RESTful Web script design

Saulius Gražulis, Algirdas Grybauskas, Andrius Merkys, Antanas Vaitkus

$Date: 2020-03-10 14:56:40 +0200 (Tue, 10 Mar 2020) $

$Revision: 6304 $

Version history

Date	Author	Version	Changes
2020-03-10	A. Merkys	v0.1.9	trunk/ Updating the section on backend database limitations as OPTiMaDe filter syntax is adapted and 'action' field is removed.
2020-01-27	S. Gražulis	v0.1.8	trunk/ Updating 'DELETE' specification.
2019-12-23	A. Merkys	v0.1.7	trunk/ Minor rewordings and fixes.
2019-12-21	S. Gražulis	v0.1.6	trunk/ Describing intended REST behaviour in GET, PUT, POST and PATCH requests.

Date	Author	Version	Changes
2019-09-20	A. Merkys	v0.1.5	trunk/ Describing newly implemented CRUD functionality. Describing input and output data formats.
2019-08-13	A. Merkys	v0.1.4	trunk/ Adding a paragraph on name limitations.
2019-07-12	A. Merkys	v0.1.3	trunk/ Converting to ReStructuredText, fixing typos.
2018-05-16	S. Gražulis, A. Grybauskas	v0.1.2	trunk/ (saulius@varanas.saulius-grazulis.lt) Describing a method to ensure idempotence of PUT and POST POST requests. trunk/ (saulius@varanas.saulius-grazulis.lt) Describing the nested revisions to keep track of the data provenance when data are transferred from another databases with the same or similar schema.

Date	Author	Version	Changes
2018-09-13	S. Gražulis, A. Merkys, T. Vaitkus	v0.1.1	trunk/ (saulius@koala.ibt.lt) Explaining behaviour when empty JSON is uploaded. trunk/ (saulius@koala.ibt.lt) Further designing the REST interface; updating the project documentation. trunk/ (saulius@varanas.saulius-grazulis.lt) Describing combination of CGI scripts and Web sites from two repositories.
2018-05-05	S. Gražulis	v0.1.0	Initial revision

Meaning of terms

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119.

Independence requirement

The main goal is to achieve independence, simplicity and easy substitutability of Web scripts that serve representations of various Web resources. For example, a resource http://example.net/db/record/12345 might be served by a CGI script db-record.pl; this script is solely responsible for the logic that represents records of this particular database, in any supported representation ((X)HTML, XML, JSON, CIF, etc.). To maintain the SPOT principle, this script MAY call appropriate libraries that implement the necessary functionality for generating the representation (for example, for generating a valid XHTML page given data

from a database record), and these libraries MAY be reused in different scripts serving different resources, but the relevant decisions on how the resource is represented MUST be implemented in the top-level script, in this case in the "db-record.pl".

CLI analogy

It seems that each of the CGI script could be though as a CLI Unix command; 'cmd –option=opt-value resource' might be thought as functionally equivalent to 'https://example.net/cmd/resource?option=opt-value'.

SPOT preservation

A script SHOULD call relevant subroutines in a tool libraries for performing common task and accessing common configurations. This fact, however, MUST be hidden under the script interface.

Web site vs. Web engine

It seems advisable to separate the CGI scripts and their libraries, together with configuration examples, into a separate repository; this repository may or may not contain a simple web site for testing the scripts. The web sites that provides useful services using these scripts will be hosted in other repositories, and will use the CGI script repository either as a public external resource or as a vendor branch. The web sites will probably run of a specific revision of the script repository, of a ^/trunk/ if the development is rapid, but for a production environment they should run from a specific release tag (released version).

Resource-script mapping

A set of HTTP redirections will be set up to maintain mapping between resources and scripts.

Form-table duality

It seems advantageous to generate both the input form (for searches or selections of the resources) and the (table) representation of the resource in the same script, to avoid spread of information that must be kept identical over several scripts.

Object-relation mapping

The underlying resource might be handled using an object in the implementation language, or as a set of functional calls. It is highly advisable that objects are used as "smart values", and not as functional frameworks; if this is the case then the two approaches are essentially equivalent.

Styles of representations

The generated representations should be as much as possible style-agnostic. For (X)HTML it means that we most probably will want to use CSS for defining the visual style.

In any case, we must strive as much as possible to keep content and its visual representation separate. This cam be achieved in two ways:

1. When using standard (X)HTML or XML and CSS, all visual styles should go to CSS; this means that HTML elements MUST be properly marked up: they SHOULD have unique IDs, and MUST have appropriate class assignments.
2. If CSS are not an option (either not supported by the target representation at all, like ODT, PDF or LaTeX, or not supported properly by the target browsers), it is permissible to generate representation with visual commands embedded (like HTML with element styles included, or LaTeX with visual formatting included), but this must be done explicitly by one function that gets data as one parameter, style list as another parameter, and generates the representation according to the style given:

   ```
   $visualHTML = generateVisualHTML( $dataForHtml, $htmlStyle );
   ```

   In this way, it should be easy to replace representation styles, and to keep the style consistent.

RESTFUL representation output

In hierarchical resource representations (such as JSON or XML), foreign keys of nested subordinate tables pointing to the main record MUST be omitted.

CRUD functionality and HTTP requests

HTTP request overview

HTTP method	Table	Record
GET	Retrieve data of all records in the table, if needed, paginated	Retrieve data of a single record
PUT	Insert record(s) into the table; fail on duplicates[1]	Insert a record with a specified ID value, fail on duplicate[2]
PATCH	Update record(s) of the table; fail if record(s) to update do(es) not exist	Update a specified record; fail if the record does not exist
DELETE	Fails; dropping tables/deleting their contents in bulk is allowed after addtional confirmation.	Deletes a specified record
POST	Same as PUT, although SHOULD upsert	Same as PATCH, although SHOULD upsert

The support of HTTP POST requests MUST be retained, as HTML forms are unable to send HTTP requests other than GET and POST. In the current implementation the action to be taken with HTTP POST is provided via fields 'Save', 'Update', 'Delete' of multipart/form-data content.

HTTP request behaviour

General requirements for responses in JSON format

The following keys MUST be returned in the "metadata" object:

- "data_returned" – contains the number of items actually returned in the query;
- "data_available" – contains the total number of items in the database that matched the query filter; this is the number of items that the client can get in principle; this number would be returned if the page size parameter ("rows") was infinitely large.

[1] Fails for JSON API requests, as HTTP PUT is not handled by JSON API v1.0
[2] Fails for JSON API requests, as HTTP PUT is not handled by JSON API v1.0

GET from a record

MUST return a single record (row) of data from a database. SHOULD return a deep (depth=-1) set of records unless the depth is explicitly requested.

The following QS parameters MUST be honoured (N is an integer):

- *depth=N* – only related tables up to level N are returned; N=0 indicates that only the requested table's data, and no related table data MUST be returned. The depth=-1 parameter indicates that infinite depth must be used, i.e. all related table data MUST be returned. However, the related data items MAY be limited by the 'rows' number;
- *rows=N* – limit the number of the rows in the related tables to N.

NOTE: currently, "metadata" JSON object is returned for the main table query, but not for the related tables. It seems very useful if the "metadata" would be returned also for related tables, in particular its "data_returned" and "data_available" items; in this way the client would know that the data from the related tables are incomplete and it needs to make additional queries to get all data if needed. For this, the "representation" URI in related table metadata would be very useful.

GET from a table

MUST return a page with multiple records, up to the 'limit' number of records. See the 'General requirements for responses in JSON format'_ for the description of the returned metadata.

The following QS parameters MUST be honoured (N is an integer):

- *rows=N* – limit the number of the rows to N;
- *offset=N* – start output with the row number N (zero-based).

The same 'rows' value SHOULD be used for the related tables.

CAVEAT: using the same 'rows' value limits the functionality; we may wish to have different 'rows' parameters for the main table and for the related tables. We need to discuss use cases for this more thoroughly.

PUT to a record

The PUT request is intended for the insertion of the *new* record. On the SQL level, this corresponds to the 'INSERT' SQL statement. The record MAY NOT be inserted into a view, only into a genuine table; if an insertion into a view is attempted, the request SHOULD return an error status, 'HTTP 400 Bad Request'. The response payload in JSON format MUST conform to the response

schema and MUST contain the 'error_code' and 'error_message' fields that indicate the nature of the error ("duplicate unique key '<ID>' ").

When PUT request is performed using a *record* URI, the posted data MUST contain exactly one record; any other number of records in the posted data is an error; in such case the server database MUST NOT be changed, and the 'HTTP 400 Bad Request' status code MUST be returned, with the error message and code identifying the problem. The error code MUST be distinct from the error code returned on duplicate IDs.

RATIONALE: using PUT request indicates that we need to create a new record that did not exist before. An existing record with the same ID is most probably an error, and should be diagnosed appropriately.

NOTE[1]: This specification **obsoletes** the idempotence requirements of the PUT request in the 'Ensuring idempotence of the write operations'_ section.

NOTE[2]: This behaviour appears to break the strict idempotence of the RESTful requests. However, the very nature of the PUT request makes it necessary to diagnose the duplication of the record. If duplication of the record is not an error, the client has a choice:

1. Process the 'HTTP 400 Bad Request' status code with the error code "duplicate unique key" as a regular situation and proceed; the client MAY assume in this case that the record is present, but it MUST NOT assume that the data in this record are the one that it attempted to insert. Most probably, another request to fetch the data is necessary. The URI to fetch this data MUST be provided in the response payload;
2. Use POST request if the SQL 'UPSERT' functionality is intended; the POST request is idempotent in the sense described in the 'POST to a record'_ section.

If the posted data contain related table records, these records MUST be inserted in a single transaction; if any of the IDs in the inserted records exists in the database, if MUST be treated as an error, in the same way in which the duplicate key of the inserted primary record is treated.

TODO: consider whether the response payload, the "data" part, SHOULD/MAY/MUST contain data from the existing record. One possibility is given below:

The "data" part of the response payload MAY be missing; if it is present, it MUST contain data from the actual record that is present in the database, as if it were returned by a GET query to the same record URI.

RATIONALE: if the insertion fails on duplicate key, the client most probably will need to know what the actual data is before it can proceed. Returning "data" with the actual record will save the client one HTTP request. However, it increases traffic and will most probably require another SQL request, which complicates the server. Thus, the server implementer may decide that this feature is too expensive and not provide it.

Furthermore:

1. After the first insertion, all further requests will be idempotent and return identical error responses;
2. The PUT request, even if it fails, ensures the invariant "the record with the requested ID exists in the database";
3. The server might be able to identify that the PUT record is identical to the one already present in the database and return the 'HTTP 200 Success' code with the existing record payload. However, this would require another SQL lookup query; it raises questions how different fields are compared for quality (e.g. floating-point numbers); and it is not clear what to do if the PUT data contain less fields than the database table. Thus, it may be prudent and less error-prone to react to the 'INSERT' failure of the SQL with a HTTP error code and the corresponding error message.

PUT to a table

PUT to a table inserts one or several records into the table. The submitted file MAY contain any number of records, including 0. All records MUST be new and unique; if a duplicate key is detected (either present in the database or among the PUT records), the entire PUT transaction MUST be aborted, the database MAY NOT be changed and the error status code and message MUST be returned following the specifications of the 'POST to a record'_ specification.

POST to a record

The POST request provides the SQL 'UPSERT' functionality; the POST requests are idempotent in the sense that subsequent identical requests sent to the server will produce the state of the resource on the server identical to that after the first request, and produce that same representation of the resource that the first query produced in the response (except maybe for different timestamps).

POST to a table

POST to a table UPSERTs multiple records into the table; the submitted file MAY contain any number of records, including 0.

If the insertion of any individual record fails, the whole request MUST fail, and the error code, RESTful error code and message MUST be returned.

PATCH to a record

The PATCH request provides the SQL 'UPDATE' functionality; the PATCH requests are idempotent in the sense that subsequent identical requests sent to the server will produce the state of the resource on the server identical to that after the first request, and produce that same representation of the resource that the first query produced in the response (except maybe for different timestamps).

PATCH to a table

The PATCH to a table modifies several records (including 0); all records MUST exist.

If the update of any individual record fails for whatever reason, the whole request MUST fail, and the error code, RESTful error code and message MUST be returned.

POST'ing to URI with a record identifier

POST'ing to URI with a record identifier MUST update the record, but NOT the identifiers. Mismatches of provided and existing identifiers MUST be checked and reported as errors.

A special QS parameter, '&override_ids=1', SHOULD be supported, and with this QS option, in case the system supports it, the database identifiers MUST be overwritten by identifiers supplied in the uploaded JSON file. This is currently the only method to change record identifiers through the RESTful interface. We should investigate the possibility to provide extra authorisation for such action.

When a JSON with only UUID identifier of the record is provided, all Public IDs (such as SOLSA ID) MUST be assigned automatically. Likewise, when a new Public ID is used to create a new record, UUIDs MUST be generated automatically.

If no IDs are provided, JSON MUST be posted to the URI with the UUID or PublicID (i.e. SOLSA ID), and this ID from the URI will be assigned to the submitted JSON data.

When inserting POST'ed data, internal primary keys ('id' columns) MUST be ignored.

Ensuring idempotence of the write operations

When the same data are PUT, POSTed or otherwise submitted to the RESTFul interface, the underlying implementation MUST ensure that submitting identical data for the second time does not change the database state. The implementation MUST ensure this when submitted file is bit-wise identical; it MAY apply some canonical transformation (e.g. removing insignificant spaces and newlines, forcing identical quotes, etc.) on the submitted file.

A possible implementation using SQL database is as follows. While comparing (diff'ing) the submitted file contents against the existing database content might be used, it will most probably be too inefficient. Thus, an implementation that uses cryptographic checksums for detecting identical data might be used. The implementation would compute, after receiving the uploaded data file, and possibly after application of some canonical transformations, checksum (SHA1 or analogous) of the incoming file. The checksums of the inserted files, along with their contents and other metadata (filename, upload date, database revision number after data insertion), would be stored in a separate database table. Before insertion of the data, a check is made whether the newly calculated checksum is already present in a data table. If yes, the actual insertion is not performed, thus preventing a spurious new revision of the database to be generated, and the the same answer as after the original insertion (e.g. the revision contents of the insert revision) is returned. Otherwise, transaction is started, a new database revision is recorded and the newly calculated cheksum and file data are inserted into the checksum table, after which all regular insertions (including the revision number) is performed. The the transaction is committed or rolled back as usual. If transactions are not used, the checksum database table MUST contain the 'committed' bit which is set to 0 (false) before the insertions are started and set to '1' (true, meaning that the file is committed) when also insertions are performed successfully.

Return of the RESTful post queries

For databases that use revision tables, a contents of the newly created revision record MAY be returned. If a request is repeated and detected as such, then the contents of the revision record when the data where actually inserted. The response returned on duplicate PUT or GET query MUST be the same as the response of the first query (when the actual insertion or update actually took place), except that the application MAY insert additional audit fields as the current query time or query ID.

Revision chaining upon data transfer

When data are transferred from one database (e.g. an on-field database) to another (e.g. a centralised archive repository), a data provenance should be kept, keeping information of the original insertion revision as well as the current transfer revision, and possibly all the intermediate revisions.

The revision table should thus have a 'previous_revision_id' column. This column MUST be NULL for the newly inserted revisions. For the data transferred from other databases, it will contain a reference to the a previous revision (or a revision chain) that came with the data (e.g. in the JSON or XML file). The old revisions will be inserted as ordinary data into the revision table.

Input and output data formats

A RESTful database is capable of supporting input and output of virtually infinite set of data formats. The principle requirement for support of a data format is an unambiguous mapping to an underlying relational data model. Tabular data formats (spreadsheets and the like; CSV, ODS, XLS, to name a few) are straightforward to map, whereas document-oriented (like XML) or loosely structured (like JSON) are more difficult to support.

A RESTful method to submit input in a specific data format in HTTP PUT/POST/PATCH is by sending its contents to a backed with 'Content-Type' HTTP header set using Internet Media Type conventions. The accepting script MUST check the supplied value of 'Content-Type' HTTP to determine whether the supplied input can be handled. If unknown, such format MUST be reported via either 400 Bad Request or 501 Not Implemented HTTP statuses. The default input format SHOULD be 'multipart/form-data'.

Output format MAY be specified by the client using one of the following methods: 'format' QS parameter or 'Accept' HTTP header. Again, unsupported formats MUST be reported. The default output format MUST be HTML.

Error handling

In no case SHOULD a script crash, cause "internal server error", or return a blank page without any error code.

Errors should cause:

1. An appropriate human- and machine- readable response;
2. A HTTP error code, so that the client software knows how to behave.

In order to prevent attempts to break into the site server by guessing passwords or extracting information via the error messages, the response MAY conceal the exact cause of the error from the client; in that case, however, it MUST mention the the exact error message will be presented in the server logs, and MUST log full error message into the log. For instance:

- HTML response:
  - "We regret but the server can not grant you access to modify the record"db/12345" on this server. Please try to log in (once more). The exact cause of the denial is recorded in the server logs; please contact admin@example.net for assistance."
- Log message:
  - "update.pl: ERROR, permission denied for user 'eve' - password incorrect."

  or:
  - "update.pl: ERROR, permission denied for user 'adam' - no such user."

Configuration

Unfortunately but inevitably Web software requires configuration, authentication and authorisation.

Configuration should proceed automatically as much as possible. If a script does not find a database in a specified location, but can write a file in that location, and the database is a file (a typical situation when the script is started for the first time after a successful installation), then the script SHOULD create the database using default schema. If a script needs a configuration file, but can not find this file, it should create a config file with as much content as possible from the defaults or from a configuration template. If a database access needs to be provided to a database server, a script should inform the user how to set up such access.

Security

One should ensure that configuration files, especially ones containing passwords, are not readable by the outside users. This MUST be actively checked during the operation. Default passwords MUST NOT be used; instead, first time passwords MUST be generated upon installation from reliable random sources and recorded in a place which is only accessible for the system administrator. If GPG or X509 certificates are available, the password MAY be encrypted using the public keys.

Possible secure configuration methods:

1. Upon the first start, the server detects missing configuration files and/or not writable directories, and presents a page with detailed step-by-step instructions on how to set up the server, having access to the server command line;
2. Upon the first start, server asks the user to fill in forms for server configuration, generates the configuration files and makes them read-only. To make sure that only legitimate user can configure the service, such configuration forms MUST be only presented on localhost, or MUST require an action to be performed on a server (e.g. a special file must be created with a random string that was presented to the user; this random string must be valid for a limited time)

Limitations for the backend databases

Due to the interplay of various namespace choices some limitations for the supported backend databases exist:

- Table and column names MUST consist only of alphanumeric symbols and an underscore symbol. Names MUST NOT start with a digit. Uppercase letters are allowed in names only if table and column names are case-insensitive. That is, all tables and columns MUST be queryable using all-lowercase names. These limitations arise from the usage of OPTiMaDe filter syntax in RestfulDB.
- As table columns and composite foreign keys are placed in the same namespace, there MUST NOT be a column and a composite foreign key with the same name in the same table. If a composite foreign key is unnamed, then the name of the parent table is used.