1. What is REST?

The REST stands for REpresentational State Transfer.

-**State** means data

-**REpresentational** means formats (such as XML, JSON, YAML, HTML, etc)

-**Transfer** means carry data between consumer and provider using HTTP protocol

REST is not an architecture; rather, it is a set of constraints that creates a software architectural style, which can be used for building distributed applications.

The REST (REpresentational State Transfer) is designed to take advantage of existing HTTP protocols when used for Web APIs. It is very flexible in that it is not tied to resources or methods and has the ability to handle different calls and data formats. Because REST API is not constrained to an XML format like SOAP, it can return multiple other formats depending on what is needed. If a service adheres to this style, it is considered a "RESTful" application. REST allows components to access and manage functions within another application.

Like any other architectural style, REST also does have it's own six guiding constraints which must be satisfied if an interface needs to be referred to as RESTful.

Let's briefly look into six REST architectural constraints:

If you follow all constraints designed by the REST architectural style your system is considered RESTful.

1. 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.

2. Stateless

The notion of statelessness is defined from the perspective of the server. 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.

3. 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.

4. Uniform Interface

The uniform interface constraint at a high level means the interface for a component needs to be as generic as possible. It simplifies and decouples the architecture, which enables each part of the architecture to evolve independently.

5. 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. Being the door to the system,Â the top layer can enforce security policies as well.

6. Code on Demand

The **code on demand** is an optional constraint. It allows a client to download and execute code from a server.

2. What is a RESTful API?

A RESTful API (or application program interface) uses HTTP requests to GET, PUT, POST, and DELETE data following the REST standards. This allows two pieces of software to communicate with each other. In essence, REST API is a set of remote calls using standard methods to return data in a specific format.

The systems that interact in this manner can be very different. Each app may use a unique programming language, operating system, database, etc. So, how do we create a system that can easily communicate and understand other apps?? This is where the Rest API is used as an interaction system.

Let's understand some terms which will help you to design and develop REST APIs.

Resources

A RESTful resource is anything that is addressable over the web. By addressable, we mean resources that can be accessed and transferred between clients and servers.

Here are some examples of REST resources:

- An employee, department, projects in IT company
- A student in a classroom in a school
- A search result for a particular item in a web index, such as Google
- Users of the system.
- Courses in which a student is enrolled.
- A user's timeline of posts.
- The users that follow another user.
- An article about programming.

REST architecture treats every content as a resource. These resources can be Text Files, Html Pages, Images, Videos or Dynamic Business Data. REST Server simply provides access to resources and REST client accesses and modifies the resources. Here each resource is identified by URIs/ global IDs. REST uses various representations to represent a resource were Text, JSON, XML. The most popular representations of resources are XML and JSON.

URI

A URI (**Uniform resource identifier**) is a string of characters used to identify a resource over the web. In simple words, the URI in a RESTful web service is a hyperlink to a resource, and it is the only means for clients and servers to exchange representations.

The client uses a URI to locate the resources over the web, sends a request to the server, and reads the response.

For example:

Method	**Url**	**Description**
GET	**/api/todos**	Get all todos which belongs to logged in user

Method	Url	Description
GET	**/api/todos/{id}**	Get todo by id (If todo belongs to logged in user)
POST	**/api/todos**	Create new todo (By logged in user)
PUT	**/api/todos/{id}**	Update todo (If todo belongs to logged in user)
DELETE	**/api/todos/{id}**	Delete todo (If todo belongs to logged in user)

The above shows the URI's for **Todo** resource on the web.

The representation of resources

The representation of resources is what is sent back and forth between clients and servers in a RESTful system. A representation is a temporal state of the actual data located in a storage device at the time of request.

Different clients can consume different representations of the same resource. Therefore, a representation can take various forms, such as an image, a text file, an XML, or a JSON format.

For example: Below is the resource JSON representation:

Sign Up -> /api/auth/signup

```
{
        "firstName": "ramesh",
        "lastName": "fadatare",
        "username": "ramesh",
        "password": "password",
        "email": "javaguides@gmail.com"
}
```

Log In -> /api/auth/signin

```
{
        "usernameOrEmail": "ramesh",
        "password": "password"
}
```

3. HTTP Methods

RESTful web service makes heavy uses of HTTP verbs to determine the operation to be carried out on the specified resource(s). REST APIs enable you to develop any kind of web application having all possible CRUD (create, retrieve, update, delete) operations.

Let's learn about HTTP methods and their role in client-server communication over HTTP.

HTTP GET Method

Use GET requests to retrieve resource representation/information only – and not to modify it in any way. As GET requests do not change the state of the resource, these are said to be safe methods. Additionally, GET APIs should be idempotent, which means that making multiple identical requests must produce the same result every time until another API (POST or PUT) has changed the state of the resource on the server.

If the address is successfully received (error-free), GET returns a JSON or XML representation in combination with the HTTP status code 200 (OK). In case of an error, the code 404 (NOT FOUND) or 400 (BAD REQUEST) is usually returned.

Examples for GET request URI's:

HTTP GET - **http://www.usermanagement/api/users/me** - Get logged in user profile

HTTP GET - **http://www.usermanagement/api/users/{username}/profile** - Get user profile by username

HTTP GET - **http://www.usermanagement/api/users/{username}/posts** - Get posts created by user

HTTP GET - **http://www.usermanagement/api/users/{username}/albums** - Get albums created by user

http://www.usermanagement/api/users/checkUsernameAvailability - Check if username is available to register

http://www.usermanagement/api/users/checkEmailAvailability - Check if email is available to register

HTTP POST Method

The HTTP POST request is most commonly used to create new resources. When talking strictly in terms of REST, POST methods are used to create a new resource into the collection of resources.

Upon successful creation, an HTTP code 201 is returned, and the address of the created resource is also transmitted in the 'Location' header.

Here are some examples for HTTP POST requests:

HTTP POST - **http://www.domain/api/users** - Create User

HTTP POST - **http://www.domain/api/posts** - Create Post

HTTP POST - **http://www.domain/api/posts/{postId}/comments** - Create new comment for post with id = postId

HTTP PUT Method

Use PUT APIs primarily to update existing resource (if the resource does not exist, then API may decide to create a new resource or not). If a new resource has been created by the PUT API, the origin server MUST inform the user agent via the HTTP response code 201 (Created) response and if an existing resource is modified, either the 200 (OK) or 204 (No Content) response codes SHOULD be sent to indicate successful completion of the request.

Here are some examples for HTTP Put requests:

HTTP PUT - **http://www.domain/api/users/{username} -** Update user

HTTP PUT - **http://www.domain/api/posts/{id}** - Update post by id

HTTP PUT - **http://www.domain/api/posts/{postId}/comments/{id}** - Update comment by id if it belongs to post with id = postId

HTTP DELETE Method

Use DELETE APIs to delete resources (identified by the Request-URI).

A successful response of DELETE requests SHOULD be HTTP response code `200 (OK)` if the response includes an entity describing the status, 202 (Accepted) if the action has been queued, or 204 (No Content) if the action has been performed but the response does not include an entity.

Here are some examples for HTTP Delete requests:

DELETE **http://www.domain/api/users/{username}** - Delete user

DELETE **http://www.domain/api/posts/{id} -** Delete post

DELETE **http://www.domain/api/posts/{postId}/comments/{id} -** Delete comment by id if it belongs to post with id = postId

4. HTTP Status Codes

For every HTTP request, the server returns a status code indicating the processing status of the request. Let's see the frequently used HTTP status codes:

1xx: Information

Communicates transfer protocol-level information

- 100: Continue

2xx: Success

This indicates that the client’s request was accepted successfully.

- 200: OK
- 201: Created
- 202: Accepted
- 204: No Content

3xx: Redirect

This indicates that the client must take some additional action in order to complete their request.

- 301: Moved Permanently
- 307: Temporary Redirect

4xx: Client Error

This category of error status codes points the finger at clients.

- 400: Bad Request
- 401: Unauthorized
- 403: Forbidden
- 404: Not found

5xx Server Error

The server takes responsibility for these error status codes.

- 500: Internal Server Error
- 501: Not Implemented
- 502: Bad Gateway
- 503: Service Unavailable
- 504: Gateway Timeout

What is Payload in REST API?

In a REST API, a payload refers to the data or information that is sent by the client in a request to the server or the data that is returned by the server in response to a request. The payload typically contains the data that needs to be processed or manipulated by the server, such as a JSON or XML object, or sometimes binary data like images or videos.

For example, when a client sends a POST request to a server to create a new resource, the payload typically contains the data for that resource in JSON or XML format. Similarly, when a client sends a PUT request to update an existing resource, the payload will contain the updated data for that resource.

The payload is an essential part of a REST API as it contains the actual data being transferred between the client and the server. It is important to ensure that the payload is properly formatted and structured according to the API's specifications to ensure proper communication and handling of data.

JSON payload format example:

```
{
    "cid": 1,
    "cname": "Ramesh",
    "email": "ramesh@gmail.com"
}
```

XML payload format example:

```
   <customer>
      <cid>1</cid>
      <cname>Ramesh</cname>
      <email>ramesh@gmail.com</email>
   </customer>
```

Payload Composition

Content:

The payload carries data, which can be structured in various formats such as JSON, XML, or even plain text. In the realm of RESTful APIs, JSON remains the most popular choice due to its lightweight nature and ease of use.

HTTP Headers:

Accompanying the payload are HTTP headers which provide meta-information about the payload, such as its type (Content-Type header, e.g., application/json) and length (Content-Length header).

The **'Content-Type'** header name in the HTTP request message is used to represent the payload format in the HTTP request message. For example JSON, XML, etc.

The **'Accept'** header name in an HTTP request message is used to represent the expected payload format in an HTTP response message. Examples: JSON, XML, plain text, HTML, etc.

Payload in Different HTTP Methods

GET: Typically, GET requests fetch data. While these requests do not usually have a request body (payload), some advanced APIs might use the body to send extensive search or filter criteria.

POST: When creating a new resource, the payload usually carries data for the new entity.

PUT: Used to update a resource, the payload here encompasses the complete updated data.

PATCH: This method is for partial updates, and the payload contains only specific fields meant for modification.

DELETE: While DELETE operations commonly don't have a payload, some APIs expect additional deletion-related data in the request body.

HTTP Request and Response Payload JSON Example

HTTP Request Message Example:

An HTTP request message with a JSON payload is typically structured with:

- A **request line**, which includes the HTTP method, the request URI, and the HTTP version.
- **Headers** that provide meta-information about the request.
- A **blank line** indicating the end of the headers.
- The **message body**, which contains the JSON payload.

Let's take a look at an example. Imagine you are developing a RESTful API for managing a collection of books, and you want to add a new book to your collection.

```
POST /api/books HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0
Accept: application/json
Content-Type: application/json
Content-Length: 112

{
  "title": "The Great Gatsby",
  "author": "F. Scott Fitzgerald",
  "year": 1925,
  "genre": "Fiction"
}
```

Explanation:

Request Line:

- POST is the HTTP method indicating we want to create a new resource.
- `/api/books` is the request URI specifying the endpoint we're targeting.
- HTTP/1.1 is the HTTP version.

Headers:

- **Host:** Specifies the domain name of the server.
- **User-Agent:** Indicates the client's software (e.g., a web browser or other client).
- **Accept:** Tells the server the type of data the client is expecting in response.
- **Content-Type:** Specifies the media type of the data being sent to the server, which in this case is `application/json`.
- **Content-Length:** Indicates the length of the request body.

Blank Line: This separates the headers from the body.

Message Body: Contains the JSON payload that represents the book details.

This is a basic example. In real-world scenarios, there might be additional headers like Authorization for authentication purposes, or other complexities depending on the API's requirements.

HTTP Response Message Example:

Let's consider a scenario in continuation of the previous book example. Suppose the server processed the request to add a new book and now wants to return a confirmation to the client.

HTTP Response Message Example:

```
HTTP/1.1 201 Created
Date: Mon, 15 Aug 2023 12:30:00 GMT
Server: Apache/2.4.1 (Unix)
Content-Type: application/json
Content-Length: 150

{
  "id": 12345,
  "title": "The Great Gatsby",
  "author": "F. Scott Fitzgerald",
  "year": 1925,
  "genre": "Fiction",
  "message": "Book successfully added!"
}
```

Explanation:

Status Line:

- **HTTP/1.1:** The HTTP version.
- **201 Created:** This is the status code and its associated message. A 201 status indicates that the request has been fulfilled, resulting in the creation of a new resource.

Headers:

- **Date:** The date and time at which the message was sent.
- **Server**: Information about the software used by the server to handle the request.
- **Content-Type:** Specifies the media type of the data being returned, in this case, application/json.
- **Content-Length:** Indicates the length of the response body.

Blank Line: This separates the headers from the body.

Message Body: Contains the JSON payload which, in this example, provides details of the added book along with a message confirming the successful addition.

Again, this is a simplistic example. Real-world scenarios might have additional headers or complexities based on what the API and application require.

HTTP Request and Response Payload XML Example

HTTP Request XML Payload Example

It is the responsibility of the consumer means client application to prepare and send HTTP request message as given below:

```
POST /SpringBootREST/customers HTTP/1.1
Accept: application/xml
Content-Type: application/xml
Content-Length: 196
User-Agent: Java/1.7.0_25
Host: 127.0.0.1:7000
Connection: keep-alive

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
   <customer>
      <cid>0</cid>
      <cname>Sumita</cname>
      <email>sumitaspandey@gmail.com</email>
 </customer>
```

From the above HTTP request, the payload means the body in the HTTP request message. It's optional and depends on the HTTP method name i.e.,

- In the case of the GET HTTP method, the HTTP request message is without a body.
- In the case of the POST HTTP method, the HTTP request message with the body

The '**Content-Type**' header name in the above HTTP request message is used to represent the payload format in the HTTP request message and the payload format is `XML`.

The '**Accept**' header name in an HTTP request message is used to represent the expected payload format in an HTTP response message and the payload format is `XML`.

Http Response XML Payload Example

It is the responsibility of the business component (developed by the service provider) to prepare and send HTTP response message as given below:

```
HTTP/1.1 200 OK
Content-Type: application/xml
Server: Apache-Coyote/1.1
Transfer-Encoding: chunked
Date: Mon, 10 Nov 2019 09:45:34 GMT
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
   <customer>
      <cid>1</cid>
      <cname>Sumita</cname>
      <email>sumitaspandey@gmail.com</email>
   </customer>
```

The HTTP response message contains any one of the HTTP status codes ranging between 100 and 599:

Informational: **1xx**

Successful: **2xx**

Redirection: **3xx**

Client-side error: **4xx**

Server-side error: **5xx**

From the above Response message, the payload means the body in the HTTP response message and the payload is in XML format.

HTTP GET Method

Use GET requests to retrieve resource representation/information only – and not to modify it in any way. As GET requests do not change the state of the resource, these are said to be safe methods. Additionally, GET APIs should be idempotent, which means that making multiple identical requests must produce the same result every time until another API (POST or PUT) has changed the state of the resource on the server.

If the address is successfully received (error-free), GET returns a JSON or XML representation in combination with the HTTP status code 200 (OK). In case of an error, the code 404 (NOT FOUND) or 400 (BAD REQUEST) is usually returned.

Examples for GET request URI's:

HTTP GET - **http://www.usermanagement/api/users/me** - Get logged in user profile

HTTP GET - **http://www.usermanagement/api/users/{username}/profile** - Get user profile by username

HTTP GET - **http://www.usermanagement/api/users/{username}/posts** - Get posts created by user

HTTP GET - **http://www.usermanagement/api/users/{username}/albums** - Get albums created by user

http://www.usermanagement/api/users/checkUsernameAvailability - Check if username is available to register

http://www.usermanagement/api/users/checkEmailAvailability - Check if email is available to register

HTTP POST Method

The HTTP POST request is most commonly used to create new resources. When talking strictly in terms of REST, POST methods are used to create a new resource into the collection of resources.

Upon successful creation, an HTTP code 201 is returned, and the address of the created resource is also transmitted in the 'Location' header.

Here are some examples for HTTP POST requests:

HTTP POST - **http://www.domain/api/users** - Create User

HTTP POST - **http://www.domain/api/posts** - Create Post

HTTP POST - **http://www.domain/api/posts/{postId}/comments** - Create new comment for post with id = postId

HTTP PUT Method

Use PUT APIs primarily to update existing resource (if the resource does not exist, then API may decide to create a new resource or not). If a new resource has been created by the PUT API, the origin server MUST inform the user agent via the HTTP response code 201 (Created) response and if an existing resource is modified, either the 200 (OK) or 204 (No Content) response codes SHOULD be sent to indicate successful completion of the request.

Here are some examples for HTTP Put requests:

HTTP PUT - **http://www.domain/api/users/{username} -** Update user

HTTP PUT - **http://www.domain/api/posts/{id}** - Update post by id

HTTP PUT - **http://www.domain/api/posts/{postId}/comments/{id}** - Update comment by id if it belongs to post with id = postId

The difference between the POST and PUT APIs can be observed in request URIs. POST requests are made on resource collections, whereas PUT requests are made on a single resource.

HTTP DELETE Method

Use DELETE APIs to delete resources (identified by the Request-URI).

A successful response of DELETE requests SHOULD be HTTP response code `200 (OK)` if the response includes an entity describing the status, 202 (Accepted) if the action has been queued, or 204 (No Content) if the action has been performed but the response does not include an entity.

Here are some examples for HTTP Delete requests:

DELETE **http://www.domain/api/users/{username}** - Delete user

DELETE **http://www.domain/api/posts/{id} -** Delete post

DELETE **http://www.domain/api/posts/{postId}/comments/{id} -** Delete comment by id if it belongs to post with id = postId

HTTP PATCH Method

HTTP PATCH requests are to make partial updates on a resource. If you see PUT requests also modify a resource entity, so to make more clear – the PATCH method is the correct choice for partially updating an existing resource, and PUT should only be used if you're replacing a resource in its entirety.

Example:

A sample patch request to update the email will be like this:

```
HTTP PATCH /users/1

[

        { “op”: “replace”, “path”: “/email”, “value”: “new.email@example.org” }

]
```

When to Use HTTP PUT and When HTTP PATCH?

When a client needs to replace an existing Resource entirely, they can use PUT. When they're doing a partial update, they can use HTTP PATCH.

For instance, when updating a single field of the Resource, sending the complete Resource representation might be cumbersome and utilizes a lot of unnecessary bandwidth. In such cases, the semantics of PATCH make a lot more sense.

Another important aspect to consider here is idempotence; PUT is idempotent; PATCH can be, but isn't required to. And, so – depending on the semantics of the operation we're implementing, we can also choose one or the other based on this characteristic.

Summary of HTTP Methods

REST API's for **User Management Application -** Following five REST APIs for User resource:

Sr. No.	API Name	HTTP Method	Path	Status Code	Description
(1)	GET Users	GET	`/api/v1/users`	200 (OK)	All User resources are fetched.
(2)	POST User	POST	`/api/v1/users`	201 (Created)	A new User resource is created.
(3)	GET User	GET	`/api/v1/users/{id}`	200 (OK)	One User resource is fetched.
(4)	PUT User	PUT	`/api/v1/users/{id}`	200 (OK)	User resource is updated.
(5)	DELETE User	DELETE	`/api/v1/users/{id}`	204 (No Content)	User resource is deleted.

HTTP Status Codes

For every HTTP request, the server returns a status code indicating the processing status of the request. In this article, we will see some of the frequently used HTTP status codes. A basic understanding of status codes will definitely help us later while designing RESTful web services:

1xx Informational

This series of status codes indicates informational content. This means that the request is received and processing is going on. Here are the frequently used informational status codes:

100 Continue:

This code indicates that the server has received the request header and the client can now send the body content. In this case, the client first makes a request (with the Expect: 100-continue header) to check whether it can start with a partial request. The server can then respond either with 100 Continue (OK) or 417 Expectation Failed (No) along with an appropriate reason.

101 Switching Protocols:

This code indicates that the server is OK for a protocol switch request from the client.

102 Processing:

This code is an informational status code used for long-running processing to prevent the client from timing out. This tells the client to wait for the future response, which will have the actual response body.

2xx Success

This series of status codes indicates the successful processing of requests. Some of the frequently used status codes in this class are as follows:

200 OK:

This code indicates that the request is successful and the response content is returned to the client as appropriate.

201 Created:

This code indicates that the request is successful and a new resource is created.

204 No Content:

This code indicates that the request is processed successfully, but there's no return value for this request. For instance, you may find such status codes in response to the deletion of a resource.

3xx Redirection

This series of status codes indicates that the client needs to perform further actions to logically end the request. A frequently used status code in this class is as follows:

304 Not Modified:

This status indicates that the resource has not been modified since it was last accessed. This code is returned only when allowed by the client via setting the request headers as If-Modified-Since or If-None-Match. The client can take appropriate action on the basis of this status code.

4xx Client Error

This series of status codes indicates an error in processing the request. Some of the frequently used status codes in this class are as follows:

400 Bad Request:

This code indicates that the server failed to process the request because of the malformed syntax in the request. The client can try again after correcting the request.

401 Unauthorized:

This code indicates that authentication is required for the resource. The client can try again with appropriate authentication.

403 Forbidden:

This code indicates that the server is refusing to respond to the request even if the request is valid. The reason will be listed in the body content if the request is not a HEAD method.

404 Not Found:

This code indicates that the requested resource is not found at the location specified in the request.

405 Method Not Allowed:

This code indicates that the HTTP method specified in the request is not allowed on the resource identified by the URI.

408 Request Timeout:

This code indicates that the client failed to respond within the time window set on the server.

409 Conflict:

This code indicates that the request cannot be completed because it conflicts with some rules established on resources, such as validation failure.

5xx Server Error

This series of status codes indicates server failures while processing a valid request. Here is one of the frequently used status codes in this class:

500 Internal Server Error:

This code indicates a generic error message, and it tells that an unexpected error occurred on the server and that the request cannot be fulfilled.

501 (Not Implemented):

The server either does not recognize the request method, or it cannot fulfill the request. Usually, this implies future availability (e.g., a new feature of a web-service API).

Summary of HTTP Status Codes

1×× Informational

- 100 Continue
- 101 Switching Protocols
- 102 Processing

2×× Success

- 200 OK
- 201 Created
- 202 Accepted
- 203 Non-authoritative Information
- 204 No Content
- 205 Reset Content
- 206 Partial Content

- 207 Multi-Status
- 208 Already Reported
- 226 IM Used

3×× Redirection

- 300 Multiple Choices
- 301 Moved Permanently
- 302 Found
- 303 See Other
- 304 Not Modified
- 305 Use Proxy
- 307 Temporary Redirect
- 308 Permanent Redirect

4×× Client Error

- 400 Bad Request
- 401 Unauthorized
- 402 Payment Required
- 403 Forbidden
- 404 Not Found
- 405 Method Not Allowed
- 406 Not Acceptable
- 407 Proxy Authentication Required
- 408 Request Timeout
- 409 Conflict
- 410 Gone
- 411 Length Required
- 412 Precondition Failed
- 413 Payload Too Large
- 414 Request-URI Too Long
- 415 Unsupported Media Type
- 416 Requested Range Not Satisfiable
- 417 Expectation Failed
- 418 I'm a teapot
- 421 Misdirected Request
- 422 Unprocessable Entity
- 423 Locked
- 424 Failed Dependency
- 426 Upgrade Required

- 428 Precondition Required
- 429 Too Many Requests
- 431 Request Header Fields Too Large
- 444 Connection Closed Without Response
- 451 Unavailable For Legal Reasons
- 499 Client Closed Request

5×× Server Error

- 500 Internal Server Error
- 501 Not Implemented
- 502 Bad Gateway
- 503 Service Unavailable
- 504 Gateway Timeout
- 505 HTTP Version Not Supported
- 506 Variant Also Negotiates
- 507 Insufficient Storage
- 508 Loop Detected
- 510 Not Extended
- 511 Network Authentication Required
- 599 Network Connect Timeout Error

You can check out all the HTTP status codes at **https://developer.mozilla.org/en-US/docs/Web/HTTP/Status**

Using HTTP Status Codes

REST API's for **User Management Application -** Following five REST APIs for User resource and their status codes (refer to status code column):

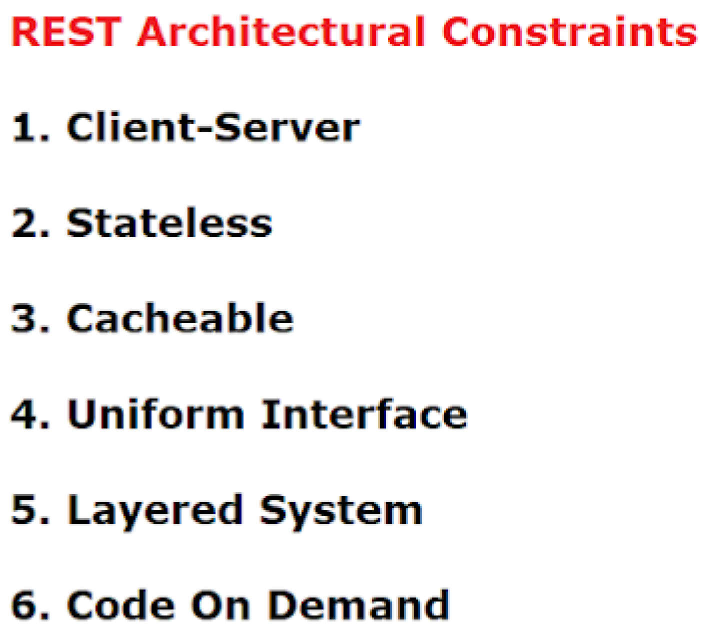
Sr. No.	API Name	HTTP Method	Path	Status Code	Description
(1)	GET Users	GET	`/api/v1/users`	200 (OK)	All User resources are fetched.
(2)	POST User	POST	`/api/v1/users`	201 (Created)	A new User resource is created.
(3)	GET User	GET	`/api/v1/users/{id}`	200 (OK)	One User resource is fetched.
(4)	PUT User	PUT	`/api/v1/users/{id}`	200 (OK)	User resource is updated.
(5)	DELETE User	DELETE	`/api/v1/users/{id}`	204 (No Content)	User resource is deleted.

Advantages of REST

- REST is simple.
- REST is widely supported.
- Resources can be represented in a wide variety of data formats (JSON, XML, etc.)
- You can make good use of HTTP cache and proxy server to help you handle high load and improve performance.
- It reduces client/server coupling.
- Browsers can interpret representations.

- Javascript can use representations.
- A rest service can be consumed by applications written in different languages.
- It makes it easy for new clients to use a RESTful application, even if the application was not designed specifically for them.
- Because of the statelessness of REST systems, multiple servers can be behind a load balancer and provide services transparently, which means increased scalability.
- Because of the uniform interface, little or no documentation of the resources and basic operations API is necessary.
- Using REST does not imply specific libraries at the client level in order to communicate with the server. With REST, all that is needed is a network connection.

REST stands for **Representational State Transfer**, a term coined by Roy Fielding in 2000. It is an architecture style for designing loosely coupled applications over HTTP, that is often used in the development of web services. REST does not enforce any rule regarding how it should be implemented at a lower level, it just put high-level design guidelines and leave you to think of your own implementation.



REST Architectural Constraints

1. **Client-Server**
2. **Stateless**
3. **Cacheable**
4. **Uniform Interface**
5. **Layered System**
6. **Code On Demand**

REST Architectural Constraints are design rules that are applied to establish the distinct characteristics of the REST architectural style.

If you follow all constraints designed by the REST architectural style your system is considered RESTful.

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.

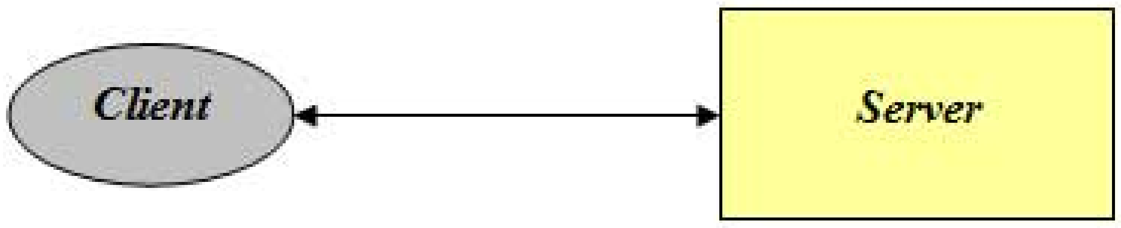


Figure: Client-Server

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.

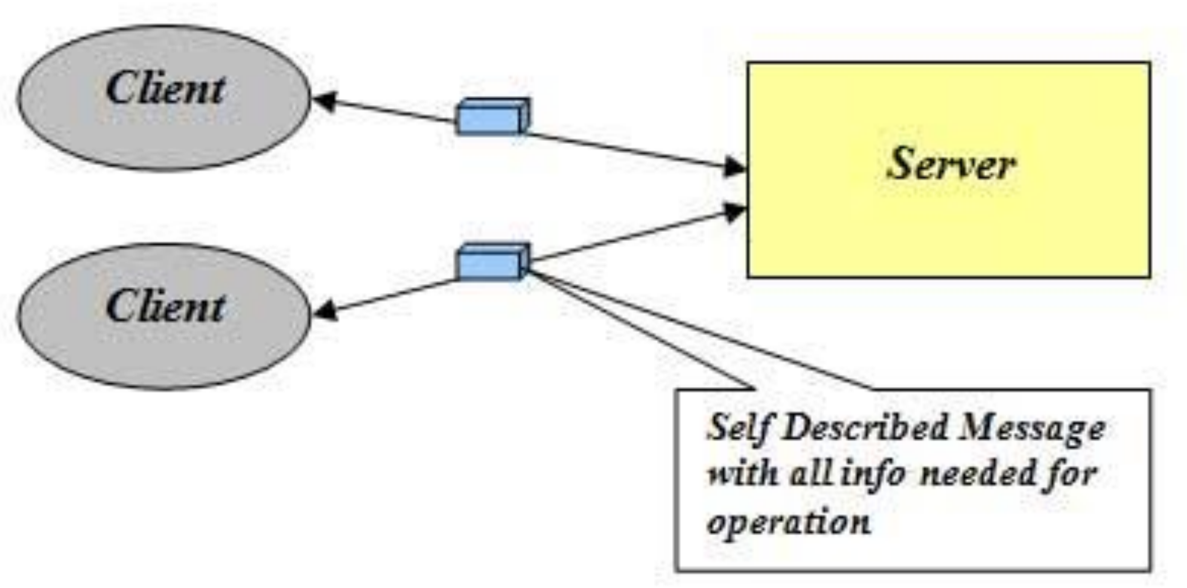


Figure: Stateless

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.

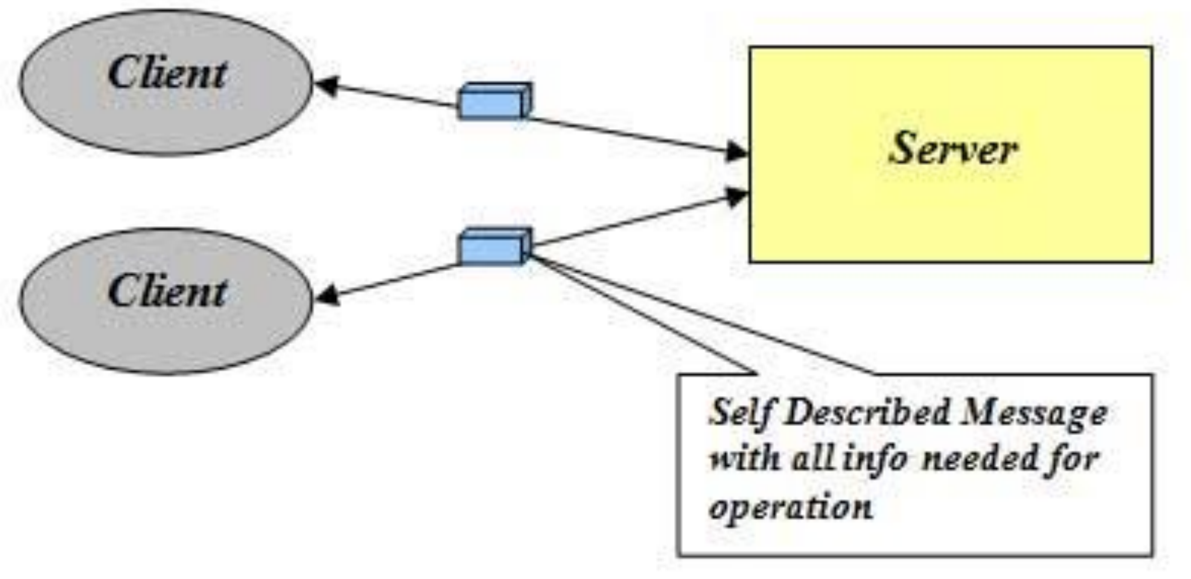


Figure: Cacheable

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.

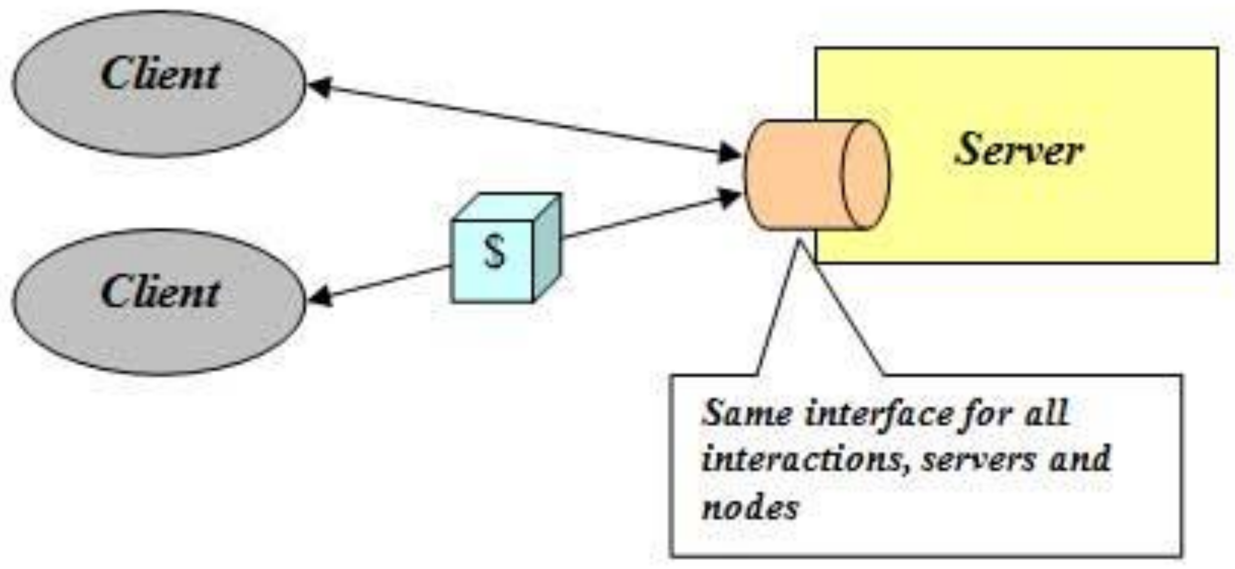


Figure: Uniform Interface

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.

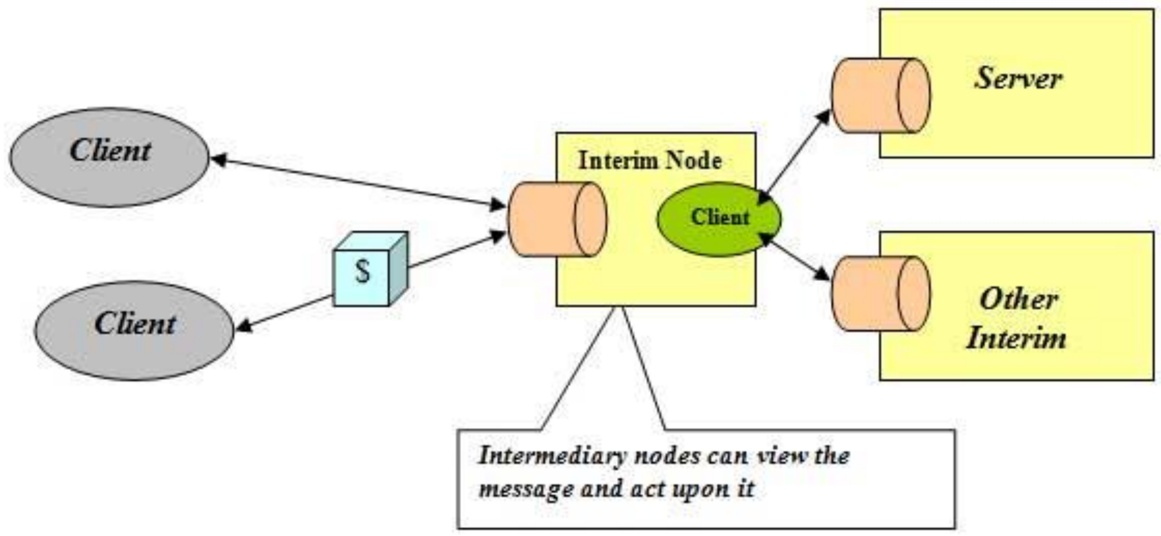


Figure: Layered System

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.

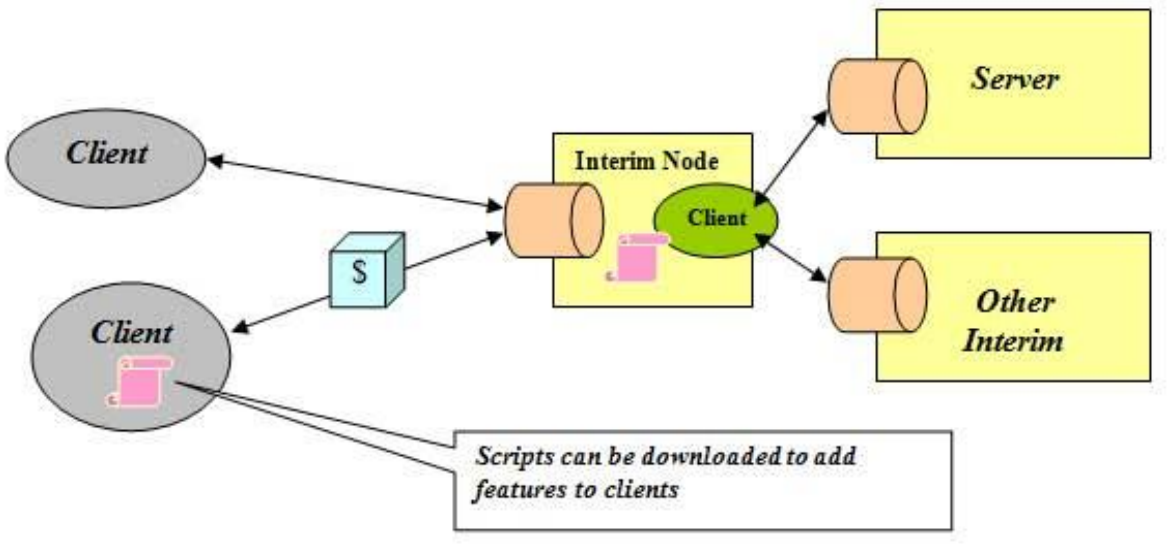
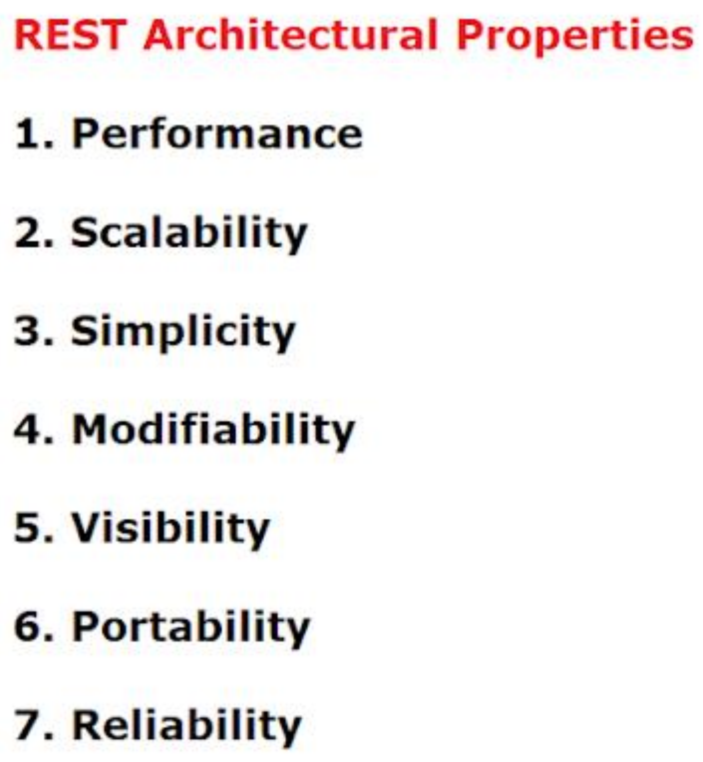


Figure: Code on demand

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.

REST Architectural Properties



REST Architectural Properties

1. Performance
2. Scalability
3. Simplicity
4. Modifiability
5. Visibility
6. Portability
7. Reliability

The constraints of the *REST* architectural style affect the following architectural properties:

1. Performance

The term *performance* means different things to different people. To end-users, high performance means that when they click on a link or click on a buy button on an e-store, they get immediate results; to developers, it means solving the typical problem of service availability and supporting architectures that are scalable.

REST can support the Performance goal by using caches to keep available data close to where it is being processed. It can further help by minimizing the overhead associated with setting up complex interactions by keeping each interaction simple and self-contained (as a request-response pair).

2 Scalability

Scalability, in the context of the Web, means to consistently provide service regardless of the increase or decrease of web users.

`REST`-style architectures support load balancing across service instances via the Layered System and Uniform Interface constraints. This can be further simplified through the Stateless constraint, which structures interactions as request-response pairs that can each be handled by services independently of other requests.

There is no need to keep directing the same service consumer to the same service instance or to synchronize session state explicitly between service instances; the session data is contained in each request.

3. Simplicity

By separating the functionality within components in a system, we can induce the simplicity of the architectural styles. If functionality can be allocated such that each component is less complex, then it will be easier to understand and implement.

The major contribution of `REST` to Simplicity is the Uniform Interface constraint. The overall system architecture is simplified and the visibility of interactions is improved. The functionality encapsulated by services is abstracted from the underlying interface implementation.

4. Modifiability

The requirements for any architecture are bound to change over time. Modifiability represents the ease at which changes can be incorporated into the architecture.

`Modifiability` in `REST`-style architecture is further broken down into the following areas:

- **Evolvability** - Represents the degree to which a component can be refactored without impacting other parts of the architecture.
- **Extensibility** - The ability to add functionality to the architecture (even while solutions are running)
- **Customizability** - The ability to temporarily modify parts of a solution to perform special types of tasks
- **Configurability** - The ability to permanently modify parts of an architecture
- **Reusability** - The ability to add new solutions to architecture that reuse existing services, middleware, methods, and media types, without modification. Modifiability is particularly important for larger distributed architectures where it is not possible to redeploy the entire environment every time an architectural improvement is made.

5. Visibility

`Visibility` refers to the ability of a component to monitor or mediate the interaction between two components. REST supports visibility through Uniform Interface constraint. Visibility can enable:

- Improved `performance` via shared caching of interactions,
- `Scalability` through layered services,
- `Reliability` through reflective monitoring
- `Security` by allowing the interactions to be inspected by mediators

6. Portability

`Portability` represents the each with which a system can be moved from one deployed location to other. portability of components by moving program code with the data; In REST architecture style, the client/server constraint helps improve the UI portability.

7. Reliability

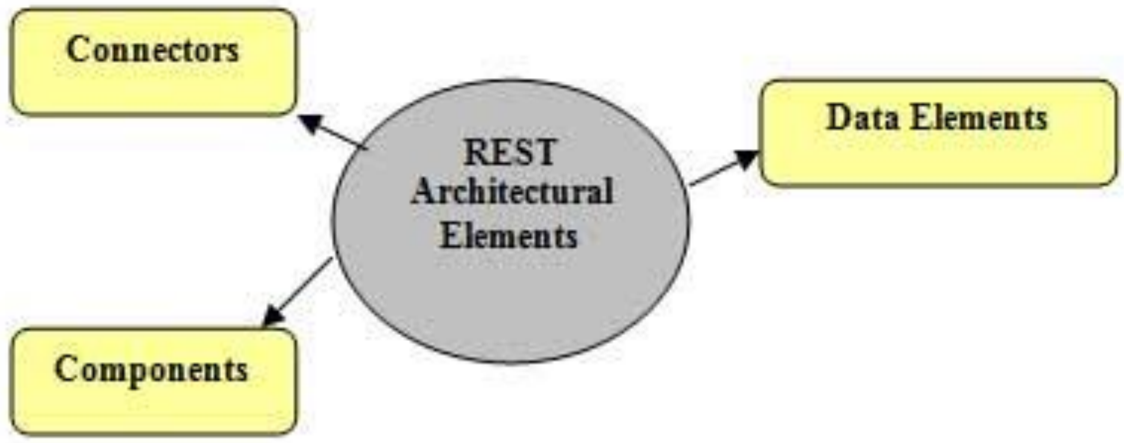
`Reliability` of a distributed architecture is the degree to which its solutions and services (and underlying infrastructure) are susceptible to failure. An architecture style can improve reliability by avoiding single points of failure, using failover mechanisms, and be relying on monitoring features that can dynamically anticipate and respond to failure conditions.

REST API - REST Architectural Elements

The Representational State Transfer (REST) style is an abstraction of the architectural elements within a distributed hypermedia system. REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements.

It encompasses the fundamental constraints upon components, connectors, and data that define the basis of the Web architecture, and thus the essence of its behavior as a network-based application. REST distinguishes three classes of architectural elements, they are:

- Connectors
- Components
- Data Elements



Connectors

Connectors represent the activities involved in accessing resources and transferring representations. Roles provide an interface for components to implement. REST encapsulates different activities of accessing and transferring representations into different connector types. The table below summarizes the connector types:

Connector Type	Description	Example
Client	Sending requests, receiving responses.	HTTP library
Server	Listening for requests, sending responses.	Web Server API
Cache	Can be located at the client or server connector to save cacheable responses, can also be shared between several clients	Browser cache
Resolver	Transforms resource identifiers into network addresses.	bind (DNS lookup library)
Tunnel	Relays requests, any component can switch from active behavior to a tunnel behavior.	SOCKS, SSL after HTTP CONNECT

Components

In REST, the various software that interacts with one another are called components. They are categorized by roles summarized in the table below:

Component Role	Description	Example
Origin Server	Uses a server connector to receive the request, and is the definitive source for representations of its resources. Each server provides a generic interface to its services as a resource hierarchy.	Apache httpd, Microsoft IIS
User Agent	Uses a client connector to initiate a request and becomes the ultimate recipient of the response.	Browser like Netscape Navigator etc
Gateway	Act as both, client and server in order to forward - with possible translation - requests and responses.	Squid, CGI, Reverse Proxy
Proxy		CERN Proxy, Netscape Proxy, Gauntlet

Data Elements

The key aspect of `REST` is the state of the data elements, its components communicate by transferring representations of the current or desired state of data elements.

`REST` identifies six data elements: `a resource, resource identifier, resource metadata, representation, representation metadata, and control data`, shown in the table below:

Data Element	Description	Example

Resource	Any information that can be named is a resource. A resource is a conceptual mapping to a set of entities, not the entity itself.	Title of a movie from IMDb, A Flash movie from YouTube, Images from Flickr etc
Resource Identifier	Every resource must have a name that uniquely identifies it. Under HTTP these are called URIs. Uniform Resource Identifier (URI) in a RESTful system is a hyperlink to a resource. It is the only means for clients and servers to exchange representations of resources. The relationship between URIs and resources is many to one. A resource can have multiple URIs which provide different information about the location of a resource.	
Resource metadata	This describes the resource. A metadata provides additional information such as location information, alternate resource identifiers for different formats or entity tag information about the resource itself.	Source link, vary
Representation	It is something that is sent back and forth between clients and servers. So, we never send or receive resources, only their representations. A representation captures the current or intended state of a resource. A particular resource may have multiple representations	Sequence of bytes, HTML document, archive document, image document
Representation metadata	This describes the representation.	Headers (media-type)

Control data	This defines the purpose of a message between components, such as the action being requested.	If-Modified-Since, If-Match

`Addressable` means anything that can be accessed and transferred between clients and servers

Details of the standardized format of a URI are as follows:

scheme://host: port/path?queryString#fragment

- **scheme** - It is the protocol you are using to communicate with. For `RESTful` web services, it is usually `HTTP` or `HTTPS`
- **host** - It is a DNS name or IP address
- **port** - This is optional and which is numeric The host and port represent the location of your resource on the network
- **path** - This expression is a set of text segments delimited by the "/" character. Think of the path expression is a directory list of a file on your machine
- **?** - This character separates the path from the queryString. queryString -This is a list of parameters represented as name/value pairs. Each pair is delimited with the "&" character. Here's an example query string within a URI: **http://somedomain.com/customers?firsttName=Manisha&zipcode=411027**
- **fragment**: It is delimited by a "#" character. The fragment is usually used to point to a certain place in the document you are querying