webstore Documentation

Release 0.2

Open Knowledge Foundation

Sep 27, 2017

Contents

webstore is a RESTful data store for tabular and table-like data. It can be used as a dynamic storage for table data, allowing filtered, partial or full retrieval and format conversion.

CHAPTER 1

API

The webstore provides an API that aims to make tabular data available as hypermedia resources following the REST paradigm. This means having a trade-off between the perfect way to represent tables and adhering to a resource-centric view of web media.

Core Resource:

```
/{user-name}/{db-name}/{table-name}
```

Table is the central exposed resource (databases are created on-the-fly and may in fact not actually be separate databases, depending on the backend vendor (e.g. we can use PostgreSQL Schemas to partition the table space and don't actually need to create distinct databases).

On the `table` resource, the following operations are supported.

Viewing and selecting data

Retrieval:

```
GET /{user-name}/{db-name}/{table-name}
```

This will read data. The desired representation should be specified as an `Accept` header (text/csv, application/json or text/html). As a fallback, a file type suffix can also be used:

```
GET /{user-name}/{db-name}/{table-name}.csv
```

The resource can also be filtered:

```
GET /{user-name}/{db-name}/{table-name}?column=value
```

To limit the number of results or to specify an offset, use these query parameters:

```
GET /{user-name}/{db-name}/{table-name}?_limit=10&_offset=20
```

The query can also be sorted, either as 'asc' (ascending order) or 'desc' (descending order):

```
GET /{user-name}/{db-name}/{table-name}?_sort=asc:amount
```

Note. It might be tempting to use '_asc' and '_desc' instead, but order is relevant and not provided for mixed query argument names in Werkzeug.

Another useful feature of the views is result counts. To get counts:

```
GET /{user-name}/{db-name}/{table-name}?_count=1
```

This will give the response an `X-Count` header specifying the number of matching records and for the json it will fill in the count field in the json object.

JSON with Padding / CORS

For JSON, a special query parameter *_callback=function* can be added to specify a JSONP padding. Be aware that webstore also supports Cross-Origin Resource Sharing (CORS) by allowing access to all HTTP methods from any host. This can make accessing resources very easy, as simple XMLHTTPRequest calls can be used. CORS is not supported by all browsers, though, and thus JSONP still has a use case.

JSON tuples (ordered JSON)

Since JSON dictionaries don't have any order associated with them, webstore offers a secondary JSON encoding, called JSONtuples. This format can both be written and read and provoked by setting the *application/json+tuples* content type or by appending the *.jsontuples* suffix to a path.

The generated output will contain a dictionary with two entries: *keys* for an ordered listing of the column names and *data*, which contains a list of tuples, each a row of data ordered in the same order as given by the *keys*.

Sub-resources of tables

Sometimes it is useful to know the number, names and types of columns in the database. To get such schema information, access the 'schema' subresource:

```
GET /{user-name}/{db-name}/{table-name}/schema
```

For reference, one can also address each row of a given table at the following location (with {line-number} set to the auto-increment `__id__` column):

```
GET /{user-name}/{db-name}/{table-name}/row/{line-number}
```

Another useful function is the distinct subcollection: for any column in a table, this will return all values of the column exactly once with a count of its occurences (ie. this is actually a GROUP BY query):

```
GET /{user-name}/{db-name}/{table-name}/distinct/{column-name}
```

For both *row* and *distinct*, query paramters such as sorting apply.

Listing databases

If a user name is known, a listing of all their databases is available at:

```
GET /{user-name}
```

This listing is based on a directory listing for the SQLite backend and prone have errors if the directory structure is modified outside of the application. Further, generating this listing may be non-trivial for cases in which another backend is used.

Writing

To create a new table, simply POST to the database:

```
POST /{user-name}/{db-name}?table={table-name}
```

The request must have an appropriate `Content-type` set. The entire request body is treated as payload. The desired table name is either given as a query parameter (see above) or by posting to a non-existent table:

```
POST /{user-name}/{db-name}/{table-name}
```

If application/json is specified as the content type, webstore will expect a list of single-level hashes:

```
[
  {"column": "value", "other_column": "other value"},
  {"column": "banana", "other_column": "split"}
]
```

To insert additional rows into a table or to update existing rows, issue a PUT request with the same type of payload used for table creation:

```
PUT /{user-name}/{db-name}/{table-name}
```

Without further arguments, this will insert new rows as necessary. If you want to update existing records, name the columns which are sufficient to uniquely identify the row(s) to be updated:

```
PUT /{user-name}/{db-name}/{table-name}?unique=id_colum&unique=date
```

This will attempt to update the database and only create a new row if the update did not affect any existing records.

To delete an entire table, simply issue an HTTP DELETE request:

```
DELETE /{user-name}/{db-name}/{table-name}
```

Please consider carefully before doing so because datakrishna gets angry when people delete data.

Data types

If the submitted data is in format that supports types (e.g. JSON), database columns are created according to the type of the first row in the data (e.g. a float JSON field will generate a FLOAT column in SQL). If the value of a column is null in the first row, a VARCHAR column is created. When new data is added, types are enforced on the incoming data. Values that can not be converted to the designated type will result in an HTTP 400 (Bad Request) error.

Executing raw SQL

Webstore can execute raw SQL statements coming from a request. Such statements have to be submitted in the body of a PUT request to the database with a content type of 'text/sql':

```
PUT /{user-name}/{db-name}
```

If the user has 'delete' authorization, any SQL statement can be executed, including potentially destructive operations such as INSERT, UPDATE and DELETE. Otherwise, only read operations - i.e. the SELECT statement - can be run.

An example of using this could look like this:

```
curl -X PUT -d "SELECT * FROM {table-name}" -i -H "Content-type: text/sql" http://
↪{host}/{user-name}/{db-name}
```

If you need to rely on SQL parameter binding or database attachment (see below), you can submit a JSON envelope instead of sending raw SQL strings. This expects the same type of put request with a content type of *application/json* and a body in the following format:

```
{
 "query": "SELECT * FROM .. WHERE x = ?",
 "params": [5],
 "attach": [{
             "user": "db_user",
             "database": "db_name",
             "alias": "short"},
            ..]
}
```

`params` can be either a list or dictionary of values to be bound in the query, depending on whether you're using positional or named parameters.

Database attachment is a special feature of SQLite that will allow queries across several SQLite files. If you attach a database with a given alias, its tables will be available with the `alias` prefix. If no alias is specified, the database name will be used. If you don't specify a user, the same user as for the main database will be assumed.

Note. All SQL queries are somewhae´t database-specific, so you need to know whether you are speaking to a PostgreSQL or SQLite-backed webstore.

Downloading the whole database (SQLite)

When SQLite is used as a backend to webstore, the whole database file (not a dump!) can be retrieved by calling the database endpoint either with the '.db' suffix or the 'Accept:' header set to 'application/x-sqlite3':

```
curl -o local.db http://{host}/{user-name}/{db-name}.db
```

Command-line usage

Uploading a spreadsheet:

```
curl --data-binary @myfile.csv -u user:password -i -H "Content-type: text/csv" http://
↪{host}/{user-name}/{db-name}?table={table-name}
```

Updating (upsert) based on a set of unique columns:

```
curl -XPUT --data-binary @myfile.csv -u user:password -i -H "Content-type: text/csv"␣
↪http://{host}/{user-name}/{db-name}/{table-name}?unique={col1}&unique={col2}
```

Get a filtered JSON representation:

```
curl -i -H "Accept: application/json" http://localhost:5000/{user-name}/{db-name}/
↪{table-name}?{col}={value}
```

Authentication and Authorization

The webstore itself does not maintain information about registered users, although users are a necessary, first-class element of the system. To still support users, authentication is delegated to another system or performed based on rules. The preferred authentication backend is CKAN, which is used by directly interacting with the platform's database. This means CKAN credentials can be used as long as they include a valid CKAN api key (when authenticating against CKAN replace the "user:password" string with your api key).

Authentication can be used via a basic auth header. In the future, support for API keys and OAuth is planned.

Authorization is based on simple rules and can be configured via the config file (AUTHORIZATION). A few common policies are this:

- Default: all users can read, owner can write
- Restricted: owner can read and write, everyone can do nothing

Possible future: config file can specify a python method / entry point to support pluggable authorization rules (TODO: method signature)

CHAPTER 2

Client Libraries

- Python: http://github.com/okfn/webstore-client
- Pypi: webstore-client
- Documentation at: http://packages.python.org/webstore-client/