**RESTful Applications Standards and Guidelines**

The standards and guidelines here sketch how to design restufl API simple, scalable, visible, portable, reliable, and high performance

**Introduction**

RESTFUL is an architectural style that defines a series of constraints for distributed systems that together achieve the properties of:

* Simplicity
* Scalability
* Modifiability
* Performance
* Visibility (to monitoring)
* Portability
* Reliability

The use of RESTFUL services in the industry has become an essential option for many service providers accommodating various business use cases. Characteristics like simplicity and scalability have led to RESTFUL wide adoption. The need for multi-client multi-device support has positioned RESTFUL as the common protocol that can support almost any front-end technology. Currently, many DFS applications have started to implement RESTFUL and it is expected to become an important building block for next generation DFS applications.

**Objective**

This document serves as a technical guide for internal developers as well as external vendors who are developing RESTFUL DFS applications.

The document outlines common standards and guidelines to implement RESTFUL interfaces consistently across the various areas and teams within DFS.

**Scope**

The rationale of when to use RESTFUL is not covered in this document. This document is intended to help the implementation of RESTFUL interface after the decision is made that RESTFUL is the technology of choice for your application.

**HTTP and Web Fundamentals**

Many of DFS developers are moving from developing thick client/server Java application and starting to acquire Web development skill set. The new emerging architecture requires basic understanding of HTTP, RESTFUL and Java Servlets concepts.

Providing RESTFUL tutorial or technical explanation of these web concepts is beyond the scope of this document, but here is a list of topics that one should be familiar with before reading this document or starting to develop RESTFUL components:

* HTTP methods: GET, POST, PUT, DELETE, HEAD, and OPTIONS
* HTTP Media Content Types: application/json, application/xml, text/plain, text/html, ...
* HTTP Status Codes: 200, 400, 500, & ...
* Java Web Container scopes: Context, Session and Request scopes
* web.xml configuration and web directory structure
* HTTP Header structure and extracting HTTP parameters: PathParam, QueryParam, HeaderParam, ...
* JAX-RS and JSR 311 annotations

**Summary of Standards and DFS REST Policies**

**Policy#1**

* All Java EE based RESTFUL applications (Websphere & Tomcat apps) should follow JAX-RS specifications. Spring and other non-java applications are not bound to JAX-RS.

**Policy#2**

* For Java EE, Spring, and other non-java applications, development teams must adhere to the standards outlined in this document. Namely: following the Verb & URL-based security pattern and resource-based design.

**Policy#3**

* JSON payload should be used. JSON is widely accepted data model for RESTFUL services while XML is viewed as well-formed and standard-based option. JSON is always preferred and should be first option. Only in very special exceptional cases where XML is entertained with REST.

**Policy#4**

* **GET** requests are safe and idempotent. Clients can repeat **GET** requests as many as necessary. Never violate this rule.

**Policy#5**

* **PUT** and **DELETE** are not safe, but idempotent. They change the state of a resource (not safe), but client retries should result in making the same changes to the resource.

**Policy#6**

* Avoid **POST overloading** or **tunneling** (single method for multiple actions). Use instead a distinct identifier (URI) for each atomic operation.

**Policy#7**

* All RESTFUL calls with business transactions or sensitive data must be conducted over TLS. Additional Field Level Encryption (FLE) is required for highly sensitive data elements (e.g. Restricted/Confidential Data or according to data classification as defined by Enterprise Data Management and Cybersecurity)

**Policy#8**

* All REST endpoints must implement authentication and authorization mechanism for all callers. Use **Filter/Proxy** (or similar technique relevant to your technology), enterprise library or API Management sidecar to achieve REST application security and to establish the security context before passing it to the application layers.

This "Chain of Responsibility" design pattern encapsulates the security context as a façade on front of the actual RESTFUL implementation, allowing the RESTFUL implementation to be more useful across channels if chosen to be chained behind different filters.

The role of the interceptor/filter/proxy is to authenticate the consumer such as a user or a calling application. Only if the caller is authenticated the application is allowed to process the request, and the caller attributes are propagated in the request context.

 This is high level design pattern for authentication and for setting up contextual variables. Detailed security design will depend on the specific use case (internal, external, B2B).

**Policy#9**

* Any API must have a version and follow versioning standard, by including the major contract version in its url and API name (see section Version a Restful API for detail). The major contract version is not necessary to change for any backward compatible changes of an API.

**Technical Details**

**Steps to design RESTFUL interface**

1. Identify Resource (URI)
   * Basic element vs. Collection
   * Abstract vs. Concrete (e.g. person > customer > cardmember)
   * Nouns vs. Verbs
   * CRUD vs. Computing process
2. Identify Methods supported for each resource
   * Get, Post, Put, Delete, Head, Options
   * Safe vs. Unsafe
   * Idempotent vs. non-idempotent
3. HTTP Request/Response headers supported
   * Accept, Content-Type, ...
4. List Return Codes for each method
   * Success: 2xx
   * Redirects: 3xx
   * Client Errors: 4xx
   * Server Errors: 5xx
5. Design or reuse a tiered application
   * AOs, BOs, EOs, DAOs
6. Secure the interface

**Version a Restul API Interface**

API interface versioning is different from its underlying code base versioning. It is popular to version the API code base by following Semantic Versioning standard. The API interface only uses the major version number of Semantic Version. Here's how you put API interface version on its API name and the context root of an API URI:

* API name is in form of [API functional name]\_v[major version number]. For example, if a complaint updater API has major version 1, then the API name is ComplaintUpdater\_v1.
* API context root is described in Context Root paragraph of section Resource (URI) Identification. The API's major version name must be the last field of the context root. Using ComplaintUpdater\_v1 API as a example, its context root is /globalpymt/consumer/complaint-updater/v1
* API implementation code can be revisioned multiple times as long as the new implementation does not break backward compatibility, the API version on API name and API context root do not need to be changed, i.e. use the same major version.
* If API implementation makes incompatible API changes, its major version must be changed (increasing major version number); therefore the corresponding API name and API context root must be changed to match the new major version.

**Resource (URI) Identification**

Fully linked RESTFUL representation is described below:

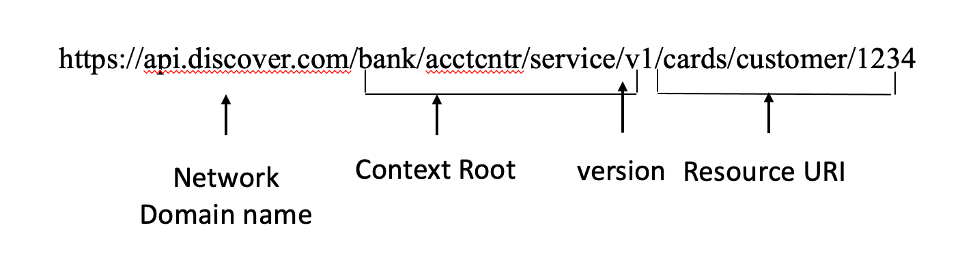
***Network Domain name***: managed via DNS servers. Used to logically group or partition resources for localization, distribution, or to enforce various monitoring or security policies. ([www.example.com](http://www.example.com/), api.example.com, gateway.example.com ...)

***Context Root:*** managed in web descriptor of a web module. API context root is in form of [business namespace]/[focused business area]/[API business function]/[major version number] and must uniquely identify the restul API application across all API implementations. Also provides operational benefits and the ability to apply certain routing, monitoring, or security policies per module.

***URL pattern***: - http(s)://{Network Domain name (:Port number)}/{Context Root with API version}/{Resource URI} used to map to the RESTFUL application handler for processing a specific API function.

***Resource URI***: is a path to identify a set of annotated resource classes and methods followed by query strings

Example:



* **Identify resources from domain nouns**

One of the first steps in developing a RESTFUL application is designing the resource model. The resource model identifies and classifies all the resources the client uses to interact with the server. Of all the aspects of designing a RESTFUL interface, such as identification of resources, choice of media types and formats, and application of the uniform interface, resource identification is the most flexible part.

Analyze your use cases to find domain nouns that can be operated using "create,"

"read," "update," or "delete" operations. Designate each noun as a resource. Use POST,

GET, PUT, and DELETE methods to implement "create," "read," "update," and "delete" operations, respectively, on each resource.

**Example:**

Consider a RESTFUL interface for handling Bank Origination Process where an applicant is applying for a loan. In this example an "application\_form" is an entity in the domain. The actions a client can perform on this entity include "create" a new application\_form, "update" an application\_form, and "view" an application\_form.

However, if you limit yourself to identifying resources based on domain nouns alone, you are likely to find that the fixed set of methods in HTTP is quite a limitation. This is what gives RESTFUL the perception that RESTFUL is suitable for CRUD-style (Create, Read, Update, Delete) applications only. In most applications, CRUD operations make only part of the interface.

* **Support for Computing/Processing Functions**

One of the most common perceptions of RESTFUL's architectural constraints is that they only apply to resources that are "things" or "entities" in the application domain. Although this may be true in a number of cases, scenarios that involve processing functions challenge that perception.

**Examples:** processing the distance between two places, ATM locator, driving direction, currency exchange, validating a credit card, verifying an applicant to a loan product.

Processing functions are not uncommon. Websites like Babel Fish (<http://babelfish.yahoo.com/>), XE.com (<http://www.xe.com/>), and Google Maps ([http://maps.google.com](https://dta.discoverfinancial.com/architectures/api-rest-standard-guideline)) take some inputs, process them with the data stored in their backend servers and some algorithms, and return results. These are all processing functions.

The solution relies in treating the processing function as a resource, and use HTTP ***GET*** to fetch a representation containing the output of the processing function. Use *query parameters* to supply inputs to the processing function.

One way to address such use cases is to treat the processing function itself as a resource.

In the first example, you can treat the distance calculator as a resource and the distance as its representation. Similarly, "direction finder," "points of interest finder," and "credit card validator" can all be resources with "directions," "points of interest," and "validation result" as representations of those resources

GET /atm/dist\_calc?lats=47.610&lngs=-122.333&late=37.788&lnge=-122.406

GET /directions?from=Chicago&to=San%20Francisco

GET /poi?lat=47.610&lng=-122.333

GET /account/validator?corrNum=1234567890123456

GET /application\_form/1234/applicants\_verifier?id=4567

* **The Use of Operator Resource**

Many applications may have the need for write operations that involve modifying more than one resource atomically, or whose mapping to PUT or DELETE is not obvious.

In this case, designate a controller resource for the distinct atomic operation (i.e. operator resource). Let clients use the HTTP method **POST** to submit a request to trigger the operation. If the outcome of the operation is the creation of a new resource, return response code 201 (Created) with a Location header referring to the URI of the newly created resource. If the outcome is the modification of one or more existing resources, return response code 303 (See Other) with a Location with a URI that clients can use to fetch a representation of those modifications. If the server cannot provide a single URI to all the modified resources, return response code 200 (OK) with a representation in the body that clients can use to learn about the outcome.

 "Tunneling" occurs whenever the client is using the same method on a single URI for different actions. Avoid tunneling at all costs. Instead, use a distinct resource (such as operator URI) for each atomic operation.

* **POST Overload**

URIs should be used as *unique* identifiers for *actions* and *resources*. If URIs become generic gateways for unspecified information and actions, this can result in improperly cached responses, possibly even the leakage of secure data that should not be shared without appropriate authentication.

* Some HTTP clients, servers, network or middleware components don't understand/support certain HTTP headers (e.g. PUT & DELETE). If you run into this scenario, use *operator URI* to create new action identifier to operate on resource.

Overloading POST to mimic a PUT or DELETE method is sometime a necessary evil, and must only be considered after the operation URI cannot be an option (e.g. if the Operator URI will lead to too many variations of URIs) Example: Use: POST /123/items/2/update Do not use: POST /123/items/2?\_method=update

* **What makes a good resource?**
* A resource only needs 2 base-URIs to be represented: one to represent an **element** and the other to represent a **collection**. (E.g. /cardmember and /cardmember**s**). The main guideline here is to be consistent in defining these base-URIs for all your resources and collections of resources.
* Sometimes the natural entities in a domain could be defined at various **abstraction levels**. Example: person > customer > cardmember > platinumCardmember. Highly abstracted resources could lead to confusion and are not very useful. On the other hand too concrete resources may lead to unnecessary complexity and repetitions. A good design will define a resource at an appropriate medium abstraction level. (E.g. /cardmember)
* For resources that have **associations** or **façade controllers**, the best practice is to keep the URI within 2 to 3 levels deep. An association is needed to represent a hierarchical relationship. More than 4 levels depth may indicate a need for a refactoring of the resource. Example of 2-level associations: /cardmember/123/statement Example of 2-level façade controller: /application\_form/123/verifier
* After attempting to define the resource at the appropriate **concrete** granularity and with the proper **association depth**, there might still be some complex variations of a resource, or a need to **filtration** based on certain attributes. In this scenario, it's best to hide all the variations complexity as a query string. (after the '?') Example: /application\_forms?status="pending"&sort="byDate"&sortDir="desc"
* Follow a **consistent naming convention** to define resources: For **resource names**, follow the search engines best practice to use under score "\_" to separate words in the URI. For **attributes** **names**, use the Javascript naming convention. E.g.: myObject.

**Supported Methods**

* **Safe Methods**

In HTTP, **safe** methods are not expected to cause side effects. Clients can send requests with safe methods without worrying about causing unintended side effects. To provide this guarantee, **implement safe methods (GET) as read-only operations**.

Safety does not mean that the server must return the same response every time. It just means that the client can make a request knowing that it is not going to change the state of the resource.

**Any client must be able to make GET, OPTIONS and HEAD requests as many times as necessary safely without changing the status of a resource.** If a server's implementation causes unexpected side effects when processing these requests, it is fair to conclude that the server's implementation of HTTP is incorrect.

* **Idempotent Methods**

Idempotency guarantees clients that repeating a request has the same effect as making a request just once. Idempotency matters most in the case of network or software failures. Clients can repeat such requests and expect the same outcome. For example, consider the case of a client updating the price of a product:

# Request

PUT /book/gone-with-the-wind/price/us HTTP/1.1

Content-Type: application/x-www-form-urlencoded

val=14.95

Now assume that because of a network failure, the client is unable to read the response.

Since HTTP says that PUT is idempotent, the client can repeat the request.

**For this approach to work, you must implement all methods except POST to be idempotent.**

In programming language terms, idempotent methods are similar to "setters."

For instance, calling the setPrice method more than once has the same effect as calling it just once.

**Any client should be able to re-try idempotent (e.g. PUT, DELETE) requests as necessary to change the status of a resource.**

Note: DELETE is idempotent method as in practice, the subsequent call will not find the resource to delete, hence not changing any status on the server.

**This is a table summarizing common implementations of the different HTTP methods:**

| **Method** | Safe? | Idempotent? |  |
| --- | --- | --- | --- |
| **GET** | Yes | Yes | For Read only & Algorithmic calculations |
| **HEAD** | Yes | Yes | Simulates GET with no response |
| **OPTIONS** | Yes | Yes | Returns headers and WADL |
| **PUT** | No | Yes | Setters-like methods |
| **DELETE** | No | Yes |  |
| **POST** | No | No |  |

Avoid trouble: According to the JAX-RS specification, you **must** not put multiple HTTP method annotations, such as @javax.ws.rs.POST or @javax.ws.rs.PUT on the same resource method Because HTTP methods have uniquely defined semantics, do not use a resource method for multiple HTTP methods

**Bank Origination - URI Design Example:**

| **URI** | **Method** | **Business Operation** |
| --- | --- | --- |
| /application\_form | POST | Creates new application. Returns app id |
| /application\_form/123 | GET | Returns complete application form object. One can design a query string to filter needed attributes. E.g.: GET /application\_form/123?q=status,amount,date |
| /application\_form/123/applicants | GET | Returns applicants info |
|  | PUT | Updates applicants info |
| /application\_form/123/applicant/456/verification | POST | Verifies Applicant. May return add'l question |
| /application\_form/123/applicant/456/question/789? answer="abcd" | GET | Validates the answer to verification question |
| /application\_form/123/fundings | GET | Returns funding info |
|  | PUT | Updates funding info |
| /application\_form/123/fulfillment | POST | Fulfills application (open acct) |

Given that the above operations are stateless. The final POST method (to create new account) **must** not make any assumptions about prior steps. It must validate that all prior steps in the process have been fulfilled. This could be achieved by using server side records of prior steps, if exist, otherwise application could generate a session token to tie the various steps and to ensure proper order of steps and completeness of the process within a session.

 Long running workflows must have server side records for the process steps.

* **HEAD & OPTIONS**

Per the JAX-RS specifications: by default runtime will automatically support the methods HEAD and OPTIONS if not explicitly implemented. (you may check your specific framework for default behavior)

For **HEAD**, the runtime will invoke the implemented GET method, if present, and ignore the response entity, if set.

For **OPTIONS**, the Allow response header will be set to the set of HTTP methods supported by the resource. In addition, the JAX-RS runtime will return a Web Application Definition Language (WADL) document describing the resource; see <http://www.w3.org/Submission/wadl/> for more information.

**REST Security**

All RESTFUL calls with business transactions or sensitive data must be conducted over HTTPS (TLS). If the restful request contains sensitive information, the sensitive information can't be put in the restful request url and must be included in the body of the request and the HTTPS request method must be POST instead of GET and "query" must be added to the end of the url to indicate this is a query(get) operation. For example:

*POST /cardissuer/acctnbr/v1/acctkey/query*

For additional security factors, external facing applications must encrypt sensitive information within the payload with field-level encryption (FLE).

Example of sensitive information like Restricted/Confidential Data or according to data classification as defined by Enterprise Data Management and Cybersecurity.

Specific implementation description is beyond the scope of this document. RESTFUL security could be implemented in various layers in the infrastructure or application such as API Management Platform (CA/Layer7 Gateway), Access Manager (CA Siteminder), and Application Servlet Filter (as described in patterns above).

Your RESTFUL implementation must design for security with the following aspects:

* Client Authentication
* User Authentication
* Client Authorization
* User Authorization
* Input Validation
* Cryptography

**Client Authentication:**

There are multiple solution patterns for your RESTFUL application to authenticate the clients. Patterns are continuously being developed and reviewed. For specific solution or use case please find the appropriate pattern by consulting with EA and Security team.

In general, if your application is using API Management platform (like Layer7) you should use **API key and secret** generated by the API platform to authenticate calling app or to implement OAuth v2 Client Credential. If your application is not going through API Management platform you can use other security mechanisms to authenticate caller applications (e.g. JWT token validation, mTLS ...)

**User Authentication:**

User authentication could also be implemented in various ways depending on infrastructure security available to your RESTFUL service.

One approach to integrate existing identity provider (e.g. company's LDAP, or Account Center user's registry) with your app is OAuth v2 Authentication Code profile.

**Client Authorization**

When using API Platform management (e.g. Layer7), the consumers are registered with your RESTFUL service and must use **API Key/Secret** to identify their client app(s). The API Platform is responsible for authorizing which consumer app(s) has access to which API(s).

In addition to API-level authorization, if using **OAuth scope**, API Platform should implement authorization for: READ, WRITE and HEALTHCHECK scopes, mapping certain methods to each scope.

**User Authorization:**

Some RESTFUL applications may have requirements to authorize end-users before allowing them to take certain high level atomic actions within their APIs. We will call this coarse-grained authorization. If a RESTFUL application has this requirement or is responsible for updating backend systems (not read only), you must use URI-based security to support coarse-grained authorization.

In this scenario, application access control will depend on two variables, URI (e.g. resource URI) AND the HTTP verb/method (e.g. Get, Post ...). **This type of access control could be implemented using Access Manager products (e.g. Siteminder) or by application code using special authorization Servlet Filter approach.**

All other methods not required (e.g PUT, DELETE) must return a generic error.

Other fine-grained authorization or data contextual authorization as required per application should be handled by application code or by Entitlement Server product.

**Insecure direct object references**

Just like any web application, all RESTFUL web services must be protected against direct object referencing.

**Cross-site request forgery protection (CSRF)**

For REST interfaces exposed to client applications (Call Center Apps), and with critical methods like PUT, POST and DELETE should be protected from CSRF attacks. Typically this is achieved using token-based approach. Please refer to the following link for more information on CSRF protection.

* [https://www.owasp.org/index.php/Cross-Site\\_Request\\_Forgery\\_(CSRF)\\_Prevention\\_Cheat\\_Sheet](https://www.owasp.org/index.php/Cross-Site%5C_Request%5C_Forgery%5C_(CSRF)%5C_Prevention%5C_Cheat%5C_Sheet)

**Input validation & Output Encoding:**

All Input must be validated for malicious input characters. Generally a whitelist of allowed characters is preferred then a blacklist. Client side input validation is good to prevent accidental exploits, but all input must be validated on server side as well. Along with input validation, the receiving client/application must implement very strong output encoding.

**Secure parsing**

In instances where the RESTFUL application is using XML for communication, secure parsing of the XML should be implemented. This will help in mitigating attacks like XXE and parser expansion attacks.

* [https://www.owasp.org/index.php/XML\\_External\\_Entity\\_(XXE)\\_Processing](https://www.owasp.org/index.php/XML%5C_External%5C_Entity%5C_(XXE)%5C_Processing)
* [http://ws-attacks.org/index.php/Main\\_Page](http://ws-attacks.org/index.php/Main%5C_Page)

**Validate incoming content-types**

Every RESTFUL POST or PUT call will have a Content-Type (e.g. application/xml or application/json) specified by the client. The service or client should not assume the Content Type and validate that Content-Type header and the actual content is the same. For an unexpected Content-Type or missing Content-Type Header, the content must be rejected with a generic error message.

The Jersey/JAX-RS framework has specific annotations to handle this without additional coding effort.

**Validate response types**

Many REST services have multiple response types (e.g. application/xml or application/json) and clients specify the order of response types by the Accept header in the request. If you're consuming a REST service and response content type is missing, do not default to the Content-Type of the request. Reject the response with an appropriate error message.

[https://www.owasp.org/index.php/RESTFUL\_Security\_Cheat\_Sheet](https://dta.discoverfinancial.com/architectures/api-rest-standard-guideline)

For more implementation details about security please consult with Enterprise Architecture and Info Security.