RESTful API

1. **REST** - **Re**presentational **S**tate **T**ransfer is not a software engineering standard, but rather an architectural style. REST has become popular because it facilitates the implementation of networked and Web-enabled applications. Any RESTful implementation should enforce the following architectural constraints:

    a. **Decouples consumers from producers** - by promoting a lightweight and implicit contract between requesters and responders and lending itself to building client server and distributed computing solutions

    b. **Stateless existence** - by ensuring each request from any consumer contains all the information necessary to service the request, and session state is maintained only on the consumer side. Resource state is maintained on the producer side.

    c. **Able to leverage a cache** - by ensuring that the use of HTTP methods follows the rules of safety and idempotency specified by the HTTP standard. Most specifically that the GET method is always guaranteed to be *safe* and *idempotent* to facilitate cache management and that responses, explicitly or implicitly, identify themselves as cacheable or not. GET responses are implicitly cacheable and all others are not.

    d. **Leverages a layered system** - by ensuring that consumers do not need to know if they are directly connected to end servers or through intermediaries, enabling easier management of load balancing, caching, and security policies.

    e. **Leverages a uniform interface** - by presenting a clear, unambiguous, and implicit application interface with the following characteristics:

        i. Identification of resources - individual resources are identified in requests. The resources themselves are conceptually separate from the representations that are returned to the consumer. For example, the server may send data from its database as JSON, which is not the server's internal representation.

        ii. Manipulation of resources through representations - when a consumer holds a representation of a resource it has sufficient information to request that the producer modify or delete the resource.

        iii. Self-descriptive requests - a request includes enough information to describe to the producer how to process the request and uses HTTP methods to specify the desired intent.

    f. **Hypermedia as the engine of application state** (HATEOAS) - Having accessed an initial URI for the REST application — analogous to a human Web user accessing the homepage of a website — a REST client should then be able to use server-provided links dynamically to discover all the available actions and resources it needs. As access proceeds, the server responds with text that includes hyperlinks to other actions that are currently available. There is no need for the client to be hard-coded with information regarding the structure or dynamics of the REST service.

2. **Our REST API** - Our challenge is to define a clear, concise, API that meets RESTful architectural constraints and our application requirements in an economical, extensible, and sustainable fashion. We actually have two APIs to provide, a RESTful API for resources and a RESTlike RPC

RESTful API

API for handling processing requests. It allows a developer to read and write data (resources). Using a RESTful API a developer can select (GET), insert (POST), update (PUT and PATCH), and delete (DELETE) rows (resources).

Building a complete application (excluding the user interface) using just JDBC is possible with heavy use of stored procedures and triggers, but difficult and clumsy to do well (I've done it and wouldn't wish to repeat the experience). The same is true of REST. You can build a complete application with browser-based code and a RESTful API, but it can be difficult and clumsy to manage using only a stateless API.

In many applications, implementation of transactional processing is critical. REST is stateless and doesn't understand transactions. This is why we need to augment a RESTful API with a procedural API. We need to create a RESTlike API that can use the plumbing (networking, HTTP request/response handling, data representations, conventions, etc.) that we set up for the RESTful API — but designed to invoke transaction-aware procedures on the server side.

a. Access, Navigate, and Manipulate Resources - This is the expressed purpose of the REST architectural pattern and operates with a noun-oriented perspective. The following example updates a "customer" resource:

Http Method: `PUT`
Relative URI: `/v1/customer/3247`
Body:

```
{
  "id": 3367,
  "lastChg": null,
  "dob": "1968-06-19",
  "startTime": "14:57:25",
  "firstName": "John",
  "lastName": "Smith",
  "addr1Type": "Home",
  "addr1Line1": "416 Maple Road",
  "addr1Line2": null,
  "addr1City": "Littleton",
  "addr1State": "MA",
  "addr1PostalCode": "02099",
  "addr2Type": null,
  "addr2Line1": null,
  "addr2Line2": null,
  "addr2City": null,
  "addr2State": null,
  "addr2PostalCode": null,
  "phone1Type": "Home",
  "phone1Number": "555 555-5555",
  "phone2Type": null,
  "phone2Number": null,
}
```

b. Issue Commands or Processing Requests - We have an additional requirement to implement RMI-style functionality with a RESTlike RPC solution — one that effectively uses our REST implementation and security strategy to accomplish verb-oriented tasks. This API is required to replace RMI and to integrate procedural functionality with a browser-based GUIs. The following examples invokes the CLI commands:

Http Method: `POST`
Relative URI: `/v1/nuvisum/cli`
Body:

```
{
  "-dosynchronize": null,
  "-host": "npci-nuvisum",
  "-port": 9051,
  "-pubname": "bank",
  "-repgrouptype": "m"
}
```

Http Method: `POST`
Relative URI: `/v1/nuvisum/cli`
Body:

```
{
  "-createpub": "bank",
  "-host": "npci-nuvisum",
  "-port": "9051",
  "-pubdbid": "1",
  "-reptype": "t",
  "-tables":["tablea","tableb","tablec"]
}
```

3. **HTTP Method Use -** The REST architectural pattern is intended to exploit HTTP and makes use of HTTP methods to identify the purpose of a request. Two important aspects of the individual methods are useful to understand. They are: 1) ***safe*** means that the request will make no changes to the state of any resource, and 2) ***idempotent*** means that making an identical request once, or more than once, will result in exactly the same resource state. See **Table 1** for details.

    a. ***GET*** is the standard HTTP method for retrieving resources. With the addition of a "columns" or "attributes" query parameter, it can be repurposed to provide us a means of requesting selected attributes of resources without transferring the entire content of each resource. This makes populating UI lists and traversing lists on the client side more efficient. It is hard to overestimate the network cost of moving arrays of entire resources across the wire when a few attributes will do. Much of the activity of a RESTful consumer is the manipulation of lists to enable action on specific individual resources.

    b. ***POST*** is the standard HTTP method for creating new resources of a specific type. We can also re-purpose it to "post" commands and processing requests (RESTlike RPC).

    c. ***PUT*** is the standard HTTP method used to replace a specific resource with a new copy. It is also sometimes used to create a new resource. We have no compelling reason to use it for that purpose and will treat that, at present, as an error.

    d. ***PATCH*** is an HTTP method created to update one or more attributes of a specific resource. It was added to provide a means of partially updating a resource in lieu of the more expensive total replacement of a resource. NOTE: An annotation for the PATCH method must be added to the Jersey JAX-RS implementation.

    e. ***DELETE*** is the standard HTTP method for removing resources. There are two types of responses commonly implemented for a DELETE request: 1) RC 200 OK and the

representation of the deleted resource, and 2) RC 204 No Content. The 2nd is the more flexible and more idempotent option for our purposes — because multiple requests will always return the same response.

f. ***HEAD*** is the standard HTTP method for requesting HTTP header information for a resource. We will be using JAX-RS 2.0's default support for HEAD.

g. OPTIONS is the standard HTTP method to enable the client to determine the options and/or requirements associated with a resource, or the capabilities of a server, without implying a resource action or initiating a resource retrieval.

Method	URI	I	S	Description	Input	Returns	RCs
GET	{resource}	Y	Y	Get an array of resources of the resource type (*recommend a default limit to the number of rows returned that can be overridden with the limit parameter*)	Optional: offset = 0 - n limit = 0 - n columns={CSV} where = {key}={CSV} orderby={CSV}	Array of JSON Objects or, when the columns parameter is present, an array of JSON maps	200 204
GET	{resource}/{ID}	Y	Y	Get a specific resource matching the ID	Optional: columns={CSV}	Array of JSON Objects or, when the columns parameter is present, an array of JSON maps	200 204
POST	{resource}	N	N	Create a new resource of the type	JSON Object	Location URI of new resource in HTTP header	201
POST	{resource}/{id}	*	*	Update a specific resource (use PUT instead)			405 Not Allowed
POST	Command URI	N	N	Issue command or processing request	JSON Object	Future token, if 202 RC	200 202
PUT	{resource}/{ID}	Y	N	Replace specific resource with a new copy	JSON Object	Location URI of updated resource in HTTP header	204
PATCH	{resource}/{ID}	Y	N	Update one or more attributes of a resource	JSON Object containing modified or deleted (null) attributes		200
DELETE	{resource}/{ID}	Y	N	Remove a specific resource			204
DELETE	{resource}	Y	N	Remove selected resources of a type (must be qualified with where)	Required: where = {key}={CSV}		204 (405 if no where)
OPTIONS	{resource} or	Y	Y	Get a list of the HTTP methods		Comma-separ	204

	{resource}/{ID}			and API versions supported for the URI and query parameters		ated lists of valid methods and versions in response header	

Table 1 - HTTP Method Use

4. Responses to Requests

a. HTTP response code usage - The standard HTTP response code conventions are appropriate to our requirements and should be used where applicable. The correct choices here can simplify programming and error handling on the consumer side. See HTTP response Codes and **Table 2**, below.

Response Code	Name	RESTful Use
2XX		*Successful*
200	OK	General purpose success response
201	Created	A new resource has been created (POST). The URI of the new resource is returned in the Location field of the HTTP header (Entity)
202	Accepted	The request has been accepted (POSTed processing request) and will be processed asynchronously. A URI for requesting the status of the request has been returned.
204	No Content	The request was successful (PUT, PATCH, DELETE, GET, HEAD, OPTIONS) but there is no data to return. *A 404, Not Found, could be used for GET, but it can make coding easier if the 400 series is used only for error conditions and Not Found is not an error as long as the resource type is valid.*
206	Partial Content	The request was successful (GET, HEAD), but we limited the number of resources returned. More can be requested using *offset* and *limit*.
4XX		*Client error*
400	Bad Request	The request cannot be understood by the server because of a malformed URI or malformed or missing data.
401	Unauthorized	The request requires authorization and either authentication or authorization failed.
404	Not Found	The server could not find a resource matching the URI. *This implies a client error. 204, No Content, is more accurate in most cases in a RESTful application. Use 404 when the resource type is not found, 204 when no instance of a known resource is found.*
405	Method Not Allowed	The request's combination of HTTP method and the URI are not allowed by the application.
409	Conflict	Processing the request would put the resource into an inconsistent state. For example: creating a duplicate resource or a resource with incorrect or conflicting attributes. *This is the appropriate response to use for data validation errors.*
5XX		*Server Error - server-side has responsibility to log error and provide diagnostic info.*

Table 2 - HTTP Response Codes

b. HTTP Header usage - The HTTP Header contains information used to manage HTTP requests and responses. Because the HTTP 1.1 protocol specifies that unrecognized header parameters be ignored, we can add our own, Application-specific parameters to help us manage our own API implementation. The standard discourages new General Header Fields, so we'll limit ourselves to the Request, Response, and Entity, Header Fields as needed. To insure that we do not conflict with any other current or future Header Field Names, we will prefix our header field names with "X-APP-". The following table lists useful custom Header Fields. *We can add more as we need them.*

Part	FieldName: Value	HTTP RC	Purpose
Response	`X-APP-Item-Count: {integer}`	200 (for GET, HEAD) 206	Number of items returned
Entity	`X-APP-Item-Class`	200 206	Java class represented by the returned JSON entity or JSON array members
Entity	`Location: {URI}`	201	The URI of a newly created resource, i.e., `/publication/43`
Response	`X-APP-Future-URI: {URI}`	202	Provide a URI that the client can use to check status/results of an asynchronous request
Response	`X-APP-Valid-Methods: {CSV}`	204	Identify the valid HTTP Methods for this URI (HTTP OPTIONS)
Response	`X-APP-Available-Versions {CSV}`	204	Identify the available versions for this URI (HTTP OPTIONS)
Response	`X-APP-Range-Offset: {integer}`	206	Zero-based offset index of returned item array (used to manage scrolling through answer set)
Response	`X-APP-Range-End: {integer}`	206	Zero-based end index of returned item array (used to manage scrolling through answer set)
Response	`X-APP-Response-Code: {string}`	4XX	Application error code
Response	`X-APP-Response-Message: {string}`	4XX	Application error message
Response	`X-APP-Version`	ALL	Version of the application module responding to request

Table 3 - API-specific HTTP Header Fields

c. Asynchronous futures usage - Response code 202 Accepted is returned when a request is accepted and will be processed asynchronously. A URI should be returned with that response to provide consumer access to the status of the asynchronous process. (Needs further definition and examples) *At some point (in the future), we may want to examine the use of WebSockets to implement producer to consumer notifications.*

d. Response body usage - At a minimum, REST response bodies should contain all the data required by the requestor to properly process the response.

5. **Security Token Use -** We would need to have very compelling reasons not to use the token-based security model used in most modern networked security solutions.

6. **Nouns - Resource URI Model** - At the heart of REST are the URI conventions we choose to identify resources. Resource models, and thus resource URIs are application specific and are composed of resource names and resource IDs. See **URI Examples**, below.

7. **Verbs - Command URI Model** - The URI conventions we use to specify RESTlike RPC requests will determine how well these features will integrate with our RESTful API. See **URI Examples**.

8. **Data Representations** - This is a key implementation issue that will influence the effectiveness of our solution. Where possible, the solutions we use here should be consistent with those we use for data streaming.

    a. JSON Object Notation - The ECMA Standard defines the JSON Data Interchange Format. JSON stands for JavaScript Object Notation and has become the *de facto* standard for REST object representations. It is also widely supported by Java, Python, and C libraries. In Java, the Apache Jackson libraries provide high performance processing of the JSON data format and provide support for Java data types not specifically included in the ECMA standard such as **`date`**, **`time`**, and **`timestamp/datetime`**. Jackson is also used to implement JAX-RS.

    b. MIME types - MIME stands for **M**ultipurpose **I**nternet **M**ail **E**xtensions. It is the standard way of identifying the type of a data being transmitted over the Internet. It is used in RESTful Web services to clearly identify the data format of passed data, i.e., application/json identifies JSON-formatted data and application/octet-stream is used to identify pure binary data.

Type	Description	Typical Subtypes
text	Represents any document that contains text and is theoretically human readable	text/plain, text/html, text/css, text/javascript
image	Represents any kind of image file. Videos are not included, though animated images (like animated gif) are described with an image type.	image/gif, image/png, image/jpeg, image/bmp, image/webp
audio	Represents any kind of audio file	audio/midi, audio/mpeg, audio/webm, audio/ogg, audio/wav
video	Represents any kind of video file	video/webm, video/ogg
application	Represents any kind of application specific binary data, i.e., application/octet-stream is the standard way to identify pure binary data.	application/json, application/octet-stream, application/pkcs12, application/vnd.mspowerpoint, application/xhtml+xml, application/xml, application/pdf ('vnd' denotes vendor-specific data)

Table 4 - MIME Types

c. Large objects (LOBs) - LOBs should not be automatically returned, but should be represented as a URI that can be requested and returned separately. The PATCH method should be used to submit large objects for update or after creates.

   i. Character (CLOB) - Represented as text/plain MIME type.

   ii. Binary (BLOB) - Represented as application/octet-stream MIME type.

9. **Data Compression -** GZIP is the standard mechanism for compressing HTTP messages and is supported by most (if not all) browsers as well as by Java. GZIP performance exhibits an excellent compression versus CPU use ratio.

10. **Testing RESTful Requests/Responses** - One of the advantages of the RESTful environment is the availability of browser-based testing tools. One that works very well is Postman which is a GUI app for the Chrome browser. The PPCD team used cURL, a command line scripting tool and Python-based Requests in their development process.

11. **Versioning Strategy** - How we choose to manage the coexistence of multiple versions of our RESTful API will impact the extensibility and sustainability of our solution.

    a. There are two version identifiers:

       i. Consumer Version is the version explicitly used by the service consumer. It is always the first part of the request URI, i.e, **`v1/customer/374`**. JAX-RS uses that URI part to route a request to the Java class that processes it.

       ii. Producer Version represents the version or versions understood by the producer and ultimately identifies the Java class that will produce the response.

    b. We use a JAX-RS 2.0 pre-matching filter to examine the consumer version and direct the request to the appropriate producer version module (or return an 404 error).

    c. This strategy requires a catalog of producer module versions and the consumer versions they support for the pre-matching filter to use. This can be created at startup, by the pre-matching filter, using custom Java annotations for producer modules.

12. **Implementation Considerations** - Existing open-source Java code can reduce the time/cost of implementing and maintaining embedded REST endpoints.

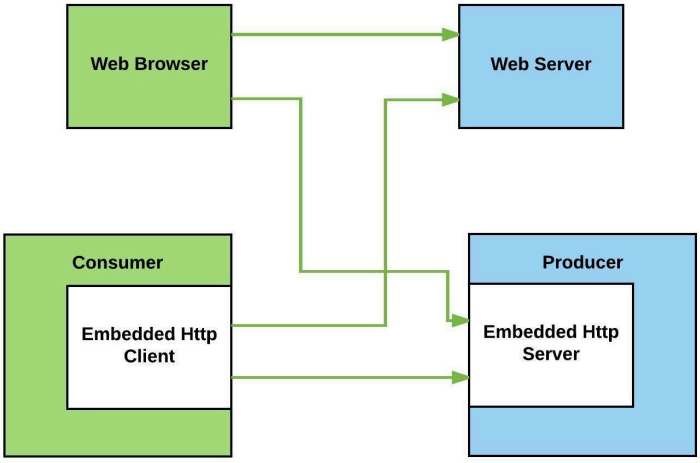

Figure 1 - REST Endpoints

a. **Producer-side** - This is the processor of HTTP requests for resources and processing/commands. It uses JAX-RS 2.0, Java API Specifications for RESTful Web Services Version 2.0.

   i. Jetty embeddable high-performance HTTP server (comes configured with Jersey and Jackson for JAX-RS 2.0 support). At a minimum Jetty, implements the same interfaces as Java HttpServer, but uses the more efficient Java NIO for files and TCP/IP. Uses a one page Java wrapper class to run standalone.

   ii. HttpServer class starting with Java 1.6 is the embeddable implementation of HTTP services that comes with the standard JRE. Needs to be configured to use Jersey and Jackson for JAX-RS 2.0 support. Can use a Java wrapper class to run standalone.

b. Consumer-side

   i. JAX-RS 2.0 HTTP Client API as implemented in the Jersey libraries.

   ii. Open source HttpClient support from Google

   iii. Open source HttpClient support from Apache Jakarta Commons

c. **JSON Libraries** - JSON handling libraries can be found at Jackson and json.org. Jersey uses Jackson to serialize and deserialize Java/JSON.

d. **Test Support** - The Jersey JAX-RS 2.0 implementation provides a test framework that can be set up to be invoked from the build process.

1. **RESTful API** - fully meets RESTful architectural constraints.

a. URI Components

i. Path - is used to identify an individual resource (/**publication/43**) or resource type (**publication**). It may be preceded by the name of the application or servlet intended to process the request (**nuvisum/rest/publication/43**).

ii. Query Parameters - are used to identify options to be used in processing the request. The absence of an option should not cause the failure of the request and, where necessary an option should have a default.

1. **offset=n** - starts a resultset array of resources at the **n**th (zero-based item). **offset** defaults to 0.

2. **limit=n** - limits the size of a resultset array of resources to **n**. Defaults to an application assigned value (perhaps 20 or 50). A **limit** of **-1** signifies unlimited (caution, might result in memory and/or network issues).

3. **columns={CSV list of columns to be selected}**. Defaults to all (*****).

4. **where=**{**and**/**or** separated key-value pairs separated by boolean_operators} - acts like a SQL “where” clause to filter a resultset of resources. No default is needed.

5. **orderby=**{comma-separated list of one or more attribute names} - sorts the resultset of resources (honored by **offset** and **limit**). No default needed.

iii. Form Parameters - not applicable to RESTful API

iv. Header Parameters - as defined

	URI	Meaning	HTTP Methods
1	**/publication?limit=20**	First 20 publications	**GET, HEAD**
2	**/publication?offset=20&limit=20**	Second 20 publications	**GET, HEAD**
3	**/publication**	All publications (default *limit*) or the publication resource type	**GET, HEAD, POST***
4	**/publication/43**	Publication with ID 43	**GET, HEAD, PUT*, PATCH*, DELETE**
5	**/publication/43/table?limit=-1**	All tables in publication 43	**GET, HEAD**
6	**/table/213**	Table with ID 213 (from publication 43)	**GET, HEAD, PUT*, PATCH*, DELETE**

** With resource in message body*

Table 5 - URI Examples

2. **RESTlike API** - does not fully meet RESTful architectural constraints but is designed to operate within the functional framework of a REST/HTTP implementation. It replaces RPC/RMI functionality in a REST application. It is used to issue commands and processing requests and supports transactionally-aware activities. Call parameters are passed in a JSON object contained in the body of the HTTP message.

    a. **URI Components**

        i. Path - is used to identify the application module intended to process the request

        ii. Query Parameters - reserved for future use

        iii. Form Parameters - not applicable to a RESTlike API (*might be used a secondary alternative to JSON objects when using simple HTML forms*)

        iv. Header Parameters - as defined

Table 6 - URI Examples

	URI	Meaning	HTTP Methods
1	`/nuvisum/cli`	Send a request to a processing program (request parameters in message body)	`POST`
2			
3			