UNIT-2

Contents

- REST API
- Protocols
- Web Servers
- Apache Modules
- Client-Server Relationship

API

- What is API??
- Key Concept
- Work
- Types of API
- Example

What is API?

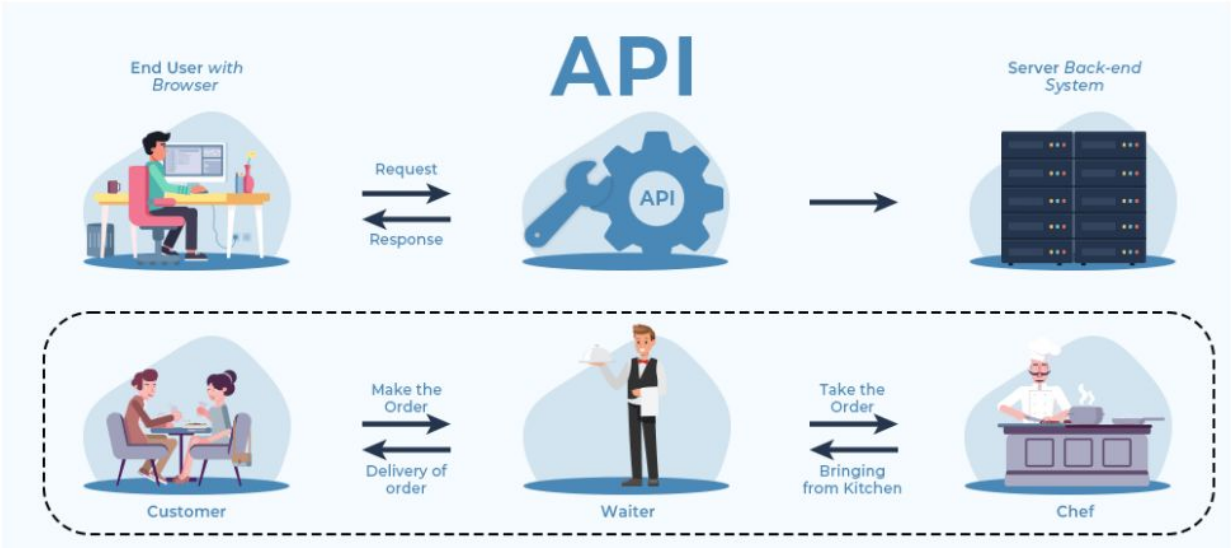
- API stands for Application Programming Interface
- It is a set of rules and protocols that allows different software applications to communicate and exchange data with each other.

Key concepts of API

- Communication
- Data Exchange
- Abstraction

API works…

- Request
- Processing
- Response



Difference between types of API

API Type	Focus	Common Use Cases	Data Format	Flexibility
REST	Resources & Actions	Web APIs, Mobile Apps	JSON	High
GraphQL	Client-driven Data	Mobile Apps, Complex Systems	GraphQL	Very High
SOAP	Complex Integrations	Enterprise, Financial	XML	Moderate
WebSockets	Real-time Communication	Chat, Live Updates	Varies	High

Examples of API

- **Web APIs**
  - The most common type, using HTTP for communication over the internet.
- **OS APIs**
  - Allow applications to interact with the operating system (e.g., file system access).
- **Library APIs**
  - Provide access to specific functionalities within a programming language.

REST API

- Web services are purpose-built web servers that support the needs of a site or any other application.
- Client programs use application programming interfaces (APIs) to communicate with web services.
- Generally speaking, an API exposes a set of data and functions to facilitate interactions between computer programs and allow them to exchange information.

REST API

- As depicted in Figure 1-1, a Web API is the face of a web service, directly listening and responding to client requests.
- The REST architectural style is commonly applied to the design of APIs for modern web services.
- A Web API conforming to the REST architectural style is a REST API.

REST API

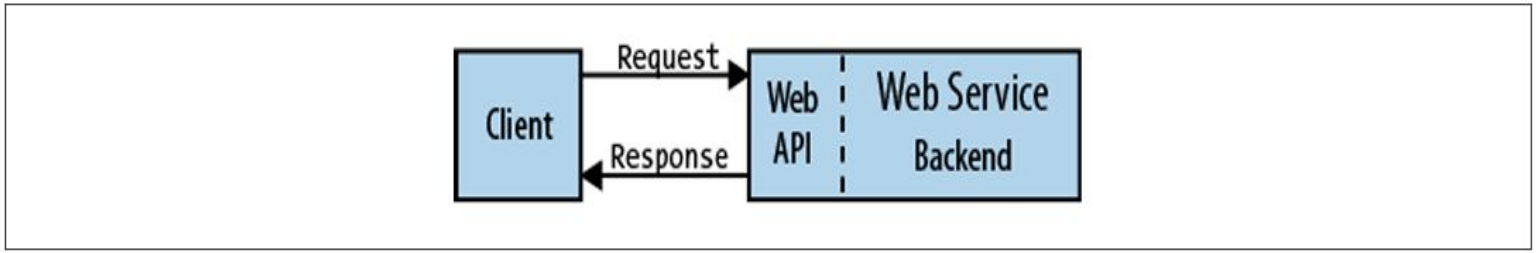


Figure 1-1. Web API

REST(Representational State Transfer) API

- Having a REST API makes a web service “RESTful.”
- A REST API consists of an assembly of interlinked resources.
- This set of resources is known as the REST API’s resource model. Well-designed REST APIs can attract client developers to use web services.

REST API

- In today’s open market where rival web services are competing for attention, an aesthetically pleasing REST API design is a must-have feature.

REST API…

- An API that follows the REST standard is called a RESTful API.
- REST Standard ..
  - Uniform Interface
  - Client-Server
  - Stateless
  - Cacheable
  - Layered System
  - Code on Demand

REST API…

❖ Uniform Interface

- ➢ **Identification of Resources:** Every resource within the application is identified by a unique global identifier (URI).
- ➢ **Representation of Resources:** Resources are represented in a standardized format (e.g., JSON, XML) that is independent of the underlying data storage or implementation.

REST API…

- **Self-descriptive Messages:** Messages exchanged between the client and server contain enough information to understand and process the message.
- **Hypermedia as the Engine of Application State (HATEOAS):** The server provides links within responses that allow the client to discover available actions and navigate the application state.

REST API…

❖ Client Server

- ➢ The client and server are distinct and independent components.
- ➢ The client is responsible for user interface and application logic.
- ➢ The server is responsible for data storage and processing.
- ➢ This separation of concerns improves scalability and maintainability.

REST API…

- **Stateless**
  - Each request from the client to the server must contain all the necessary information for the server to understand and process the request.
  - The server does not store any session state about the client between requests.
  - This makes the system more scalable and reliable, as the server does not need to maintain state for each client.

REST API…

- **Cacheable**
  - Responses from the server can be cached by intermediate components (e.g., browsers, proxies) to improve performance and reduce load on the server.
  - The server must indicate which responses are cacheable and for how long.

REST API…

❖ Layered System

- ➢ The architecture is composed of multiple layers of components (e.g., clients, proxies, servers).
- ➢ Each layer interacts only with adjacent layers.
- ➢ This improves modularity and allows for the introduction of new components without affecting other layers.

REST API…

- **Code on Demand(Optional)**
  - The server can optionally transfer executable code to the client to extend or customize client functionality.
  - This is often used for applets or browser extensions.

Concept Resource Modelling

Resource: A fundamental unit of data within your API. It represents a specific entity or collection of entities.

Examples:

- Single user: /user/123
- Collection of users: /users
- Order: /orders/456
- Product: /products

Concept Resource Modelling

HTTP Methods:

- **GET:** Retrieve data (read-only)
- **POST:** Create new resources
- **PUT:** Update an entire resource
- **PATCH:** Update specific fields of a resource
- **DELETE:** Remove a resource

Concept Resource Modelling

HTTP Methods:

Example: “Account” resource example there can be a options like open, close,deposit. All these three words can be a verbs for given noun ‘Account’.

In given example, we can use HTTP methods.

Concept Resource Modelling

HTTP Methods:

- If API consumer request for open account than HTTP POST method will be apply.
- If API consumer request for deposit than HTTP PUT/PATCH/POST will be apply.

Concept Resource Modelling

Status Codes: Indicate the success or failure of an API request.

Examples:

- **200 OK:** Successful request
- **404 Not Found:** Resource not found
- **400 Bad Request:** Invalid request data
- **500 Internal Server Error:** Server-side issue

Concept Resource Modelling

Representations: The format in which resources are exchanged. **Common formats:**

- JSON (JavaScript Object Notation) - Most widely used
- XML

URI Paths…

Filtering and Sorting: Allow clients to filter and sort data using query parameters:

- /users?limit=10&offset=0
- /users?name=John&age=30
- /users?sort=name

URI Paths…

Filtering and Sorting…

- /users?limit=10&offset=0

/users: This is the base path of the resource being requested. It indicates that we are interested in retrieving information about users.

?: This character separates the base path from the query string.

URI Paths…

limit=10: This parameter specifies that the server should return a maximum of 10 user records in the response. This is useful for paginating results, especially when dealing with large datasets.

&: This character separates multiple parameters within the query string.

URI Paths…

offset=0: This parameter indicates that the server should start returning results from the very beginning of the user list.

URI Paths…

- **Pagination:** Implement pagination for large datasets to improve performance and user experience.
- **Versioning:**
  - Use URL versioning (e.g., /api/v1/users) or header-based versioning.
  - Allows you to make changes without breaking existing clients.

Usage of HTTP

- Use *GET* requests to retrieve resource representation/information only – and not modify it in any way. As GET requests do not change the resource's state, these are said to be safe methods.
- Additionally, GET APIs should be idempotent. 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.

Usage of HTTP

- If the Request-URI refers to a data-producing process, it is the produced data that shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.

```
HTTP GET http://www.appdomain.com/users
HTTP GET http://www.appdomain.com/users?size=20&page=5
HTTP GET http://www.appdomain.com/users/123
HTTP GET http://www.appdomain.com/users/123/address
```

Usage of HTTP

- Use POST APIs to create new subordinate resources, e.g., a file is subordinate to a directory containing it or a row is subordinate to a database table.
- When talking strictly about REST, POST methods are used to create a new resource into the collection of resources.

Usage of HTTP

- Responses to this method are not cacheable unless the response includes appropriate Cache-Control or Expires header fields.
- POST is neither safe nor idempotent, and invoking two identical POST requests will result in two different resources containing the same information (except resource ids).

Usage of HTTP

POST API Response Codes

- Ideally, if a resource has been created on the origin server, the response SHOULD be HTTP response code 201 (Created) and contain an entity that describes the status of the request and refers to the new resource, and a Location header.

Usage of HTTP

POST API Response Codes

- Many times, the action performed by the POST method might not result in a resource that can be identified by a URI. In this case, either HTTP response code 200 (OK) or 204 (No Content) is the appropriate response status.

```
HTTP POST http://www.appdomain.com/users
HTTP POST http://www.appdomain.com/users/123/accounts
```

Usage of HTTP

- Use PUT APIs primarily to update an existing resource (if the resource does not exist, then API may decide to create a new resource or not).
- If the request passes through a cache and the Request-URI identifies one or more currently cached entities, those entries SHOULD be treated as stale. Responses to PUT method are not cacheable.

Usage of HTTP

PUT API Response Codes

- 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.
- 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.

Usage of HTTP

PUT API Response Codes

```
HTTP PUT http://www.appdomain.com/users/123
HTTP PUT http://www.appdomain.com/users/123/accounts/456
```

Usage of HTTP

- As the name applies, DELETE APIs delete the resources (identified by the Request-URI).
- DELETE operations are idempotent. If you DELETE a resource, it’s removed from the collection of resources.
- Some may argue that it makes the DELETE method non-idempotent. It’s a matter of discussion and personal opinion.

Usage of HTTP

- If the request passes through a cache and the Request-URI identifies one or more currently cached entities, those entries SHOULD be treated as stale. Responses to this method are not cacheable.

Usage of HTTP

DELETE API Response Codes

- A successful response of DELETE requests SHOULD be an HTTP response code 200 (OK) if the response includes an entity describing the status.
- The status should be 202 (Accepted) if the action has been queued.
- The status should be 204 (No Content) if the action has

Usage of HTTP

DELETE API Response Codes

been performed but the response does not include an entity.

- Repeatedly calling DELETE API on that resource will not change the outcome – however, calling DELETE on a resource a second time will return a 404 (NOT FOUND) since it was already removed.

Usage of HTTP

DELETE API Response Codes

```
HTTP DELETE http://www.appdomain.com/users/123
HTTP DELETE http://www.appdomain.com/users/123/accounts/456
```

URI

- A URI, or Uniform Resource Identifier, is a string of characters that uniquely identifies a resource. URIs can be used to identify resources on the internet, such as web pages, images, and videos.
- REST APIs use Uniform Resource Identifiers (URIs) to address resources. On today's Web, URI designs range from masterpieces that clearly communicate the API's resource model like:

URI Paths

http://api.example.restapi.org/france/paris/louvre/leonardo-da-vinci/mona-lisa

to those that are much harder for people to understand, such as: http://api.example.restapi.org/68dd0-a9d3-11e0-9f1c-0800200c9a66

URI Format

- The rules presented in this section pertain to the format of a URI. RFC 3986* defines the generic URI syntax as shown below:

```
URI = scheme "://" authority "/" path [ "?" query ] [ "#" fragment ]
```

URI Format Rules

Rule: Forward slash separator (/) must be used to indicate a hierarchical relationship

- The forward slash (/) character is used in the path portion of the URI to indicate a hierarchical relationship between resources. For example:

```
http://api.canvas.restapi.org/shapes/polygons/quadrilaterals/squares
```

URI Format Rules

Rule: Hyphens (-) should be used to improve the readability of URIs

- To make your URIs easy for people to scan and interpret, use the hyphen (-) character to improve the readability of names in long path segments. Anywhere you would use a space or hyphen in English, you should use a hyphen in a URI. For example:

```
http://api.example.restapi.org/blogs/mark-masse/entries/this-is-my-first-post
```

URI Format Rules

Rule: Underscores (_) should not be used in URIs

- Text viewer applications (browsers, editors, etc.) often underline URIs to provide a visual cue that they are clickable.
- Depending on the application's font, the underscore (_) character can either get partially obscured or completely hidden by this underlining. To avoid this confusion, use hyphens (-) instead of underscores.

URI Format Rules

Rule: Lowercase letters should be preferred in URI paths

- When convenient, lowercase letters are preferred in URI paths since capital letters can sometimes cause problems. RFC 3986 defines URIs as case-sensitive except for the scheme and host components. For example:

URI Format Rules

Rule: Lowercase letters should be preferred in URI paths

```
http://api.example.restapi.org/my-folder/my-doc  ❶
HTTP://API.EXAMPLE.RESTAPI.ORG/my-folder/my-doc  ❷
http://api.example.restapi.org/My-Folder/my-doc  ❸
```

❶ This URI is fine.

❷ The URI format specification (RFC 3986) considers this URI to be identical to URI #1.

❸ This URI is *not* the same as URIs 1 and 2, which may cause unnecessary confusion.

URI Format Rules

Rule: File extensions should not be included in URIs

- On the Web, the period (.) character is commonly used to separate the file name and extension portions of a URI.
- A REST API should not include artificial file extensions in URIs to indicate the format of a message's entity body.
- Instead, they should rely on the media type, as communicated through the Content-Type header, to determine how to process the body's content.

URI Format Rules

Rule: File extensions should not be included in URIs

```
http://api.college.restapi.org/students/3248234/transcripts/2005/fall.json  ❶
http://api.college.restapi.org/students/3248234/transcripts/2005/fall       ❷
```

❶ File extensions should *not* be used to indicate format preference.

❷ REST API clients should be encouraged to utilize HTTP's provided format selection

URI Format Rules

Rule: Consistent subdomain names should be used for your APIs

- The top-level domain and first subdomain names (e.g., soccer.restapi.org) of an API should identify its service owner.
- The full domain name of an API should add a subdomain named api. For example:

```
http://api.soccer.restapi.org
```

URI Format Rules

Rule: Consistent subdomain names should be used for your client developer portal

- Many REST APIs have an associated website, known as a developer portal, to help onboard new clients with documentation, forums, and self-service provisioning of secure API access keys.
- If an API provides a developer portal, by convention it should have a subdomain labeled developer. For example: `http://developer.soccer.restapi.org`

URI Path Design

Rule: A singular noun should be used for document names

- A URI representing a document resource should be named with a singular noun or noun phrase path segment.
- For example, the URI for a single player document would have the singular form:

```
http://api.soccer.restapi.org/leagues/seattle/teams/trebuchet/players/claudio
```

URI Path Design

Rule: A plural noun should be used for collection names

- A URI identifying a collection should be named with a plural noun, or noun phrase, path segment. A collection's name should be chosen to reflect what it uniformly contains.
- For example, the URI for a collection of player documents uses the plural noun form of its contained resources:

```
http://api.soccer.restapi.org/leagues/seattle/teams/trebuchet/players
```

URI Path Design

Rule: A plural noun should be used for store names

- A URI identifying a store of resources should be named with a plural noun, or noun phrase, as its path segment. The URI for a store of music playlists may use the plural noun form as follows:

```
http://api.music.restapi.org/artists/mikemassedotcom/playlists
```

URI Path Design

Rule: A verb or verb phrase should be used for controller names

- Like a computer program's function, a URI identifying a controller resource should be named to indicate its action. For example:

```
http://api.college.restapi.org/students/morgan/register
http://api.example.restapi.org/lists/4324/dedupe
```

URI Path Design

Rule: Variable path segments may be substituted with identity-based values

- Some URI path segments are static; meaning they have fixed names that may be chosen by the REST API's designer.
- Other URI path segments are variable, which means that they are automatically filled in with some identifier that may help provide the URI with its uniqueness.

URI Path Design

Rule: Variable path segments may be substituted with identity-based values

- The URI Template syntax allows designers to clearly name both the static and variable segments.
- A URI template includes variables that must be substituted before resolution.
- URI template example with three variables (leagueId, teamId, and playerId):

URI Path Design

Rule: Variable path segments may be substituted with identity-based values

```
http://api.soccer.restapi.org/leagues/{leagueId}/teams/{teamId}/players/{playerId}
```

- The substitution of a URI template's variables may be done by a REST API or its clients. Each substitution may use a numeric or alphanumeric identifier, as shown in the examples:

URI Path Design

Rule: Variable path segments may be substituted with identity-based values

```
http://api.soccer.restapi.org/leagues/seattle/teams/trebuchet/players/21  ❶
http://api.soccer.restapi.org/games/3fd65a60-cb8b-11e0-9572-0800200c9a66  ❷
```

❶ Conceptually, the value `21` occupies a variable path segment slot named *`playerId`*.

❷ The UUID value fills in the *`gameId`* variable.

URI Path Design

Rule: CRUD function names should not be used in URIs

- URIs should not be used to indicate that a CRUD function is performed. URIs should be used to uniquely identify resources, and they should be named as described in the rules above.
- HTTP request methods should be used to indicate which CRUD function is performed.

URI Path Design

Rule: CRUD function names should not be used in URIs

- For example, this API interaction design is preferred:

```
DELETE /users/1234
```

The following anti-patterns exemplify what *not* to do:

```
GET /deleteUser?id=1234
GET /deleteUser/1234
DELETE /deleteUser/1234
POST /users/1234/delete
```

URI Query Design

Rule: The query component of a URI may be used to filter collections or stores

- A URI's query component is a natural fit for supplying search criteria to a collection or store.
- For example:

```
GET /users ❶
GET /users?role=admin ❷
```

❶ The response message's state representation contains a listing of all the users in the collection.

❷ The response message's state representation contains a filtered list of all the users in the collection with a "role" value of `admin`.

URI Query Design

Rule: The query component of a URI should be used to paginate collection or store results

- A REST API client should use the query component to paginate collection and store results with the pageSize and pageStartIndex parameters.
- The pageSize parameter specifies the maximum number of contained elements to return in the response.
- The pageStartIndex parameter specifies the zero-based index of the first element to return in the response.

URI Query Design

Rule: The query component of a URI should be used to paginate collection or store results

- For example: `GET /users?pageSize=25&pageStartIndex=50`
- When the complexity of a client's pagination (or filtering) requirements exceeds the simple formatting capabilities of the query part, consider designing a special controller resource that partners with a collection or store.

URI Query Design

Rule: The query component of a URI should be used to paginate collection or store results

- For example, the following controller may accept more complex inputs via a request's entity body instead of the URI's query part:

```
POST /users/search
```

URI Query Design

Rule: The query component of a URI should be used to paginate collection or store results

- This design allows for custom range types and special sort orders to be easily specified in the client request message body.

Response Status code

- REST APIs use the Status-Line part of an HTTP response message to inform clients of their request's overarching result.
- RFC 2616 defines the Status-Line syntax as shown below:

  Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF

Response Status code

- HTTP defines forty standard status codes that can be used to convey the results of a client's request. The status codes are divided into the five categories.

Category	Description
1xx: Informational	Communicates transfer protocol-level information.
2xx: Success	Indicates that the client's request was accepted successfully.
3xx: Redirection	Indicates that the client must take some additional action in order to complete their request.
4xx: Client Error	This category of error status codes points the finger at clients.
5xx: Server Error	The server takes responsibility for these error status codes.

Response Status code

Rule:301 (“Moved Permanently”) should be used to relocate resources

- The 301 status code indicates that the REST API’s resource model has been significantly redesigned and a new permanent URI has been assigned to the client’s requested resource.
- The REST API should specify the new URI in the response’s Location header.

Response Status code

Rule: 302 (“Found”) should not be used

- The intended semantics of the 302 response code have been misunderstood by programmers and incorrectly implemented in programs since version 1.0 of the HTTP protocol.
- The confusion centers on whether it is appropriate for a client to always automatically issue a follow-up GET request to the URI in response’s Location header, regardless of the original request’s method.

Response Status code

Rule: 302 (“Found”) should not be used

- For the record, the intent of 302 is that this automatic redirect behavior only applies if the client’s original request used either the GET or HEAD method.

Response Status code

Rule: 303 (“See Other”) should be used to refer the client to a different URI

- A 303 response indicates that a controller resource has finished its work, but instead of sending a potentially unwanted response body, it sends the client the URI of a response resource.
- This can be the URI of a temporary status message, or the URI to some already existing, more permanent, resource.

Response Status code

Rule: 303 (“See Other”) should be used to refer the client to a different URI

- The 303 status code allows a REST API to send a reference to a resource without forcing the client to download its state. Instead, the client may send a GET request to the value of the Location header.

Response Status code

Rule: 304 (“Not Modified”) should be used to preserve bandwidth

- This status code is similar to 204 (“No Content”) in that the response body must be empty.
- The key distinction is that 204 is used when there is nothing to send in the body, whereas 304 is used when there is state information associated with a resource but the client already has the most recent version of the representation.

Response Status code

Rule: 307 (“Temporary Redirect”) should be used to tell clients to resubmit the request to another URI

- HTTP/1.1 introduced the 307 status code to reiterate the originally intended semantics of the 302 (“Found”) status code.
- A 307 response indicates that the REST API is not going to process the client’s request.
- Instead, the client should resubmit the request to the URI specified by the response message’s Location header.

Response Status code

Rule: 400 (“Bad Request”) may be used to indicate nonspecific failure

- 400 is the generic client-side error status, used when no other 4xx error code is appropriate.

Response Status code

Rule: 401 (“Unauthorized”) must be used when there is a problem with the client’s credentials

- A 401 error response indicates that the client tried to operate on a protected resource without providing the proper authorization.
- It may have provided the wrong credentials or none at all.

Response Status code

Rule: 403 (“Forbidden”) should be used to forbid access regardless of authorization state

- A 403 error response indicates that the client’s request is formed correctly, but the REST API refuses to honor it.
- A 403 response is not a case of insufficient client credentials; that would be 401 (“Unauthorized”).
- REST APIs use 403 to enforce application-level permissions.
- For example, a client may be authorized to interact with

Response Status code

Rule: 403 (“Forbidden”) should be used to forbid access regardless of authorization state

some, but not all of a REST API’s resources.

- If the client attempts a resource interaction that is outside of its permitted scope, the REST API should respond with 403.

Response Status code

Rule: 404 (“Not Found”) must be used when a client’s URI cannot be mapped to a resource

- The 404 error status code indicates that the REST API can’t map the client’s URI to a resource.

Response Status code

Rule: 405 (“Method Not Allowed”) must be used when the HTTP method is not supported

- The API responds with a 405 error to indicate that the client tried to use an HTTP method that the resource does not allow.
- For instance, a read-only resource could support only GET and HEAD, while a controller resource might allow GET and POST, but not PUT or DELETE.
- A 405 response must include the Allow header, which

Response Status code

Rule: 405 (“Method Not Allowed”) must be used when the HTTP method is not supported

lists the HTTP methods that the resource supports. For example: **Allow: GET, POST**

Response Status code

Rule: 406 (“Not Acceptable”) must be used when the requested media type cannot be served

- The 406 error response indicates that the API is not able to generate any of the client’s preferred media types, as indicated by the Accept request header.
- For example, a client request for data formatted as **application/xml** will receive a 406 response if the API is only willing to format data as application/json.

Response Status code

Rule: 409 (“Conflict”) should be used to indicate a violation of resource state

- The 409 error response tells the client that they tried to put the REST API’s resources into an impossible or inconsistent state.
- For example, a REST API may return this response code when a client tries to delete a non-empty store resource.

Response Status code

Rule: 412 (“Precondition Failed”) should be used to support conditional operations

- The 412 error response indicates that the client specified one or more preconditions in its request headers, effectively telling the REST API to carry out its request only if certain conditions were met.
- A 412 response indicates that those conditions were not met, so instead of carrying out the request, the API sends this status code.

Response Status code

Rule: 415 (“Unsupported Media Type”) must be used when the media type of a request’s payload cannot be processed

- The 415 error response indicates that the API is not able to process the client’s supplied media type, as indicated by the Content-Type request header.
- For example, a client request including data formatted as **application/xml** will receive a 415 response if the API is only willing to process data formatted as application/json.

Response Status code

Rule: 500 (“Internal Server Error”) should be used to indicate API malfunction

- Most web frameworks automatically respond with this response status code whenever they execute some request handler code that raises an exception.
- A 500 error is never the client’s fault and therefore it is reasonable for the client to retry the exact same request that triggered this response, and hope to get a different response.

Representation

- A REST API commonly uses a response message's entity body to help convey the state of a request message's identified resource.
- REST APIs often employ a text-based format to represent a resource state as a set of meaningful fields.
- The most commonly used text formats are XML and JSON. XML, like HTML, organizes a document's information by nesting angle-bracketed tag pairs.

Representation

- Well-formed XML must have tag pairs that match perfectly.
- This “buddy system” of tag pairs is XML’s way of holding a document’s structure together.
- JSON uses curly brackets to hierarchically structure a document’s information.
- Most programmers are used to this style of scope expression, which makes the JSON format feel natural to folks that are oriented to think in terms of object-based structures.

Representation

Rule: JSON should be supported for resource representation

- As a format for data exchange, JSON supports lightweight and simple interoperation. JSON is a popular format that is commonly used in REST API design.
- JSON borrows some of JavaScript's good parts and benefits from seamless integration with the browser's native runtime environment.
- If there is not already a standard format for a given

Representation

Rule: JSON should be supported for resource representation

resource type (e.g., image/jpeg for JPEG-compressed image resources), a REST API should use the JSON format to structure its information.

- This rule is in regard to the JSON data format only and does not necessarily imply that the application/json media type should be used as the value of an HTTP message's Content-Type header

Representation

Rule: JSON must be well-formed

- A JSON object is an unordered set of name-value pairs.
- The JSON object syntax defines names as strings which are always surrounded by double quotes.
- This is a less lenient formatting rule than that of object literals in JavaScript, and this difference often leads to malformed JSON.

Representation

Rule: JSON must be well-formed

- The following example shows well-formed JSON with all names enclosed in double quotes.

```
{
    "firstName" : "Osvaldo",
    "lastName" : "Alonso",
    "firstNamePronunciation" : "ahs-VAHL-doe",
    "number" : 6,  ❶
    "birthDate" : "1985-11-11"  ❷
}
```

❶ JSON supports number values directly, so they do not need to be treated as strings.

❷ JSON does not support date-time values, so they are typically formatted as strings.

Representation

Rule: JSON must be well-formed

- JSON names should use mixed lower case and should avoid special characters when ever possible.
- In JavaScript, JSON names like fooBar are preferred since they allow the use of the cleaner dot notation for property access.
  - For example: **var.fooBar**

Representation

Rule: JSON must be well-formed

- Names like foo-bar require the use of JavaScript's less elegant bracket notation to access the property, such as: var["foo-bar"]

Representation

Rule: XML and other formats may optionally be used for resource representation

- JSON should be a supported representation format for clients. REST APIs may optionally support XML, HTML, and other languages as alternative formats for resource representation.
- Clients should express their desired representation using media type negotiation.

Representation

Rule: Additional envelopes must not be created

- A REST API must leverage the message “envelope” provided by HTTP. In other words, the body should contain a representation of the resource state, without any additional, transport-oriented wrappers.

HyperMedia Representation

- Much like the Web's HTML-based hyperlinks (links) and forms, REST APIs employ hypermedia within representations.
- A REST API response message's body includes links to indicate the associations and actions that are available for a given resource, in a given state.

HyperMedia Representation

- Included along with other fields of a resource's state representation, links convey the relationships between resources and offer clients a menu of resource-related actions, which are context-sensitive.
- On the Web, users click on links to navigate a universe of interconnected resources.

HyperMedia Representation

- Despite the Web’s ever-increasing number of diverse resources, a few simple and uni formly structured HTML elements convey everything the browser needs to know in order to facilitate navigation.