Generic REST interface for CSPA services

Statistics Netherlands
Jan van der Laan, Edwin de Jonge

Istat
Monica Scannapieco, Diego Zardetto

March 3, 2016

Abstract

A generic REST interface is proposed which can be used by many of the services in CSPA. Furthermore, it is possible to develop a generic implementation which can be used to wrap the REST interface around an existing service making implementation much more simple. A generic interface used by most of the services would make it easier to implement systems that use the services.

1 Introduction

In this report we propose a generic REST (REpresentational State Transfer) interface, which could be used by a large number of services created in the CSPA project (United Nations Economic Commission for Europe (UNECE), 2013a). Many statistical services are simple data processing steps. As input a service receives one or more datasets with some parameters. It will perform some sort of computation, which could take a substantial amount of time. When finished, the service returns one or more result data sets with some logging information. Since many services, especially in phase 5 of the GSBPM (United Nations Economic Commission for Europe (UNECE), 2013b), follow this pattern, it is possible to design a generic REST interface for data processing services. For other services such a data services (another important class of services) another interface will probably be more appropriate. A homogeneous interface for most CSPA services would make using and documenting the services much simpler.

As just mentioned, the interface will focus on services which will process (large amounts of) data. These services have some specific issues that the proposed interface tries to solve. First, the data sources passed to the service and generated by the service are too large to send directly (using a POST request) to the service. Therefore, a reference to the data sources is passed to the service and the service is responsible for retrieving the data. This is also a common pattern in REST where not the contents of resources are passed around but

We thank Dick Woensdrecht for his helpful comments.

URLs to the resources. Second, processing generally takes too long to handle synchronously. Therefore, processing is done asynchronously.

Besides proposing a generic interface for data processing services, we believe that it is possible to create a generic REST wrapper implementing this interface that can be used to wrap a command-line CPSA service with its parameters (CLI). This would reduce the time to implement a CSPA service considerably. This also means that institutes can work together on implementing and perfecting this wrapper, creating a much more robust and tested interface and that institutes implementing a service do not need to have extensive knowledge on implementing REST interfaces.

RESTful applications use HTTP requests for CRUD (Create, Read, Update and Delete) operations and map these on the HTTP verbs POST, GET, PUT and DELETE. All state is modeled with resources. We consider the following as a resource:

- Input data are *data resources* needed by the service.
- Results are *data resources* generated by the service.
- Log files are *text resources* generated by the service[1].
- One processing step is modeled as a *job resource*[2] located at the service.

Therefore, invoking the REST service consists of posting a job resource to the REST service. This job contains the parameters and references to the data resources that are needed by the (CLI) service. The REST service updates the job resource with information on the status of the job and, when the job has finished, references to the log files and result data resources. Each job has a unique id and URL where the current status of the job can be retrieved (using a GET).

2 REST interface

Since the REST interface revolves around the job resource, we give first a description of this resource. After that a description of the REST interface is given.

2.1 Job

The key resource for the service is the job resource. The job closely corresponds to the Process Step Instance from GSIM (United Nations Economic Commission for Europe (UNECE), 2013c). A job resource contains the following information:

id Job id to be used in querying the job properties.

url Unique URL to this job (includes job id).

[1] As such they can also be considered as *data resources* generated by the service. However, they contain information in human readable form on the process. In GSIM (United Nations Economic Commission for Europe (UNECE), 2013c) terms they can be considered to be Information Resources.

[2] The *job resources* corresponds closely to the Process Step Instance of GSIM (United Nations Economic Commission for Europe (UNECE), 2013c).

name Human readable name of the calling process.

version Version of the service that created this job.

input The input is service specific and can consist of:

- Data resources: URL endpoints to input data and its describing meta data, that are to be retrieved using GET by the service
- Configuration parameters. Simple types (such as string, integer or float) can be passed by value. Complex configuration objects can be passed by reference (URL)[3]

result The results are service specific and will consist of:

- Data resources: URL endpoints to generated output data that are to be serviced by the service.

log Contains a URL to the log file of the job.

status Can have one of the following values: 'created', 'scheduled', 'running', 'finished', 'error'.

created Time at which the job was created (in UTC).

started Time at which the job was started (in UTC)

finished Time at which the job has finished (in UTC)

on_end A user can optionally provide a callback URL which gets called (post of the job URL) when the job has finished. This is a common pattern in asynchronous processing.

Some of this information is only available when relevant. For example, the *result* of the job resource is only available when the job has finished (and the job *status* is 'finished').

The current implementation returns the job in JSON (JavaScript Object Notation) format although other formats such as XML could be considered. JSON seems to be more common in REST interfaces. Because it is more lightweight and because it is better supported by many tools (suchs as R, Python, SAS, web frontends), it is easier to use. An example of a complete job description as could be returned by a service (we used the Linear Rule Checking service as an example):

```
{
  "id" : "1234",
  "url" : "http://example.com/LRC/job/1234",
  "version": "0.0.1",
  "name" : "my_process",
  "status" : "finished",
  "input" : {
    "data" : {
      "type" : "ddi+csv",
```

[3]The distinction between data and configuration is not always clear cut. For example, a regional classification used by a geographic service, is that a configuration or a data resource? For the service this distinction is not that important as both can be passed by reference to the service and the service does not really know the difference between configuration and data resources.

```
      "meta" : "http://previous/service/job/1/result/output/meta",
      "data" : "http://previous/service/job/1/result/output/data",
    },
    "rules": {
      "type" : "text",
      "data" : "http://allthedatayouneed.com/myrules.txt"
    }
  },
  "result": {
    "checks" : {
      "type" : "ddi+csv",
      "meta" : "http://example.com/LRC/job/1234/result/checks/meta",
      "data" : "http://example.com/LRC/job/1234/result/checks/data"
    }
  },
  "log": {
    "type":"text"
    "url" : "http://example.com/LRC/job/1234/log",
  },
  "created": "2014-01-01T12:00" ,
  "started": "2014-01-01T12:00" ,
  "finished": "2014-01-01T12:05" ,
  "on_end" : "http://callback.com"
}
```

One thing that can be noted is that the references to the data resources consist of (two or) three parts: type, data and meta. The exact method of passing data around in CSPA is currently undecided. Section 2.3.1 further discusses this topic. For the REST interface the exact method is not important. The only thing that is important is that data is passed by reference, e.g. by passing URLs to data resources.

When creating a new job, only a subset of the information above needs to be passed to the service, namely: *name*, *input* and optionally *on_end*. The service expects the job in JSON format, for example:

```
{
  "name" : "my_process",
  "input" : {
    "data" : {
      "type" : "ddi+csv",
      "meta" : "http://previous/service/job/1/result/output/meta",
      "data" : "http://previous/service/job/1/result/output/data",
    },
    "rules": {
      "type" : "text",
      "data" : "http://allthedatayouneed.com/myrules.txt"
    }
  },
  "on_end" : "http://callback.com"
}
```

The additional information is added to the job description as it becomes available to the service.

2.2 Description of the interface

Table 1 gives an overview of the proposed REST interface. A service can be invoked by posting a job description to `/<servicename>/job`. This will create the job and schedule it for execution. On successful creation of the job a URL to the newly created job is returned. These URLs have the following form: `/<servicename>/job/<jobid>`. Using this URL information on the job can be requested and the job can be deleted.

Any data generated by the job, such as data sets and logging information, can be obtained from URLs nested within the job. For example, logging information can always (all jobs should be required to generate some logging information) be obtained from `/<servicename>/job/<jobid>/log`.

Documentation of the services is very important. Therefore, we suggest that each service will also provide an interface for documentation. We propose `/<servicename>/help` for human readable help and `/<servicename>/example` for an example including example data sets. This information is probably (at least partially) also available in the service catalog.

Since a job can run for a substantial amount of time the orchestrator needs to know when the service is finished. There are two possible methods for this. The simplest is to poll regularly to check if the service has finished. Since the frequency of this polling can be quite low, this will probably not introduce any significant load on the server. Furthermore, the service could use the `Expires` header to indicate the time after which the job is expected to have finished. A second possibility is to pass a callback to the service using the `on_end` parameter of the job. The URL in the `on_end` parameter will receive a POST with the URL to the job when the job has finished.

2.3 Other points, open ends and possible extensions

2.3.1 Data resources

In the current description of the REST interface, data resources consist of three parts: a type, a URL to the meta data and a URL to the data.

It's important to clarify that metadata have two different components, both of which must be provided to the REST interface in order to fully specify them, namely: (i) a model to which data are assumed to be compliant and (ii) a format. For instance, data can be represented as an XML file, according to an XSD schema (defining the data model) that in turn is in the XML format. As a further example, SDMX data can be represented as an XML file, according to a Data Structure Definition that is in the XML format. In the examples above DDI is used for the meta data format and CSV as the data format. The current specification of CSPA does not prescribe a single data format. However, if we want to be able to connect services with a little effort as possible (which is one of the key goals of CSPA) some sort of agreement has to be reached on the data model and format used for exchange between the services.

The exact data and meta data specification used by the CSPA services, is not important for the exact interface used, except for the following two points:

- It is probably best to transfer the meta data and the data separately. First, (depending on the meta data specification used) parsing the meta data is difficult for large data sets when the data and meta data are in the same

Table 1: REST interface for CSPA services

Resource	HTTP verb	Response code	Description
/<servicename>/job			
	POST		Creates a job. Uses *input parameters package*
		201	Job successfully created. Returns URL to created job. Job execution is started.
		412	Precondition failed. Job is not created.
	GET		Returns a list of all jobs at the service
/<servicename>/help			
	GET		Returns human-readable help describing input and output parameters of the service.
		200	OK
/<servicename>/example			
	GET		Returns a working test example for the service
		200	OK
/<servicename>/example/input/<datasetname>			
	GET		Returns a working test example data set for the service
		200	OK
		404	NotFound
/<servicename>/job/<jobid>			
	GET		Returns *information response package* regarding specific job
		200	OK
		404	NotFound
	DELETE		
		200	OK
		401	Unauthorized
/<servicename>/job/<jobid>/result/<datasetname>/data			
	GET		Returns the physical data set
		200	OK
		204	NoContent. The data set is incomplete and is not returned.
		404	NotFound
/<servicename>/job/<jobid>/result/<datasetname>/meta			
	GET		Returns the meta data of the data set
		200	OK
		204	NoContent. The data set is incomplete and is not returned.
		404	NotFound
/<servicename>/job/<jobid>/log			
	GET		Returns the log file
		200	OK. Returns the complete log file
		206	PartialContent. Returns an incomplete log file.
		404	NotFound

file. Second, services will often be called with different data sets while the meta data will remain the same, e.g. running the same service each month for the new monthly figures.

- Data and meta data should be passed by reference as it is one of the key concepts of REST that not the resource itself but the URL to the resource is passed around. Furthermore, for large data sets this might also be more efficient as the service might not need all of the data and can also be passed a query in the URL[4]

2.3.2 Data resource ownership

In the current proposal the services will also serve the data sets generated by the service. This raises questions on the responsibilities of the statistical service for the management of the data sets, e.g. do the data sets remain at the service indefinitely or are they automatically removed after a certain period?

Another solution is that besides the statistical services there is also a data service which is responsible for the data resources. However, even then the question remains who is responsible for transferring the data to the data service? The statistical service or the orchestrator?

2.3.3 Chaining services

In the current design it is relatively easy to chain the services. A service B which uses the output from service A can be passed a URL to the output of service A. All the orchestrator has to do is create the jobs on the different services passing in the correct URLs.

2.3.4 Versioning

Clients can ask for a specific version of the service in the HEAD of the POST message by using the `Accept-Version=<version number>` header. When omitted the service will use the latest version.

2.3.5 Requesting a specific output format

Using `Accept=<mimetype>` in the HEAD of a GET request clients can ask for specific representations of a resource. In case of the CSPA service this could mean the asking for `application/json` on `/<servicename>/job` will return a list of jobs in JSON format, while asking for `text/html` will return a (possibly interactive) web page showing the status of all jobs.

[4]How far one can go in this depends on how much the service knows about the data resource and which data resources will be allowed by CSPA. When passing the query directly in the URL passed to the service (as in `http://dataresource/dataset?gender=male`), the service has to know little about the data resource: it can simple perform a GET on the data resource and retrieve the data. However, one could also think of a service which is implemented by performing SQL queries directly on the data resources. In that case the service has to know (or assume) at lot more about the data resource. In the current proposal we assume that the data needed by the service can be retrieved using a GET on the data URL, but this also depend on the data exchange formats allowed by CSPA.

2.3.6 Machine readable description of input and output parameters

The interface contains a help page which gives a human readable description of input and output parameters. It might be of interest to also generate a machine readable description. This could for example be used to automatically generate controls (e.g. input fields) on a orchestrator. One way to implement this would be to allow a client request JSON data from the help page (see 2.3.5).

2.3.7 Authentication

Http allows for authentication. This allows for restrictions on creating jobs, deleting jobs, listing jobs, etc and can also be used to restrict access to data. Since it is the service which needs to access the data, the service needs to be authorized to access certain data resources. The best method for handling this needs to be investigated.

2.4 Generic implementation

The interface as discussed above can be used for a large number of services. Any service that accepts a number of input data sources with some parameters, performs some sort of computation and then returns one or more output data-sets can probably be served by the interface presented above. Most services in phase 5 of the GSBPM (United Nations Economic Commission for Europe (UNECE), 2013b), will follow this pattern. Furthermore, many statistical programs can be run from the command line by passing in some arguments and data sets. For example, SPSS, R and SAS code can be run in this way. All the REST service has to do is to translate the input parameters of the job into the command line arguments, start the code and subsequently serve the output files generated. The amount of configuration needed to translate the input arguments of the submitted job into the command line arguments can vary depending on the the execution model of the different programming languages. However, generally speaking, such effort should be quite limited.

The advantage of such a generic implementation is that all services using this implementation will have the same structured interface, except for its input and output parameters. Also, implementation is much easier. Not everybody implementing a service for CSPA will have to be able to implement a REST service. When the implementation is used by multiple services, this will also mean that the implementation is better tested and will therefore be much more stable.

3 Conclusion

In the previous paragraphs a REST interface is proposed which can be used by many statistical services. We feel that in order to let CSPA succeed it is necessary to have uniform interfaces and uniform data formats. Therefore, we propose that the REST interface proposed above is used as a starting point for a generic REST interface that is to be used by most of the CSPA services.

References

United Nations Economic Commission for Europe (UNECE) (2013a). Common statistical production architecture. Technical report. Version 1.0 December 2013.

United Nations Economic Commission for Europe (UNECE) (2013b). Generic statistical business process model (GSBPM). Technical report. Version 5.0 December 2013.

United Nations Economic Commission for Europe (UNECE) (2013c). Generic statistical information model (GSIM). Technical report. Version 1.1 December 2013.