**Rest API Standards**

Contents

[1 Document Control 3](#_Toc92185729)

[1.1 Change Record 3](#_Toc92185730)

[1.2 Reviewer 3](#_Toc92185731)

[1.3 Approver 3](#_Toc92185732)

[2 Document Purpose 4](#_Toc92185733)

[3 Accept And Respond with JSON 4](#_Toc92185734)

[4 Rest Archetypes 4](#_Toc92185735)

[4.1 Document 4](#_Toc92185736)

[4.2 Collection 4](#_Toc92185737)

[4.3 Store 4](#_Toc92185738)

[4.4 Controller 5](#_Toc92185739)

[5 API Endpoints 5](#_Toc92185740)

[5.1 Organize the API design around resources 5](#_Toc92185741)

[6 Resource Modeling 5](#_Toc92185742)

[7 Http Methods 6](#_Toc92185743)

[8 Http Response Status Codes 6](#_Toc92185744)

[9 Media Type 7](#_Toc92185745)

[Below example describes the media type 7](#_Toc92185746)

[10 Asynchronous Operations 8](#_Toc92185747)

[Applications of SignalR 8](#_Toc92185748)

[11 Sort, Filter, Pagination 8](#_Toc92185749)

[11.1 Sort 8](#_Toc92185750)

[11.2 Filter 9](#_Toc92185751)

[11.3 Pagination 9](#_Toc92185752)

[11.3.1 Offset Pagination 9](#_Toc92185753)

[11.3.2 Keyset Pagination 9](#_Toc92185754)

[11.3.3 Seek Pagination 9](#_Toc92185755)

[11.4 Large query string parameters 10](#_Toc92185756)

[11.4.1 Encoding 10](#_Toc92185757)

[12 Sensitive Information in HTTP Requests 10](#_Toc92185758)

[12.1.1 Encryption 11](#_Toc92185759)

[12.1.2 Password Hashing 11](#_Toc92185760)

[13 Versioning 11](#_Toc92185761)

[13.1 URI Versioning 11](#_Toc92185762)

[13.2 Query String Versioning 11](#_Toc92185763)

[14 General Rules 11](#_Toc92185764)

# Document Control

## Change Record

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Author** | **Version** | **Change reference** |
| 11/25/2021 | Prajeesh T S | 1.1 | Initial version |

## Reviewer

|  |  |  |
| --- | --- | --- |
| **Name** | **Role** | **Approval/Review Date** |
| **Anoop Jose** | Staff Software Architect |  |

## Approver

|  |  |  |
| --- | --- | --- |
| **Name** | **Role** | **Approval/Review Date** |
| **Anoop Jose** | Staff Software Architect |  |

# Document Purpose

Purpose of the document is to define the standards and general guidelines to define & implement RestAPIs with OpenAPI standards.

# Accept And Respond with JSON

REST APIs should accept JSON for request payload and also send responses to JSON. JSON is the standard for transferring data. Almost every networked technology can use it. JavaScript has built-in methods to encode and decode JSON either through the Fetch API or another HTTP client. Server-side technologies have libraries that can decode JSON.

To make sure that when our REST API app responds with JSON that clients interpret it as such, we should set Content-Type in the response header to application/json after the request is made. Many server-side app frameworks set the response header automatically. Some HTTP clients look at the Content-Type response header and parse the data according to that format.

# Rest Archetypes

## Document

Use “singular” name to denote document resource archetype.

## Collection

Use “plural” name to denote collection resource archetype.

Below URI’s identifies the collection resource

* api.microsoft.org/departments
* api.microsoft.org/departments/hr/managers

## Store

Use “plural” name to denote store resource archetype.

Example

*There are resources for an organization and we need to assign existing employee for manager(mentor) as mentee.*

***Employee****:  
Rahul[id:35]****Manager****:*

*Victor[id:2]*

*URI:*

*PUT-api/****managers****/2/****mentee****/employees/35  
PUT-api/****managers****/2/****mentee****/employees/Rahul  
PUT-api/****managers****/victor/****mentee****/employees/35*

## Controller

A verb or verb phrase should be used for controller name

# API Endpoints

## Organize the API design around resources

* *The paths should contain the plural form of resources and the HTTP method should define the kind of action to be performed on the resource.*
* *The URL should only contain resources(nouns) not actions or verbs.*
* *HTTP methods (GET, POST, DELETE, PUT) also called as verbs should play the role of actions*

Examples

<https://adventure-works.com/employees/getAllEmployess> // Bad

[https://adventure-works.com/employees/addEmployee //](https://adventure-works.com/employees/addEmployee%20%20%20%20%20%20%20%20//) Bad

<https://adventure-works.com/employees/deleteEmployee> // Bad

Http Get Method <https://adventure-works.com/employees/> ->Returns all employees

Http Post Method <https://adventure-works.com/employee/> ->Add new employee

Http Delete method <https://adventure-works.com/employee/>45 ->Delete employee with id 45.

# Resource Modeling

Calls across the network may be expensive, so to minimize them, coarse-grained APIs may be the best fit, as each request from the client forces lot of work at the server side, and in fine-grained, many calls are required to do the same amount of work at the client side.

The coarse-grained approach:

* GET /user/{id}, to get the user details along with his contacts
* POST /users, to add a new user along with his contacts
* PUT /users/{id}, to update a user along with his contacts

The fine-grained approach:

* GET /user/{id}, to get the user details
* GET /user/{id}/contacts, to get the user's contacts
  + POST /user, to add a new user
  + PUT /user/{id}, to update a user
  + POST /user/{id}/contacts, to add a new user contact
  + DELETE /user/{id}/contact/{id}, to delete a user contact

# Http Methods

* GET - Retrieves a representation of the resource at the specified URI. The body of the response message contains the details of the requested resource.
* POST - Creates a new resource at the specified URI. The body of the request message provides the details of the new resource.
* PUT - Either creates or replaces the resource at the specified URI. The body of the request message specifies the resource to be created or updated.
* PATCH- Performs a partial update of a resource. The request body specifies the set of changes to apply to the resource.
* DELETE - Removes the resource at the specified URI.

# Http Response Status Codes

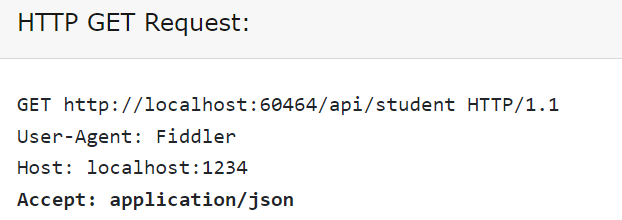
|  |  |  |  |
| --- | --- | --- | --- |
| Method | Success | Failure / Partial Process | Invalid Request |
| GET | 200 (OK) | 404 (Not Found) |  |
| POST | 201 (Created) | 200,204 (No Content) | 400 (Bad Request) |
| PUT | 201 (Created) | 409 (Conflict) |  |
| PATCH | 204 (No Content) | 409 (Conflict) | 400 (Bad Request) |
| DELETE | 204 (No Content) | 404 (Not Found) |  |

# Media Type

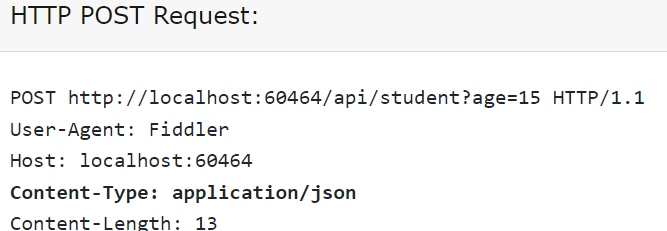
Media types allow an API to inform the client how to interpret the data in the payload.

In HTTP request, MIME type is specified in the request header using **Accept** and **Content-Type** attribute. The Accept header attribute specifies the format of response data which the client expects and the Content-Type header attribute specifies the format of the data in the request body so that receiver can parse it into appropriate format.

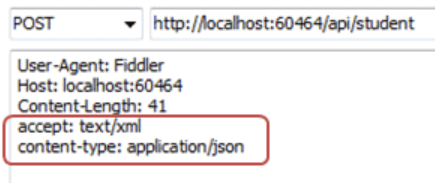
If a client wants response data in JSON format then it will send following GET HTTP request with Accept header to the Web API.



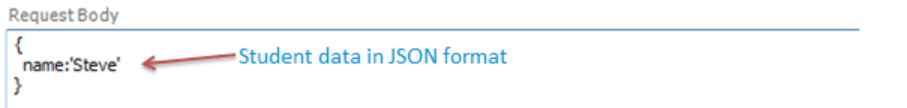
If a client includes JSON data in the request body to send it to the receiver then it will send following POST HTTP request with Content-Type header with JSON data in the body.



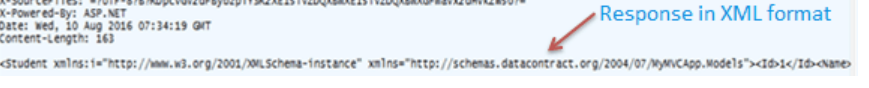
# Below example describes the media type



Request Body



Response



# Asynchronous Operations

POST, PUT, PATCH, or DELETE operation might require processing that takes a while to complete. If we wait for completion before sending a response to the client, it may cause unacceptable latency. If so, consider making the operation asynchronous. Return HTTP status code 202 (Accepted) to indicate the request was accepted for processing but is not completed.

We should expose an endpoint that returns the status of an asynchronous request, so the client can monitor the status by polling the status endpoint. Include the URI of the status endpoint in the Location header of the 202 response

We can also consider Signlr for an option to know the request is complete. This will eliminate the process of polling the request, the Signlr hub will send the notification of the process and the connected clients can consume the notification.

SignalR simplifies the process of adding real-time web functionality to web applications, where the server code pushes content to connected clients as soon as it becomes available.

## Applications of SignalR

* User notifications
* Sending high-frequency updates to clients
* Dashboards containing real-time charts and graphs
* Collaborative applications, such as chat and messaging services
* Games and entertainment applications
* Alerting mechanisms

# Sort, Filter, Pagination

## Sort

Should sort the dataset using query string

https://adventure-works.com/v2/employees?sort=rank\_asc

Sort multiple columns/parameters

To sort multiple fields, specify a comma-separated list of fields.

https://adventure-works.com/v2/employees? orderby=Created,Modified

https://adventure-works.com/v2/employees? orderby= Created asc,Modified desc

## Filter

Filtering dataset should be like below

https://adventure-works.com/v2/employees?location=India

## Pagination

When the dataset is too large, we divide the data set into smaller chunks, which helps in improving the performance.

### Offset Pagination

Offset pagination is one of the simplest to implement. It’s achieved using the limit and offset commands.

https://adventure-works.com/v2/employees?offset=0&limit=15

### Keyset Pagination

Keyset pagination uses the filter values of the previous page to determine the next set of items. Those results will then be indexed.

Consider this example:

1. Client requests most recent items GET /items? Limit=20
2. Upon clicking the next page, the query finds the minimum created date of 2019–01–20T00:00:00. This is then used to create a query filter for the next page. GET /items? Limit=20&created: lte:2019-01-20T00:00:00

The benefits of this approach are that it doesn’t require additional backend logic. It only requires one limit URL parameter. It also features consistent ordering, even when new items are added to the database. It also works smoothly with large offset values.

### Seek Pagination

Seek pagination is the next step beyond keyset pagination. Adding the queries after\_id and before\_id, you can remove the constraints of filters and sorting.

Consider this example:

1. Client requests a list of the most recent items GET items?limit=20
2. Client requests a list of the next 20 items, using the results of the first query GET /items?limit=20&after\_id=20
3. Client requests the next/scroll page, using the final entry from the second page as the starting point GET /items?limit=20&after\_id=40

## Large query string parameters

Evaluate whether longest query string is longer than 2000 characters. If it doesn't, and you don't expect it to be, go with GET. It might seem ugly but it has all the advantages derived from the method' semantics (idempotence, safe and caching) and bookmarking.

For large query strings, encoding is the better option if the data is not sensitive, for sensitive data create new POST method with search parameters as body.

If the filter criteria are less, attaching the operators [and, or etc.] in the query string is the right choice, if it is larger Microsoft OData would be the right choice.

### Encoding

* Build the JSON with all the filters and normalise it.
* Parse it to string
* Encode it
* Send the encoded JSON as request param (/search=SGVsbG8gV29ybGQh....).
* On the server-side, decode.
* Deserialize the JSON string

# Sensitive Information in HTTP Requests

RESTful web services should be careful to prevent leaking credentials. Passwords, security tokens, and API keys should not appear in the URL, as this can be captured in web server logs, which makes them intrinsically valuable.

* In POST/PUT requests sensitive data should be transferred in the request body or request headers.
* In GET requests sensitive data should be transferred in an HTTP Header.
* Sending the username and password in the body of the POST request is the best practice (Never use GET to send sensitive information such as Credentials).
* Sending the api token in the HTTP request and response headers is the best practice (Again never use GET to send sensitive information such as session tokens)
* The time out of the API token should be short in case of idle user. (5 ~ 15 mins are the averages based on the criticality of the application)
* The length of the API token should be long string approx. 30 ~ 40 characters.
* The API token generation must be randomized and hard to predict to protect from (session prediction attacks.)

### Encryption

#### AES Encryption

AES is a symmetric algorithm which uses the same 128, 192, or 256 bits key for both encryption and decryption (the security of an AES system increases exponentially with key length). With even a 128-bit key, the task of cracking AES by checking each of the 2128 possible key values (a “brute force” attack) is so computationally intensive that even the fastest supercomputer would require, on average, more than 100 trillion years to do it. In fact, AES has never been cracked, and based on current technological trends, is expected to remain secure for years to come.

### Password Hashing

Bcrypt is a password-hashing function designed by Niels Provos and David Mazières, based on the Blowfish cipher and presented at USENIX in 1999.[[1]](https://en.wikipedia.org/wiki/Bcrypt#cite_note-provos-1) Besides incorporating a salt to protect against rainbow table attacks, bcrypt is an adaptive function: over time, the iteration count can be increased to make it slower, so it remains resistant to brute-force search attacks even with increasing computation power.

# Versioning

## URI Versioning

Each time we modify the web API or change the schema of resources, we add a version number to the URI for each resource. The previously existing URIs should continue to operate as before, returning resources that conform to their original schema.

Example

Old API to return customer details - https://adventure-works.com/employees/3.

Suppose the response of the request is changed and we should add the versioning like below

 https://adventure-works.com/v2/employees/3.

## Query String Versioning

Rather than providing multiple URIs, you can specify the version of the resource by using a parameter within the query string appended to the HTTP request, such as

 https://adventure-works.com/employees/3?version=2.

# General Rules

* A trailing forward slash (/) should not be included in URIs
* Hyphens (-) should be used to improve the readability of URIs
  + http://api.example.restapi.org/blogs/mark-masse/entries/my-first-post
* Underscores (\_) should not be used in URIs
* Lowercase letters should be preferred in URI paths
  + http://api.example.restapi.org/my-folder/my-doc
* File extensions should not be included in URIs
* Consistent subdomain names should be used for your APIs