DataGator Specification: RESTful API[*]

```
author: LIU Yu <liuyu@opencps.net>
revision: 0.8
```

Background

The RESTful API of DataGator is a JSON-based interface for accessing and operating DataGator's computing infrastructure in a programmatic way. This document specifies web service endpoints and protocols for invoking the RESTful API of DataGator. Targeted readers of this document are developers experienced in web programming, especially, composing HTTP requests with tools such as cURL.

Status of Backend Service

URL Endpoint[1]: `^api/v1/`

GET: get the runtime status of DataGator's backend service.

On success, the response is a Message object[2] with status code 200, e.g.,

```
$ curl -i https://www.data-gator.com/api/v1/
HTTP/1.1 200 OK
Content-Type: application/json

{
    "kind": "datagator#Status",
    "code": 200,
    "version": "v1",
    "service": "datagator.wsgi.api"
}
```

URL Endpoint: `^api/v1/schema`

GET: get the JSON schema being used by DataGator's backend service.

On success, the response is the JSON schema being used by DataGator's backend service for validating data submitted to, or returned from, the RESTful API.

[*]This document is copyrighted by the Frederick S. Pardee Center for International Futures (abbr. *Pardee*) at University of Denver, and distributed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (CC BY-NC-ND 4.0).

[1]Unless otherwise specified, URL endpoints are relative to `http(s)://www.data-gator.com/`.

[2]Responses from DataGator's RESTful API are HTTP messages with JSON objects as payload, which all conform to the JSON schema defined at http://www.data-gator.com/api/v1/schema.

Repository Operations

URL Endpoint: `^api/v1/<repo>`

`GET:` list all `DataSet`'s within the `Repo` identified by `<repo>`.

On success, the response is a `Repo` object, e.g.,

```
$ curl -i https://www.data-gator.com/api/v1/Pardee
HTTP/1.1 200 OK
Content-Type: application/json

{
    "kind": "datagator#Repo",
    "name": "Pardee",
    "items": [
    {
        "kind": "datagator#DataSet",
        "name": "Embassies",
        "repo": {
            "kind": "datagator#Repo",
            "name": "Pardee"
        },
        "id": "7f76285d-b31c-40ee-8493-b5692474579c",
        "rev": 5,
        "itemsCount": 5
    },
    ...
    }, {
        "kind": "datagator#DataSet",
        "name": "Treaties",
        "repo": {
            "kind": "datagator#Repo",
            "name": "Pardee"
        },
        "id": "8282a5c2-f02c-4364-8aeb-d535dd0d0502",
        "rev": 11,
        "itemsCount": 11
    }],
    "itemsCount": 4
}
```

On failure, the response is a `Message` object with error code and description, e.g.,

```
$ curl -i https://www.data-gator.com/api/v1/\
> NonExistence
HTTP/1.1 404 NOT FOUND
Content-Type: application/json

{
    "kind": "datagator#Error",
    "code": 404,
    "message": "Invalid repository 'NonExistence'",
    "service": "datagator.wsgi.api"
}
```

PUT: create a new DataSet within the Repo identified by <repo>. Note that PUT is a *committal* operation requiring authentication, e.g.,

```
    $ curl -i -X PUT -u "Pardee:<concealed>" -d @payload \
    > https://www.data-gator.com/api/v1/Pardee
```

where, the payload SHOULD contain a valid DataSet object, e.g.,

```
    {
        "kind": "datagator#DataSet",
        "name": "IGO_Members",
        "repo": {
            "kind": "datagator#Repo",
            "name": "Pardee"
        }
    }
```

On success, the response is a Message object with status code 202, where, the Location entry in the HTTP message header is the URL of a corresponding Task object and can be used to monitor the creation status of the DataSet. Note that DataSet creation is an *asynchronous* operation, namely, the DataSet MAY NOT be available until the Task completes. Refer to *Task Operations* for more details.

```
    HTTP/1.1 202 ACCEPTED
    Location: https://www.data-gator.com/api/v1/task/
        a7302504-5842-4a4d-b0bc-3f8a05863b76
    Content-Type: application/json

    {
        "kind": "datagator#Status",
        "code": 202,
        "service": "datagator.wsgi.api",
        "message": "Accepted and pending creation."
    }
```

On failure, the response is also a Message object, but may bear a diversity of error codes. For instance, if the operation was initiated by someone (i.e. David) not having *committal* privileges to <repo>, then the error code will be 403, i.e.,

```
    $ curl -i -X PUT -u "David:<concealed>" -d @payload \
    > https://www.data-gator.com/api/v1/Pardee
    HTTP/1.1 403 FORBIDDEN
    Content-Type: application/json

    {
        "kind": "datagator#Error",
        "code": 403,
        "message": "Permission mismatch.",
        "service": "datagator.wsgi.api"
    }
```

For another instance, if the `DataSet` object submitted within the `payload` specifies a `<repo>` (i.e. `Pardee`) other than the one indicated by the URL (i.e. `David`), then `DataGator` will respond with error code 400, e.g.,

```
    $ curl -i -X PUT -u "David:<concealed>" -d @payload \
    > https://www.data-gator.com/api/v1/David
    HTTP/1.1 400 BAD REQUEST
    Content-Type: application/json

    {
        "kind": "datagator#Error",
        "code": 400,
        "message": "Invalid dataset repository 'Pardee'.",
        "service": "datagator.wsgi.api"
    }
```

Remarks: `DataGator` accepts `PUT` to a `Repo` that already contains the submitted `DataSet` object. This ensures the *idempotence* of the `PUT` method as specified by RFC 2616, namely, the side effects of $N > 0$ identical `PUT` requests SHOULD be the same as for a single `PUT` request. In the future, the schema of `DataSet` MAY be extended to contain additional properties such as keywords, copyright notices, etc. The semantics of the `PUT` method will, then, naturally extend to updating the `DataSet`.

Data Set Operations

URL Endpoint:

```
    ^api/v1/<repo>/<dataset>
    ^api/v1/<repo>/<dataset>.<rev>
```

`GET`: get the `HEAD` revision of the `DataSet` identified by `<repo>/<dataset>`, or, get the *historical* revision identified by `<repo>/<dataset>.<rev>`.

On success, the response is the requested `DataSet` object, e.g.,

```
    $ curl -i https://www.data-gator.com/api/v1/\
    > Pardee/IGO_Members
    HTTP/1.1 200 OK
    Content-Type: application/json
    Etag: c6956d5c718f55259efd001891a5795b
    Last-Modified: Sun, 18 Jan 2015 09:19:52 GMT

    {
        "kind": "datagator#DataSet",
        "name": "IGO_Members",
        "repo": {
            "kind": "datagator#Repo",
            "name": "Pardee"
        },
        "id": "46833464-4b62-4dd1-96f6-bd0d1adf2696",
        "rev": 5,
        "created": "2015-01-12T15:11:52Z",
        "items": [
        {
            "kind": "datagator#Matrix",
            "name": "IMF"
        },
        ...
        {
            "kind": "datagator#Matrix",
            "name": "WTO"
        }],
        "itemsCount": 5
    }
```

On failure, the response is a `Message` object with error code and description. For instance, if the requested revision does *not* exist for the specified `DataSet`, then `DataGator` will respond with error code 404, e.g.,

```
    $ curl -i https://www.data-gator.com/api/v1/\
    > Pardee/IGO_Members.100
    HTTP/1.1 404 NOT FOUND
    Content-Type: application/json

    {
        "kind": "datagator#Error",
        "code": 404,
        "message": "No such revision '100'",
        "service": "datagator.wsgi.api"
    }
```

PUT: commit a new revision to the `DataSet` identified by `<repo>/<dataset>`. Note that `PUT` is a *committal* operation requiring authentication, e.g.,

```
    $ curl -i -X PUT -u "Pardee:<concealed>" -d @payload \
    > https://www.data-gator.com/api/v1/Pardee/IGO_Members
```

where, the `payload` SHOULD contain a dictionary of `<key>`, `<value>` pairs, each specifying one of the three operations as follows,

- **create:** If (i) `<value>` is a valid `DataItem` object, and (ii) the current `HEAD` revision of the `DataSet` does *not* contain a `DataItem` named `<key>`, then, the pending revision of the `DataSet` will incorporate a new `DataItem` with `<key>` as identifier and `<value>` as content.

- **update:** If (i) `<value>` is a valid `DataItem` object, and (ii) the current `HEAD` revision of the `DataSet` already contains a `DataItem` named `<key>`, then, the content of the `DataItem` named `<key>` will be replaced with `<value>` in the pending revision.

- **remove:** If (i) `<value>` is equal to `null`, and (ii) the current `HEAD` revision of the `DataSet` contains a `DataItem` named `<key>`, then, the `DataItem` named `<key>` will be eliminated in the pending revision. If, otherwise, the `HEAD` does *not* contain a `DataItem` named `<key>`, the operation itself will be ignored, thus not affecting the pending revision.

A `PUT` request MAY involve one or more of the above-mentioned operations. For instance, the following `payload` will (i) **create / update** a `Matrix` named `NATO`, and (ii) **remove** the `DataItem` named `WTO`, in the targeted `DataSet` (i.e. `Pardee/IGO_Members`). All other `DataItem`'s in the current `HEAD` revision of the targeted `DataSet` will be preserved *as-is* in the pending revision.

```
{
    "NATO": {
        "kind": "datagator#Matrix",
        "columnHeaders": 1,
        "rowHeaders": 1,
        "rows": [
            ["Country", 1816, 1817, ..., 2013],
            ["Abkhazia", null, null, ..., 0],
            ...
            ["Zimbabwe", null, null, ..., 0]
        ],
        "rowsCount": 337,
        "columnsCount": 199
    },
    "WTO": null
}
```

On success, the response is a `Message` object with status code 202, where, the `Location` entry in the HTTP message header is the URL of a corresponding `Task` object and can be used to monitor the status of revision. Note that `DataSet` revision is an *asynchronous* operation, namely, the new `HEAD` revision MAY NOT be available until the `Task` completes. Refer to *Task Operations* for more details.

```
HTTP/1.1 202 ACCEPTED
Location: https://www.data-gator.com/api/v1/task/
    c6266af4-d4fa-4764-8481-b189c1dfe999
Content-Type: application/json

{
    "kind": "datagator#Status",
    "code": 202,
    "service": "datagator.wsgi.api",
    "message": "Accepted and pending commit."
}
```

On failure, the response is also a `Message` object, but may bear a diversity of error codes. For instance, if the `PUT` request targets a *historical* revision of the `DataSet`, instead of the `HEAD` revision, then `DataGator` will respond with error code 400, e.g.,

```
$ curl -i -X PUT -u "Pardee:<concealed>" -d @payload \
> https://www.data-gator.com/api/v1/Pardee/IGO_Members.1
HTTP/1.1 400 BAD REQUEST
Content-Type: application/json

{
    "kind": "datagator#Error",
    "code": 400,
    "message":
        "Cannot commit to history revision '2'.",
    "service": "datagator.wsgi.api"
}
```

Remarks: All operations from the same `PUT` request will be committed in a single transaction by the *asynchronous* `Task`. Namely, if any of the operations fails, then the pending revision will be revoked entirely, and the `HEAD` revision of the targeted `DataSet` will remain intact. In case a `PUT` request contains conflicting operations on the same `DataItem` -- e.g., both **update** and **remove**, or multiple **update**'s with distinct `<value>`'s -- the `Task` MAY still succeed, but the outcome of revision is *undefined*. In addition, if the `PUT` request is *trivial* -- i.e., the enclosed operations not yielding effective changes to the `HEAD` revision of the targeted `DataSet`, such as (i) **update** operations with `<value>`'s identical to those found in the `HEAD` revision, (ii) **delete** operations whose `<key>`'s are not present in the `HEAD` revision, or (iii) payload being an empty dictionary -- then the pending revision will *not* be committed.

The semantics of `DataSet` revision fits the definition of the `PATCH` method from RFC 5789, i.e., to submit a set of changes (i.e. **create / update / delete**) to be applied to the targeted resource (i.e. `DataSet`). `DataGator` chooses `PUT` over `PATCH` for two reasons, (i) standardized in 2010, the `PATCH` method is not as widely supported as other HTTP methods from RFC 2616, especially on some older browsers and AJAX libraries, and (ii) the implementation of `DataSet` revision is guaranteed to be *idempotent*. Namely, instead of raising an error, `DataGator` silently ignores the attempt to delete a non-existent `DataItem`, thus ensuring the side effects of $N > 0$ identical `PUT` requests is the same as for a single `PUT` request. As such, using the `PUT` method to represent `DataSet` revision is theoretically justified. In the future, when the support of `PATCH` method becomes mainstream, `DataGator` MAY re-implement `DataSet` revision using `PATCH`, possibly in `v2` of its RESTful API.

Data Item Operations

URL Endpoint:

```
^api/v1/<repo>/<dataset>/<key>
^api/v1/<repo>/<dataset>.<rev>/<key>
```

GET: get the `DataItem` object by `<key>` from the `HEAD` revision of the `DataSet` identified by `<repo>/<dataset>`, or, get the `DataItem` from the *historical* revision of the `DataSet` identified by `<repo>/<dataset>.<rev>`.

On success, the response is the requested `DataItem` object, e.g.,

```
$ curl -i https://www.data-gator.com/api/v1/\
> Pardee/IGO_Members/UN
HTTP/1.1 200 OK
Content-Type: application/json
Content-Disposition: attachment; filename="UN.json"
Etag: 870214b768af595b9c91bd8306fee2c1
Last-Modified: Sun, 18 Jan 2015 09:19:52 GMT

{
    "kind": "datagator#Matrix",
    "columnHeaders": 1,
    "rowHeaders": 1,
    "rows": [
        ["Country", 1816, 1817, ..., 2013],
        ["Abkhazia", null, null, ..., 0],
        ...
        ["Zimbabwe", null, null, ..., 1]
    ],
    "columnsCount": 199,
    "rowsCount": 337
}
```

To facilitate cache control, a `GET` request MAY specify *conditional request* headers `If-None-Match` and `If-Modified-Since` as specified by RFC 7232, e.g.,

```
$ curl -i \
> -H "If-None-Match: 870214b768af595b9c91bd8306fee2c1" \
> https://www.data-gator.com/api/v1/Pardee/IGO_Members/UN
HTTP/1.1 304 NOT MODIFIED
Etag: 870214b768af595b9c91bd8306fee2c1
```

And likewise,

```
$ curl -i \
> -H "If-Modified-Since: Sun, 18 Jan 2015 09:19:52 GMT" \
> https://www.data-gator.com/api/v1/Pardee/IGO_Members/UN
HTTP/1.1 304 NOT MODIFIED
Etag: 870214b768af595b9c91bd8306fee2c1
```

Remarks: `DataItem` objects can be considerably large in volume. It is recommended to cache `DataItem` objects on the client side and use conditional requests whenever possible to avoid repetitive transmission.

Task Operations

URL Endpoint:

^api/v1/task/<repo>

^api/v1/task/<repo>/<startIndex>

GET: get a Page of Task objects operating on the Repo identified by <repo>, starting with the most recent (or the <startIndex>-th recent) Task.

On success, the response is a Page object containing exactly #itemsCount Task objects. Task objects are sorted in decreasing order of the time of creation.

```
$ curl -i https://www.data-gator.com/api/v1/task/Pardee
HTTP/1.1 200 OK
Cache-Control: no-cache
Content-Type: application/json

{
    "kind": "datagator#Page",
    "items": [
    {
        "kind": "datagator#Task",
        "id": "61b8b259-9574-4c64-937d-5db28f1d350a",
        ...
    },
    ...
    {
        "kind": "datagator#Task",
        "id": "0ab7deec-c842-4932-a4c9-34e9e8f1fef5",
        ...
    }],
    "startIndex": 0,
    "itemsPerPage": 10,
    "itemsCount": 10
}
```

URL Endpoint: ^api/v1/task/<id>

GET: get the Task object identified by <id>.

```
$ curl -i https://www.data-gator.com/api/v1/task/\
> 61b8b259-9574-4c64-937d-5db28f1d350a
HTTP/1.1 200 OK
Cache-Control: no-cache
Content-Type: application/json

{
    "kind": "datagator#Task",
    "repo": {
        "kind": "datagator#Repo",
        "name": "Pardee"
    },
    "id": "61b8b259-9574-4c64-937d-5db28f1d350a",
    "created": "2015-01-18T10:22:00Z",
    "status": "SUC",
    "handler": "datagator.ext.taskqueue.handlers.commit",
    "options": {"retry": 2},
    "args": [...],
    "kwargs": {}
}
```

Remarks: The `status` of a `Task` object is subject to change over time. In other words, `Task` objects SHOULD NOT be cached on the client side.