# Best practices for RESTful web API design

05/09/2025

A RESTful web API implementation is a web API that employs Representational State Transfer (REST) architectural principles to achieve a stateless, loosely coupled interface between a client and service. A web API that is RESTful supports the standard HTTP protocol to perform operations on resources and return representations of resources that contain hypermedia links and HTTP operation status codes.

A RESTful web API should align with the following principles:

- **Platform independence**, which means that clients can call the web API regardless of the internal implementation. To achieve platform independence, the web API uses HTTP as a standard protocol, provides clear documentation, and supports a familiar data exchange format such as JSON or XML.

- **Loose coupling**, which means that the client and the web service can evolve independently. The client doesn't need to know the internal implementation of the web service, and the web service doesn't need to know the internal implementation of the client. To achieve loose coupling in a RESTful web API, use only standard protocols and implement a mechanism that allows the client and the web service to agree on the format of the data to exchange.

This article describes best practices for designing RESTful web APIs. It also covers common design patterns and considerations for building web APIs that are easy to understand, flexible, and maintainable.

## RESTful web API design concepts

To implement a RESTful web API, you need to understand the following concepts.

- **Uniform Resource Identifier (URI):** REST APIs are designed around resources, which are any kind of object, data, or service that the client can access. Each resource is represented by a URI that uniquely identifies that resource. For example, the URI for a particular customer order might be:

  HTTP
  ```
  https://api.contoso.com/orders/1
  ```

- **Resource representation** defines how a resource that's identified by its URI is encoded and transported over the HTTP protocol in a specific format, such as XML or JSON.

Clients that want to retrieve a specific resource must use the resource's URI in the request to the API. The API returns a resource representation of the data that the URI indicates. For example, a client can make a GET request to the URI identifier `https://api.contoso.com/orders/1` to receive the following JSON body:

```JSON
{"orderId":1,"orderValue":99.9,"productId":1,"quantity":1}
```

- **Uniform interface** is how RESTful APIs achieve loose coupling between client and service implementations. For REST APIs that are built on HTTP, the uniform interface includes using standard HTTP verbs to perform operations like `GET`, `POST`, `PUT`, `PATCH`, and `DELETE` on resources.

- **Stateless request model:** RESTful APIs use a stateless request model, which means that HTTP requests are independent and might occur in any order. For this reason, keeping transient state information between requests isn't feasible. The only place where information is stored is in the resources themselves, and each request should be an atomic operation. A stateless request model supports high scalability because it doesn't need to retain any affinity between clients and specific servers. However, the stateless model can also limit scalability because of challenges with web service back-end storage scalability. For more information about strategies to scale out a data store, see Data partitioning.

- **Hypermedia links:** REST APIs can be driven by hypermedia links that are contained in each resource representation. For example, the following code block shows a JSON representation of an order. It contains links to get or update the customer that's associated with the order.

```JSON
{
  "orderID":3,
  "productID":2,
  "quantity":4,
  "orderValue":16.60,
  "links": [
    {"rel":"product","href":"https://api.contoso.com/customers/3", "ac-
tion":"GET" },
    {"rel":"product","href":"https://api.contoso.com/customers/3", "ac-
tion":"PUT" }
  ]
}
```

## Define RESTful web API resource URIs

A RESTful web API is organized around resources. To organize your API design around resources, define resource URIs that map to the business entities. When possible, base resource URIs on nouns (the resource) and not verbs (the operations on the resource).

For example, in an e-commerce system, the primary business entities might be customers and orders. To create an order, a client sends the order information in an HTTP POST request to the resource URI. The HTTP response to the request indicates whether the order creation is successful.

The URI for creating the order resource might be something like:

HTTP
```http
https://api.contoso.com/orders // Good
```

Avoid using verbs in URIs to represent operations. For example, the following URI isn't recommended:

HTTP
```http
https://api.contoso.com/create-order // Avoid
```

Entities are often grouped together into collections like customers or orders. A collection is a separate resource from the items inside the collection, so it should have its own URI. For example, the following URI might represent the collection of orders:

```
https://api.contoso.com/orders
```

After the client retrieves the collection, it can make a GET request to the URI of each item. For example, to receive information on a specific order, the client performs an HTTP GET request on the URI `https://api.contoso.com/orders/1` to receive the following JSON body as a resource representation of the internal order data:

JSON
```json
{"orderId":1,"orderValue":99.9,"productId":1,"quantity":1}
```

### Resource URI naming conventions

When you design a RESTful web API, it's important that you use the correct naming and relationship conventions for resources:

- **Use nouns for resource names.** Use nouns to represent resources. For example, use `/orders` instead of `/create-order`. The HTTP GET, POST, PUT, PATCH, and DELETE methods already imply the verbal action.

- **Use plural nouns to name collection URIs.** In general, it helps to use plural nouns for URIs that reference collections. It's a good practice to organize URIs for collections and items into a hierarchy. For example, `/customers` is the path to the customer's collection, and `/customers/5` is the path to the customer with an ID that equals 5. This approach helps keep the web API intuitive. Also, many web API frameworks can route requests based on parameterized URI paths, so you can define a route for the path `/customers/{id}`.

- **Consider the relationships between different types of resources and how you might expose these associations.** For example, the `/customers/5/orders` might represent all of the orders for customer 5. You can also approach the relationship in the other direction by representing the association from an order to a customer. In this scenario, the URI might be `/orders/99/customer`. However, extending this model too far can become cumbersome to implement. A better approach is to include links in the body of the HTTP response message so that clients can easily access related resources. Use Hypertext as the Engine of Application State (HATEOAS) to enable navigation to related resources describes this mechanism in more detail.

- **Keep relationships simple and flexible.** In more complex systems, you might be inclined to provide URIs that allow the client to navigate through several levels of relationships, such as `/customers/1/orders/99/products`. However, this level of complexity can be difficult to maintain and is inflexible if the relationships between resources change in the future. Instead, try to keep URIs relatively simple. After an application has a reference to a resource, you should be able to use this reference to find items related to that resource. You can replace the preceding query with the URI `/customers/1/orders` to find all the orders for customer 1, and then use `/orders/99/products` to find the products in this order.

  **Tip**

  Avoid requiring resource URIs that are more complex than *collection/item/collection*.

- **Avoid a large number of small resources.** All web requests impose a load on the web server. The more requests, the bigger the load. Web APIs that expose a large number of small resources are known as *chatty web APIs*. Try to avoid these APIs because they require a client application to send multiple requests to find all of the data that it requires. Instead, consider denormalizing the data and combining related information into bigger resources that can be retrieved via a single request. However, you still need to balance this approach against the overhead of fetching data that the client doesn't need. Large object retrieval can increase the latency of a request and incur more bandwidth costs. For more information about these performance antipatterns, see Chatty I/O and Extraneous Fetching.

- **Avoid creating APIs that mirror the internal structure of a database.** The purpose of REST is to model business entities and the operations that an application can perform on those entities. A client shouldn't be exposed to the internal implementation. For example, if your data is stored in a relational database, the web API doesn't need to expose each table as a collection of resources. This approach increases the attack surface and might result in data leakage. Instead, think of the web API as an abstraction of the database. If necessary, introduce a mapping layer between the database and the web API. This layer ensures that client applications are isolated from changes to the underlying database schema.

**Tip**

It might not be possible to map every operation implemented by a web API to a specific resource. You can handle these *nonresource* scenarios through HTTP requests that invoke a function and return the results as an HTTP response message.

For example, a web API that implements simple calculator operations such as add and subtract can provide URIs that expose these operations as pseudo resources and use the query string to specify the required parameters. A GET request to the URI */add?operand1=99&operand2=1* returns a response message with the body containing the value 100.

However, you should use these forms of URIs sparingly.

## Define RESTful web API methods

RESTful web API methods align with the request methods and media types defined by the HTTP protocol. This section describes the most common request methods and the media types used in RESTful web APIs.

### HTTP request methods

The HTTP protocol defines many request methods that indicate the action that you want to perform on a resource. The most common methods used in RESTful web APIs are GET, POST, PUT, PATCH, and DELETE. Each method corresponds to a specific operation. When you design a RESTful web API, use these methods in a way that is consistent with the protocol definition, the resource that's being accessed, and the action that's being performed.

It's important to remember that the effect of a specific request method should depend on whether the resource is a collection or an individual item. The following table includes some conventions that most RESTful implementations use.

> **Important**
>
> The following table uses an example e-commerce `customer` entity. A web API doesn't need to implement all of the request methods. The methods that it implements depend on the specific scenario.

Expand table

| Resource | POST | GET | PUT | DELETE |
|---|---|---|---|---|
| /customers | Create a new customer | Retrieve all customers | Bulk update of customers | Remove all customers |
| /customers/1 | Error | Retrieve the details for customer 1 | Update the details of customer 1 if it exists | Remove customer 1 |
| /customers/1/orders | Create a new order for customer 1 | Retrieve all orders for customer 1 | Bulk update of orders for customer 1 | Remove all orders for customer 1 |

## GET requests

A GET request retrieves a representation of the resource at the specified URI. The body of the response message contains the details of the requested resource.

A GET request should return one of the following HTTP status codes:

Expand table

| HTTP status code | Reason |
|---|---|
| 200 (OK) | The method has successfully returned the resource. |

| HTTP status code | Reason |
|---|---|
| 204 (No Content) | The response body doesn't contain any content, such as when a search request returns no matches in the HTTP response. |
| 404 (Not Found) | The requested resource can't be found. |

## POST requests

A POST request should create a resource. The server assigns a URI for the new resource and returns that URI to the client.

> **Important**
>
> For POST requests, a client shouldn't attempt to create its own URI. The client should submit the request to the URI of the collection, and the server should assign a URI to the new resource. If a client attempts to create its own URI and issues a POST request to a specific URI, the server returns HTTP status code 400 (BAD REQUEST) to indicate that the method isn't supported.

In a RESTful model, POST requests are used to add a new resource to the collection that the URI identifies. However, a POST request can also be used to submit data for processing to an existing resource, without the creation of any new resource.

A POST request should return one of the following HTTP status codes:

Expand table

| HTTP status code | Reason |
|---|---|
| 200 (OK) | The method has done some processing but doesn't create a new resource. The result of the operation might be included in the response body. |
| 201 (Created) | The resource was created successfully. The URI of the new resource is included in the Location header of the response. The response body contains a representation of the resource. |
| 204 (No Content) | The response body contains no content. |
| 400 (Bad Request) | The client has placed invalid data in the request. The response body can contain more information about the error or a link to a URI that provides more details. |

| HTTP status code | Reason |
|---|---|
| 405 (Method Not Allowed) | The client has attempted to make a POST request to a URI that doesn't support POST requests. |

## PUT request

A PUT request should update an existing resource if it exists or, in some cases, create a new resource if it doesn't exist. To make a PUT request:

1. The client specifies the URI for the resource and includes a request body that contains a complete representation of the resource.
2. The client makes the request.
3. If a resource that has this URI already exists, it's replaced. Otherwise, a new resource is created, if the route supports it.

PUT methods are applied to resources that are individual items, such as a specific customer, instead of collections. A server might support updates but not creation via PUT. Whether to support creation via PUT depends on whether the client can meaningfully and reliably assign a URI to a resource before it exists. If it can't, then use POST to create resources and have the server assign the URI. Then use PUT or PATCH to update the URI.

**Important**

PUT requests must be *idempotent*, which means that submitting the same request multiple times always results in the same resource being modified with the same values. If a client resends a PUT request, the results should remain unchanged. In contrast, POST and PATCH requests aren't guaranteed to be idempotent.

A PUT request should return one of the following HTTP status codes:

Expand table

| HTTP status code | Reason |
|---|---|
| 200 (OK) | The resource was updated successfully. |
| 201 (Created) | The resource was created successfully. The response body might contain a representation of the resource. |

| HTTP status code | Reason |
|---|---|
| 204 (No Content) | The resource was updated successfully, but the response body doesn't contain any content. |
| 409 (Conflict) | The request couldn't be completed because of a conflict with the current state of the resource. |

**Tip**

Consider implementing bulk HTTP PUT operations that can batch updates to multiple resources in a collection. The PUT request should specify the URI of the collection. The request body should specify the details of the resources to be modified. This approach can help reduce chattiness and improve performance.

## PATCH requests

A PATCH request performs a partial update to an existing resource. The client specifies the URI for the resource. The request body specifies a set of changes to apply to the resource. This method can be more efficient than using PUT requests because the client only sends the changes and not the entire representation of the resource. PATCH can also create a new resource by specifying a set of updates to an empty or *null* resource if the server supports this action.

With a PATCH request, the client sends a set of updates to an existing resource in the form of a patch document. The server processes the patch document to perform the update. The patch document specifies only a set of changes to apply instead of describing the entire resource. The specification for the PATCH method, RFC 5789, doesn't define a specific format for patch documents. The format must be inferred from the media type in the request.

JSON is one of the most common data formats for web APIs. The two main JSON-based patch formats are JSON patch and JSON merge patch.

JSON merge patch is simpler than JSON patch. The patch document has the same structure as the original JSON resource, but it only includes the subset of fields that should be changed or added. In addition, a field can be deleted by specifying `null` for the field value in the patch document. This specification means that merge patch isn't suitable if the original resource can have explicit null values.

For example, suppose the original resource has the following JSON representation:

JSON

```json
{
    "name":"gizmo",
    "category":"widgets",
    "color":"blue",
    "price":10
}
```

Here's a possible JSON merge patch for this resource:

JSON

```json
{
    "price":12,
    "color":null,
    "size":"small"
}
```

This merge patch tells the server to update `price`, delete `color`, and add `size`. The values for `name` and `category` aren't modified. For more information about JSON merge patch, see RFC 7396. The media type for JSON merge patch is `application/merge-patch+json`.

Merge patch isn't suitable if the original resource can contain explicit null values because of the special meaning of `null` in the patch document. The patch document also doesn't specify the order that the server should apply the updates. Whether this order matters depends on the data and the domain. JSON patch, defined in RFC 6902, is more flexible because it specifies the changes as a sequence of operations to apply, including add, remove, replace, copy, and test to validate values. The media type for JSON patch is `application/json-patch+json`.

A PATCH request should return one of the following HTTP status codes:

Expand table

| HTTP status code | Reason |
|---|---|
| 200 (OK) | The resource was updated successfully. |
| 400 (Bad Request) | Malformed patch document. |
| 409 (Conflict) | The patch document is valid, but the changes can't be applied to the resource in its current state. |
| 415 (Unsupported Media Type) | The patch document format isn't supported. |

## DELETE requests

A DELETE request removes the resource at the specified URI. A DELETE request should return one of the following HTTP status codes:

Expand table

| HTTP status code | Reason |
|---|---|
| 204 (NO CONTENT) | The resource was deleted successfully. The process has been successfully handled and the response body contains no further information. |
| 404 (NOT FOUND) | The resource doesn't exist. |

## Resource MIME types

Resource representation is how a resource that's identified by URI is encoded and transported over the HTTP protocol in a specific format, such as XML or JSON. Clients that want to retrieve a specific resource must use the URI in the request to the API. The API responds by returning a resource representation of the data indicated by the URI.

In the HTTP protocol, resource representation formats are specified by using media types, also called MIME types. For nonbinary data, most web APIs support JSON (media type = `application/json`) and possibly XML (media type = `application/xml`).

The Content-Type header in a request or response specifies the resource representation format. The following example demonstrates a POST request that includes JSON data:

HTTP

```http
POST https://api.contoso.com/orders
Content-Type: application/json; charset=utf-8
Content-Length: 57

{"Id":1,"Name":"Gizmo","Category":"Widgets","Price":1.99}
```

If the server doesn't support the media type, it should return HTTP status code 415 (Unsupported Media Type).

A client request can include an Accept header that contains a list of media types that the client accepts from the server in the response message. For example:

HTTP

```http
GET https://api.contoso.com/orders/2
```

```
Accept: application/json, application/xml
```

If the server can't match any of the listed media types, it should return HTTP status code 406 (Not Acceptable).

## Implement asynchronous methods

Sometimes a POST, PUT, PATCH, or DELETE method might require processing that takes time to complete. If you wait for completion before you send a response to the client, it might cause unacceptable latency. In this scenario, consider making the method asynchronous. An asynchronous method should return HTTP status code 202 (Accepted) to indicate that the request was accepted for processing but is incomplete.

Expose an endpoint that returns the status of an asynchronous request so that the client can monitor the status by polling the status endpoint. Include the URI of the status endpoint in the Location header of the 202 response. For example:

HTTP

```
HTTP/1.1 202 Accepted
Location: /api/status/12345
```

If the client sends a GET request to this endpoint, the response should contain the current status of the request. Optionally, it can include an estimated time to completion or a link to cancel the operation.

HTTP

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "status":"In progress",
    "link": { "rel":"cancel", "method":"delete", "href":"/api/status/12345" }
}
```

If the asynchronous operation creates a new resource, the status endpoint should return status code 303 (See Other) after the operation completes. In the 303 response, include a Location header that gives the URI of the new resource:

HTTP

```
HTTP/1.1 303 See Other
Location: /api/orders/12345
```

For more information, see Provide asynchronous support for long-running requests and Asynchronous Request-Reply pattern.

## Implement data pagination and filtering

To optimize data retrieval and reduce payload size, implement data pagination and query-based filtering in your API design. These techniques allow clients to request only the subset of data that they need, which can improve performance and reduce bandwidth usage.

- **Pagination** divides large datasets into smaller, manageable chunks. Use query parameters like `limit` to specify the number of items to return and `offset` to specify the starting point. Make sure to also provide meaningful defaults for `limit` and `offset`, such as `limit=25` and `offset=0`. For example:

  HTTP

  ```http
  GET /orders?limit=25&offset=50
  ```

  - **`limit`**: Specifies the maximum number of items to return.

    > **Tip**
    >
    > To help prevent denial-of-service attacks, consider imposing an upper limit on the number of items returned. For example, if your service sets `max-limit=25`, and a client requests `limit=1000`, your service can either return 25 items or an HTTP BAD-REQUEST error, depending on the API documentation.

  - **`offset`**: Specifies the starting index for the data.

- **Filtering** allows clients to refine the dataset by applying conditions. The API can allow the client to pass the filter in the query string of the URI:

  HTTP

  ```http
  GET /orders?minCost=100&status=shipped
  ```

  - **`minCost`**: Filters orders that have a minimum cost of 100.
  - **`status`**: Filters orders that have a specific status.

Consider the following best practices:

- **Sorting** allows clients to sort data by using a `sort` parameter like `sort=price`.

ⓘ Important

The sorting approach can have a negative effect on caching because query string parameters form part of the resource identifier that many cache implementations use as the key to cached data.

- **Field selection for client-defined projections** enable clients to specify only the fields that they need by using a `fields` parameter like `fields=id,name`. For example, you can use a query string parameter that accepts a comma-delimited list of fields, such as */orders?fields=ProductID,Quantity*.

Your API must validate the requested fields to ensure that the client is allowed to access them and won't expose fields that aren't normally available through the API.

## Support partial responses

Some resources contain large binary fields, such as files or images. To overcome problems caused by unreliable and intermittent connections and to improve response times, consider supporting the partial retrieval of large binary resources.

To support partial responses, the web API should support the Accept-Ranges header for GET requests for large resources. This header indicates that the GET operation supports partial requests. The client application can submit GET requests that return a subset of a resource, specified as a range of bytes.

Also, consider implementing HTTP HEAD requests for these resources. A HEAD request is similar to a GET request, except that it only returns the HTTP headers that describe the resource, with an empty message body. A client application can issue a HEAD request to determine whether to fetch a resource by using partial GET requests. For example:

HTTP
```
HEAD https://api.contoso.com/products/10?fields=productImage
```

Here's an example response message:

HTTP
```
HTTP/1.1 200 OK

Accept-Ranges: bytes
Content-Type: image/jpeg
Content-Length: 4580
```

The Content-Length header gives the total size of the resource, and the Accept-Ranges header indicates that the corresponding GET operation supports partial results. The client application can use this information to retrieve the image in smaller chunks. The first request fetches the first 2,500 bytes by using the Range header:

HTTP
```
GET https://api.contoso.com/products/10?fields=productImage
Range: bytes=0-2499
```

The response message indicates that this response is partial by returning HTTP status code 206. The Content-Length header specifies the actual number of bytes returned in the message body and not the size of the resource. The Content-Range header indicates which part of the resource is returned (bytes 0-2499 out of 4580):

HTTP
```
HTTP/1.1 206 Partial Content

Accept-Ranges: bytes
Content-Type: image/jpeg
Content-Length: 2500
Content-Range: bytes 0-2499/4580

[...]
```

A subsequent request from the client application can retrieve the remainder of the resource.

## Implement HATEOAS

One of the primary reasons to use REST is the ability to navigate the entire set of resources without prior knowledge of the URI schema. Each HTTP GET request should return the information necessary to find the resources related directly to the requested object through hyperlinks included in the response. The request should also be given information that describes the operations available on each of these resources. This principle is known as HATEOAS, or Hypertext as the Engine of Application State. The system is effectively a finite state machine, and the response to each request contains the information necessary to move from one state to another. No other information should be necessary.

> ⓘ **Note**
>
> There are no general-purpose standards that define how to model the HATEOAS principle. The examples in this section illustrate one possible, proprietary solution.

For example, to handle the relationship between an order and a customer, the representation of an order might include links that identify the available operations for the customer of the order. The following code block is a possible representation:

JSON

```json
{
  "orderID":3,
  "productID":2,
  "quantity":4,
  "orderValue":16.60,
  "links":[
    {
      "rel":"customer",
      "href":"https://api.contoso.com/customers/3",
      "action":"GET",
      "types":["text/xml","application/json"]
    },
    {
      "rel":"customer",
      "href":"https://api.contoso.com/customers/3",
      "action":"PUT",
      "types":["application/x-www-form-urlencoded"]
    },
    {
      "rel":"customer",
      "href":"https://api.contoso.com/customers/3",
      "action":"DELETE",
      "types":[]
    },
    {
      "rel":"self",
      "href":"https://api.contoso.com/orders/3",
      "action":"GET",
      "types":["text/xml","application/json"]
    },
    {
      "rel":"self",
      "href":"https://api.contoso.com/orders/3",
      "action":"PUT",
      "types":["application/x-www-form-urlencoded"]
    },
    {
      "rel":"self",
      "href":"https://api.contoso.com/orders/3",
      "action":"DELETE",
      "types":[]
    }]
}
```

In this example, the `links` array has a set of links. Each link represents an operation on a related entity. The data for each link includes the relationship ("customer"), the URI (`https://api.contoso.com/customers/3`), the HTTP method, and the supported MIME types. The client application needs this information to invoke the operation.

The `links` array also includes self-referencing information about the retrieved resource. These links have the relationship *self*.

The set of links that are returned can change depending on the state of the resource. The idea that hypertext is the *engine of application state* describes this scenario.

## Implement versioning

A web API doesn't remain static. As business requirements change, new collections of resources are added. As new resources are added, the relationships between resources might change, and the structure of the data in resources might be amended. Updating a web API to handle new or different requirements is a straightforward process, but you must consider the effects that such changes have on client applications that consume the web API. The developer who designs and implements a web API has full control over that API, but they don't have the same degree of control over client applications built by partner organizations. It's important to continue to support existing client applications while allowing new client applications to use new features and resources.

A web API that implements versioning can indicate the features and resources that it exposes, and a client application can submit requests that are directed to a specific version of a feature or resource. The following sections describe several different approaches, each of which has its own benefits and trade-offs.

### No versioning

This approach is the simplest and can work for some internal APIs. Significant changes can be represented as new resources or new links. Adding content to existing resources might not present a breaking change because client applications that aren't expecting to see this content ignore it.

For example, a request to the URI `https://api.contoso.com/customers/3` should return the details of a single customer that contains the `id`, `name`, and `address` fields that the client application expects:

HTTP

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
```

```
{"id":3,"name":"Fabrikam, Inc.","address":"1 Microsoft Way Redmond WA 98053"}
```

> **Note**
>
> For simplicity, the example responses shown in this section don't include HATEOAS links.

If the `DateCreated` field is added to the schema of the customer resource, then the response looks like:

HTTP

```http
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":3,"name":"Fabrikam, Inc.","dateCreated":"2025-03-
22T12:11:38.0376089Z","address":"1 Microsoft Way Redmond WA 98053"}
```

Existing client applications might continue to function correctly if they can ignore unrecognized fields. Meanwhile, new client applications can be designed to handle this new field. However, more drastic modifications to the schema of resources, including field removals or renaming, could occur. Or the relationships between resources might change. These updates can constitute breaking changes that prevent existing client applications from functioning correctly. In these scenarios, consider one of the following approaches:

- URI versioning
- Query string versioning
- Header versioning
- Media type versioning

## URI versioning

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

For example, the `address` field in the previous example is restructured into subfields that contain each constituent part of the address, such as `streetAddress`, `city`, `state`, and `zipCode`. This version of the resource can be exposed through a URI that contains a version number, such as `https://api.contoso.com/v2/customers/3`:

HTTP

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":3,"name":"Fabrikam, Inc.","dateCreated":"2025-03-
22T12:11:38.0376089Z","address":{"streetAddress":"1 Microsoft
Way","city":"Redmond","state":"WA","zipCode":98053}}
```

This versioning mechanism is simple but depends on the server to route the request to the appropriate endpoint. However, it can become unwieldy as the web API matures through several iterations and the server has to support many different versions. From a purist's point of view, in all cases, the client applications fetch the same data (customer 3), so the URI shouldn't differ according to the version. This schema also complicates the implementation of HATEOAS because all links need to include the version number in their URIs.

## Query string versioning

Instead of providing multiple URIs, you can specify the version of the resource by using a parameter within the query string appended to the HTTP request, such as `https://api.contoso.com/customers/3?version=2`. The version parameter should default to a meaningful value, like 1, if older client applications omit it.

This approach has the semantic advantage that the same resource is always retrieved from the same URI. However, this method depends on the code that handles the request to parse the query string and send back the appropriate HTTP response. This approach also complicates the implementation of HATEOAS in the same way as the URI versioning mechanism.

> **Note**
>
> Some older web browsers and web proxies don't cache responses for requests that include a query string in the URI. Uncached responses can degrade performance for web applications that use a web API and run from within an older web browser.

## Header versioning

Instead of appending the version number as a query string parameter, you can implement a custom header that indicates the version of the resource. This approach requires that the client application adds the appropriate header to any requests. However, the code that handles the client request can use a default value, like version 1, if the version header is omitted.

The following examples use a custom header named *Custom-Header*. The value of this header indicates the version of web API.

Version 1:

HTTP

```
GET https://api.contoso.com/customers/3
Custom-Header: api-version=1
```

HTTP

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":3,"name":"Fabrikam, Inc.","address":"1 Microsoft Way Redmond WA 98053"}
```

Version 2:

HTTP

```
GET https://api.contoso.com/customers/3
Custom-Header: api-version=2
```

HTTP

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":3,"name":"Fabrikam, Inc.","dateCreated":"2025-03-
22T12:11:38.0376089Z","address":{"streetAddress":"1 Microsoft
Way","city":"Redmond","state":"WA","zipCode":98053}}
```

Similar to URI versioning and query string versioning, you must include the appropriate custom header in any links to implement HATEOAS.

## Media type versioning

When a client application sends an HTTP GET request to a web server, it should use an Accept header to specify the format of the content that it can handle. Usually, the purpose of the Accept header is to allow the client application to specify whether the body of the response should be XML, JSON, or some other common format that the client can parse. However, it's possible to define custom media types that include information that lets the client application indicate which version of a resource it expects.

The following example shows a request that specifies an Accept header with the value `application/vnd.contoso.v1+json`. The `vnd.contoso.v1` element indicates to the web server that it should return version 1 of the resource. The `json` element specifies that the format of the response body should be JSON:

HTTP

```http
GET https://api.contoso.com/customers/3
Accept: application/vnd.contoso.v1+json
```

The code that handles the request is responsible for processing the Accept header and honoring it as much as possible. The client application can specify multiple formats in the Accept header, which allows the web server to choose the most appropriate format for the response body. The web server confirms the format of the data in the response body by using the Content-Type header:

HTTP

```http
HTTP/1.1 200 OK
Content-Type: application/vnd.contoso.v1+json; charset=utf-8

{"id":3,"name":"Fabrikam, Inc.","address":"1 Microsoft Way Redmond WA 98053"}
```

If the Accept header doesn't specify any known media types, the web server can generate an HTTP 406 (Not Acceptable) response message or return a message with a default media type.

This versioning mechanism is straightforward and well-suited for HATEOAS, which can include the MIME type of related data in resource links.

> ⓘ **Note**
>
> When you select a versioning strategy, implications, especially in relation to web server caching. The URI versioning and query string versioning schemas are cache-friendly because the same URI or query string combination refers to the same data each time.
>
> The header versioning and media type versioning mechanisms typically require more logic to examine the values in the custom header or the Accept header. In a large-scale environment, many clients using different versions of a web API can result in a significant amount of duplicated data in a server-side cache. This problem can become acute if a client application communicates with a web server through a proxy that implements caching and only forwards a request to the web server if it doesn't currently contain a copy of the requested data in its cache.

## Multitenant web APIs

A *multitenant* web API solution is shared by multiple tenants, such as distinct organizations that have their own groups of users.

Multitenancy significantly affects web API design because it dictates how resources are accessed and discovered across multiple tenants within a single web API. Design an API with multitenancy in mind to help avoid the need for future refactoring to implement isolation, scalability, or tenant-specific customizations.

A well-architected API should clearly define how tenants are identified in requests, whether through subdomains, paths, headers, or tokens. This structure ensures a consistent yet flexible experience for all users within the system. For more information, see Map requests to tenants in a multitenant solution.

Multitenancy affects endpoint structure, request handling, authentication, and authorization. This approach also influences how API gateways, load balancers, and back-end services route and process requests. The following strategies are common ways to achieve multitenancy in a web API.

## Use subdomain or domain-based isolation (DNS-level tenancy)

This approach routes requests by using tenant-specific domains. Wildcard domains use subdomains for flexibility and simplicity. Custom domains, which allow tenants to use their own domains, provide greater control and can be tailored to meet specific needs. Both methods rely on proper DNS configuration, including `A` and `CNAME` records, to direct traffic to the appropriate infrastructure. Wildcard domains simplify configuration, but custom domains provide a more branded experience.

Preserve the hostname between the reverse proxy and back-end services to help avoid problems like URL redirection and prevent exposing internal URLs. This method ensures correct routing of tenant-specific traffic and helps protect internal infrastructure. DNS resolution is crucial for achieving data residency and ensuring regulatory compliance.

HTTP

```http
GET https://adventureworks.api.contoso.com/orders/3
```

## Pass tenant-specific HTTP headers

Tenant information can be passed through custom HTTP headers like `X-Tenant-ID` or `X-Organization-ID` or through host-based headers like `Host` or `X-Forwarded-Host`, or it can be extracted from JSON Web Token (JWT) claims. The choice depends on the routing capabilities of your API gateway or reverse proxy, with header-based solutions requiring a Layer 7 (L7) gateway to inspect each request. This requirement adds processing overhead, which increases compute costs when traffic scales. However, header-based isolation provides key benefits. It enables centralized authentication, which simplifies security management across multitenant APIs. By using SDKs or API clients, tenant context is dynamically managed at runtime, which reduces client-side configuration complexity. Also, keeping tenant context in headers results in a cleaner, more RESTful API design by avoiding tenant-specific data in the URI.

An important consideration for header-based routing is that it complicates caching, particularly when cache layers rely solely on URI-based keys and don't account for headers. Because most caching mechanisms optimize for URI lookups, relying on headers can lead to fragmented cache entries. Fragmented entries reduce cache hits and increase back-end load. More critically, if a caching layer doesn't differentiate responses by headers, it can serve cached data that's intended for one tenant to another and create a risk of data leakage.

HTTP

```
GET https://api.contoso.com/orders/3
X-Tenant-ID: adventureworks
```

or

HTTP

```
GET https://api.contoso.com/orders/3
Host: adventureworks
```

or

HTTP

```
GET https://api.contoso.com/orders/3
Authorization: Bearer <JWT-token including a tenant-id: adventureworks claim>
```

## Pass tenant-specific information through the URI path

This approach appends tenant identifiers within the resource hierarchy and relies on the API gateway or reverse proxy to determine the appropriate tenant based on the path segment. Path-based isolation is effective, but it compromises the web API's RESTful design and introduces more complex routing logic. It often requires pattern matching or regular expressions to parse and canonicalize the URI path.

In contrast, header-based isolation conveys tenant information through HTTP headers as key-value pairs. Both approaches enable efficient infrastructure sharing to lower operational costs and enhance performance in large-scale, multitenant web APIs.

```HTTP
GET https://api.contoso.com/tenants/adventureworks/orders/3
```

## Enable distributed tracing and trace context in APIs

As distributed systems and microservice architectures become the standard, the complexity of modern architectures increases. Using headers, such as `Correlation-ID`, `X-Request-ID`, or `X-Trace-ID`, to propagate trace context in API requests is a best practice to achieve end-to-end visibility. This approach enables tracking of requests as they flow from the client to back-end services. It facilitates rapid identification of failures, monitors latency, and maps API dependencies across services.

APIs that support the inclusion of trace and context information enhance their observability level and debugging capabilities. By enabling distributed tracing, these APIs allow for a more granular understanding of system behavior and make it easier to track, diagnose, and resolve problems across complex, multiple-service environments.

```HTTP
GET https://api.contoso.com/orders/3
Correlation-ID: aaaa0000-bb11-2222-33cc-444444dddddd
```

```HTTP
HTTP/1.1 200 OK
...
Correlation-ID: aaaa0000-bb11-2222-33cc-444444dddddd

{...}
```

## Web API maturity model

In 2008, Leonard Richardson proposed what is now known as the Richardson Maturity Model (RMM) for web APIs. The RMM defines four levels of maturity for web APIs and is based on the principles of REST as an architectural approach to designing web services. In the RMM, as the level of maturity increases, the API becomes more RESTful and more closely follows the principles of REST.

The levels are:

- **Level 0:** Define one URI, and all operations are POST requests to this URI. Simple Object Access Protocol web services are typically at this level.
- **Level 1:** Create separate URIs for individual resources. This level isn't yet RESTful, but it begins to align with RESTful design.
- **Level 2:** Use HTTP methods to define operations on resources. In practice, many published web APIs align approximately with this level.
- **Level 3:** Use hypermedia (HATEOAS). This level is truly a RESTful API, according to Fielding's definition.

## OpenAPI Initiative

The OpenAPI Initiative was created by an industry consortium to standardize REST API descriptions across vendors. The standardizing specification was called Swagger before it was brought under the OpenAPI Initiative and renamed to the OpenAPI Specification (OAS).

You might want to adopt OpenAPI for your RESTful web APIs. Consider the following points:

- The OAS comes with a set of opinionated guidelines for REST API design. The guidelines are advantageous for interoperability but require you to ensure that your design conforms with the specifications.

- OpenAPI promotes a contract-first approach instead of an implementation-first approach. Contract-first means that you design the API contract (the interface) first and then write code that implements the contract.

- Tools like Swagger (OpenAPI) can generate client libraries or documentation from API contracts. For an example, see ASP.NET Core web API documentation with Swagger/OpenAPI.

## Next steps

- See detailed recommendations for designing REST APIs on Azure.
- See a checklist of items to consider when you design and implement a web API.
- Build software as a service and multitenant solution architectures on Azure.