REST Architectural Constraints

REST defines 6 architectural constraints that make any web service – a true RESTful API.

- Client-Server
- Stateless
- Cacheable
- Uniform Interface
- Layered System
- Code On Demand (Optional)

Client-Server

This constraint keeps the client and server loosely coupled. In this case, the client does not need to know the implementation details in the server, and the server is not worried about how the data is used by the client. However, a common interface is maintained between the client and server to ease communication. This constraint is based on the principle of Separation of concerns.

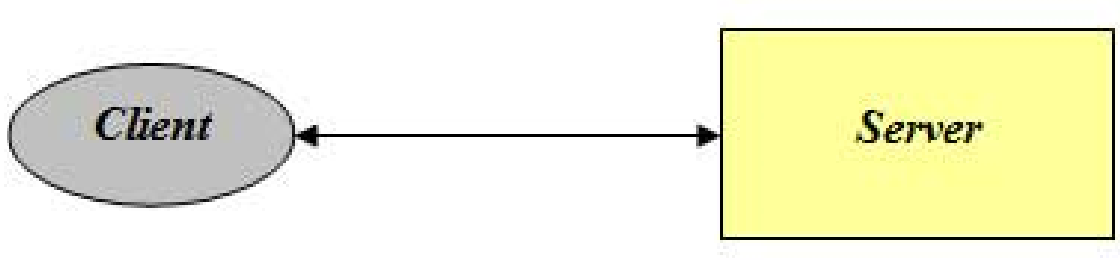

Figure: Client-Server

Applying separation of concerns:

- Separates user interface concerns from data storage concerns.
- Improves portability of interface across multiple platforms.
- Improves scalability by simplifying server components.
- Allows the components to evolve independently.

Stateless

There should be no need for the service to keep user sessions. In other words, the constraint says that the server should not remember the state of the application. As a consequence, the client should send all information necessary for execution along with

each request, because the server cannot reuse information from previous requests as it didn't memorize them. All info needed is in the message.

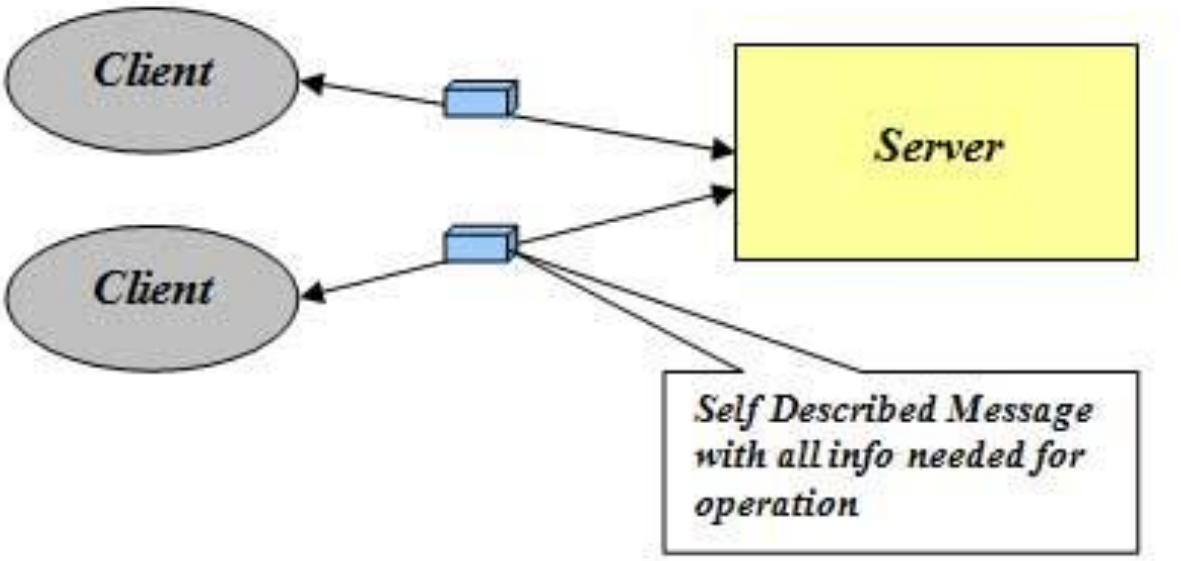

Figure: Stateless

By applying statelessness constraint:

- Session state is kept entirely on the client.
- Visibility is improved since a monitoring system does not have to look beyond a single request.
- Reliability is improved due to easier recoverability from partial failures.
- Scalability is improved due to not having to allocate resources for storing state
- The server does not have to manage resource usage across requests.

Cacheable

This constraint has to support a caching system. The network infrastructure should support a cache at different levels. Caching can avoid repeated round trips between the client and the server for retrieving the same resource.

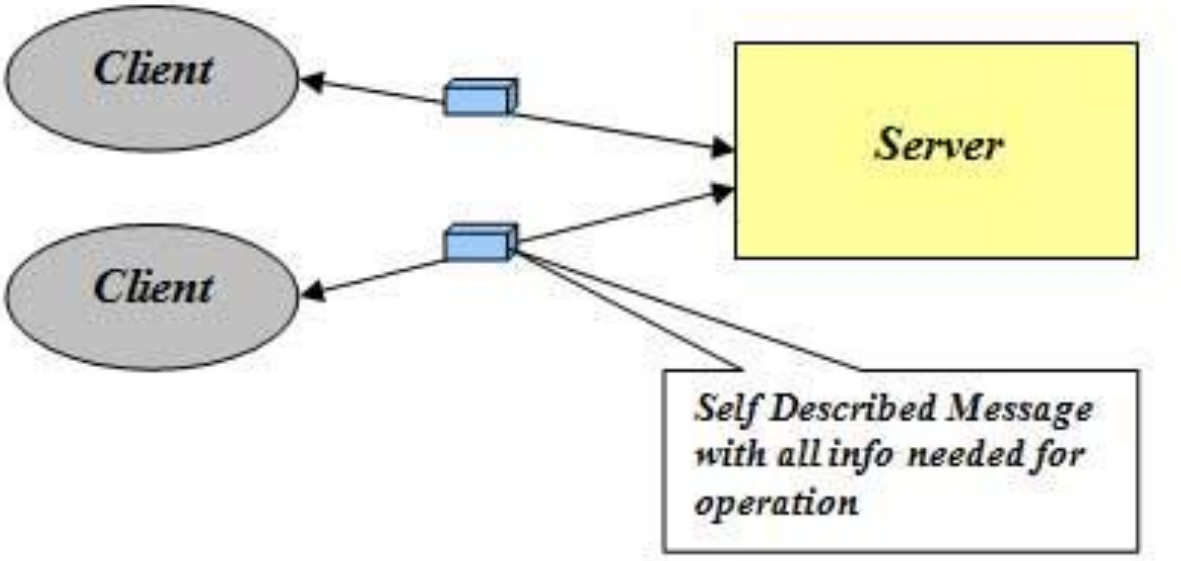

Figure: Cacheable

By adding optional non-shared caching:

- Data within a response to a request is implicitly or explicitly labeled as cacheable or non-cacheable.
- If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.
- Improves efficiency, scalability, and user-perceived performance.
- Tradeoff: cacheable constraint reduces Reliability.

Uniform Interface

This constraint indicates a generic interface to manage all the interactions between the client and server in a unified way, which simplifies and decouples the architecture. This constraint indicates that each resource exposed for use by the client must have a unique address and should be accessible through a generic interface. The client can act on the resources by using a generic set of methods.

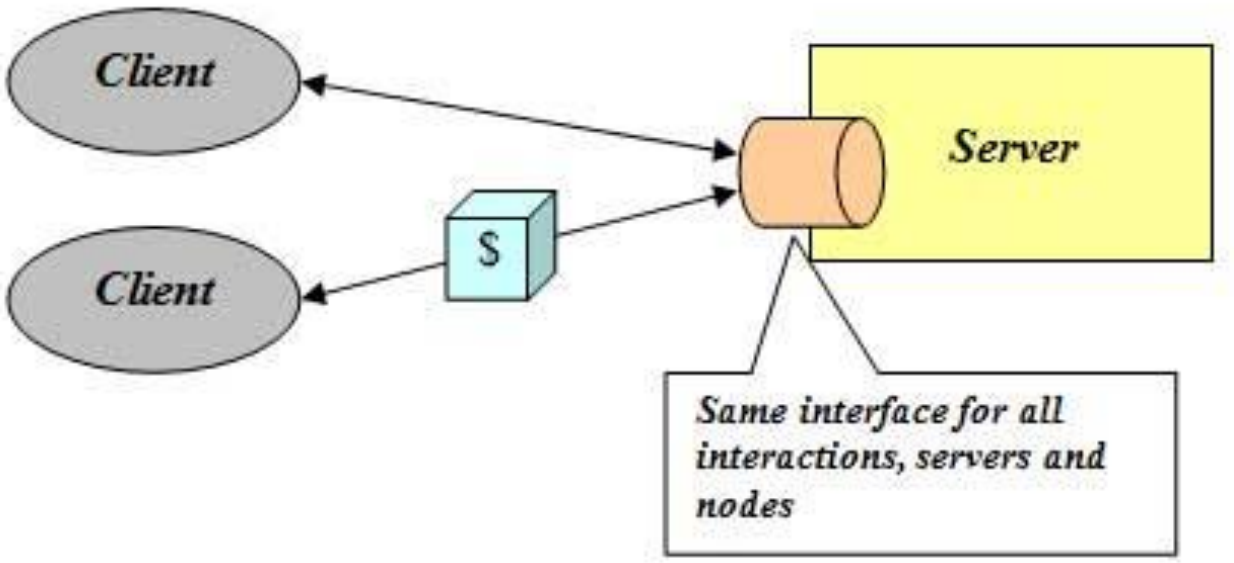

Figure: Uniform Interface

By applying uniform interface constraint:
The overall system architecture is simplified and the visibility of interactions is improved. Implementations are decoupled from the services they provide and encourage independent evolvability.

Trade-off: Degrades efficiency since information is transferred in a standardized form rather than one which is specific to the application's needs. Further, a uniform interface has four sub-constraints

- Identification of resources
- Manipulation of resources through representations
- Self-descriptive messages
- Hypermedia as the engine of application state (HATEOAS)

Identification of resources

Each resource must have a specific and cohesive URI to be made available. REST APIs are designed around resources, which are any kind of object, data, or service that can be accessed by the client.

A resource has an identifier, which is a URI that uniquely identifies that resource.

For example, the URI for a particular customer order might be:

```
http://adventure-works.com/orders/1
```

Resource representation – This is how the resource will return to the client. This representation can be in HTML, XML, JSON, TXT, and more. Clients interact with a service by exchanging representations of resources. Many web APIs use JSON as the exchange format. For example, a GET request to the URI listed above might return this response body:

```
{"orderId":1,"orderValue":99.90,"productId":1,"quantity":1}
```

Self-descriptive Messages

Each message includes enough information to describe how to process the message. Beyond what we have seen so far, the passage of meta information is needed (metadata) in the request and response. Some of this information are: HTTP response code, Host, Content-Type etc. Taking as an example the same URI as we have just seen:

```
GET /#!/user/ramesh HTTP/1.1
User-Agent: Chrome/37.0.2062.94
Accept: application/json
Host: exampledomain.com
```

Hypermedia as the Engine of Application State (HATEOAS)

REST APIs are driven by hypermedia links that are contained in the representation. For example, the following shows a JSON representation of an order. It contains links to get or update the customer associated with the order.

Just one example:

```
{
  "orderID":3,
  "productID":2,
  "quantity":4,
  "orderValue":16.60,
  "links": [
    {"rel":"product","href":"http://adventure-works.com/customers/3", "action":"GET" },
    {"rel":"product","href":"http://adventure-works.com/customers/3", "action":"PUT" }
  ]
}
```

Layered System

The server can have multiple layers for implementation. This layered architecture helps to improve scalability by enabling load balancing. It also improves the performance by providing shared caches at different levels.

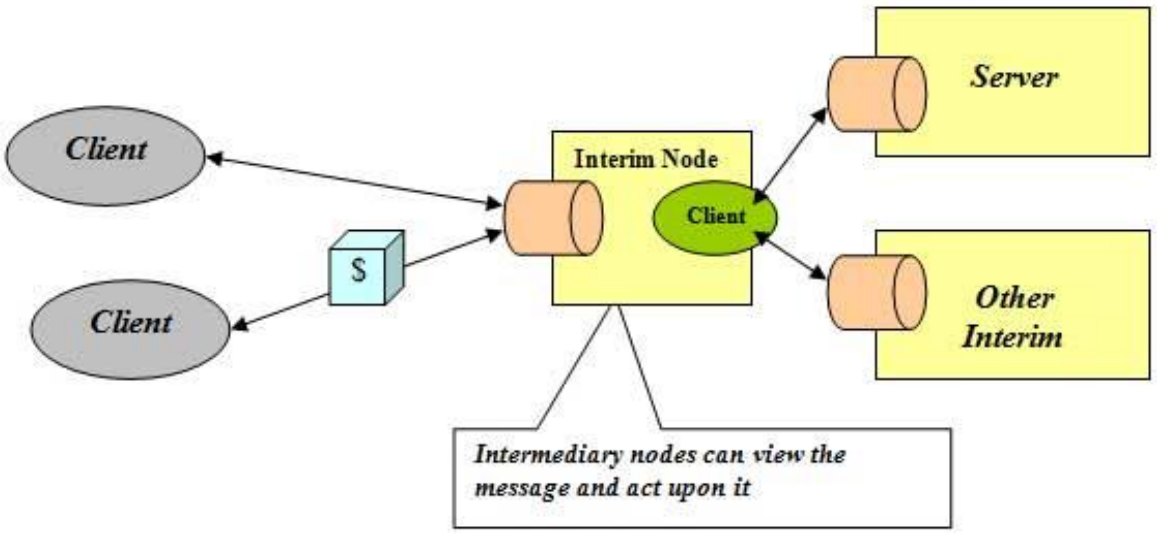

Figure: Layered System

By applying a layered system constraint:

- Similar to the client-server constraint this constraint improves simplicity by separating concerns.
- Can be used to encapsulate legacy services or protect new services from legacy clients.
- Intermediaries can be used to improve system scalability by enabling load balancing
- Placing shared caches at the boundaries of the organizational domain can result in significant benefits. Can also enforce security policies e.g. firewall.
- Intermediaries can actively transform message content since messages are self-descriptive and their semantics are visible to the intermediaries Tradeoff: Adds overhead and latency and reduce user-perceived performance.

Code on Demand

This constraint is optional. This constraint indicates that the functionality of the client applications can be extended at runtime by allowing a code download from the server and executing the code. Some examples are the applets and the JavaScript code that get transferred and executed at the client-side at runtime.

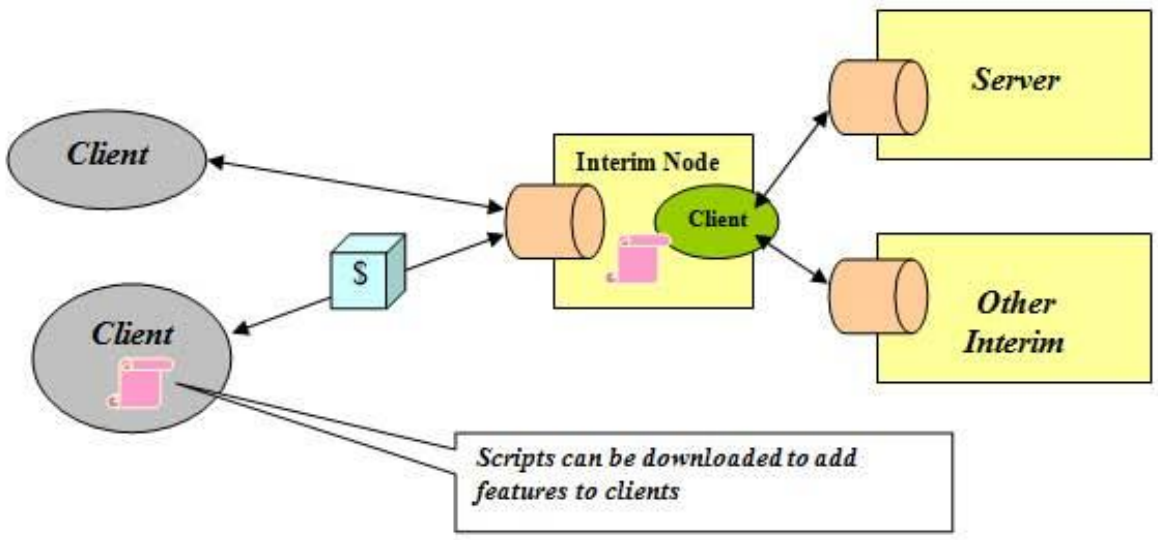

Figure: Code on demand

By applying Code on demand constraint:

- Simplifies clients, hence promote the reduced coupling of features.
- Improves scalability by virtue of the server off-loading work onto the clients.
- Trade-off: Reduces visibility generated by the code itself, which is hard for an intermediary to interpret.

How to Assign HTTP methods to REST Resources

GET

The HTTP GET method is used to 'read '(or retrieve) a representation of a resource. In the "happy" (or non-error) path, GET returns a representation in XML or JSON and an HTTP response code of 200 (OK). In an error case, it most often returns a 404 (NOT FOUND) or 400 (BAD REQUEST).

According to the design of the HTTP specification, GET (along with HEAD) requests are used only to read data and not change it. Therefore, when used this way, they are considered safe. That is, they can be called without risk of data modification or corruption—calling it once has the same effect as calling it 10 times or none at all. Additionally, GET (and HEAD) is idempotent, which means that making multiple identical

requests ends up having the same result as a single request.

Do not expose unsafe operations via GET—it should never modify any resources on the server.
Examples:

- GET *http://www.example.com/customers/12345*
- GET *http://www.example.com/customers/12345/orders*
- GET *http://www.example.com/buckets/sample*

POST

The POST verb is most-often utilized to 'create' new resources. In particular, it's used to create subordinate resources. That is subordinate to some other (e.g. parent) resource. In other words, when creating a new resource, POST to the parent and the service takes care of associating the new resource with the parent, assigning an ID (new resource URI), etc.

On successful creation, return HTTP status 201, returning a Location header with a link to the newly-created resource with the 201 HTTP status.

The POST is neither safe nor idempotent. It is therefore recommended for non-idempotent resource requests. Making two identical POST requests will most-likely result in two resources containing the same information.

Examples:

- POST-http://www.example.com/customers
- POST-http://www.example.com/customers/12345/orders

PUT

PUT is most-often utilized for **update** capabilities, PUT-ing to a known resource URI with the request body containing the newly-updated representation of the original resource.

However, *PUT* can also be used to create a resource in the case where the resource ID is chosen by the client instead of by the server. In other words, if the PUT is to a URI that contains the value of a non-existent resource ID. Again, the request body contains a resource representation. Many feel this is convoluted and confusing. Consequently, this method of creation should be used sparingly, if at all.

On successful update, return 200 (or 204 if not returning any content in the body) from a PUT. If using PUT for creating, return HTTP status 201 on successful creation. A body in the response is optional—providing one consumes more bandwidth. It is not necessary to

return a link via a Location header in the creation case since the client already set the resource ID.

Examples:

- *PUT http://www.example.com/customers/12345*
- *PUT http://www.example.com/customers/12345/orders/98765*
- *PUT http://www.example.com/buckets/secret_stuff*

DELETE

DELETE is pretty easy to understand. It is used to **delete** a resource identified by a URI. On successful deletion, return HTTP status 200 (OK) along with a response body, perhaps the representation of the deleted item (often demands too much bandwidth), or a wrapped response (see Return Values below). Either that or return HTTP status 204 (NO CONTENT) with no response body. In other words, a 204 status with nobody, or the JSEND-style response and HTTP status 200 are the recommended responses.

Examples:

- DELETE *http://www.example.com/customers/12345*
- DELETE *http://www.example.com/customers/12345/orders*
- DELETE *http://www.example.com/bucket/sample*

PATCH

PATCH is used for **modify** capabilities. The PATCH request only needs to contain the changes to the resource, not the complete resource.

This resembles PUT, but the body contains a set of instructions describing how a resource currently residing on the server should be modified to produce a new version. This means that the PATCH body should not just be a modified part of the resource but in some kind of patch languages like JSON Patch or XML Patch.

Examples:

- PATCH http://www.example.com/customers/12345
- PATCH http://www.example.com/customers/12345/orders/98765
- PATCH http://www.example.com/buckets/secret_stuff

How to Model JSON Representation Format

This is the fourth step in *Design Restful API*. In this post, we will learn how to model *JSON* representation. There are many representations but in this post, we mainly focus on popularly used *JSON* representation.
Most the time we use *JSON* as the format for displaying a resource. Let's understand what is naming convention for JSON field names

JSON Field name

It is recommended to use "***lower camel case***" as the JSON field name. It is recommended on considering its compatibility with JavaScript, which is assumed as one of the client applications. The JSON sample with field name set in "lower camel case", is as given below. In "lower camel case", the first letter of the word is in lowercase and subsequent first letters of words are in uppercase.

```
{
  "memberId" : "M000000001"
}
```

NULL and blank characters

It is recommended to differentiate *NULL* and blank characters as *JSON* values. Although, as the application process, NULL and blank characters are often equated, it is advisable to differentiate NULL and blank characters as the value to be set in JSON.
Example: JSON sample wherein NULL and blank characters are differentiated.

```
{
  "dateOfBirth" : null,
  "address1" : ""
}
```

Date format

It is recommended to use the extended *ISO-8601* format as the *JSON* date field format. Format other than extended ISO-8601 format can be used. However, it is advisable to use the extended ISO-8601 format, if there is no particular reason otherwise. There are two formats in ISO-8601 namely, basic format and extended format, however, readability is higher in an extended format. Basically, there are the following three formats.

1. **yyyy-MM-dd**

```
{
  "dateOfBirth" : "1977-03-12"
}
```

2. yyy-MM-dd'T'HH:mm:ss.SSSZ

```
{
  "lastModifiedAt" : "2014-03-12T22:22:36.637+09:00"
}
```

3. **yyyy-MM-dd'T'HH:mm:ss.SSS'Z' (format for UTC)**

```
{
  "lastModifiedAt" : "2014-03-12T13:11:27.356Z"
}
```

Hypermedia link format

It is recommended to use the following format to create a *`hypermedia link`*.

A sample of recommended format is as given below.

```
{
  "links" : [
    {
      "rel" : "ownerMember",
      "href" : "http://example.com/api/v1/memebers/M000000001"
    }
  ]
}
```

- Link object consisting of 2 fields - "*`rel`*" and "*`href`*" is retained in collection format.
- Link name for identifying the link is specified in "rel".
- URI to access the resource is specified in "href".
- "links" is the field which retains the Link object in collection format.

The format at the time of error response

You can follow below is an example of the response format when an error is detected.

```
{
 "code" : "e.ex.fw.7001",
 "message" : "Validation error occurred on item in the request body.",
 "details" : [ {
  "code" : "ExistInCodeList",
  "message" : "\"genderCode\" must exist in code list of CL_GENDER.",
  "target" : "genderCode"
 } ]
}
```


In the above example,

- Error code (code)
- Error message (message)
- Error details list (details) are provided as error response formats. It is assumed that the error details list is used when input validation error occurs. It is a format that can retain details like the field in which the error occurred and the information of the error.

Sample JSON Representation


```
{
 "links" : [ {
  "rel" : "self",
  "href" : "http://localhost:8080/rest-api-web/api/v1/members/M000000001"
 } ],
 "memberId" : "M000000001",
 "firstName" : "John",
 "lastName" : "Smith",
 "genderCode" : "1",
 "dateOfBirth" : "2013-03-14",
 "emailAddress" : "user1394794959984@test.com",
 "telephoneNumber" : "09012345678",
 "zipCode" : "1710051",
 "address" : "Tokyo",
 "credential" : {
  "signId" : "user1394794959984@test.com",
  "passwordLastChangedAt" : "2014-03-14T11:02:41.477Z",
  "lastModifiedAt" : "2014-03-14T11:02:41.477Z"
 },
 "createdAt" : "2014-03-14T11:02:41.477Z",
 "lastModifiedAt" : "2014-03-14T11:02:41.477Z"
}
```