

WHITE PAPER



## Best Practices for Building RESTful Web services



## Introduction

Representational State Transfer (REST) is an architectural style for designing loosely coupled web services. It is mainly used to develop lightweight, fast, scalable, and easy to maintain, web services that often use HTTP as the means of communication.

**REST is an architectural style, which provides direction for building distributed and loosely coupled services**

**REST is not linked to any particular platform or technology – it's an idea to develop services to function similar to the Web**

In many ways, the World Wide Web itself, which is based on HTTP, is the best example of REST-based architecture.


RESTful applications use HTTP requests to post data (create / update), read data (making queries), and delete data. Hence, REST uses HTTP for all four CRUD (Create / Read / Update / Delete) operations.

REST defines the Web as a distributed hypermedia (hyperlinks within hypertext) application, whose linked resources communicate by exchanging representations of the resource state. The REST architectural style provides guiding principles for building distributed and loosely coupled applications.

REST is not a standard in itself but instead is an architectural style that uses standards like HTTP, XML / HTML / JSON / GIF (Representations of Resources), text / html, text / xml, and image / jpeg (MIME Types). This is why you will never see organizations selling REST-based toolkits.

We should design REST web-services in a way that results in loosely coupled web services, which follow web standards. It should also be development-friendly and flexible enough to be used for a variety of new applications.

In this paper, we will mainly focus on the best practices in REST, and share some quick tips, which can be used for REST web services design.



**The difference between a web service and a website is about who accesses it.**

**The latter is accessed by human beings and former is accessed by programmed clients**

## REST Vs SOAP: When to choose REST?

Simple Object Access Protocol (SOAP) depends primarily on XML to provide messaging services. SOAP uses different protocols for communication, such as HTTP, SMTP, or FTP.

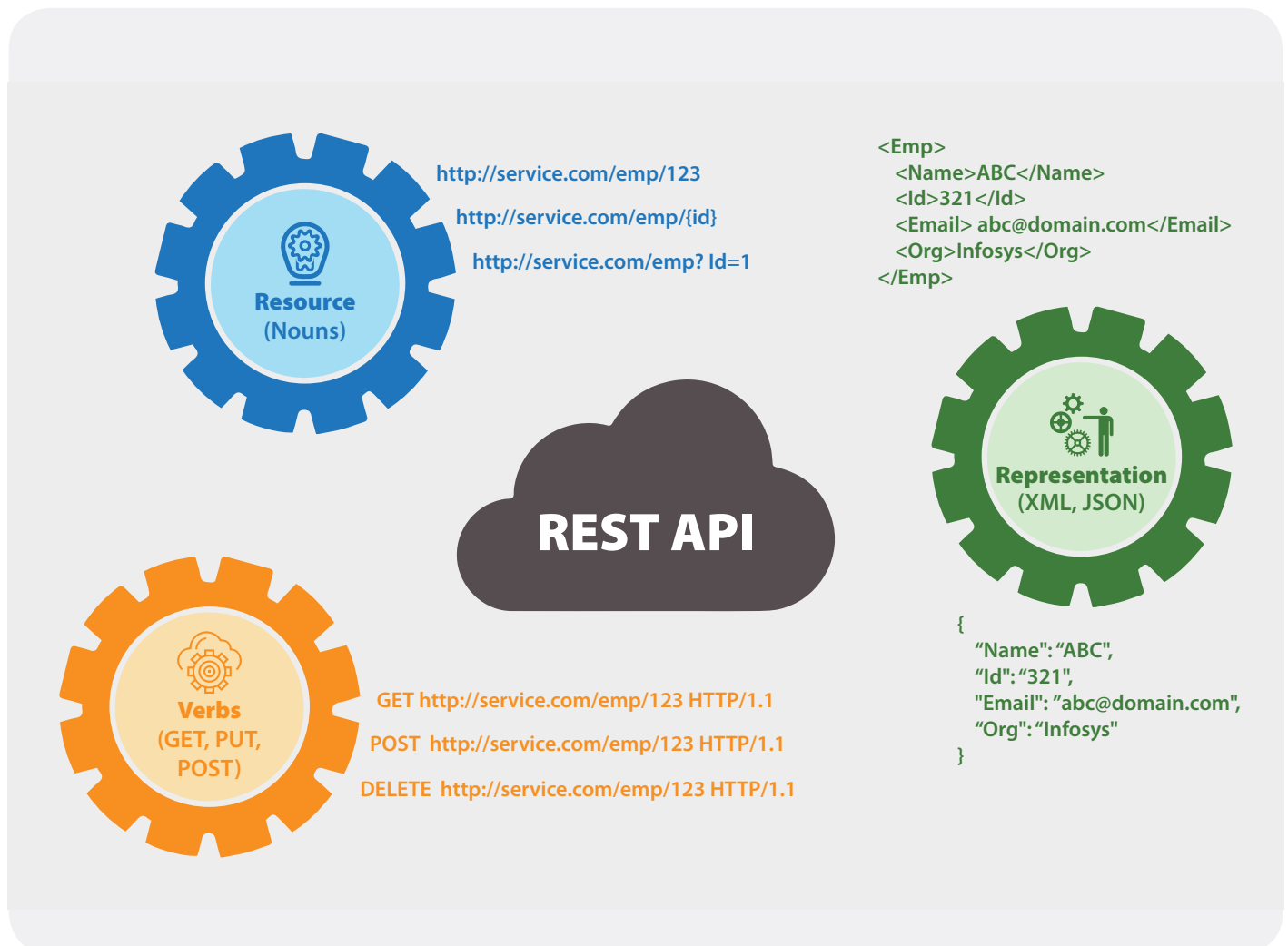
REST on the other hand, is an architectural style, which uses existing HTTP actions and methods; and does not create any new standards. SOAP on the other hand, is a protocol.

REST is more flexible compared to SOAP web services. It has the following benefits over SOAP:

- SOAP uses only XML for messages. REST supports different formats
- REST messages are smaller in size and consume lesser bandwidth
- REST is better in terms of performance with better caching support
- No third party tool is required to access REST web services. Also with REST-based services, learning is easier when compared to SOAP
- There is less coupling between REST Clients (browsers) and Servers; feature-

extensions and changes can be made easily. The SOAP client however, is tightly coupled with the server and the integration would break if a change is made at either end.

REST should be chosen when you have to develop a highly secure and complex API, which supports different protocols. Although SOAP may be a good choice, REST may be better when you have to develop lightweight APIs with great performance and support for CRUD operations.



# REST is like a three-wheeler that rests on Resources, Representation, and Verbs

## Resources

Resources are the fundamental elements of the web platform. While working on REST, the first task is to identify the resources and find out how they are linked with each other. Every resource has a unique identifier on the web platform, which is known as the universal resource identifier (URI) and the best example on the Web is a uniform resource locator (URL). There is no limit on the number of URIs that can refer to a resource. For example we can access a particular domain page (of course, a resource) using `http://yahoo.com` and `http://www.yahoo.com`.

In REST web services, we use nouns to identify a type of resource. Employee information from EmpDB can be accessed using the below URL:`http://EmployeeService/Employee/1`

## Verb

Verb is an HTTP action like POST, GET PUT, DELETE, OPTIONS, etc.

Let's first revisit the HTTP Request.

Example of a GET Request:

```
GET http://www.w3schools.com/ : HTTP/1.1
Status: HTTP/1.1 200 OK
Accept text/xml,text/html;
Accept-Encoding gzip, deflate, sdch
Accept-Language en-US,en;
```

Using URLs, the identity of the target server can be determined for communication,

but HTTP verbs only tell you which action needs to be performed on the host. There are many actions that a client can trigger on the host.

These verbs are –

- GET: retrieve an existing resource
- POST: create a new entry of resource
- PUT: modify an existing resource
- DELETE: remove an existing resource

## Representation

The third and final wheel is about determining a way to showcase these resources to clients. REST supports all formats without any restrictions; so you can use any format for representing the resources.

Based on the client's and server's ability to work with the formats, you can go with JSON, XML, or any other format.

## Best Practices

Here we come up with a few recommendations / best practices that can be used to develop flexible, easy-to-use, and loosely coupled REST APIs.

## Use nouns for Resources and not verbs

Verbs should not be used for resources because doing this will give a huge list of URLs with no pattern – which makes maintenance very difficult. For easy

understanding, use nouns for every resource. Additionally, do not mix up singular and plural nouns, and always use plural nouns for consistency:

```
GET parts/1
GET orders/123
GET seats?id=3
```

## How to handle asynchronous tasks

The Hypertext Transfer Protocol (HTTP) is a synchronous and stateless protocol. The server and client get to know each other during the current request. After this, both of them forget about the request. Because of this behavior, retaining information between requests is not possible at the client and server-side.

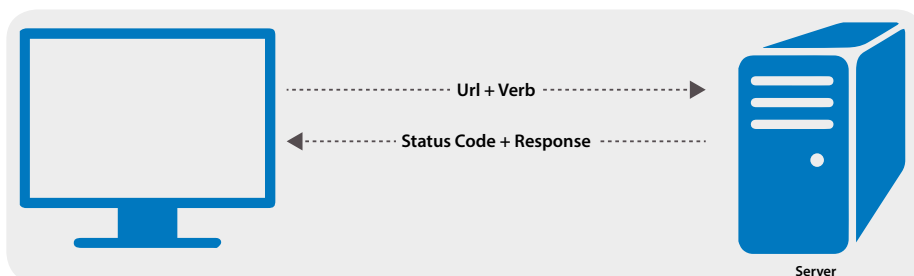
For asynchronous requests (that take too long to complete) follow the steps detailed below –

- Place a GET / Post request which takes too long to complete
- Create a new task and return status code 202 with a representation of the new resource so the client can track the status of the asynchronous task
- On completion of the request, return response code 303 and a location header containing a URI of resource that displayed the result set
- On request failure, return response code 200 (OK) with a representation of the task resource informing that the process has failed. Clients will look at the body to find the reason for the failure.

Here, an example is provided for a file-upload web service, which supports asynchronous model.

Let's start with the client submitting a POST request to initiate a multi file upload task:

```
# Request
POST /files/ HTTP/1.1
Host: www.service.com
```



A response is received, which reflects that the process has started. Response code 202 indicates that the server has accepted the request for processing:

```
# Response
HTTP/1.1 202 Accepted
Content-Type:
application/xml;charset=UTF-8
Content-Location:
http://www.example.org/files/1
<status>
  <state>pending</state>
  <message xml:lang="en">
    File Upload process is started
    and to get status refresh page
    after sometime.
  </message>
</status>
```

The client can check the status by passing a GET request, but if the server is still processing the file upload, it will return the same response.

Once the server successfully completes the file upload process, it redirects the client to the new page. The response code 303 states that the result exists at the URI available in the location header:

```
# Request
GET /file/1 HTTP/1.1
Host: www.service.com
```

```
# Response
HTTP/1.1 303
Location:
www.service.com/file/1
content-Location:
www.service.com/file/ process/1
<status>
  <state>completed</state>
  <message> File Upload is
  completed</message>
</status>
```

## How to combine resources

Composite resources can be used to reduce the number of client / server round-trips. These composites can be built by combining information from other resources. For example, to display your personalized Yahoo page, first aggregate news, blogs, weather, tips, meetings, and then display them as a composite resource.

For the Amazon customer page, you can design a "Customer View" composite resource that aggregates all the information and presents it to the customer. An example of this is provided below:

```
# Request
GET /amazon/customer/0004532/
view HTTP/1.1
Host: www.amazon.com
```

```
# Response
HTTP/1.1 200 OK
Content-Type: application/xml
<view>
  <customer>
    <id>0004532</id>
    <atom:link rel="self"
    href="www.amazon.com/
    customer/0004532">
    <name>ABCD</name>
    <dob>25th July</dob>
  </customer>
  <orders>
    <atom:link href=
    "www.amazon.com/
    customer/0004532/orders" />
    <order>
      <id>...</id>
      ...
    </order>
    ...
  </orders>
  <rewardpoints>
    <atom:link href="www.
    amazon.com/customer
    0004532/rewards">
  </rewardpoints>
  <favorite >
    <atom:link href="www.
    amazon.com/customer/
    0004532/favpages">
  </favorite>
</view>
```

## How to choose the right Representation Format and Media Type

Determine the format and media type, which best matches your requirements and the client's needs. No single format may be right for all kinds of requirements.

In case of the unavailability of requirements, extensively used formats such as XML (application/xml), or JSON (application/json) can be used. To get the right media type,

check IANA website. Designing resource representations is also very important as it defines the relationships between the resources.

XML is the most commonly used format across the applications. On the other hand, JSON (JavaScript Object Notation) is very popular across browsers as it is easier to consume, because it is based on JavaScript

Be flexible while choosing the variety of media types and formats, because we need multiple formats for some representations. For instance, managing parts of automobiles need the following representations:

- HTML pages to describe the parts
- XML-formatted representation for each part
- Parts specification in PDF format
- An Atom feed of all the new parts

## Error Handling

When a server shows some error because of problems within the server, or due to a client request, always return a representation that describes the error in detail. This includes the response status code, response headers, and a body containing the description of the error.

To present more information to the user about the error, include a link to that page; if you are logging errors somewhere, include an identifier of the same.

HTTP 1.1 defines two classes of error codes:

### 1. 4xx: Client Error

4xx codes are used when there is an error / exception at the client's side.

This happens because of requesting an unavailable resource or placing a bad request.

### 2. 5xx: Server Error

5xx codes are used when there is an error / exception at the server-side while interpreting the request

While working with the responses for errors / exceptions, it is better to include the error identifier, error description, optional link to

the error's details, or information to resolve it. Here, an example is provided to return XML when some invalid key is passed to the service:

```
# Response
HTTP/1.1
<?xml version="1.0" encoding="UTF-8" ?>
<error>
  <error_code>2002</error_code>
  <error_msg>Invalid key
  supplied</error_msg>
  <more_info>http://www.service.
  com/docs/error-2002</more_
  info>
</error>
```

## URIs Design for Queries

URIs should be meaningful and well structured. When designing URIs, use path variables to separate the elements of a hierarchy. We should also use query parameters to apply filters, sort, and select specific resources.

Here, are some examples for getting camera from an e-commerce site:

Select all five rated cameras	<code>http://www.service.com/Cameras?review=5</code>
Select all cameras from Nikon brand	<code>http://www.service.com/Cameras?brand=Nikon</code>
Select cameras which were released in the year 2015, in ascending order	<code>http://www.service.com/Cameras?year=2015&amp; sortByASC=release date</code>
Select cameras which have 20X zoom	<code>http://www.service.com/Cameras?zoom=20X</code>

```
# Request
POST /parts/engine/copy;t=<token>
HTTP/1.1
Host: D
```

```
# Response
HTTP/1.1 201 Created
Content-Type:application/xml;
Location:www.service.com/parts
<parts>
  <link rel="self" href="
  www.service.com /parts/
  engine"/>
  ...
</parts>
```

A moving operation is used when one resource needs to be moved to some

## When to use URI Templates

When server does not have all the information to generate a valid URI, we should consider URL Template. URI Templates are designed to return semi-opaque URIs to clients, which allow clients to fill in the missing pieces to generate valid URIs:

```
Query Parameters
http://www.service.com/
part ?queryParam1={qp1}&
queryParam2={qp2}
```

```
Matrix parameters
http://www.service.com/
part;queryParam1={qp1};
queryParam2={qp2}
```

```
URL Path parameters
http://www.service.com/part{t1}/subpart
```

## How to Copy, Merge, or Move a Resource

Consider copying a resource when the client would like to duplicate the resource

other location on the same or a different server. The original resource should also be removed.

In this example, the server uses a URI Template for the client to specify a category for the resource to be moved to:

```
# Request
POST /parts/engine/XYZ/move;t=<token>?
group=Jeep HTTP/1.1
Host: www.service.com
```

Consider merging resources when the client would like to merge two or more resources presented to the server.

and make some changes to the newly created copy. To implement this, we should design a controller to make a copy of the resource and include a link to it for representation.

Request to fetch a representation of the resource and copy the link that you get:

```
# Request
GET /parts/engine
Host: www.service.com

# Response
HTTP/1.1 200 OK
Content-Type:application/xml
<parts
  <link
  href="http://www.service.com/
  parts/engine /copy;
  t=<token>"/>
  ...
</parts>
```

The URI for the controller resource carries a token to make the request conditional. The server may use the token to ensure that duplicate requests are not considered. After this, place a POST request to copy the resource.

In this example, a request is placed to merge one part with another part:

```
# Request
POST /parts/merge?src=part/
XYZ&dest=part/ABCHHTTP/1.1
Host: www.service.com
```

## When to use Link Relation Types

A link relation type describes the role or purpose of a link. Links are not useful if correct semantics are not assigned to them. All relation type values are case insensitive. Multiple values for each relation can be considered.



about	Information about resource
alternate	Replacement Identifier for original resource
current	Current resource in a row of resources
first	First resource in a row of resources
last	Last resource in a row of resources.
prev	Previous resource in a row of resources
next	Next resource in a row of resources
original	Identifier of original resource

In the following example, relation types are used to apply paging on the products:

```
<product xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link rel="current" href="http://service.com/product/122"/>
  <atom:link rel="prev" href="http://service.com/product/121"/>
  <atom:link rel="next" href="http://service.com/product/123"/>
</product>
```

### Security

REST web services should be designed in such a way that they can authenticate users and authorize the resources they are allowed to access and use. Ensure the

confidentiality and integrity of information from the moment it is collected, until the time it is stored, and later presented to the authorized persons. HTTP carries some inherited authentication mechanisms, it allows Basic, Digest, and Custom Authentication.

If an anonymous user tries to access the resources, the service will return a 401 unauthorized response and refuse access. Here is a request example from a client attempting to access a resource that needs authentication:

```
# Request
GET /parts HTTP/1.1
Host: www.service.com
```

**# Response**

```
401 Unauthorized
Content-Type: application/xml; charset=UTF-8
<error xmlns:atom="http://www.w3.org/2005/Atom">
  <message>Unauthorized.</message>
</error>
```

This is an example of when the client passes a request which contains the Authorization header:

**# Request**

```
GET /parts HTTP/1.1
Host: www.service.com
Authorization: Basic aFGHRFKLnvascdubf2536fgsfHGFHG=^&vnbvb%%
```

**# Response**

```
HTTP/1.1 200 OK
```

In Basic Authentication, passwords are passed over the network, in a simple plain text format, which is highly unsecured. To overcome this issue, we may choose an HTTPS protocol, which encrypts the HTTP pipe carrying the passwords.

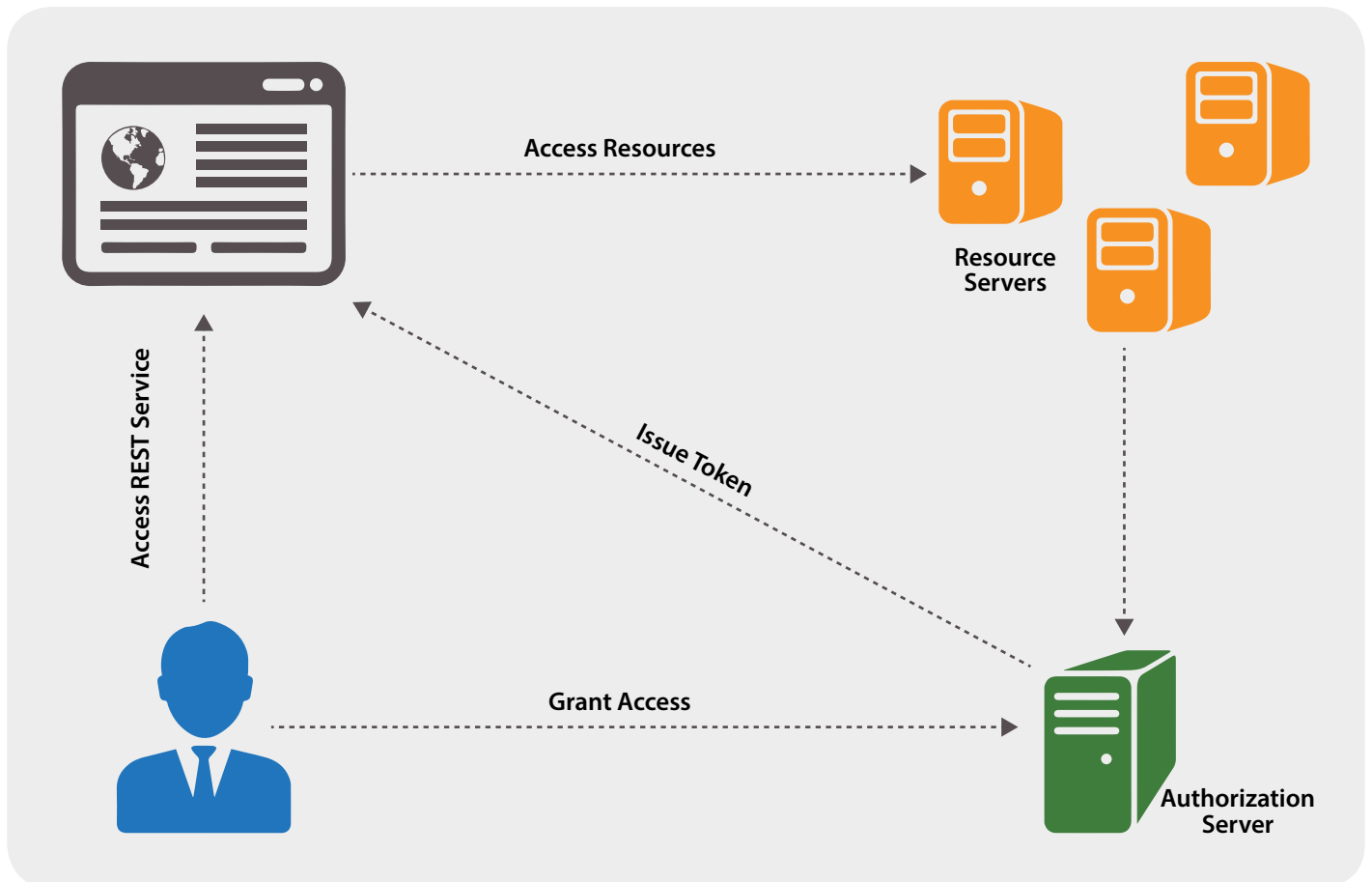
In Digest Authentication, the client sends a digest of the credentials to the server. By default, clients use MD5 to compute the digest. It is better than Basic Authentication.

Applications that demand high security, should implement a custom authentication scheme. This scheme uses an HMAC (custom Hash Message Authentication Code) approach, where the server passes the client a user-ID and a secret key.

This secret key can then be used for all further sign-in requests.

We should also follow an Application Security principle, like validating all inputs on the server. It would be good if we validate TOP 10 OWASP security requirements, and log all the suspicious activities.

Currently, OAuth is widely used for authentication. OAuth (<http://oauth.net>) is a delegated authorization protocol, which enables services and applications to interact with resources hosted securely in third-party services, without requiring the owners of those resources to share their credentials.





## Versioning

Versioning should be considered when the servers are unable to maintain compatibility. It can also be considered when the client needs a special behavior with respect to other clients.

It is important to be careful while doing versioning as it may require code changes at the client-side. You also have to maintain a code base at the server-side for each version.

Versioning can be accomplished via a version number in the URI itself, where the client indicates the version of a resource they need directly in the URL. Facebook and Google use the URL versioning technique.

A few examples of URL versioning:

```
http://service/v1/part/123
http://service/v2/part/123
http://service/part/123?version=v3
```

Some applications prefer using Accept and Content-Type with version identifiers, instead of using version identifiers in URIs. Content Type header is used to define a request and response body format (from both client and server-side) and Accept header is used to define supported media type by clients:

```
# Request
GET http://service/parts/123
Accept: application/json; version=1
```

```
# Response
HTTP/1.1 200 OK
Content-Type:
application/json; version=1
{"partId":"123","name":"Engine"}
```

Now, to retrieve version 2 of the same resource in JSON format:

```
# Request
GET http://service/parts/123
Accept: application/json; version=2
```

```
# Response
HTTP/1.1 200 OK
Content-Type:
application/json; version=2
{"partId":"123",
"name":"Engine","type":"Diesel"}
```

Now the client requires an XML representation with the Accept header that would be set to 'application/xml' along with the required version:

```
# Request
GET http://service/parts/123
Accept: application/json; version=1,
application/xml; version=1
```

The above request assumes that the server supports one or both of the requested types. In the response below, the server favors application/xml:

```
# Response
HTTP/1.1 200 OK
Content-Type:
application/xml; version=1
<part>
  <partId>123</partId>
  <name> Engine </name>
</part>
```

Here, the same URI is being used, with the Accept header to indicate the format of the required response.

## Caching

HTTP provides a built-in caching framework. Therefore, as long as you are using HTTP as defined, you should be able to add a caching layer without making any code changes. Caching can be established on the client or the server-side, and a proxy server can be placed between them.

Header parameters are defined below to control caching:

Consider setting expiration caching headers for responses of GET and HEAD requests for all successful response codes. Although POST is cacheable, caches consider this method as non-cacheable. Also, consider adding caching headers to the 3xx and 4xx response codes. This will help reduce the amount of error-triggering traffic from clients. This is called negative caching.

Avoid implementing caching layer at the client-side because it will make the client slower and client-side caching implementation could lead to security vulnerabilities. Instead, place a forward proxy cache between your clients and the servers. This does not involve any development activity and you get the benefits of a well-tested and robust caching infrastructure.

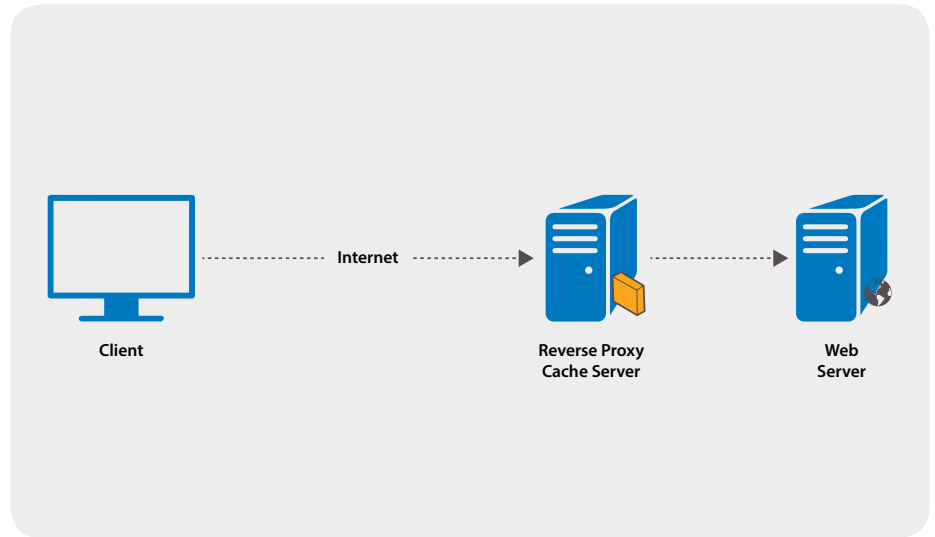
There is also the possibility to reverse proxy cache server at the server-side. The advantage of implementing a cache proxy server is that you can share the

Header	Parameter Meaning
Last Modified	This parameter gives the Date and Time when the server last updated the representation.
Cache-Control	This is used for HTTP 1.1 header to control caching.
Date	Date and time when this representation was initially generated.
Expires	Date and time when representation will expire. (HTTP 1.0 clients)
Age	Total time in seconds since the representation was retrieved from the server.

cache generated by a client with any other client on the planet performing the same request.

To keep the cache always fresh and updated, synchronize its expiry with the frequency of updates. Additionally, implement background processes to watch for database updates and schedule GET requests to refresh caches.

Try to keep static contents like images, CSS, JS cacheable, with expiration date of 1–3 days, and never keep expiry date too high. Dynamic content should only be cached for 1–4 days.



## Application integration using REST and a perfect use case for designing RESTful web services in the right manner

These days, REST is used everywhere – from desktops to mobiles and even in the likes of Facebook, Google, and Amazon. REST provides a lighter-weight alternative for application integration. The REST architecture allows working in a variety of scenarios and it is very useful in cloud and mobile development.

Here, a real-time example is provided for creating RESTful web service for a complex system. This example is about Banking

Account Application, and presents the number of operations that are possible while working with a banking application.

- createAccountHolderProfile
- getAccountHolderProfile
- updateAccountHolderProfile
- doLogin
- doLogout
- getAccountSummary
- getLoanAccounts
- getAllAccounts
- billPayment
- cancelPayment
- completePayment
- fundTransfer
- addPayee
- updatePayee
- deletePayee
- createFixedDeposit
- preCloserFixedDeposit

The second step would be to design the URLs, mapped with the business operations:

RESTful URL	HTTP Action	Noun	Business Operation
/Accounts/Profiles/;<profileData>	POST	Profile	createAccountHolderProfile
/Accounts/Profiles/{profile_id}	GET	Profile	getAccountHolderProfile
/Accounts/Profiles/{profile_id};< profileData>	PUT	Profile	updateAccountHolderProfile
/Accounts/{acc_id}	GET	Account	getAccountSummary
/Accounts/Loans/	GET	Loan	getLoanAccounts
/Accounts/	GET	Account	getAllAccounts
/Accounts/Bills/;<BillData>	POST	BILL	billPayment
/Accounts/Payments/{paymentId}	DELETE	Payment	cancelPayment
/Accounts/Payees/ ;<payee data>	POST	Payee	addPayee
/Accounts/Payees/{payee_id};<payee data>	PUT	Payee	updatePayee
/Accounts/Payee/{payee_id}	DELETE	Payee	deletePayee
/Accounts/fd;<FD Data>	POST	FD	createFixedDeposit
/Accounts/fd{fd_id};<FD Data>	PUT	FD	preCloserFixedDeposit

As a first step towards creating RESTful interface, identify nouns out of the application requirements:

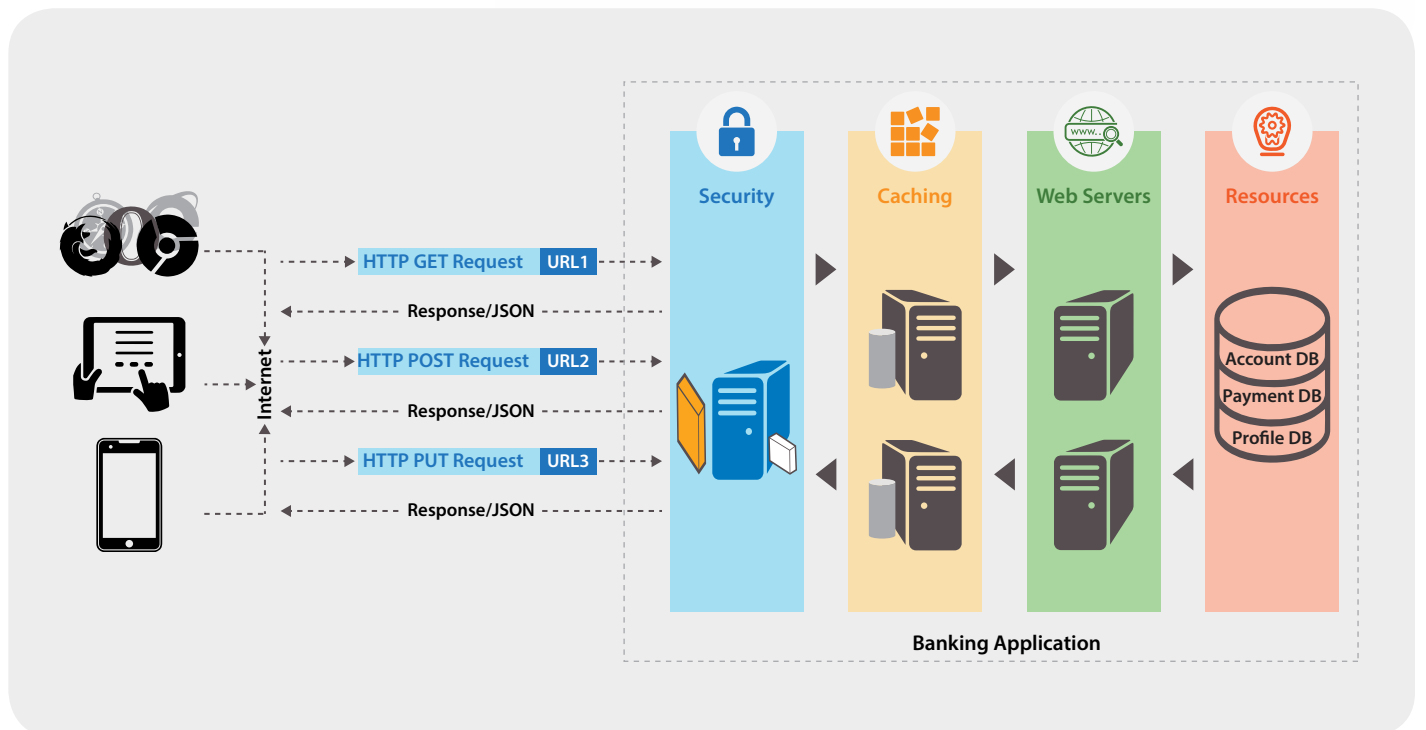
- Account
- Profile
- Bill
- Loan
- Payee
- Fund
- Fixed Deposit
- Payment

As all browsers support JSON, and it is very lightweight, we are going to use it as a representation format.

For caching, we will use a caching proxy server to present frequently accessed information to users. For example – interest rates and policies.

HTTPS is going to be used for communication, which means that our transport layer is encrypted. In addition, a Token Based Authentication will be used to secure the applications.

In the case of an exception at the client-side or server-side, include a link of the error page, which contains the detailed error description.



## Conclusion

By designing web services through adopting RESTful guidelines and best practices, your application can best utilize the in-built features of a web platform and the HTTP protocol. REST provides a superb way to implement services with inherited features such as uniform interface and caching. Developers can enhance productivity and develop loosely coupled web services by adopting the best REST practices.

## About the Author



### Deepak Kumar

*Senior Technology Architect, Infosys Digital*

Deepak Kumar is the Senior Technology Architect with Infosys Digital group. He has 11+ years of IT industry experience in Software Design and Development. He has good hands-on experience in designing SharePoint™, Microsoft .NET based CMS, .NET, and J2EE applications. His strength lies in the ability to handle Multi-tier Application Design, Content Management, Internet / UI based applications, Object Oriented Design, and Component Based Design.

You can reach out to him at [deepak\\_kumar14@infosys.com](mailto:deepak_kumar14@infosys.com)

For more information, contact [askus@infosys.com](mailto:askus@infosys.com)



© 2017 Infosys Limited, Bengaluru, India. All Rights Reserved. Infosys believes the information in this document is accurate as of its publication date; such information is subject to change without notice. Infosys acknowledges the proprietary rights of other companies to the trademarks, product names and such other intellectual property rights mentioned in this document. Except as expressly permitted, neither this documentation nor any part of it may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, printing, photocopying, recording or otherwise, without the prior permission of Infosys Limited and/ or any named intellectual property rights holders under this document.