RESTful HTTP in practice

This item in japanese

This article gives a short overview about the basics of RESTful HTTP and discusses typical issues that developers face when they design RESTful HTTP applications. It shows how to apply the REST architecture style in practice. It describes commonly used approaches to name URIs, discusses how to interact with resources through the Uniform interface, when to use PUT or POST and how to support non-CRUD operations.

REST is a style, not a standard. There is neither a REST RFC, nor a REST protocol specification nor something similar. The REST architecture style has been described in the dissertation of Roy Fielding, one of the principal authors of the HTTP and URI specification. An architecture style such as REST defines a set of high-level architectures decisions which is implemented by an application. Applications which implement a dedicated architecture style will use the same patterns and other architectural elements such as caching or distribution strategies in the same way. Roy Fielding described REST as an architecture style which attempts “to minimize latency and network communication, while at the same time maximizing the independence and scalability of component implementations"

Even though REST is heavily influenced by the Web-Technology, in theory the REST architecture style is not bound to HTTP. However, HTTP is the only relevant instance of the REST. For this reason this article describes REST implemented by using HTTP. Often this is called RESTful HTTP.

The idea behind RESTful HTTP is to use the existing features and capabilities of the WEB. REST does not invent new technologies, components or services. RESTful HTTP defines the principles and constrains to use the existing WEB-Standards in a better way.

Resources

Resources are the key abstractions in REST. They are the remote accessible objects of the application. A resource is a unit of identification. Everything that might be accessed or be manipulated remotely could be a resource. Resources can be static, which means the state of the resource will not change over the time. On the other side other resources can have a high degree of variance in their state over time. Both types of resources are vaild types.

For instance, the classes, shown in Figure 1, could easily be mapped to such resources. Mapping entity classes such as `Hotel` or `Room` to resources will not be very comprehensible for object oriented designers. The same is true for mapping control classes which represent coordination, transactions, or control of other classes.

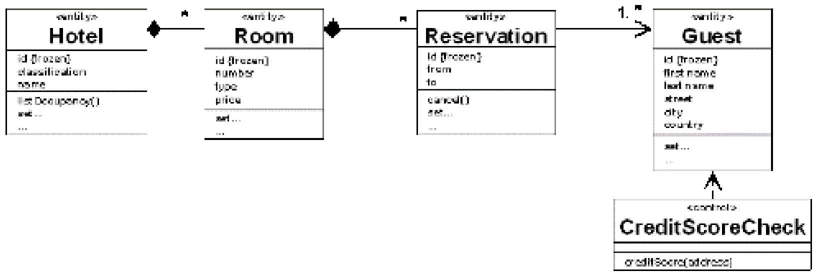

Figure 1: Example analysis model

The analysis model is a good
starting point
for identifying resources. However, there is not necessarily a one-to-one mapping. For instance, the `<Hotel>.listOccupancy()` operation can also be modelled as resources. Further more there could also be resources which represents (parts of) some entities. The primary drivers of the resource design are networking aspects and not the object model.

Any important resource is reachable through a unique identifier. RESTful HTTP uses URIs to identify resources. URIs are providing identification that is common across the Web. They contain everything the client needs to interact with the referred resource.

How to name Resource Identifiers?

Even though RESTful HTTP does not specify how a URI path have to be structured, in practice often specific naming schemas for the URI path is used. URI naming schemas help to debug and trace applications. Often a URI contains the resource type name followed by an identifier to address a dedicated resource. Such a URI will
not
contain verbs which indicate a business operation to process. It is only used to address resources. Figure (a1) shows an example URI of a `Hotel`resource. Alternatively the same `Hotel`can be accessed by URI (a2). A resource can be refered by more than one URI.

```
(a1) http://localhost/hotel/656bcee2-28d2-404b-891b

(a2) http://127.0.0.1/hotel/656bcee2-28d2-404b-891b

(b) http://localhost/hotel/656bcee2-28d2-404b-891b/Room/4

(c) http://localhost/hotel/656bcee2-28d2-404b-891b/Reservation/15

(d) http://localhost/hotel/656bcee2-28d2-404b-891b/Room/4/Reservation/15

(e) http://localhost/hotel/656bcee2-28d2-404b-891b/Room/4/Reservation/15v7

(f) http://localhost/hotel/656bcee2-28d2-404b-891bv12
```

Figure 2: Examples of addressing resources

URIs can also be used by resources to establish relationships between resource representations. For instance a `Hotel`representation will refer the assigned `Room` resources by using a URI, not by using a plain `Room` id. Using a plain id would force the caller to construct a URI by accessing the resource. The caller would not be able to access the resource without additional context knowledge such as the host name or the base URI path.

Hyperlinks are used by clients to navigate through the resources. RESTful APIs are
hypertext-driven
, which means by getting a `Hotel`representation the client will be able to navigate to the assigned `Room` representations and the assigned `Reservation` representations.

In practice, classes such as shown in figure 1 will often be mapped in the sense of business objects. This means the URI stays persistent throughout the lifecycle of the business object. If an new resource is created, a new URI will be allocated. After deleting the resource the URI becomes invalid. The URI (a), (b), (c) and (d) are examples of such identifiers. On the other side a URI can also be used to referring object snapshots. For instance the URI (e) and (f) would refer such a snapshot by including a version identifier within the URI.

URIs can also addresses
"sub" resources
as shown in example (b), (c), (d) and (e). Often aggregated objects will be mapped to sub-resources such as the `Room` which is aggregated by the `Hotel`. Aggregated objects do not have their own lifecycle and if the parent object is deleted, all aggregated objects will also be deleted.

However, if a
“sub"
resource can be moved from one parent resource to another one it should not include the parent resource identifier within the URI. For instance the `Reservation`, shown in Figure 1 can be assigned to another `Room`. A `Reservation` resource URI which contains the `Room` identifier such as shown in (d) will become invalid, if the `Room` instance identifier changes. If such a `Reservation` URI is referred by another resource, this will be a problem. To avoid invalid URIs the `Reservation` could be addressed such as shown in (c).

Normally the resource URIs are controlled by the server. The clients do not have to understand the resource URI namespace structure to access the resource. For instance using the URI structure (c) or the URI structure (d) will have the same effects for the client.

Uniform Resource interface

To simplify the overall system architecture the REST architecture style includes the concept of a
Uniform Interface
. The
Uniform Interface
consists of a constrained set of well-defined operations to access and manipulate resources. The same interface is used regardless of the resource. If the client interacts with a `Hotel`resource, a `Room` resource or a `CreditScore` resource the interface will be the same. The
Uniform Interface
is independent to the resource URI. No IDL-like files are required describing the available methods.

The interface of RESTful HTTP is widely used and very popular. It consists of the standard HTTP methods such as GET, PUT or POST which is used by internet browsers to retrieve pages and to send data. Unfortunately a lot of developers believe implementing a RESTful application just means to use HTTP in a direct way, which it is not. For instance the HTTP methods have to be implemented according to the HTTP specification. Using a GET method to create or to modify objects violates the HTTP specification.

Uniform Interface applied

Fielding's dissertation does not include a table, a list or something else which describes in detail when and how to use the different HTTP verbs. For the most methods such as GET or DELETE it becomes clear by reading the HTTP specification. This is not true for POST and partial updates. In practice different approaches exists to perform partial updates on resources which will be discussed below.

Table 1 list the typical usage of the most important methods GET, DELETE, PUT and POST

Important Methods	**Typical Usage**	**Typical Status Codes**	**Safe?**	**Idempotent**?
GET	- retrieve a representation - retrieve a representation if modified (caching)	200 (OK) - the representation is sent in the response 204 (no content) - the resource has an empty representation 301 (Moved Permanently) - the resource URI has been updated 303 (See Other) - e.g. load balancing 304 (not modified) - the resource has not been modified (caching) 400 (bad request) - indicates a bad request (e.g. wrong parameter) 404 (not found) - the resource does not exits	yes	yes

		406 (not acceptable) - the server does not support the required representation 500 (internal server error) - generic error response 503 (Service Unavailable) - The server is currently unable to handle the request		
DELETE	- delete the resource	200 (OK) - the resource has been deleted 301 (Moved Permanently) - the resource URI has been updated 303 (See Other) - e.g. load balancing 400 (bad request) - indicates a bad request 404 (not found) - the resource does not exits 409 (conflict) - general conflict 500 (internal server error) - generic error response 503 (Service Unavailable) - The server is currently unable to handle the request	no	yes
PUT	- create a resource with client-side managed instance id - update a resource by replacing	200 (OK) - if an existing resource has been updated 201 (created) - if a new	no	yes

	- update a resource by replacing if not modified (optimistic locking)	resource is created 301 (Moved Permanently) - the resource URI has been updated 303 (See Other) - e.g. load balancing 400 (bad request) - indicates a bad request 404 (not found) - the resource does not exits 406 (not acceptable) - the server does not support the required representation 409 (conflict) - general conflict 412 (Precondition Failed) e.g. conflict by performing conditional update 415 (unsupported media type) - received representation is not supported 500 (internal server error) - generic error response 503 (Service Unavailable) - The server is currently unable to handle the request		

POST	- create a resource with server-side managed (auto generated) instance id - create a sub-resource - partial update of a resource - partial update a resource if not modified (optimistic locking)	200 (OK) - if an existing resource has been updated 201 (created) - if a new resource is created 202 (accepted) - accepted for processing but not been completed (Async processing) 301 (Moved Permanently) - the resource URI has been updated 303 (See Other) - e.g. load balancing 400 (bad request) - indicates a bad request 404 (not found) - the resource does not exits 406 (not acceptable) - the server does not support the required representation 409 (conflict) - general conflict 412 (Precondition Failed) e.g. conflict by performing conditional update 415 (unsupported media type) - received representation is not supported	no	no

		500 (internal server error) - generic error response 503 (Service Unavailable) - The server is currently unable to handle the request		

Table 1: Example of a Uniform Interface

Representations

Resources will always be manipulated through representations. A resource will never be transmitted over the network. Instead representations of a resource are transmitted. A representation consists of data and metadata describing the data. For instance the Content-Type header of a HTTP message is such a metadata attribute.

Figure 3 shows how to retrieve a representation by using Java. This example uses the HttpClient of the Java HTTP library
xLightweb
which is maintained by the author.

```
HttpClient httpClient = new HttpClient();

IHttpRequest request = new GetRequest(centralHotelURI);
IHttpResponse response = httpClient.call(request);
```

Figure 3: Java example to retrieve a representation

By performing the HTTP client's call method, an http request will be sent, which requests a representation of the `Hotel` resource. The returned representation, shown in Figure 4, also includes a Content-Type header which indicates the media type of the entity-body.

```
REQUEST:
GET /hotel/656bcee2-28d2-404b-891b HTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6


RESPONSE:
HTTP/1.1 200 OK
Server: xLightweb/2.6
Content-Length: 277
Content-Type: application/x-www-form-urlencoded


classification=Comfort&name=Central&RoomURI=http%3A%2F%2Flocalhost%2Fhotel%2F
```

```
656bcee2-28d2-404b-891b%2FRoom%2F2&RoomURI=http%3A%2F%2Flocalhost%2Fhotel%2F6
56bcee2-28d2-404b-891b%2FRoom%2F1
```

Figure 4: RESTful HTTP interaction

How to support specific representations?

Sometimes only a reduced set of attributes should be received to avoid transferring large data sets. In practice, one approach to determine the attributes of a representation is to support addressing specific attributes as shown in figure 5.

```
REQUEST:
GET /hotel/656bcee2-28d2-404b-891b/classification HTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6
Accept: application/x-www-form-urlencoded


RESPONSE:
HTTP/1.1 200 OK
Server: xLightweb/2.6
Content-Length: 26
Content-Type: application/x-www-form-urlencoded; charset=utf-8


classification=Comfort
```

Figure 5: Attribute filtering

The GET call, shown in figure 5, requests only one attribute. To request more than one attribute the required attributes could be separated by using a comma as shown in figure 6.

```
REQUEST:
GET /hotel/656bcee2-28d2-404b-891b/classification,name HTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6
Accept: application/x-www-form-urlencoded


RESPONSE:
HTTP/1.1 200 OK
Server: xLightweb/2.6
Content-Length: 43
Content-Type: application/x-www-form-urlencoded; charset=utf-8


classification=Comfort&name=Central
```

Figure 6: Multiattribute filtering

Another way to determine the required attributes is to use a query parameter which lists the required attributes as shown in figure 7. Query parameter will also be used to define query conditions or more complex filter or query criteria.

```
REQUEST:
GET /hotel/656bcee2-28d2-404b-891b?reqAttr=classification&reqAttr=name HTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6
Accept: application/x-www-form-urlencoded


RESPONSE:
HTTP/1.1 200 OK
Server: xLightweb/2.6
Content-Length: 43
Content-Type: application/x-www-form-urlencoded; charset=utf-8


classification=Comfort&name=Central
```

Figure 7: Query-String

In the examples above the server always returns a representation which is encoded by the media type `application/x-www-form-urlencoded`. Essentially this media type encodes an entity as a list of key-value-pairs. The key-value approach is very easy to understand. Unfortunately it will not fit well, if more complex data structures have to be encoded. Further more this media type does not support a binding of scalar data types such as `Integer, Boolean or Date`. For this reason often XML, JSON or Atom is used to represent resources (JSON also does not define the binding of the `Date` type).

```
HttpClient httpClient = new HttpClient();


IHttpRequest request = new GetRequest(centralHotelURI);
request.setHeader("Accept", "application/json");


IHttpResponse response = httpClient.call(request);


String jsonString = response.getBlockingBody().readString();
JSONObject jsonObject = (JSONObject) JSONSerializer.toJSON(jsonString);
HotelHotel= (Hotel) JSONObject.toBean(jsonObject, Hotel.class);
```

Figure 8: Requesting a JSON representation

By setting the request accept header, the client is able to request for a specific representation encoding. Figure 8 shows how to request a representation of the media type `application/json`. The returned response message shown in figure 9 will be mapped to a `Hotel`bean by using the library
JSONlib
.

```
REQUEST:
GET /hotel/656bcee2-28d2-404b-891b HTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6
Accept: application/json


RESPONSE:
HTTP/1.1 200 OK
Server: xLightweb/2.6
Content-Length: 263
Content-Type: application/json; charset=utf-8


{"classification":"Comfort",
"name":"Central",
"RoomURI":["http://localhost/hotel/656bcee2-28d2-404b-891b/Room/1",
       "http://localhost/hotel/656bcee2-28d2-404b-891b/Room/2"]}
```

Figure 9: JSON representation

How to signal errors?

What happens if the server does not support the required representation? Figure 10 shows a HTTP interaction which requests for a XML representation of the resource. If the server does not support the required representation, it will return a HTTP `406` response indicating to refuse to service the request.

```
REQUEST:
GET /hotel/656bcee2-28d2-404b-891b HTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6
Accept: text/xml


RESPONSE:
HTTP/1.1 406 No match for accept header
Server: xLightweb/2.6
Content-Length: 1468
Content-Type: text/html; charset=iso-8859-1


<html>
```

```
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1"/>
      <title>Error 406 No match for accept header</title>
   </head>
   <body>
       <h2>HTTP ERROR: 406</h2><pre>No match for accept header</pre>


         ...
   </body>
</html>
```

Figure 10: Unsupported representation

A RESTful HTTP server application has to return the status code according to the HTTP specification. The first digit of the status code identifies the type of the result. `1xx` indicates a provisional response, `2xx` a successful response, `3xx` a redirect, `4xx` a client error and `5xx` a server error. Misusing the response code or always returning a `200` response, which contains an application specific response in the body is a bad idea.

Client agents and intermediaries also evaluate the response code. For instance xLightweb's HttpClient pools persistent HTTP connections by default. After an HTTP interaction a persistent HTTP connection will be returned into an internal pool for reuse. This will only be done for healthy connection. For instance connections will not be returned if a 5xx status code is received.

Sometimes specific clients require a more precise status code. One approach to do this is to add an X-Header, which details the HTTP status code as shown in figure 11.

```
REQUEST:
POST /Guest/ HTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6
Content-Length: 94
Content-Type: application/x-www-form-urlencoded


zip=30314&lastName=Gump&street=42+Plantation+Street&firstName=Forest&country=US&
city=Baytown&state=LA



RESPONSE:
HTTP/1.1 400 Bad Request
Server: xLightweb/2.6
Content-Length: 55
Content-Type: text/plain; charset=utf-8
X-Enhanced-Status: BAD_ADDR_ZIP
```

```
AddressException: bad zip code 99566
```

Figure 11: Enhanced staus code

Often the detailed error code is only necessary to diagnose programming errors. Although a HTTP status code is often less expressive than a detailed error code, in most cases they are sufficient for the client to handle the error correctly. Another approach is to include the detailed error code into the response body

PUTting or POSTing?

In contrast to popular RPC approaches the HTTP methods do not only vary in the method name. Properties such as idempotency or safety play an important role for HTTP methods. Idempotency and safety varies for the different HTTP methods.

```
HttpClient httpClient = new HttpClient();


String[] params = new String[] { "firstName=Forest",
                           "lastName=Gump",
                           "street=42 Plantation Street",
                           "zip=30314",
                           "city=Baytown",
                           "state=LA",
                           "country=US"};
IHttpRequest request = new PutRequest(gumpURI, params);
IHttpResponse response = httpClient.call(request);
```

Figure 12: Performing a PUT method

For instance figure 12 and 13 show a PUT interaction to create a new `Guest` resource. A PUT method stores the
enclosed
resource under the supplied Request-URI. The URI will be determined on the client-side. If the Request-URI refers to an already existing resource, this resource will be replaced by the new one. For this reason the PUT method will be used to create a new resource as well as to update an existing resource. However, by using PUT, the complete state of the resource has to be transferred. The update request to set the `zip` field has to include all other fields of the `Guest` resource such as `firstName` or `city`.

```
REQUEST:
PUT Hotel/guest/bc45-9aa3-3f22d HTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6
Content-Length: 94
Content-Type: application/x-www-form-urlencoded


zip=30314&lastName=Gump&street=42+Plantation+Street&firstName=Forest&country=US&
```

```
city=Baytown&state=LA



RESPONSE:
HTTP/1.1 200 OK
Server: xLightweb/2.6
Content-Length: 36
Content-Type: text/plain; charset=utf-8
Location: http://localhost/guest/bc45-9aa3-3f22d


The guest resource has been updated.
```

Figure 13: HTTP PUT interaction

The PUT method is idempotent. An idempotent method means that the result of a successful performed request is independent of the number of times it is executed. For instance you can execute a PUT method to update the `Hotel`resource as many times as you like, the result of a successful execution will always be the same. If two PUT methods occur simultaneously, one of them will win and determine the final state of the resource. The DELETE method is also idempotent. If a PUT method occurs concurrently to a DELETE method, the resourced will be updated or deleted, but nothing in between.

If you are not sure if the execution of a PUT or DELETE was successful and you did not get a status code such as `409 (Conflict)` or `417 (Expectation Failed)`, re-execute it. No additional reliability protocols are necessary to avoid duplicated request. In general a duplicated request does not matter.

This is not true for the POST method, because the POST method is not idempotent. Take care by executing the same POST method twice. The missing idempotency is the reason why a browser always pops up a warning dialog when you retry a POST request. The POST method will be used to create a resource without determining an instance-specific id on the client-side. For instance figure 14 shows a HTTP interaction to create a `Hotel`resource by performing a POST method. Typically the client sends the POST request by using a URI which contains the URI base path and the resource type name.

```
REQUEST:
POST /HotelHTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6
Content-Length: 35
Content-Type: application/x-www-form-urlencoded; charset=utf-8
Accept: text/plain



classification=Comfort&name=Central



RESPONSE:
HTTP/1.1 201 Created
```

```
Server: xLightweb/2.6
Content-Length: 40
Content-Type: text/plain; charset=utf-8
Location: http://localhost/hotel/656bcee2-28d2-404b-891b


the Hotelresource has been created
```

Figure 14: HTTP POST interaction (create)

Often the POST method will also be used to update parts of the resource. For instance sending a PUT requests which contains only the `classification` to update the `Hotel`resource violates HTTP. This is not true for the POST method. The POST method is neither idempotent nor safe. Figure 15 shows such a partial update by using a POST method.

```
REQUEST:
POST /hotel/0ae526f0-9c3d HTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6
Content-Length: 19
Content-Type: application/x-www-form-urlencoded; charset=utf-8
Accept: text/plain

classification=First+Class



RESPONSE:
HTTP/1.1 200 OK
Server: xLightweb/2.6
Content-Length: 52
Content-Type: text/plain; charset=utf-8


the Hotelresource has been updated (classification)
```

Figure 15: HTTP POST interaction (update)

Partial update can also be performed by using the PATCH method. The PATCH method is a specialized method to apply partial modifications to a resource. A PATCH request includes a patch document which will be applied to the resource identified by the Request-URI. However, the PATCH RFC is in draft.

Using HTTP caching

To improve the scalability and to reduce the server load RESTful HTTP applications can make use of the WEB-Infrastructure caching features. HTTP recognizes caching as an integral part of the WEB infrastructure. For instance the HTTP protocol defines specific message headers to support caching. If the server sets such headers, clients such as HTTP clients or Web caching proxies will be able to support efficient caching strategies.

```
HttpClient httpClient = new HttpClient();
httpClient.setCacheMaxSizeKB(500000);


IHttpRequest request = new GetRequest(centralHotelURI + "/classification");
request.setHeader("Accept", "text/plain");


IHttpResponse response = httpClient.call(request);
String classification = response.getBlockingBody.readString();


// ... sometime later re-execute the request
response = httpClient.call(request);
classification = response.getBlockingBody.readString();
```

Figure 16: Client-side caching interaction

For instance figure 16 shows a repeated GET call. By setting the cache max size larger than 0 the caching support of the HttpClient is activated. If the response contains freshness headers such as `Expires` or `Cache-Control: max-age`, the response will be cached by the HttpClient. These headers tell how long the associated representation is fresh for. If the same request is performed within this period of time, the HttpClient will serve the request using the cache and avoid a repeated network call. On the network, shown in figure 17, only one HTTP interaction in total occurs. Caching intermediaries such as WEB proxies implement the same behaviour. In this case the cache can be shared between different clients.

```
REQUEST:
GET /hotel/656bcee2-28d2-404b-891b/classification HTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6
Accept: text/plain


RESPONSE:
HTTP/1.1 200 OK
Server: xLightweb/2.6
Cache-Control: public, max-age=60
Content-Length: 26
Content-Type: text/plain; charset=utf-8


comfort
```

Figure 17: HTTP response including an expire header

The expiration model works very well for static resources. Unfortunately, this is not true for dynamic resources where changes in resource state occur frequently and unpredictably. HTTP supports caching dynamic resources by validation headers such as `Last-Modified` and `ETag`. In contrast to the expiration model, the validation model do not save a network request. However, executing a conditional GET can safe expensive operations to generate and transmit a response body. The conditional GET shown in figure 18 (2. request) contains an additional Last-Modified header which holds the last modified date of the cached response. If the resource is not changed, the server will reply with a `304 (Not Modified)` response.

```
1. REQUEST:
GET /hotel/656bcee2-28d2-404b-891b/Reservation/1 HTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6
Accept: application/x-www-form-urlencoded


1. RESPONSE:
HTTP/1.1 200 OK
Server: xLightweb/2.6
Content-Length: 252
Content-Type: application/x-www-form-urlencoded
Last-Modified: Mon, 01 Jun 2009 08:56:18 GMT


from=2009-06-01T09%3A49%3A09.718&to=2009-06-05T09%3A49%3A09.718&guestURI=
http%3A%2F%2Flocalhost%2Fguest%2Fbc45-9aa3-3f22d&RoomURI=http%3A%2F%2F
localhost%2Fhotel%2F656bcee2-28d2-404b-891b%2FRoom%2F1


2. REQUEST:
GET /hotel/0ae526f0-9c3d/Reservation/1 HTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6
Accept: application/x-www-form-urlencoded
If-Modified-Since: Mon, 01 Jun 2009 08:56:18 GMT


2. RESPONSE:
HTTP/1.1 304 Not Modified
Server: xLightweb/2.6
Last-Modified: Mon, 01 Jun 2009 08:56:18 GMT
```

Figure 18: Validation-based caching

Do not store application state on the server-side

A RESTful HTTP interaction has to be stateless. This means each request contains all information which is required to process the request. The client is responsible for the
application state
. A RESTful server does not have to retain the application state between requests. The Server is responsible for the
resource state
not for the
application state
. Servers and intermediaries are able to understand the request and response in isolation. Web caching proxies do have all the information to handle the messages correctly and to manage their caches.

This stateless approach is a fundamental principle to implement high-scalable, high-available applications. In general statelessness enables that each client request can be served by different servers. A server can be replaced by another one for each request. As traffic increases, new servers are added. If a server fails, it will be remove from the cluster. For a more detailed explanation on load balancing and fail-over refer to the article Server load balancing architectures.

Supporting non-CRUD operations

Often developers wonder how to map non-CRUD (
C
reate-
R
ead-
U
pdate-
D
elete) operations to resources. It is obviously that `Create, Read, Update and Delete` operations will map very well to resource methods. However, RESTful HTTP is not limited to CRUD-oriented applications.

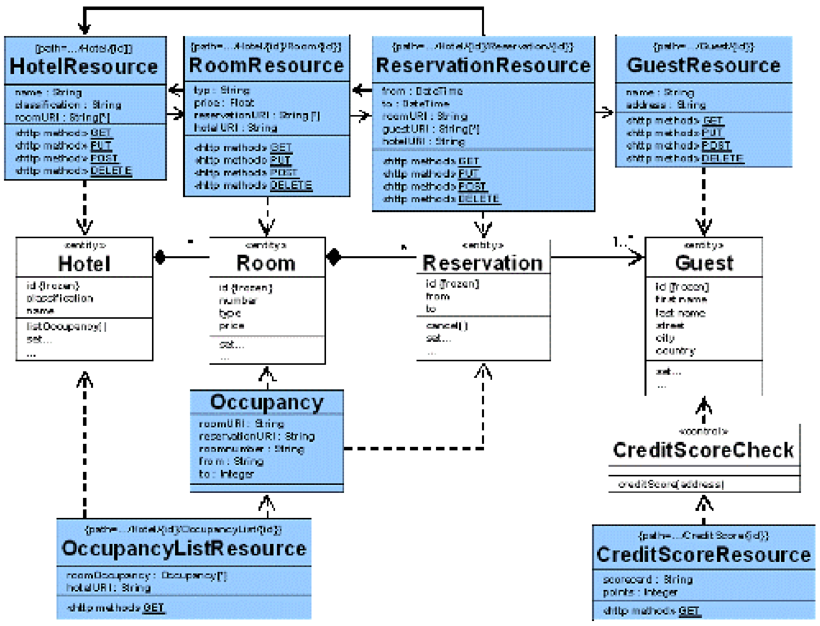

Figure 19: RESTful HTTP Resources

For instance the `creditScoreCheck` class shown in figure 19 provides a non-CRUD operation `creditScore`(...) which consumes an address, calculates the score and returns it. Such an operation can be implemented by a `CreditScoreResource` which represents the result of the computation. Figure 20 shows the GET call which passes over the address to process and retrieves the `CreditScoreResource` representation. The query parameters are used to identify the `CreditScoreResource`. The GET method is safe and cacheable which fits very well to non-functional behaviour of the `CreditScore Check's` `creditScore`(...) method. The result of the score calculation can be cached for a period of time. As shown in figure 20 the response includes a cache header to enable clients and intermediaries to cache the response.

```
REQUEST:
GET /CreditScore/?zip=30314&lastName=Gump&street=42+Plantation+Street&
              firstName=Forest&country=US&city=Baytown&state=LA HTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6
Accept: application/x-www-form-urlencoded


RESPONSE:
HTTP/1.1 200 OK
Server: xLightweb/2.6
Content-Length: 31
Content-Type: application/x-www-form-urlencoded
Cache-Control: public, no-transform, max-age=300


scorecard=Excellent&points=92
```

Figure 20: Non-CRUD HTTP GET interaction

This example also shows the limit of the GET method. Although the HTTP specification does not specify any maximum length of a URL, practical limits are imposed by clients, intermediaries and servers. For this reason sending a large entity by using a GET query-parameter can fail caused by intermediary and servers which limits the URL length.

An alternative solution is performing a POST method which will also be cacheable, if indicated. As shown in figure 21 first a POST request will be performed to create a virtual resource `CreditScoreResource`. The input address data is encoded by the mime type text/card. After calculating the score the server sends a 201 (created) response which includes the URI of the created `CreditScoreResource`. The POST response is cacheable if indicated as shown in the example. By performing a GET request the credit score will be fetched. The GET response also includes a cache control header. If the client re-executes these two requests immediately, all responses can be served by the cache.

```
1. REQUEST:
POST /CreditScore/ HTTP/1.1
Host: localhost
```

```
User-Agent: xLightweb/2.6
Content-Length: 198
Content-Type: text/x-vcard
Accept: application/x-www-form-urlencoded


BEGIN:VCARD
VERSION:2.1
N:Gump;Forest;;;;
FN:Forest Gump
ADR;HOME:;;42 Plantation St.;Baytown;LA;30314;US
LABEL;HOME;ENCODING=QUOTED-PRINTABLE:42 Plantation St.=0D=0A30314 Baytown=0D=0ALA US
END:VCARD


1. RESPONSE:
HTTP/1.1 201 Created
Server: xLightweb/2.6
Cache-Control: public, no-transform, max-age=300
Content-Length: 40
Content-Type: text/plain; charset=utf-8
Location: http://localhost/CreditScore/l00000001-l0000005c


the credit score resource has been created



2. REQUEST:
GET /CreditScore/l00000001-l0000005c HTTP/1.1
Host: localhost
User-Agent: xLightweb/2.6


2. RESPONSE:
HTTP/1.1 200 OK
Server: xLightweb/2.6
Content-Length: 31
Content-Type: application/x-www-form-urlencoded
Cache-Control: public, no-transform, max-age=300


scorecard=Excellent&points=92
```

Figure 21: Non-CRUD HTTP POST interaction

There are also some variants of this approach. Instead of returning a `201` response a `301 (Moved Permanently)` redirect response could be returned. The 301 redirect response is cacheable by default. Another variant which avoids a second request is to add the representation of the newly create `CreditScoreResource` to the `201` response.

Conclusion

Most SOA architectures such as SOAP or CORBA try to map the class model, such as shown in Figure 1, more or less one-to-one for remote access. Typically, such SOA architectures are highly focused on transparent mapping of programming language objects. The mapping is easy to understand and very traceable. However aspects such as distribution and scalability are reduced to playing a second role.

In contrast, the major driver of the REST architecture style is distribution and scalability. The design of a RESTful HTTP interface is driven by networking aspects, not by language binding aspects. RESTful HTTP does not try to encapsulate aspects, which are difficult to hide such as network latency, network robustness or network bandwidth.

RESTful HTTP applications use the HTTP protocol in a direct way without any abstraction layer. There are no REST specific data field such as error fields or security token fields. RESTful HTTP applications will just use the capability of the WEB. Designing RESTful HTTP interfaces means that the remote interface designer has to think in HTTP. Often this leads to an additional step within the development cycle.

However, RESTful HTTP allows implementing very scalable and robust applications. Especially companies which provide web applications for a very large user group such as WebMailing or SocialNetworking applications can benefit from the REST architecture style. Often such applications have to scale very high and very fast. Further more, such companies often have to run their application on a low-budget infrastructure which is built on widely-used standard components and software.

About the author

Gregor Roth
, creator of the xLightweb HTTP library, works as a software architect at United Internet group, a leading European Internet Service Provider to which GMX, 1&1, and Web.de belong. His areas of interest include software and system architecture, enterprise architecture management, object-oriented design, distributed computing, and development methodologies.

Literature

Roy Fielding - Architectural Styles and the Design of Network-based Software Architectures

Steve Vinoski - REST Eye for the SOA Guy

Steve Vinoski - Presentation: Steve Vinoski on REST, Reuse and Serendipity

Stefan Tilkov -A Brief Introduction to REST

Wikipedia - Fallacies of Distributed Computing

Gregor Roth - Server load balancing architectures

Gregor Roth - Asynchronous HTTP and Comet architectures

JSON-lib

xLightweb

12