Design Guidelines and Best Practices

RESTful Applications Standards and Guidelines

Revision 1.0

01/31/2013

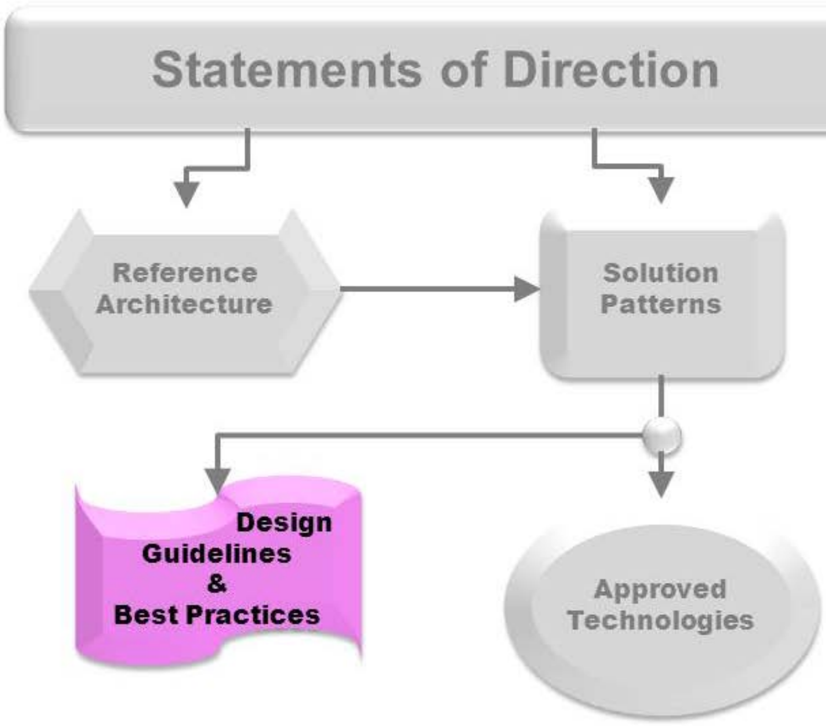


Table of Contents

Table of Contents 2
1.0 Scope 3
1.1. HTTP and Web Fundamentals 3
2.0 Purpose 4
3.0 Introduction 5
4.0 Summary of Standards and DFS Design Patterns 6
5.0 Technical Details 8
5.1. DFS Standard REST Development Framework 8
6.0 Step to Design REST Interface 10
6.1. Resource (URI) Identification 10
6.2. Supported Methods 14
6.3. REST Security 16
7.0 Glossary 18
Revision History and Contributors 19
Meta Tags 20

List of Figures

Figure 4-1: Chain of Responsibility 7
Figure 5-1: web.xml sample 9

List of Tables

Table 6-1: Summary of Common Implementation of HTTP Methods 15
Table 6-2: Bank Organization - URI Design Example 16

1.0 Scope

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

1.1. 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, REST and Java Servlets concepts.

Providing REST 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 REST components:

- HTTP methods: GET, POST, PUT, DELETE
- HTTP Media Content Types: application/xml, text/plain, text/html
- HTTP Return Codes
- 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 tutorials

2.0 Purpose

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

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

3.0 Introduction

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

4.0 Summary of Standards and DFS Design Patterns

At the time of writing this document, **Jersey 1.2** is the approved standard framework under WAS 6.1/Java 5 platform. To help future migration or upgrades efforts, development teams must adhere to the standards outlined in this document to maximize usage of JAX-RS specifications and maintain minimal dependency on Jersey-specific functionalities.

Contact Enterprise Architecture for updated directions regarding REST approved framework under WAS8/Java 6 and beyond.

Either JSON or XML payload may be used. While XML is preferred as more robust and standard-based option, usually the front-end client technology will drive this decision. For example, JSON has a lighter and more appropriate footprint for mobile devices, while Desktop applications can process XML without the need to transform the backend data model.

- **GET** requests are safe and idempotent. Clients can repeat **GET** requests as many as necessary. Never violate this rule.
- **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.
- Avoid **POST overloading** or **tunneling** (single method for multiple actions). Use instead a distinct identifier (URI) for each atomic operation.
- Use **Servlet Filter** to achieve RESTful application authentication 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 REST implementation, allowing the REST implementation to be more useful across channels if chosen to be chained behind different filters.

The role of the filter is to authenticate the user, to generate a new token for valid sessions as well as new requests, to set the security context (javax.ws.rs.core.SecurityContext) and to add it to the HttpServletRequest object if the caller is authenticated.

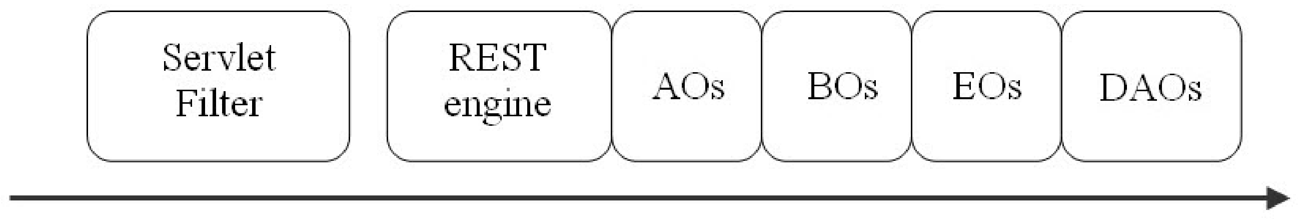
This is high level design pattern for authentication and for setting up contextual variables. Detailed security design will depend on the specific use case (internal, external, B2B). For example, OAuth is an attractive option for authorization delegation but it requires more management of the REST layer that need to be justified by a relevant business case.

REST application must follow DFS naming convention, and packaging best practices. Code organization must follow DFS JEE **Tiered Architecture:**

- Presentation Tier - REST annotated class
- Process Tier – Application Objects (AO) and Business Objects (BO)
- Persistence Tier – Entity Objects (EO) and Data Access Objects (DAOs)

This tiered pattern enables applications to support multiple contexts if needed, or to implement multiple AO objects/methods to control various sequencing of calls to BO methods. Atomic transactions could be started at AO or EO layers as needed.

Figure 4-1: Chain of Responsibility



5.0 Technical Details

5.1. DFS Standard REST Development Framework

The standard REST development framework for WAS6/Java 5 platform is **Jersey 1.2**.

Jersey is Sun's production quality reference implementation for **JSR 311**: JAX-RS (Java API for RESTful Web Services). Jersey implements support for the annotations defined in JSR-311, making it easy for developers to build RESTful applications with Java. Jersey also adds additional features not specified by the JSR which must NOT be used.

JAX-RS/JSR311 specifications became official part of JEE 6 which will become available in future WAS platform along with Java 6. Please refer to WAS8 migration guidelines for updated direction regarding development framework.

To help future migration or upgrades efforts, development teams must adhere to the following to maximize usage of JAX-RS specifications and maintain minimal dependency on Jersey-specific functionalities and configurations:

- Only implement JAX-RS / JSR311 annotations. List of standard annotations is provided at http://jsr311.java.net/nonav/releases/1.1/index.html
- Only use JAX-RS standard APIs and packages.
  A quick look at the import section of your REST handler class must not include dependencies from framework extensions like "jersey, sun, ibm, wink, apache, etc…" It should only contain packages from JAX-RS standard packages javax.ws.rs..*
- Do not use Jersey specific features like "Filters". Use instead the JEE Servlet Filter.
- Do not use @SecurityContext to extract the security context object. Extract it instead from the Request object. ******* snippet *******
- Use Web Deployment Descriptor (web.xml) configuration that's compatible with future JEE specification:

Figure 5-1: web.xml sample

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">

    <display-name>HelloWorld Application</display-name>
    <description>
        This is a simple web application with a source code organization
        based on the recommendations of the Application Developer's Guide.
    </description>

    <servlet>
        <servlet-name>HelloServlet</servlet-name>
        <servlet-class>examples.Hello</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>HelloServlet</servlet-name>
        <url-pattern>/hello</url-pattern>
    </servlet-mapping>

</web-app>
```

6.0 Step to Design REST 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
   - 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

6.1. Resource (URI) Identification

Fully linked REST representation is described below:

- 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, api.example.com, gateway.example.com)
- ***Context Root:*** managed in web descriptor of a web module. Help in modularizing your applications under same domain/sub-domain. Also provides operational benefits and the ability to apply certain routing, monitoring, or security policies per module.
- ***URL pattern***: servlet mapping to the REST application handler.
- ***REST URI***: annotated resource classes and methods followed by query strings

```
https://domain-name.com/contextRoot/resources/restURI
              ↑              ↑          ↑         ↑
           dns name     web module    REST      @PATH
                        context root  engine    annotation
                                      servlet
                                      mapping
```

Example:

```
https://m.discoverbank.com/acctcntr/resources/customer
              ↑              ↑          ↑         ↑
        dns name        web module    REST      @PATH
                              root    engine    annotation
```

6.1.1. 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 REST 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 REST the perception that REST is suitable for CRUD-style (Create, Read, Update, Delete) applications only. In most applications, CRUD operations make only part of the interface.

6.1.2. Support for Computing/Processing Functions

One of the most common perceptions of REST'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**) take some inputs, process them with the help of 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

6.1.3. 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.

6.1.4. 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 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

6.1.5. 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 the base-URIs for all 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.

6.2. Supported Methods

6.2.1. 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.

6.2.2. Idempotent Methods

Idempotency guarantees clients that repeating a request have 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.

6.2.3. HEAD & OPTIONS

Per the JEE6/JAX-RS specifications: By default, the JAX-RS runtime will automatically support the methods HEAD and OPTIONS if not explicitly implemented.

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.

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

Table 6-1: Summary of Common Implementation of HTTP Methods

METHOD	**Safe?**	**IDEMPOTENCE?**	
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

Table 6-2: Bank Organization - 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
/application_form/123/applicants	GET	Returns applicants info
	PUT	Updates applicants info
/application_form/123/applicant/456/verification	POST	Verifies Applicant. May return additional 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.

6.3. REST Security

Note: REST does not have predefined security methods so developers define their own, and often, developers in a hurry to just get... services deployed don't treat them with the same level of diligence as they treat web applications.

- For browser based clients, authenticate using a user password and a one-time-password (OTP) generated by a soft or hard token generator. Plain passwords are just not viable. A combination of password plus random key allows users to have simple-to-remember passwords yet be secured.
- After authentication, the browser client should obtain an expirable cookie that it forwards with each request that contains authentication information the server will use to authenticate subsequent requests.
- For non-browser clients, consider using digitally signed requests. Verification of a digitally signed request would be the authentication mechanism. The advantage of this is that credentials are different per request since they are

part of the attached signature. A nonce and/or expiration can be included within the digital signature to avoid replay attacks.

- Non-browser clients should make requests on behalf of other services to other services. Attaching multiple signatures to a request would also be advantageous.
- Employ the same security mechanisms for existing APIs as for any other web application. For example, if filtering for XSS on the web front-end, also do it for the APIs, preferably with the same tools.
- Use a framework or existing library that has been peer-reviewed and tested..
- Unless the API is a free, read-only public API, do not use single key-based authentication. It's not enough. Add a password requirement.
- Don't pass unencrypted static keys. If using HTTP Basic and sending it across the wire; encrypt it.
- Use hash-based message authentication code (HMAC) – it is the most secure.

7.0 Glossary

Note: Also refer to the DFS Glossary

Revision History and Contributors

Revision Date	Approved By	Author	Changes	Revision
01/31/2013		R Hauenstein	Reformat for initial release	1.0

Reviewers/Contributors:

Name	Department/Contribution

Meta Tags

Note: Meta Tag in bold is the unique identifier

BT_rest_standards_and_guidelines

BT_rest

BT_standards

BT_guidelines

BT_best_practices