'. It is all fine and the request will work as expected. However, it won't technically be idempotent because /restapi/auth/users/name=fred/ will no longer resolve (Fred is now Joe). This would also happen if you used PATCH, but PATCH isn't expected to be idempotent, while PUT is.

Referring to that user by their sequential number would make the request idempotent:

```
PUT /restapi/auth/users/2/name/ HTTP/1.0
...
value=joe
```

Such a request would be idempotent.

Due to the JSON representation of data and expressing manipulations with it using JSON patch, it's not possible for a request like

```
DELETE /restapi/auth/users/2/ HTTP/1.0
```

to be idempotent: deleting a value from a JSON array causes reindexing of successive elements, so user #3 becomes user #2, user #4 becomes user #3, etc. You should not rely on DELETE to be idempotent in this case. We have no concept of 'identity' for parameters stored in JSON arrays.

A better way to delete a user could be by name, e.g.

```
DELETE /restapi/auth/users/name=fred/ HTTP/1.0
```

Such a request would be idempotent (it would still have the reindexing side effect though).

If you really want to change properties of objects by which you selected them in a matrix URI, or delete objects by their sequential number, and want to notify any intermediaries that the request is not actually idempotent, you should wrap them in POST requests using HTTP verb tunneling.

1.7.3 Semantically side-effect-free calls still need POST

All function calls are performed by POST requests to the function's URI. However, some of these functions, like getting a history of a meter's values over a period of time, are semantically nilpotent (have no side effects), so GET could be used. Unfortunately, most of these functions have arguments, which are to be sent in the request body for content type detection to be meaningful, and sending a body in a GET request is not standard. So, POST is used instead.

1.7.4 PATCH is allowed on read-only resources

We allow PATCH to operate on resources which are read-only by nature, or writes to which are denied by policy. The reason for this is that a patch doesn't have to contain modifying operations, just test operations, and may even not contain any operations at all, in which case such resources are valid PATCH targets.

1.7.5 Some PATCH operations

The JSON patch "move" and "replace" operations are specified [4] to be functionally identical to a "remove" operation, followed by an "add" operation. There are obvious problems when attempting to apply such principles to additionally constrained objects with internal behaviour.

Some objects (e.g. the administrative user) can not be subjected to a "remove" operation, and attempting to move them e.g. for simple reordering may fail. Additionally, such operations will erase any internal state of the removed object, so e.g. you can safely move or replace only a disabled AutoPing entry.

Additionally, heterogeneous containers cannot have their fields removed, so such "replace" behaviour would be useless for the fields.

1.7.6 Templated and 'hackable' URIs

Strict adherence to principles of REST would require all possible links to be exposed to the client, but it's simply not feasible to enumerate all use cases for matrix URIs. Technically we could give you so-called opaque URIs for all resources from the entry point, like:

```
<a href="/restapi/55237aa40998e90a093e8190cd8d3bf8/" rel="dli-outlet">Outlet 1 state</a>
```

We could document the link relations, content subtypes types and all that, and become more HATEOAS compliant, but this doesn't seem practical. Rather, we choose to make URI structure represent the data model, and rely on you to apply common sense and patterns discussed in this reference to the information you obtain from the API to generate URIs.

In case some part of the URI scheme changes, we expect to be able to emulate existing resource links, or redirect you from old locations using standard HTTP 30x Redirect responses, so you will at least be notified that the U↩ RI scheme has changed (correctly configured clients would understand the redirect). Resources that were once present but have been discontinued would yield 410 Gone HTTP responses.

References

[1] T. Berners-Lee. Matrix URIs – Ideas about Web architecture, December 1996. 14

[2] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (INTERNET STANDARD), January 2005. Updated by RFCs 6874, 7320. 2

[3] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159 (Proposed Standard), March 2014. 3, 10

[4] P. Bryan and M. Nottingham. JavaScript Object Notation (JSON) Patch. RFC 6902 (Proposed Standard), April 2013. 11, 17

[5] P. Bryan, K. Zyp, and M. Nottingham. JavaScript Object Notation (JSON) Pointer. RFC 6901 (Proposed Standard), April 2013. 11

[6] L. Dusseault. HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV). RFC 4918 (Proposed Standard), June 2007. Updated by RFC 5689. 15

[7] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Authentication. RFC 7235 (Proposed Standard), June 2014. 10

[8] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication. RFC 2617 (Draft Standard), June 1999. Updated by RFC 7235. 10

[9] OWASP. Cross-site request forgery (CSRF), December 2014. 13

[10] J. Snell. Prefer Header for HTTP. RFC 7240 (Proposed Standard), June 2014. 12

[11] Wikipedia. Representational state transfer – Wikipedia, the free encyclopedia, December 2014. 2