WHAT EXACTLY IS A WEB API?

AND HOW TO TEST THEM

Table of contents

What exactly is a web API?

The notion of an Application Program Interface (API) has been around for a long time. The concept is related to the software engineering principle of information hiding, which dates back at least to the early 1980s.

> "The term hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate only the information necessary to achieve the software function."

APIs information hiding allows for the creation of a minimal interface that is relatively stable that can be used by other software systems to access or manipulate the underlying systems or data. This allows for enhancements to the underlying systems or data without disturbing the software systems that use the API.

> "In computing, an interface is a shared boundary across which two or more separate components of a computer system exchange information. The exchange can be between software, computer hardware, peripheral devices, humans, and combinations of these."

In the context of APIs, the word *Application* refers to any software with a distinct function. *Interface* can be thought of as a contract of service between two applications. This contract defines how the two communicate with each other. Lastly their API documentation contains information on how programmers are to structure that communication.

What is REST?

REST is an acronym that stands for Representational State Transfer which is an architectural style for building web services.

The term was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation. It means that a server will respond with the representation of a resource (today, it will most often be an HTML, XML or JSON document) and that resource will contain hyperlinks that can be followed to make the state of the system change. Any such request will in turn receive the representation of a resource, and so on.

An important consequence is that the only identifier that needs to be known is the identifier of the first resource requested, and all other identifiers are discovered. This means that these identifiers can change without informing the client, and that there can only be loose coupling between client and server.

> *"In computing and systems design, a loosely coupled system is one:*
>
> 1. *In which components are weakly associated (have breakable relationships) with each other, and thus changes in one component least affect existence or performance of another component;*
> 2. *In which each of its components has, or makes use of, little or no knowledge of the definitions of other separate components."*

The REST architectural style is designed for network-based applications, specifically client-server applications. But more than that, it is designed for Internet-scale usage, so the coupling between the user agent (client) and the origin server must be as loose as possible to facilitate large-scale adoption.

These are the most popular and flexible APIs found on the web today.

Architectural constraints

Like other architectural styles, REST has its guiding principles and constraints. These principles must be satisfied if a service interface needs to be referred to as RESTful.

Client–Server

The client–server model is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients.

Separation of concerns is the principle behind the client-server constraints. By separating the user interface concerns (client) from the data storage concerns (server), portability of the user interface is thus improved across multiple platforms, but more importantly it allows components to evolve independently. While the client and the server evolve, it's crucial to make sure that the interface/contract between the client and the server does not break.

Stateless

This is a constraint to the client-server interaction. A stateless protocol is a communications protocol in which no session information is retained by the receiver, usually a server. This means that each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any context stored on the server. Session state is therefore kept entirely on the client.

Cache

A Web cache (or HTTP cache) is a system for optimizing the World Wide Web. It is implemented both client-side and server-side. Cache constraints require that the data within a response to a request be 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. The advantage of adding cache constraints is that they have the potential to partially or completely eliminate some interactions, improving efficiency, scalability

Layered system

The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond the immediate layer with which they are interacting. By restricting knowledge of the system to a single layer, a bound is placed on the overall system complexity and promotes substrate independence. Layers can be used to encapsulate legacy services and to protect new services from legacy clients, simplifying components by moving infrequently used functionality to a shared intermediary. Intermediaries can improve system scalability by enabling load balancing of services across multiple networks and processors. Also, security can be added as a layer on top of the web services, separating business logic from security logic.

Uniform interface

This constraint is fundamental to the design of any RESTful system. By applying the principle of generality to the components interface, the overall system architecture is simplified and the visibility of interactions is improved.

In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components. REST is defined by four interface constraints:

→ **Identification of resources -** The key abstraction of information in REST is a resource. REST uses a resource identifier to identify the particular resource involved in an interaction between components. The naming authority that assigned the resource identifier, making it possible to reference the resource, is responsible for maintaining the semantic validity of the mapping over time

→ **Manipulation of resources through representations -** REST components perform actions on a resource by using a representation to capture the current or intended state of that resource and transferring that representation between components. A representation consists of data, metadata describing the data, and, on occasion, metadata to describe the metadata (usually for the purpose of verifying message integrity). Metadata is in the form of name-value pairs, where the name corresponds to a standard that defines the value's structure and semantics. Response messages may include both representation metadata and resource metadata: information about the resource that is not specific to the supplied representation.

→ **Self-descriptive messages -** All REST interactions are stateless. That is, each request contains all of the information necessary for a connector to understand the request, independent of any requests that may have preceded it.

→ **Hypermedia as the engine of application state -** A client interacts with a network application whose application servers provide information dynamically through hypermedia. A REST client needs little to no prior knowledge about how to interact with an application or server beyond a generic understanding of hypermedia. The server achieves this by sending hyperlinks in the representation so that clients can dynamically discover more resources.

Code on demand

REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented. Allowing features to be downloaded after deployment improves system extensibility. However, it also reduces visibility, and thus is only an optional constraint within REST.

REST and HTTP are not the same thing

Given the new, rapidly growing demand and use case for the web, many working groups were formed to develop the Web further. One of these was the HTTP Working Group, which worked on the new requirements to support the future of the growing World Wide Web (WWW). One member of the HTTP Working Group was Roy Thomas Fielding, who was simultaneously working on a broader architectural concept called Representational State Transfer (REST).

HTTP is a contract; a communication protocol, and REST is a concept; an architectural style that can use HTTP, FTP or other communication protocols, but is widely used with HTTP simply because REST was inspired by the WWW, which largely used HTTP before REST was defined, so it's easier to implement REST API style with HTTP.

RESTful web services are just web services that follow the REST architecture.

Another misconception is the use of HTTP methods to perform CRUD operations on the server. This has nothing to do with REST itself and is simply part of the HTTP specification.

What is HTTP?

Hypertext Transfer Protocol (HTTP) is an application-layer protocol for transmitting hypermedia documents.

> *"A protocol is a system of rules that define how data is exchanged within or between computers. Communications between devices require that the devices agree on the format of the data that is being exchanged."*

Development of HTTP was initiated by Tim Berners-Lee in 1989 and summarized in a simple document describing the behavior of a client and a server using the first HTTP protocol version that was named 0.9.

Development of early HTTP Requests for Comments (RFCs) started a few years later and it was a coordinated effort by the Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C), with work later moving to the IETF.

HTTP functions as a request–response protocol in the client–server model. Each message is either a request or a response. A client constructs request messages that communicate its intentions and routes those messages toward an identified origin server. A server listens for requests, parses each message received, interprets the message semantics in relation to the identified target resource, and responds to that request with one or more response messages. The client examines received responses to see if its intentions were carried out, determining what to do next based on the status codes and content received.

HTTP is a stateless protocol, meaning that the server does not keep any data (state) between two requests.

It is the foundation of any data exchange on the Web.

Key definitions

Resources

The target of an HTTP request is called a resource. HTTP does not limit the nature of a resource; it merely defines an interface that might be used to interact with resources.

A resource is any identifiable entity (digital, physical, or abstract) present or connected to the World Wide Web. Most resources are identified by a Uniform Resource Identifier (URI).

The concept of a web resource has evolved during the Web's history, from the early notion of static addressable documents or files, to a more generic and abstract definition, now encompassing every thing or entity that can be identified, named, addressed or handled, in any way whatsoever, in the web at large, or in any networked information system. In other words, any concept that might be the target of an author's hypertext reference must fit within the definition of a resource.

Representations

A representation is information that is intended to reflect a past, current, or desired state of a given resource, in a format that can be readily communicated via the protocol. A representation consists of a set of representation metadata and a potentially unbounded stream of representation data.

Connections, clients, and servers

HTTP is a client/server protocol that operates over a reliable transport or session-layer "connection".

An HTTP "client" is a program that establishes a connection to a server for the purpose of sending one or more HTTP requests. An HTTP "server" is a program that accepts connections in order to service HTTP requests by sending HTTP responses.

The terms client and server refer only to the roles that these programs perform for a particular connection. The same program might act as a client on some connections and a server on others.

Messages

HTTP is a stateless request/response protocol for exchanging “messages” across a connection. The terms sender and recipient refer to any implementation that sends or receives a given message, respectively.

A client sends requests to a server in the form of a request message with a method and request target. The request might also contain header fields for request modifiers, client information, and representation metadata, content intended for processing in accordance with the method, and trailer fields to communicate information collected while sending the content.

A server responds to a client's request by sending one or more response messages, each including a status code. The response might also contain header fields for server information, resource metadata, and representation metadata, content to be interpreted in accordance with the status code, and trailer fields to communicate information collected while sending the content.

User agents

The term "user agent" refers to any of the various client programs that initiate a request.

The most familiar form of user agent is the general-purpose Web browser, but that's only a small percentage of implementations.

Being a user agent does not imply that there is a human user directly interacting with the software agent at the time of a request.

Origin server

The term "origin server" refers to a program that can originate authoritative responses for a given target resource.

The most familiar form of origin server are large public websites. However, like user agents being equated with browsers, it is easy to be misled into thinking that all origin servers are alike.

Caches

A "cache" is a local store of previous response messages and the subsystem that controls its message storage, retrieval, and deletion. A cache stores cacheable responses in order to reduce the response time and network bandwidth consumption on future, equivalent requests.

HTTP specifications

Specification	Title	Category
RFC9110	HTTP Semantics	Internet Standard
RFC9111	HTTP Caching	Internet Standard

HTTP header fields

HTTP uses fields to provide data in the form of extensible name/value pairs with a registered key namespace. Fields are sent and received within the header and trailer sections of messages. Field names are case-insensitive

Fields that are sent or received before the content are referred to as "header fields" or just "headers", colloquially.

The “header section” of a message consists of a sequence of header field lines. Each header field might modify or extend message semantics, describe the sender, define the content, or provide additional context.

HTTP headers let the client and the server pass additional information with an HTTP request or response. An HTTP header consists of its case-insensitive name followed by a colon (:), then by its value. Whitespace before the value is ignored. The whole header, including the value, consists of one single line, which can be quite long.

Custom proprietary headers have historically been used with an X- prefix, but this convention was deprecated in June 2012 because of the inconveniences it caused when nonstandard fields became standard in RFC 6648.

Headers can be grouped according to their contexts:

Request headers

A request header is an HTTP header that can be used in an HTTP request to provide information about the request context, so that the server can tailor the response. For example, the `accept-*` headers indicate the allowed and preferred formats of the response. Other headers can be used to supply authentication credentials (e.g. `authorization`), to control caching, or to get information about the user agent or referrer, etc.

Not all headers appearing in a request are categorized request headers by the specification.

Response headers

A response header is an HTTP header that can be used in an HTTP response and that doesn't relate to the content of the message. Response headers, like Age, Location or Server are used to give a more detailed context of the response.

Not all headers appearing in a response are categorized as response headers by the specification.

Representation headers

Describe the particular representation of the resource sent in an HTTP message body. Clients specify the formats that they prefer to be sent during content negotiation (using `accept-*` headers), and the representation headers tell the client the format of the selected representation they actually received.

Representation headers may be present in both HTTP request and response messages. If sent as a response to a HEAD request, they describe the body content that would be selected if the resource was actually requested.

Representation headers include: `content-type`, `content-encoding`, `content-language`, and `content-location`.

Payload headers

A payload header is an HTTP header that describes the payload information related to safe transport and reconstruction of the original resource representation, from one or more messages. This includes information like the length of the message payload, which part of the resource is carried in this payload (for a multi-part message), any encoding applied for transport, message integrity checks, etc.

Payload headers may be present in both HTTP request and response messages (i.e. in any message that is carrying payload data).

Payload headers include: `content-length`, `content-range`, `trailer`, and `transfer-encoding`.

HTTP methods

The request method token is the primary source of request semantics; it indicates the purpose for which the client has made this request and what is expected by the client as a successful result.

Unlike distributed objects, the standardized request methods in HTTP are not resource-specific, since uniform interfaces provide for better visibility and reuse in network-based systems (REST). Once defined, a standardized method ought to have the same semantics when applied to any resource, though each resource determines for itself whether those semantics are implemented or allowed.

HTTP is designed to be usable as an interface to distributed object systems. The request method invokes an action to be applied to a target resource in much the same way that a remote method invocation can be sent to an identified object.

The HTTP specification defines a number of standardized methods that are commonly used in HTTP messages, as outlined by the following table:

Method name	Description
GET	Transfer a current representation of the target resource.
HEAD	Same as GET, but do not transfer the response content.
POST	Perform resource-specific processing on the request content.
PUT	Replace all current representations of the target resource with the request content.
DELETE	Remove all current representations of the target resource.
CONNECT	Establish a tunnel to the server identified by the target resource.
OPTIONS	Describe the communication options for the target resource.
TRACE	Perform a message loop-back test along the path to the target resource.

The set of methods allowed by a target resource can be listed in an `allow` header field. However, the set of allowed methods can change dynamically. An origin server that receives a request method that is unrecognized or not implemented should respond with the '501 (Not Implemented)' status code. An origin server that receives a request method that is recognized and implemented, but not allowed for the target resource, should respond with the '405 (Method Not Allowed)' status code.

Common methods properties

Safe methods

Request methods are considered safe if their defined semantics are essentially read-only. The client does not request, and does not expect, any state change on the origin server as a result of applying a safe method to a target resource. Likewise, reasonable use of a safe method is not expected to cause any harm, loss of property, or unusual burden on the origin server.

This definition of safe methods does not prevent an implementation from including behavior that is potentially harmful, that is not entirely read-only, or that causes side effects while invoking a safe method. What is important, however, is that the client did not request that additional behavior and cannot be held accountable for it.

Of the request methods defined by the HTTP specification, the GET, HEAD, OPTIONS, and TRACE methods are defined to be safe.

Idempotent methods

A request method is considered idempotent if the intended effect on the server of multiple identical requests with that method is the same as the effect for a single such request. Of the request methods defined by the HTTP specification, PUT, DELETE, and all safe request methods are idempotent.

Like the definition of safe, the idempotent property only applies to what has been requested by the user; a server is free to log each request separately, retain a revision control history, or implement other non-idempotent side effects for each idempotent request.

Idempotent methods are distinguished because the request can be repeated automatically if a communication failure occurs before the client is able to read the server's response. For example, if a client sends a PUT request and the underlying connection is closed before any response is received, then the client can establish a new connection and retry the idempotent request. It knows that repeating the request will have the same intended effect, even if the original request succeeded, though the response might differ.

Method definitions

GET

The GET method requests transfer of a current selected representation for the target resource. A successful response reflects the quality of "sameness" identified by the target URI. Hence, retrieving identifiable information via HTTP is usually performed by making a GET request on an identifier associated with the potential for providing that information in a ‘200 (OK)’ response.

HEAD

The HEAD method is identical to GET except that the server must not send content in the response. HEAD is used to obtain metadata about the selected representation without transferring its representation data, often for the sake of testing hypertext links or finding recent modifications.

POST

The POST method requests that the target resource process the representation enclosed in the request according to the resource's own specific semantics. For example, POST is used for the following functions (among others):

→ Providing a block of data, such as the fields entered into an HTML form, to a data-handling process;

→ Posting a message to a bulletin board, newsgroup, mailing list, blog, or similar group of articles;

→ Creating a new resource that has yet to be identified by the origin server; and

→ Appending data to a resource's existing representation(s).

If one or more resources has been created on the origin server as a result of successfully processing a POST request, the origin server should send a ‘201 (Created)’ response containing a Location header field that provides an identifier for the primary resource created and a representation that describes the status of the request while referring to the new resource(s).

If the result of processing a POST would be equivalent to a representation of an existing resource, an origin server may redirect the user agent to that resource by sending a '303 (See Other)' response with the existing resource's identifier in the Location field.

PUT

The PUT method requests that the state of the target resource be created or replaced with the state defined by the representation enclosed in the request message content.

A successful PUT of a given representation would suggest that a subsequent GET on that same target resource will result in an equivalent representation being sent in a '200 (OK)' response.

If the target resource does not have a current representation and the PUT successfully creates one, then the origin server must inform the user agent by sending a 201 (Created) response. If the target resource does have a current representation and that representation is successfully modified in accordance with the state of the enclosed representation, then the origin server must send either a '200 (OK)' or a '204 (No Content)' response to indicate successful completion of the request.

The fundamental difference between the POST and PUT methods is highlighted by the different intent for the enclosed representation. The target resource in a POST request is intended to handle the enclosed representation according to the resource's own semantics, whereas the enclosed representation in a PUT request is defined as replacing the state of the target resource.

DELETE

The DELETE method requests that the origin server remove the association between the target resource and its current functionality.

If the target resource has one or more current representations, they might or might not be destroyed by the origin server, and the associated storage might or might not be reclaimed, depending entirely on the nature of the resource and its implementation by the origin server.

Relatively few resources allow the DELETE method — its primary use is for remote authoring environments, where the user has some direction regarding its effect. For example, a resource that was previously created using a PUT request, or identified via the Location

header field after a 201 (Created) response to a POST request, might allow a corresponding DELETE request to undo those actions.

CONNECT

The CONNECT method requests that the recipient establish a tunnel to the destination origin server identified by the request target and, if successful, thereafter restrict its behavior to blind forwarding of data, in both directions, until the tunnel is closed. Tunnels are commonly used to create an end-to-end virtual connection, through one or more proxies, which can then be secured using TLS (Transport Layer Security).

CONNECT is intended for use in requests to a proxy. The recipient can establish a tunnel either by directly connecting to the server identified by the request target or, if configured to use another proxy, by forwarding the CONNECT request to the next inbound proxy. An origin server may accept a CONNECT request, but most origin servers do not implement CONNECT.

OPTIONS

The OPTIONS method requests information about the communication options available for the target resource, at either the origin server or an intervening intermediary. This method allows a client to determine the options and/or requirements associated with a resource, or the capabilities of a server, without implying a resource action.

An OPTIONS request with an asterisk ("*") as the request target applies to the server in general rather than to a specific resource.

TRACE

The TRACE method requests a remote, application-level loop-back of the request message. The final recipient of the request SHOULD reflect the message received, excluding some fields described below, back to the client as the content of a 200 (OK) response.

TRACE allows the client to see what is being received at the other end of the request chain and use that data for testing or diagnostic information. The value of the `via` header field is of particular interest, since it acts as a trace of the request chain.

HTTP status codes

The status code of a response is a three-digit integer code that describes the result of the request and the semantics of the response, including whether the request was successful and what content is enclosed (if any). All valid status codes are within the range of 100 to 599, inclusive.

The first digit of the status code defines the class of response. The last two digits do not have any categorization role. There are five values for the first digit:

→ **1xx (informational) -** The request was received, continuing process

→ **2xx (successful) -** The request was successfully received, understood, and accepted

→ **3xx (redirection) -** Further action needs to be taken in order to complete the request

→ **4xx (client error) -** The request contains bad syntax or cannot be fulfilled

→ **5xx (server error) -** The server failed to fulfill an apparently valid request

HTTP status codes are extensible. A client is not required to understand the meaning of all registered status codes, though such understanding is obviously desirable. However, a client MUST understand the class of any status code, as indicated by the first digit, and treat an unrecognized status code as being equivalent to the x00 status code of that class.

Informational 1XX

The 1xx (Informational) class of status code indicates an interim response for communicating connection status or request progress prior to completing the requested action and sending a final response.

A 1xx response is terminated by the end of the header section; it cannot contain content or trailers.

→ **100 Continue -** The initial part of a request has been received and has not yet been rejected by the server. The server intends to send a final response after the request has been fully received and acted upon.

Successful 2XX

The 2xx (Successful) class of status code indicates that the client's request was successfully received, understood, and accepted.

→ **200 OK -** The request has succeeded. The content sent in a 200 response depends on the request method.

→ **201 Created -** The request has been fulfilled and has resulted in one or more new resources being created.

→ **204 No Content -** The request has been accepted for processing, but the processing has not been completed.

Redirection 3XX

The 3xx (Redirection) class of status code indicates that further action needs to be taken by the user agent in order to fulfill the request.

→ **301 Moved Permanently -** The target resource has been assigned a new permanent URI and any future references to this resource ought to use one of the enclosed URIs

→ **302 Found -** The URI of requested resource has been changed temporarily. Further changes in the URI might be made in the future. Therefore, this same URI should be used by the client in future requests.

→ **304 Not Modified -** There is no need to retransmit the requested resources. It is an implicit redirection to a cached resource.

→ **307 Temporary Redirect -** The server sends this response to direct the client to get the requested resource at another URI with the same method that was used in the prior request. This has the same semantics as the 302 Found HTTP response code, with the exception that the user agent must not change the HTTP method used

→ **308 Permanent Redirect -** The resource is now permanently located at another URI, specified by the `location` header field. This has the same semantics as the 301 Moved Permanently HTTP response code, with the exception that the user agent must not change the HTTP method used

Client Error 4XX

The 4xx (Client Error) class of status code indicates that the client seems to have erred. Except when responding to a HEAD request, the server should send a representation containing an explanation of the error situation, and whether it is a temporary or permanent condition. These status codes are applicable to any request method. User agents should display any included representation to the user.

→ **400 Bad Request -** The server cannot or will not process the request due to something that is perceived to be a client error (e.g., malformed request syntax, invalid request message framing, or deceptive request routing).

→ **401 Unauthorized -** The request has not been applied because it lacks valid authentication credentials for the target resource.

→ **403 Forbidden -** The server understood the request but refuses to fulfill it.

→ **404 Not Found -** The server did not find a current representation for the target resource or is not willing to disclose that one exists.

→ **405 Method Not Allowed -** The method received in the request-line is known by the origin server but not supported by the target resource.

→ **409 Conflict -** The request could not be completed due to a conflict with the current state of the target resource. This code is used in situations where the user might be able to resolve the conflict and resubmit the request. The server should generate content that includes enough information for a user to recognize the source of the conflict.

Server Error 5XX

The 5xx (Server Error) class of status code indicates that the server is aware that it has erred or is incapable of performing the requested method. Except when responding to a HEAD request, the server SHOULD send a representation containing an explanation of the error situation, and whether it is a temporary or permanent condition. A user agent should display any included representation to the user. These status codes are applicable to any request method.

→ **500 Internal Server Error -** The server encountered an unexpected condition that prevented it from fulfilling the request.

What is a Uniform Resource Identifier?

A Uniform Resource Identifier (URI) is an identifier consisting of a sequence of characters matching the generic URI syntax that consists of a hierarchical sequence of components referred to as the scheme, authority, path, query, and fragment.

It enables uniform identification of resources via a separately defined extensible set of naming schemes. How that identification is accomplished, assigned, or enabled is delegated to each scheme specification.

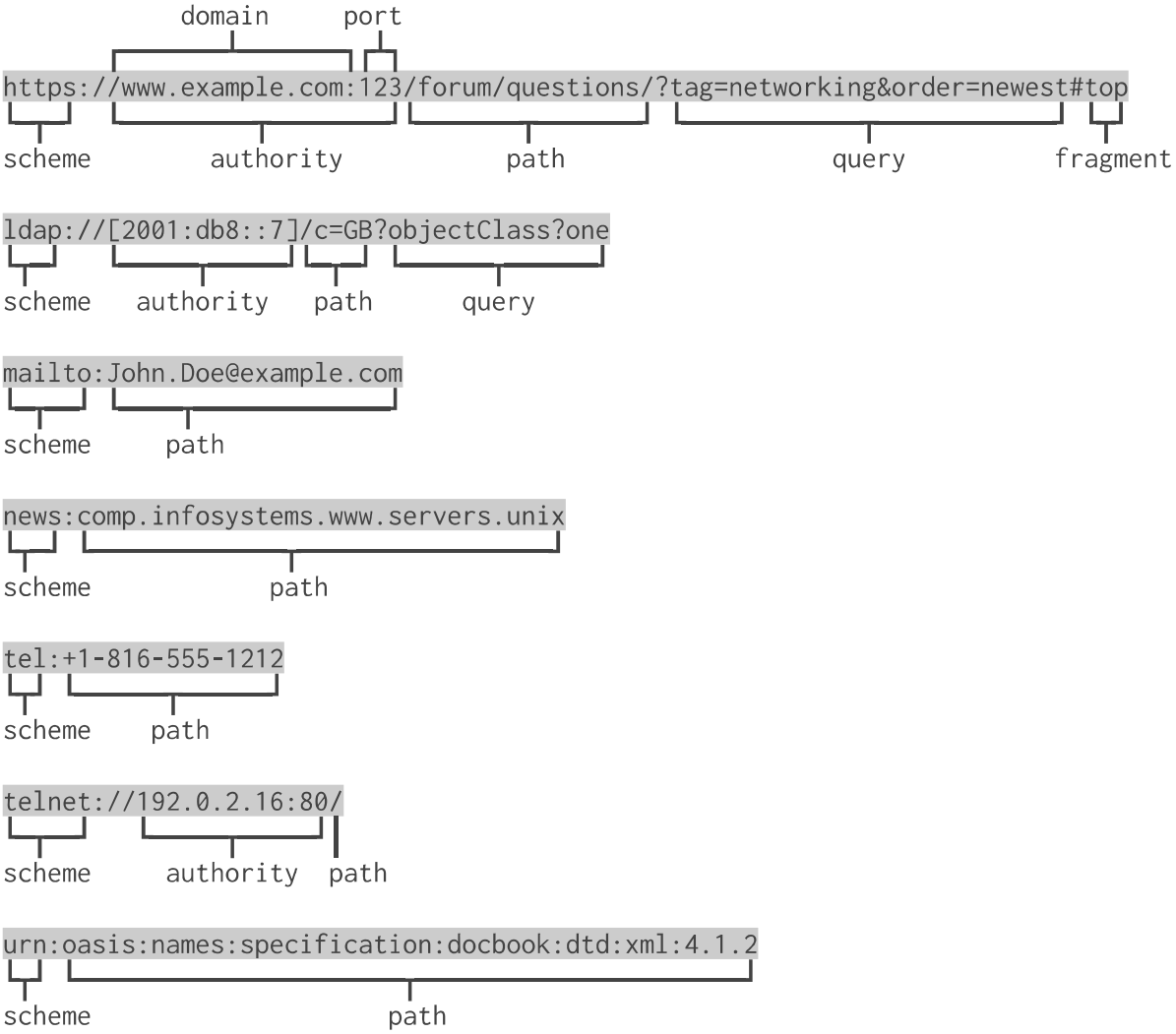
Some URIs provide only a unique name, without a means of locating or retrieving the resource or information about it, these are Uniform Resource Names (URN).

Other URIs provide a means of locating and retrieving information resources on a network; these are Uniform Resource Locators (URL).

A common misunderstanding of URIs is that they are only used to refer to accessible resources. The URI itself only provides identification; access to the resource is neither guaranteed nor implied by the presence of a URI.

Given a URI, a system may attempt to perform a variety of operations on the resource, as might be characterized by words such as "access", "update", "replace", or "find attributes". Such operations are defined by the protocols that make use of URIs.

Here are some examples of URIs:



Uniform resource name

A Uniform Resource Name (URN) is a subset of URIs that uses the urn scheme. URNs are globally unique persistent identifiers assigned within defined namespaces so they will be available for a long period of time, even after the resource which they identify ceases to exist or becomes unavailable. URNs cannot be used to directly locate an item and need not be resolvable, as they are simply templates that another parser may use to find an item.

A URN is rendered in URI (Uniform Resource Identifier) syntax. The first part of the syntax signifies the name of the resource, followed by information that is separated by a colon:

→ `urn:<nid>:<nss></nss></nid>`

The following three items are included in a URN syntax:

→ **The leading scheme -** The leading scheme is the urn:, which is case-sensitive;

→ **Namespace identifier (NID) -** The NID is the namespace identifier and can include letters, digits and dashes;

→ **Namespace-specific string (NSS) -** The NID is followed by the NSS, whose representation depends on the specified namespace and may contain American Standard Code for Information Interchange letters and digits, certain punctuation and special characters.

While other URI schemes may allow resource identifiers to be freely chosen and assigned, such is not the case for URNs. The syntactical correctness of a name starting with "urn:" is not sufficient to make it a URN. In order to ensure the name's global uniqueness, the namespace identifier (NID) is required to be registered with the Internet Assigned Numbers Authority.

Here are some examples of URNs:

→ `urn:isbn:0143039431`
  ↪ Here, the URN is used to identify a “2006 version of The Grapes of Wrath” by its International Standard Book Number (ISBN).
→ `urn:issn:1476-4687`
  ↪ Here, the scientific journal called “Nature” is identified by its International Standard Serial Number (ISSN).

Uniform resource locator

The term "Uniform Resource Locator" (URL) refers to the subset of URIs that, in addition to identifying a resource, provide a means of locating the resource by describing its primary access mechanism. URLs are used to ‘locate’ resources, by providing an abstract identification of the resource location.

With Hypertext and HTTP, URL is one of the key concepts of the Web. It is the mechanism used by browsers to retrieve any published resource on the web.

Anatomy of a URL

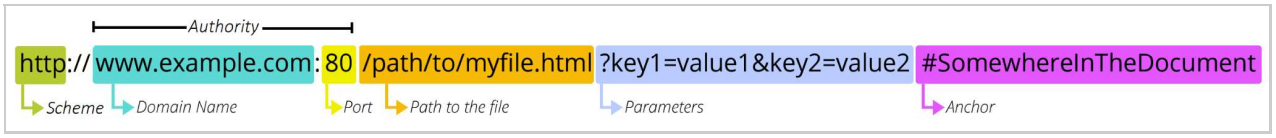
Here are some examples of URLs:

→ `https://developer.mozilla.org`

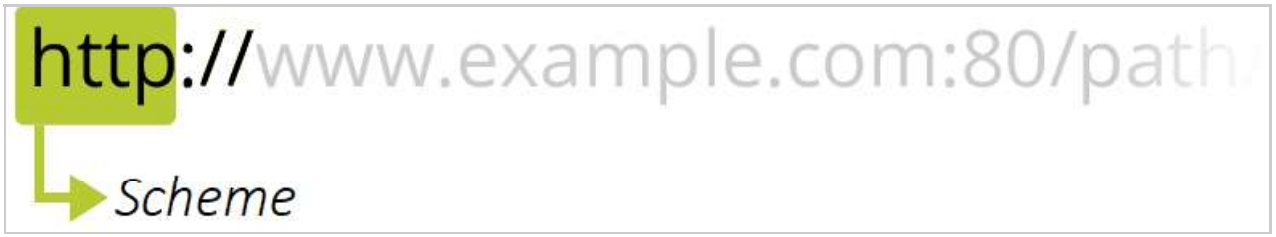
→ `https://developer.mozilla.org/en-US/docs/Learn/`

→ `https://developer.mozilla.org/en-US/search?q=name`

A URL is composed of different parts, some mandatory and others optional.



Scheme



The first part of the URL is the scheme, which indicates the protocol that the browser must use to request the resource. Usually for websites the protocol is HTTPS or HTTP (its unsecured version). Addressing web pages requires one of these two, but browsers also know how to handle other schemes.

Authority

Next follows the authority, which is separated from the scheme by the character pattern **://**. If present, the authority includes both the domain and the port, separated by a colon:

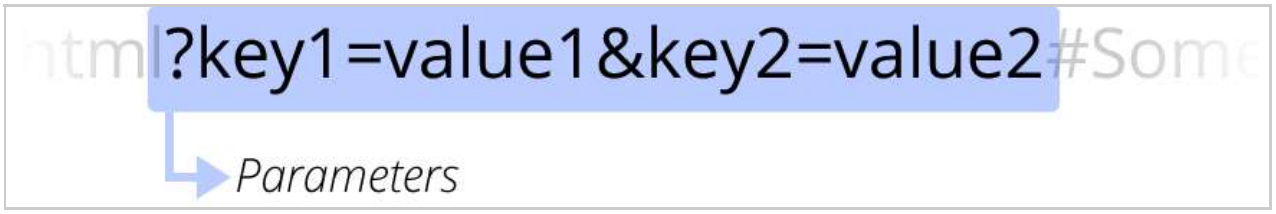
The **domain** indicates which Web server is being requested. Usually this is a domain name, but an IP address may also be used (but this is rare as it is much less convenient).

The **port** indicates the technical gate used to access the resources on the web server. It is usually omitted if the web server uses the standard ports of the HTTP protocol (80 for HTTP and 443 for HTTPS) to grant access to its resources. Otherwise it is mandatory.

Path to resource

`/path/to/myfile.html` is the path to the resource location on the Web server.

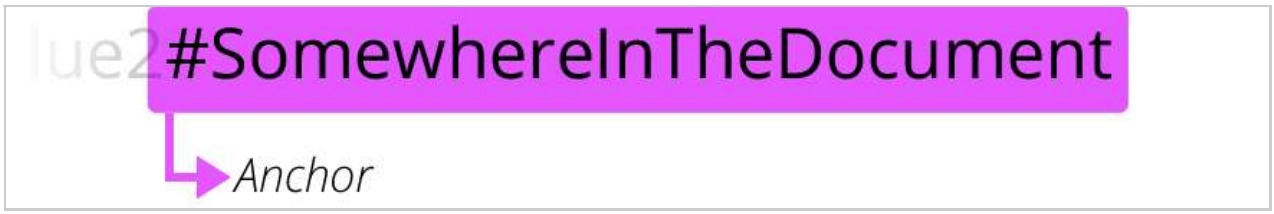
Parameters



`?key1=value1&key2=value2` are extra parameters provided to the Web server. Those parameters are a list of key/value pairs separated with the & symbol. The Web server can use those parameters to do extra operations before returning the resource. Each Web server has

its own rules regarding parameters, and the only reliable way to know if a specific Web server is handling parameters is by reviewing the server documentation.

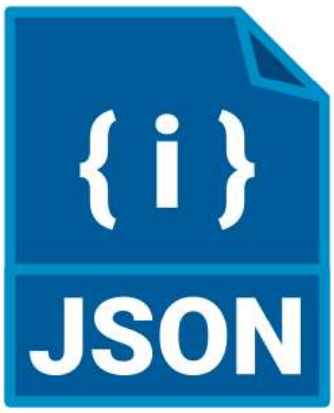
Anchor



`#SomewhereInTheDocument` is an anchor to another part of the resource itself. An anchor represents a sort of "bookmark" inside the resource, giving the browser the directions to show the content positioned at that specified location. On an HTML document, for example, the browser will scroll to the point where the anchor is defined; on a video or audio document, the browser will try to go to the time the anchor represents. It is worth noting that the part after the #, also known as the fragment identifier, is never sent to the server with the request.

URIs specifications

Specification	Title	Category
RFC3986	Uniform Resource Identifier (URI): Generic Syntax	Standards Track
RFC8141	Uniform Resource Names (URNs)	Standards Track
RFC1738	Uniform Resource Locators (URL)	Standards Track



What is JSON?

JSON (JavaScript Object Notation) grew out of a need for a real-time server-to-browser session communication protocol without using browser plugins such as Flash or Java applets, the dominant methods used in the early 2000s.

JSON is a lightweight data-interchange format. It is easy for humans to read and write and for machines to parse and generate. It is based on a subset of the JavaScript programming language. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers.

JSON is built on two structures:

→ **A collection of name/value pairs** - In most languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
→ **An ordered list of values** - In most languages, this is realized as an array, vector, list, or sequence.

These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.

It is used primarily to transmit data between a server and web application, as an alternative to XML.

In JSON, values must be one of the following data types:

Strings

A string begins and ends with quotation marks. All Unicode characters may be placed within the quotation marks. Any character may be escaped.

```JavaScript
{
	"name": "John"
}
```

Numbers

The representation of numbers is similar to that used in most programming languages. A number is represented in base 10 using decimal digits. It contains an integer component that may be prefixed with an optional minus sign, which may be followed by a fraction part and/or an exponent part. Leading zeros are not allowed.

A fraction part is a decimal point followed by one or more digits.

An exponent part begins with the letter E in uppercase or lowercase, which may be followed by a plus or minus sign. The E and optional sign are followed by one or more digits.

```JavaScript
{
	"number_1" : 210,
	"number_2" : -210,
	"number_3" : 21.05,
	"number_4" : 1.0E+2
}
```

Objects

An object structure is represented as a pair of curly brackets surrounding zero or more unordered name/value pairs (or members). A name is a string. A single colon comes after each name, separating the name from the value. A single comma separates a value from a following name. The names within an object should be unique. When the names within an object are not unique, the behavior of software that receives such an object is unpredictable.

```JavaScript
{
	"employee": {
		"name": "John",
		"age": 30,
		"city": "New York"
	}
}
```

Arrays

An array structure is represented as square brackets surrounding zero or more values (or elements). Elements are separated by commas.

There is no requirement that the values in an array be of the same type.

```JavaScript
{
	"employees": [
		"John",
		"Anna",
		"Peter"
	]
}
```

Booleans

Booleans values can be either true or false. Boolean values are not surrounded by quotes and will be treated as string values.

```JavaScript
{
	"visibility": true
}
```

Null

Although technically not a value type, null is a special value in JSON. When there is no value to assign to a key, it can be treated as null.

null value should not be surrounded by quotes.

```JavaScript
{
	"visibility" : true,
	"popularity" : null,
	"id" : 210
}
```

As described above, JSON is a string whose format very much resembles JavaScript object literal format. A JSON value must be an object, array, number, or string, or one of the following three literal names: false; null; true.

Whitespace is allowed and ignored around or between syntactic elements (values and punctuation, but not within a string value).

This allows you to construct a data hierarchy, like so:

```JavaScript
{
	"squadName": "Super hero squad",
	"homeTown": "Metro City",
	"formed": 2016,
	"secretBase": "Super tower",
	"active": true,
	"members": [
		{
			"name": "Molecule Man",
			"age": 29,
			"secretIdentity": "Dan Jukes",
			"powers": [
				"Radiation resistance",
				"Turning tiny",
				"Radiation blast"
			]
		},
		{
			"name": "Madame Uppercut",
			"age": 39,
			"secretIdentity": "Jane Wilson",
			"powers": [
				"Million tonne punch",
				"Damage resistance",
				"Superhuman reflexes"
			]
		},
```

```
            {
                  "name": "Eternal Flame",
                  "age": 300,
                  "secretIdentity": "Unknown",
                  "powers": [
                        "Immortality",
                        "Heat Immunity",
                        "Inferno",
                        "Teleportation",
                        "Interdimensional travel"
                  ]
            }
      ]
}
```

If we loaded this string into a JavaScript program and parsed it into a variable called superHeroes for example, we could then access the data inside it using the dot/bracket notation.

> "The object name acts as the namespace — it must be entered first to access anything inside the object. Next you write a dot, then the item you want to access — this can be the name of a simple property, an item of an array property, or a call to one of the object's methods".

For example:

JavaScript

```
superHeroes.homeTown

superHeroes['members'][1]['powers'][2]
```

Other notes

→ JSON is purely a string with a specified data format; it contains only properties, no methods;

→ JSON requires double quotes to be used around strings and property names. Single quotes are not valid other than surrounding the entire JSON string;

→ Even a single misplaced comma or colon can cause a JSON file to go wrong, and not work. You should be careful to validate any data you are attempting to use;

→ JSON can actually take the form of any data type that is valid for inclusion inside JSON, not just arrays or objects. So for example, a single string or number would be valid JSON;

→ Unlike in JavaScript code in which object properties may be unquoted, in JSON only quoted strings may be used as properties.

JSON specifications

Specification	Title	Category
RFC8259	The JavaScript Object Notation (JSON) Data Interchange Format	Internet Standard

What is the JSON Schema?

JSON Schema is a vocabulary that allows you to annotate and validate JSON documents. It defines how a JSON should be structured, making it easy to ensure that a JSON is formatted correctly, and it is useful for automated testing and validating. In addition, JSON Schema provides clear human- and machine-readable documentation.

Key definitions

JSON document

A JSON document is an information resource (series of octets) described by the application/json media type. In JSON Schema, the terms JSON document, JSON text, and JSON value are interchangeable because of the data model it defines.

JSON Schema is only defined over JSON documents.

Instance

A JSON document to which a schema is applied is known as an instance.

JSON Schema is defined over application/json or compatible documents, including media types with the "+json" structured syntax suffix.

Among these, this specification defines the "application/schema-instance+json" media type which defines handling for fragments in the URI.

Instance data model

JSON Schema interprets documents according to a data model. A JSON value interpreted according to this data model is called an "instance".

An instance has one of six primitive types, and a range of possible values depending on the type:

→ **Null** - A JSON "null" value

→ **Boolean -** A "true" or "false" value, from the JSON "true" or "false" value

→ **Object -** An unordered set of properties mapping a string to an instance, from the JSON "object" value

→ **Array -** An ordered list of instances, from the JSON "array" value

→ **Number -** An arbitrary-precision, base-10 decimal number value, from the JSON "number" value

→ **String -** A string of Unicode code points, from the JSON "string" value

JSON schema objects and keywords

Object properties that are applied to the instance are called keywords, or schema keywords. Broadly speaking, keywords fall into one of five categories:

→ **Identifiers -** control schema identification through setting a URI for the schema and/or changing how the base URI is determined

→ **Assertions -** produce a boolean result when applied to an instance

→ **Annotations -** attach information to an instance for application use

→ **Applicators -** apply one or more subschemas to a particular location in the instance, and combine or modify their results

→ **Reserved locations -** do not directly affect results, but reserve a place for a specific purpose to ensure interoperability

Getting started

Introduction

The following example is by no means definitive of all the value JSON Schema can provide. For this you will need to go deep into the specification itself.

Let's pretend we're interacting with a JSON based product catalog. This catalog has a product which has:

→ An identifier: `productId`

→ A product name: `productName`

→ A selling cost for the consumer: `price`

→ An optional set of tags: `tags`

For example:

```JavaScript
{
  "productId": 1,
  "productName": "A green door",
  "price": 12.50,
  "tags": [ "home", "green" ]
}
```

While generally straightforward, the example leaves some open questions. Here are just a few of them:

→ What is `productId`?

→ Is `productName` required?

→ Can the price be zero (0)?

→ Are all of the tags string values?

Starting the schema

We start with four properties called keywords which are expressed as JSON keys.

→ The `$schema` keyword states that this schema is written according to a specific draft of the standard and used for a variety of reasons, primarily version control.

→ The `$id` keyword defines a URI for the schema, and the base URI that other URI references within the schema are resolved against.

→ The `title` and `description` annotation keywords are descriptive only. They do not add constraints to the data being validated. The intent of the schema is stated with these two keywords.

→ The `type` validation keyword defines the first constraint on our JSON data and in this case it has to be a JSON Object.

JavaScript

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/product.schema.json",
  "title": "Product",
  "description": "A product in the catalog",
  "type": "object"
}
```

We introduce the following pieces of terminology when we start the schema:

→ **Schema Keyword**: `$schema` and `$id`.

→ **Schema Annotations**: `title` and `description`.

→ **Validation Keyword**: `type`.

Defining the properties

`productId` is a numeric value that uniquely identifies a product. Since this is the canonical identifier for a product, it doesn't make sense to have a product without one, so it is required.

In JSON Schema terms, we update our schema to add:

→ The `properties` validation keyword.

→ The `productId` key.

↪ `description` schema annotation and `type` validation keyword is noted – we covered both of these in the previous section.

→ The `required` validation keyword listing `productId`.

JavaScript

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/product.schema.json",
  "title": "Product",
  "description": "A product from Acme's catalog",
  "type": "object",
  "properties": {
    "productId": {
      "description": "The unique identifier for a product",
      "type": "integer"
    }
  },
  "required": [ "productId" ]
}
```

→ `productName` is a string value that describes a product. Since there isn't much to a product without a name it also is required.

→ Since the `required` validation keyword is an array of strings we can note multiple keys as required; We now include productName.

→ There isn't really any difference between `productId` and `productName` – we include both for completeness since computers typically pay attention to identifiers and humans typically pay attention to names.

```JavaScript
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/product.schema.json",
  "title": "Product",
  "description": "A product from Acme's catalog",
  "type": "object",
  "properties": {
    "productId": {
      "description": "The unique identifier for a product",
      "type": "integer"
    },
    "productName": {
      "description": "Name of the product",
      "type": "string"
    }
  },
  "required": [ "productId", "productName" ]
}
```

Going deeper with properties

According to the store owner there are no free products. ;)

→ The `price` key is added with the usual `description` schema annotation and `type` validation keywords covered previously. It is also included in the array of keys defined by the `required` validation keyword.

→ We specify the value of `price` must be something other than zero using the `exclusiveMinimum` validation keyword.

↪ If we wanted to include zero as a valid price we would have specified the `minimum` validation keyword.

```JavaScript
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/product.schema.json",
  "title": "Product",
  "description": "A product from Acme's catalog",
  "type": "object",
  "properties": {
    "productId": {
      "description": "The unique identifier for a product",
      "type": "integer"
    },
    "productName": {
      "description": "Name of the product",
      "type": "string"
    },
    "price": {
      "description": "The price of the product",
      "type": "number",
      "exclusiveMinimum": 0
    }
  },
  "required": [ "productId", "productName", "price" ]
}
```

Next, we come to the `tags` key.

The store owner has said this:

→ If there are tags there must be at least one tag,

→ All tags must be unique; no duplication within a single product.

→ All tags must be text.

→ Tags are nice but they aren't required to be present.

Therefore:

→ The `tags` key is added with the usual annotations and keywords.

→ This time the `type` validation keyword is `array`.

→ We introduce the `items` validation keyword so we can define what appears in the array. In this case: `string` values via the `type` validation keyword.

→ The `minItems` validation keyword is used to make sure there is at least one item in the array.

→ The `uniqueItems` validation keyword notes all of the items in the array must be unique relative to one another.

→ We did not add this key to the `required` validation keyword array because it is optional.

JavaScript

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/product.schema.json",
  "title": "Product",
  "description": "A product from Acme's catalog",
  "type": "object",
  "properties": {
    "productId": {
      "description": "The unique identifier for a product",
      "type": "integer"
    },
    "productName": {
      "description": "Name of the product",
      "type": "string"
    },
    "price": {
      "description": "The price of the product",
      "type": "number",
      "exclusiveMinimum": 0
    },
    "tags": {
      "description": "Tags for the product",
      "type": "array",
      "items": {
        "type": "string"
      },
      "minItems": 1,
      "uniqueItems": true
    }
  },
  "required": [ "productId", "productName", "price" ]
}
```

Nesting data structures

Up until this point we've been dealing with a very flat schema – only one level. This section demonstrates nested data structures.

→ The `dimensions` key is added using the concepts we've previously discovered. Since the `type` validation keyword is an `object` we can use the `properties` validation keyword to define a nested data structure.

↪ We omitted the `description` annotation keyword for brevity in the example. While it's usually preferable to annotate thoroughly in this case the structure and key names are fairly familiar to most developers.

→ You will note the scope of the `required` validation keyword is applicable to the dimensions key and not beyond.

JavaScript

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/product.schema.json",
  "title": "Product",
  "description": "A product from Acme's catalog",
  "type": "object",
  "properties": {
    "productId": {
      "description": "The unique identifier for a product",
      "type": "integer"
    },
    "productName": {
      "description": "Name of the product",
      "type": "string"
    },
    "price": {
      "description": "The price of the product",
      "type": "number",
      "exclusiveMinimum": 0
    },
    "tags": {
      "description": "Tags for the product",
      "type": "array",
      "items": {
        "type": "string"
      },
      "minItems": 1,
      "uniqueItems": true
    },
    "dimensions": {
      "type": "object",
```

```
      "properties": {
        "length": {
          "type": "number"
        },
        "width": {
          "type": "number"
        },
        "height": {
          "type": "number"
        }
      },
      "required": [ "length", "width", "height" ]
    }
  },
  "required": [ "productId", "productName", "price" ]
}
```

JSON schema specifications

Specification	Title	Category
JSON schema core	JSON Schema: A Media Type for Describing JSON Documents	Internet-Draft