HTTP methods and REST APIs

After reading this document, you should be able to:

- Define terms related to HTTP methods
- Explain guidelines and best practices for writing REST APIs

The internet relies on a client-server architecture. The end-user interfaces with the client, while the servers house the *services*[i] that operate the applications, the business logic, and the data. Clients communicate with the servers to achieve desired functionality for the end user. Data is transferred between client and server using hypertext transfer protocol, more commonly known as *HTTP*[ii]. This communication usually happens via *APIs*[iii]. In 2000, a set of guidelines for writing these APIs for a client-server architecture was developed called *REST*[iv] APIs.

The acronym "REST" stands for REpresentational State Transfer. Before explaining this term in more detail, let's discuss HTTP methods and some terminology first.

In a client/server architecture, the applications are composed of one or more services that reside on the servers. These services contain *resources*[v]. The client makes a *request*[vi] for a resource via a *request object*[vii] using a *route*[viii] that has an *endpoint*[ix] within the service. The application sends a *response object*[x] back in *response*[xi] to the client to honor that request.

A request object contains three parts, a *URL*[xii], a *request header*[xiii], and a *request body*[xiv]. The server uses the URL to identify the service and the endpoint within the service being acted upon. The URL contains four parts: a *protocol*[xv], a *hostname*[xvi], a *path*[xvii], and a *query string*[xviii]. The request header contains metadata about the resource of the requesting client, such as the *user agent*[xix], *host*[xx], *content type*[xxi], *content length*[xxii], and what type of data the client should expect in the response.

The server responds with a response object consisting of a *header*[xxiii], a *body,*[xxiv] and a *status code*[xxv]. The response object body often contains a *JSON*[xxvi] *payload*[xxvii] to provide the data back to the client.

There are a number of HTTP methods that can be used in the REST API that allows interaction between the client and a service. The most common methods are *GET*[xxviii], *POST*[xxix], *PUT*[xxx], *DELETE*[xxxi], and *PATCH*[xxxii]. The name of the method describes what happens to the resource when the method is applied. GET is used to read the data. It should never change the data. POST is used to create new data. PUT and PATCH are used to updated the data. DELETE is used to delete the data. PUT and DELETE methods are *idempotent*[xxxiii] meaning that if the same API method is called multiple times it will return the same result.

HTTP has three ways to pass parameters: the *URL path parameter*[xxxiv], the *URL query parameter*[xxxv], and the *header parameter*[xxxvi]. The path and query parameters are passed as part of the URL, but the header parameter is passed by the browser directly to the service.

When the service completes a request, it returns a response. An HTTP status code should be part of that response. The HTTP status code indicates whether the response has been completed or not. Response code categories are shown in the following table.

Status code range	Meaning
200-299	Everything is OK
300-399	Resource has moved
400-499	Client-side error
500-599	Server-side error

As mentioned earlier, REST is a set of guidelines. There are five requirements for an API to be considered RESTful, plus one optional criterion:

1. The API leverages a client-server architecture made up of resources that are managed and delivered via HTTP
2. Communication between client and server is *stateless*[xxxvii]
3. Data is *cacheable*[xxxviii] to improve performance on the client side
4. The interface is transferred in a standard format such that the requested resources stored on the server are separated from the representation sent to the client. The representation sent to the client contains sufficient data so that the client can manipulate the representation
5. Requests and responses communicate through different layers, such as *middleware*[xxxix]. The client and server often do not communicate directly with each other.
6. (Optional) Resources are usually static but can also contain executable code. The code should only be executed when the client requests it

When the client makes a request, it also must pass information about its state to the server. Every communication between the client and server should contain all the information needed to perform the request. The client, not the service, maintains the state. Each request must contain the requisite information so the server understands the request. So, for example, if the user is viewing a database record and that record needs to be updated, the client must also send which record needs updating. The server doesn't know which record is currently being viewed.

When defining a RESTful service, every framework as a way to designate a route. This project uses the Python Flask framework. In Flask, a Python decorator called: @app.route() method is used. The @app.route() method takes two parameters, the route from the URL to the service being acted upon and an optional HTTP method parameter such as POST, GET, and so on. The route parameter can also include a variable in the path, such as <username>, for example. The root of a route is '/'. So, for example, if you want the route to be www.mywebsite.com/accounts, you only need to specify "/accounts" as the route parameter in the @app.route function.

REST APIs have the following characteristics:

- Resource-based; that is, they describe sets of resources
- Only contain nouns, not verbs
- Use singular nouns when referring to a singular resource or plural nouns when referring to a collection of resources
- Always identified by URLs

NOT RESTful APIs	**RESTful equivalents**
GET http://api.myapp.com/getUser/123	GET http://api.myapp.com/users/123
POST http://api.myapp.com/addUser	POST http://api.myapp.com/users
GET http://api.myapp.com/removeUser/123	DELETE http://api.myapp.com/users/123

URL format guidelines

- Should use a slash '/' to denote a hierarchical relationship in the directory structure
- Should avoid using a trailing slash, e.g., /resource/
- Should use hyphens, not camel case, e.g., /my-resource, not /myResource
- Should not use an underscore '_' in the URL, e.g., /my-resource, not /my_resource
- Should use lowercase
- Should not use a period '.' in a URL
- May contain multiple subordinate resources and IDs in the URL, e.g., GET /resource/{id}/subordinate/{id}

Here are some guidelines for implementing the lifecycle methods of Create, Read, Update, Delete, and List for a resource named Account.

Action	Method	Return code	Body	URL Endpoint
List	GET	200_OK	Array of accounts [{...}]	GET /accounts
Create	POST	201_CREATED	An account as json {...}	POST /accounts
Read	GET	200_OK	An account as json {...}	GET /accounts/{id}
Update	PUT	200_OK	An account as json {...}	PUT /accounts/{id}

Action	Method	Return code	Body	URL Endpoint
Delete	DELETE	204_NO_CONTENT	""	DELETE /accounts/{id}

List is used to list all of the accounts. It is implemented using the GET method. It could have a query string to filter the list. It should return the list as an array, and a return code of 200_OK if the request completed successfully. If no accounts are found, it should return an empty array [] and 200_OK. It should not return 404_NOT_FOUND. Clients of this call are expecting a collection to be returned even if that collection is empty. The route should be: GET /accounts

Create is used to create a new account. It is implemented using the POST method. The data for the request is sent in the body. It should return the account that was created and a return code of 201_CREATED if the request is completed successfully. If the database id is generated, it should also return a Location header with the URL of the newly created account. If the client passes in the database id, it may return a 409_CONFLICT if the id already exists. The route should be: POST /accounts

Read is used to retrieve the data for an account. It is implemented using the GET method and has a path parameter specifying the id of the account to read. There is no data sent in the body of the request as it is read-only. GET should never modify the data in any way. It should return the account that was found and a return code of 200_OK if the request is completed successfully. It should return 404_NOT_FOUND if the account cannot be found. The route should be: GET /accounts/{id}

Update is used to modify the data of an account. It is implemented using the PUT method and has a path parameter specifying the id of the account to update. The data for the request is sent in the body. It should return the updated account and a return code of 200_OK if the request is completed successfully.

Note that PUT is idempotent. If the same PUT request is made multiple times, it should result in the account being in the exact same state every time. You can also use PATCH to update an account. The difference between PUT and PATCH is that PUT requires all of the data for an account to be sent with the request, while PATCH only requires the data elements that you want to change. The route should be: PUT /accounts/{id}

Delete is used to delete an account. It is implemented using the DELETE method and has a path parameter specifying the id of the account to delete. There is no data sent in the body of the request. Delete should return an empty body and a return code of 204_NO_CONTENT if the request is completed successfully.

Note that DELETE is also idempotent. That means if the request is made multiple times for the same account, it should return the same results. It should never return 404_NOT_FOUND. If the account cannot be found, it must have been deleted. Therefore the response should always be 204_NO_CONTENT with an empty body. The route should be: DELETE /accounts/{id}

---

[i] **Service:** A software component that makes up part of an application and serves a specific purpose. Generally, a service takes input from a client or another service and produces an output.

[ii] **HTTP:** The protocol used by a client-server architecture on the internet for fetching or exchanging resources' data.

[iii] **API**: An Application Programming Interface is a set of definitions and protocols that allow two services to communicate with each other. The API requests data that can be exchanged between a resource and the results that are returned from that resource.

[iv] **REST**: A set of architectural guidelines that describe how to write an interface (API) between two components, usually a client and server, that describe how these components communicate with each other. REST stands for REpresentational State Transfer. REST describes a standard way to identify and manipulate resources. REST ensures the messages passed between the client and server are self-descriptive and define how the client interacts with the server to access resources on the server.

[v] **Resource**: A *resource* is the fundamental concept of a RESTful API. It is an object that has a defined type, associated data, relationships to other resources, and a set of methods that can operate on it. A resource is commonly defined in JSON format but can also be XML.

[vi] **Request**: A *request* is made by a client to a host on a server to access a *resource*. The client uses parts of a URL to determine the information needed from the *resource*. Most common request methods include GET, POST, PUT, PATCH, and DELETE, but also include HEAD, CONNECT, OPTIONS, TRACE, and PATCH.

[vii] **Request Object**: Contains the HTTP request data. It contains three parts: a URL, a header, and a body.

[viii] **Route:** The combination of an HTTP method and the path to the resource from the root of the path.

[ix] **Endpoint**: The location of the *resource* specified by a REST API that is being accessed on the server. It is usually identified through the URL in the HTTP method of the API.

[x] **Response Object**: Contains the HTTP response data in response to a *request*. It contains a header, a body, and a status.

[xi] **Response**: A *response* is made by a server and sent to a client to either provide the client with the requested resource, tell the client the requested action has been completed, or let the client know there has been an error processing the *request*.

[xii] **URI/URL**: A "Uniform Resource Identifier" is used interchangeably with the term URL. They are part of a RESTful API that locates the endpoint of the requested resource and contains the data about how that endpoint should be manipulated. The client issues an HTTP request using the URI/URL to manipulate the resource. They should consist of four parts: the hostname, the path, the header, and a query string.

[xiii] **Request header**: information passed to the server about the retrieved resource or the requesting client

- the method with endpoint i.e., POST /car-reviews

- User-agent:,
- the host, i.e., www.edmunds.com
- contentType (what am I sending, ex: application data in a form, could also be JSON)
- content length: how many bytes of data
- Accept-Encoding: what kind of data the client is expecting in return. Ex: application/json data
- Information about the connection

[xiv] **Request body*:*** provides the data being passed to the server.

[xv] **Protocol:** Tells the service how the data is to be transferred between the server and the client.

[xvi] **Hostname:** The name of a device on a network, also often called the site name.

[xvii] **Path**: The path identifies the location of the resource in the service and its endpoint. For example: https://www.customerservice/customers/{customer_id}

[xviii] **Query String**: The part of a URL that contains the query.

[xix] **User-agent:** The type of browser the client is using.

[xx] **Host:** A computer on a network that communicates with other hosts.

[xxi] **Content type:** The media type of a resource such as text, audio, or an image.

[xxii] **Content length:** The number of bytes of data being sent in a response.

[xxiii] **Response header:** contains metadata about the response, such as a time stamp, caching control, security info, content type, and the number of bytes in the response body.

[xxiv] **Response body:** the data from the requested resource is sent back to the client.

[xxv] **Response status:** the return code that communicates the result of the request's status to the client.

[xxvi] **JSON**: "JavaScript Object Notation" is a format for storing and transporting data, usually as a way to send data from a service on a server to the client. It consists of key-value pairs and is self-describing. The format of JSON data is the same as the code for creating JavaScript objects, making it easy to convert this data into JavaScript objects but can be written in any programming language. JSON has 3 data types: scalars (numbers, strings, Booleans, null), arrays, and objects.

[xxvii] **Payload**: The *payload* is the data in the body of a response being transported from a server to the client due to an API request.

[xxviii] **GET**: The *GET method* is used as a *request* that retrieves a representation of a *resource*. GET() should never modify a resource and only return a representation of the requested resource.

[xxix] **POST**: HTTP method that sends data to the server to create a resource and should return 201_CREATED status code.

[xxx] **PUT**: HTTP method that updates a resource or replaces an existing one. Calling *PUT* multiple times in a row does not have side effects, whereas *POST* does. It should return a 200_OK code if the resource exists and can be updated or return 404_NOT_FOUND code if the resource doesn't exist.

[xxxi] **DELETE**: HTTP method that deletes a *resource* and returns 204_NO_CONTENT if the resource exists and can be deleted by the server or if the resource cannot be found, which means it has already been deleted.

[xxxii] **PATCH**: HTTP method that applies partial modifications to a resource.

[xxxiii] **Idempotent**: Describes an element of a set that remains unchanged when making multiple identical requests.

[xxxiv] **URL Path Parameter:** Passed into the operation by the client as a variable in the URL's path.

[xxxv] **URL Query Parameter:** Contains key-value pairs, usually in JSON format, and are separated from the path by a '?'. If there are multiple key-value pairs, they should be separated by an '&'. The query can be used to pass in a filter to be applied to the results that are returned by the operation.

[xxxvi] **Header Parameter**: contains additional metadata about the query, such as identifying the client that is calling the operation.

[xxxvii] **Stateless**: All requests from a client to a server for resources happen in isolation from each other. The server is unaware of the application's state on the client, so this information needs to be passed with every request.

[xxxviii] **Cacheable**: The ability to store data on the client so that data can be used in a future request.

[xxxix] **Middleware**: Middleware is software that sits between applications, databases, or services and allows those different technologies to communicate.