Diseño de API web RESTful

Artículo • 18/10/2022 • 29 minutos para leer

La mayoría de las aplicaciones web modernas exponen API que los clientes pueden usar para interactuar con la aplicación. Una API web bien diseñada debe apuntar a admitir:

- **Independencia de la plataforma** . Cualquier cliente debería poder llamar a la API, independientemente de cómo se implemente la API internamente. Esto requiere el uso de protocolos estándar y tener un mecanismo mediante el cual el cliente y el servicio web puedan acordar el formato de los datos a intercambiar.

- **Evolución del servicio** . La API web debería poder evolucionar y agregar funcionalidad independientemente de las aplicaciones cliente. A medida que evoluciona la API, las aplicaciones cliente existentes deben seguir funcionando sin modificaciones. Toda la funcionalidad debe ser detectable para que las aplicaciones cliente puedan usarla por completo.

Esta guía describe los problemas que debe tener en cuenta al diseñar una API web.

¿Qué es REST?

En 2000, Roy Fielding propuso la transferencia de estado representacional (REST) como un enfoque arquitectónico para diseñar servicios web. REST es un estilo arquitectónico para construir sistemas distribuidos basados en hipermedia. REST es independiente de cualquier protocolo subyacente y no está necesariamente vinculado a HTTP. Sin embargo, las implementaciones de API REST más comunes usan HTTP como protocolo de aplicación, y esta guía se centra en el diseño de API REST para HTTP.

Una ventaja principal de REST sobre HTTP es que utiliza estándares abiertos y no vincula la implementación de la API o las aplicaciones cliente a ninguna implementación específica. Por ejemplo, un servicio web REST podría escribirse en ASP.NET y las aplicaciones cliente pueden usar cualquier lenguaje o conjunto de herramientas que pueda generar solicitudes HTTP y analizar respuestas HTTP.

Estos son algunos de los principales principios de diseño de las API RESTful que utilizan HTTP:

- REST APIs are designed around *resources*, which are any kind of object, data, or service that can be accessed by the client.

- A resource has an *identifier*, which is a URI that uniquely identifies that resource. For example, the URI for a particular customer order might be:

```
HTTP

https://adventure-works.com/orders/1
```

- Clients interact with a service by exchanging *representations* of resources. Many web APIs use JSON as the exchange format. For example, a GET request to the URI listed above might return this response body:

```
JSON

{"orderId":1,"orderValue":99.90,"productId":1,"quantity":1}
```

- REST APIs use a uniform interface, which helps to decouple the client and service implementations. For REST APIs built on HTTP, the uniform interface includes using standard HTTP verbs to perform operations on resources. The most common operations are GET, POST, PUT, PATCH, and DELETE.

- REST APIs use a stateless request model. HTTP requests should be independent and may occur in any order, so keeping transient state information between requests is not feasible. The only place where information is stored is in the resources themselves, and each request should be an atomic operation. This constraint enables web services to be highly scalable, because there is no need to retain any affinity between clients and specific servers. Any server can handle any request from any client. That said, other factors can limit scalability. For example, many web services write to a backend data store, which may be hard to scale out. For more information about strategies to scale out a data store, see Horizontal, vertical, and functional data partitioning.

- REST APIs are driven by hypermedia links that are contained in the representation. For example, the following shows a JSON representation of an order. It contains links to get or update the customer associated with the order.

```
JSON

{
    "orderID":3,
    "productID":2,
    "quantity":4,
```

```
    "orderValue":16.60,
    "links": [
        {"rel":"product","href":"https://adventure-works.com/customers/3",
"action":"GET" },
        {"rel":"product","href":"https://adventure-works.com/customers/3",
"action":"PUT" }
    ]
}
```

In 2008, Leonard Richardson proposed the following maturity model for web APIs:

- Level 0: Define one URI, and all operations are POST requests to this URI.
- Level 1: Create separate URIs for individual resources.
- Level 2: Use HTTP methods to define operations on resources.
- Level 3: Use hypermedia (HATEOAS, described below).

Level 3 corresponds to a truly RESTful API according to Fielding's definition. In practice, many published web APIs fall somewhere around level 2.

Organize the API design around resources

Focus on the business entities that the web API exposes. For example, in an e-commerce system, the primary entities might be customers and orders. Creating an order can be achieved by sending an HTTP POST request that contains the order information. The HTTP response indicates whether the order was placed successfully or not. When possible, resource URIs should be based on nouns (the resource) and not verbs (the operations on the resource).

HTTP

```
https://adventure-works.com/orders // Good

https://adventure-works.com/create-order // Avoid
```

A resource doesn't have to be based on a single physical data item. For example, an order resource might be implemented internally as several tables in a relational database, but presented to the client as a single entity. Avoid creating APIs that simply mirror the internal structure of a database. The purpose of REST is to model entities and the operations that an application can perform on those entities. A client should not be exposed to the internal implementation.

Entities are often grouped together into collections (orders, customers). A collection is a separate resource from the item within the collection, and should have its own URI. For example, the following URI might represent the collection of orders:

```
HTTP

https://adventure-works.com/orders
```

Sending an HTTP GET request to the collection URI retrieves a list of items in the collection. Each item in the collection also has its own unique URI. An HTTP GET request to the item's URI returns the details of that item.

Adopt a consistent naming convention in 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 customers collection, and `/customers/5` is the path to the customer with ID equal to 5. This approach helps to keep the web API intuitive. Also, many web API frameworks can route requests based on parameterized URI paths, so you could define a route for the path `/customers/{id}`.

Also 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 could also go in the other direction, and represent the association from an order back to a customer with a URI such as `/orders/99/customer`. However, extending this model too far can become cumbersome to implement. A better solution is to provide navigable links to associated resources in the body of the HTTP response message. This mechanism is described in more detail in the section Use HATEOAS to enable navigation to related resources.

In more complex systems, it can be tempting to provide URIs that enable a 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. Once an application has a reference to a resource, it should be possible to use this reference to find items related to that resource. The preceding query can be replaced with the URI `/customers/1/orders` to find all the orders for customer 1, and then `/orders/99/products` to find the products in this order.

Tip

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

Another factor is that all web requests impose a load on the web server. The more requests, the bigger the load. Therefore, try to avoid "chatty" web APIs that expose a large number of small resources. Such an API may require a client application to send multiple requests to find all of the data that it requires. Instead, you might want to denormalize the data and combine related information into bigger resources that can be retrieved with a single request. However, you need to balance this approach against the overhead of fetching data that the client doesn't need. Retrieving large objects can increase the latency of a request and incur additional bandwidth costs. For more information about these performance antipatterns, see Chatty I/O and Extraneous Fetching.

Avoid introducing dependencies between the web API and the underlying data sources. 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. In fact, that's probably a poor design. 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. That way, client applications are isolated from changes to the underlying database scheme.

Finally, it might not be possible to map every operation implemented by a web API to a specific resource. You can handle such *non-resource* 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 could provide URIs that expose these operations as pseudo resources and use the query string to specify the parameters required. For example, a GET request to the URI */add?operand1=99&operand2=1* would return a response message with the body containing the value 100. However, only use these forms of URIs sparingly.

Define API operations in terms of HTTP methods

The HTTP protocol defines a number of methods that assign semantic meaning to a request. The common HTTP methods used by most RESTful web APIs are:

- **GET** retrieves a representation of the resource at the specified URI. The body of the response message contains the details of the requested resource.
- **POST** creates a new resource at the specified URI. The body of the request message provides the details of the new resource. Note that POST can also be used to trigger operations that don't actually create resources.

- **PUT** either creates or replaces the resource at the specified URI. The body of the request message specifies the resource to be created or updated.
- **PATCH** performs a partial update of a resource. The request body specifies the set of changes to apply to the resource.
- **DELETE** removes the resource at the specified URI.

The effect of a specific request should depend on whether the resource is a collection or an individual item. The following table summarizes the common conventions adopted by most RESTful implementations using the e-commerce example. Not all of these requests might be implemented—it depends on the specific scenario.

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

The differences between POST, PUT, and PATCH can be confusing.

- A POST request creates a resource. The server assigns a URI for the new resource, and returns that URI to the client. In the REST model, you frequently apply POST requests to collections. The new resource is added to the collection. A POST request can also be used to submit data for processing to an existing resource, without any new resource being created.

- A PUT request creates a resource *or* updates an existing resource. The client specifies the URI for the resource. The request body contains a complete representation of the resource. If a resource with this URI already exists, it is replaced. Otherwise a new resource is created, if the server supports doing so. PUT requests are most frequently applied to resources that are individual items, such as a specific customer, rather than collections. A server might support updates but not creation via PUT. Whether to support creation via PUT depends on whether the client can meaningfully assign a URI to a resource before it exists. If not, then use POST to create resources and PUT or PATCH to update.

- 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 can be more efficient than using PUT, because the client only sends the changes, not the entire representation of the resource. Technically PATCH can also create a new resource (by specifying a set of updates to a "null" resource), if the server supports this.

PUT requests must be idempotent. If a client submits the same PUT request multiple times, the results should always be the same (the same resource will be modified with the same values). POST and PATCH requests are not guaranteed to be idempotent.

Conform to HTTP semantics

This section describes some typical considerations for designing an API that conforms to the HTTP specification. However, it doesn't cover every possible detail or scenario. When in doubt, consult the HTTP specifications.

Media types

As mentioned earlier, clients and servers exchange representations of resources. For example, in a POST request, the request body contains a representation of the resource to create. In a GET request, the response body contains a representation of the fetched resource.

In the HTTP protocol, formats are specified through the use of *media types*, also called MIME types. For non-binary 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 format of the representation. Here is an example of a POST request that includes JSON data:

HTTP

```
POST https://adventure-works.com/orders HTTP/1.1
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 the client will accept from the server in the response message. For example:

```
HTTP

GET https://adventure-works.com/orders/2 HTTP/1.1
Accept: application/json
```

If the server cannot match any of the media type(s) listed, it should return HTTP status code 406 (Not Acceptable).

GET methods

A successful GET method typically returns HTTP status code 200 (OK). If the resource cannot be found, the method should return 404 (Not Found).

If the request was fulfilled but there is no response body included in the HTTP response, then it should return HTTP status code 204 (No Content); for example, a search operation yielding no matches might be implemented with this behavior.

POST methods

If a POST method creates a new resource, it returns HTTP status code 201 (Created). The URI of the new resource is included in the Location header of the response. The response body contains a representation of the resource.

If the method does some processing but does not create a new resource, the method can return HTTP status code 200 and include the result of the operation in the response body. Alternatively, if there is no result to return, the method can return HTTP status code 204 (No Content) with no response body.

If the client puts invalid data into the request, the server should return HTTP status code 400 (Bad Request). The response body can contain additional information about the error or a link to a URI that provides more details.

PUT methods

If a PUT method creates a new resource, it returns HTTP status code 201 (Created), as with a POST method. If the method updates an existing resource, it returns either 200 (OK) or 204 (No Content). In some cases, it might not be possible to update an existing resource. In that case, consider returning HTTP status code 409 (Conflict).

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, and the request body should specify the details of the resources to be modified. This approach can help to reduce chattiness and improve performance.

PATCH methods

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 doesn't describe the whole resource, only a set of changes to apply. The specification for the PATCH method (RFC 5789) doesn't define a particular format for patch documents. The format must be inferred from the media type in the request.

JSON is probably the most common data format for web APIs. There are two main JSON-based patch formats, called *JSON patch* and *JSON merge patch*.

JSON merge patch is somewhat simpler. The patch document has the same structure as the original JSON resource, but includes just 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. (That means merge patch is not suitable if the original resource can have explicit null values.)

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

JSON

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

Here is a possible JSON merge patch for this resource:

JSON

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

This tells the server to update `price`, delete `color`, and add `size`, while `name` and `category` are not modified. For the exact details of JSON merge patch, see RFC 7396 . The media type for JSON merge patch is `application/merge-patch+json`.

Merge patch is not suitable if the original resource can contain explicit null values, due to the special meaning of `null` in the patch document. Also, the patch document doesn't specify the order that the server should apply the updates. That may or may not matter, depending on the data and the domain. JSON patch, defined in RFC 6902 , is more flexible. It specifies the changes as a sequence of operations to apply. Operations include add, remove, replace, copy, and test (to validate values). The media type for JSON patch is `application/json-patch+json`.

Here are some typical error conditions that might be encountered when processing a PATCH request, along with the appropriate HTTP status code.

Error condition	HTTP status code
The patch document format isn't supported.	415 (Unsupported Media Type)
Malformed patch document.	400 (Bad Request)
The patch document is valid, but the changes can't be applied to the resource in its current state.	409 (Conflict)

DELETE methods

If the delete operation is successful, the web server should respond with HTTP status code 204 (No Content), indicating that the process has been successfully handled, but that the response body contains no further information. If the resource doesn't exist, the web server can return HTTP 404 (Not Found).

Asynchronous operations

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

You should expose an endpoint that returns the status of an asynchronous request, so the client can monitor the status by polling the status endpoint. Include the URI of the status endpoint in the Location header of the 202 response. 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 could also 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 on how to implement this approach, see Providing asynchronous support for long-running requests and the Asynchronous Request-Reply pattern.

Filter and paginate data

Exposing a collection of resources through a single URI can lead to applications fetching large amounts of data when only a subset of the information is required. For example, suppose a client application needs to find all orders with a cost over a specific value. It might retrieve all orders from the */orders* URI and then filter these orders on the client side. Clearly this process is highly inefficient. It wastes network bandwidth and processing power on the server hosting the web API.

Instead, the API can allow passing a filter in the query string of the URI, such as */orders?minCost=n*. The web API is then responsible for parsing and handling the `minCost` parameter in the query string and returning the filtered results on the server side.

GET requests over collection resources can potentially return a large number of items. You should design a web API to limit the amount of data returned by any single request. Consider supporting query strings that specify the maximum number of items to retrieve and a starting offset into the collection. For example:

```
HTTP

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

Also consider imposing an upper limit on the number of items returned, to help prevent Denial of Service attacks. To assist client applications, GET requests that return paginated data should also include some form of metadata that indicate the total number of resources available in the collection.

You can use a similar strategy to sort data as it is fetched, by providing a sort parameter that takes a field name as the value, such as */orders?sort=ProductID*. However, this approach can have a negative effect on caching, because query string parameters form part of the resource identifier used by many cache implementations as the key to cached data.

You can extend this approach to limit the fields returned for each item, if each item contains a large amount of data. For example, you could use a query string parameter that accepts a comma-delimited list of fields, such as */orders?fields=ProductID,Quantity*.

Give all optional parameters in query strings meaningful defaults. For example, set the `limit` parameter to 10 and the `offset` parameter to 0 if you implement pagination, set the

sort parameter to the key of the resource if you implement ordering, and set the `fields` parameter to all fields in the resource if you support projections.

Support partial responses for large binary resources

A resource may contain large binary fields, such as files or images. To overcome problems caused by unreliable and intermittent connections and to improve response times, consider enabling such resources to be retrieved in chunks. To do this, 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://adventure-works.com/products/10?fields=productImage HTTP/1.1
```

Here is 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 2500 bytes by using the Range header:

```
HTTP
```

```
GET https://adventure-works.com/products/10?fields=productImage HTTP/1.1
Range: bytes=0-2499
```

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

Use HATEOAS to enable navigation to related resources

One of the primary motivations behind REST is that it should be possible to navigate the entire set of resources without requiring prior knowledge of the URI scheme. 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, and it should also be provided with 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**
>
> Currently there are no general-purpose standards that define how to model the HATEOAS principle. The examples shown 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 could include links that identify the available operations for the customer of the order. Here is a possible representation:

JSON


```
{
  "orderID":3,
  "productID":2,
  "quantity":4,
  "orderValue":16.60,
  "links":[
    {
      "rel":"customer",
      "href":"https://adventure-works.com/customers/3",
      "action":"GET",
      "types":["text/xml","application/json"]
    },
    {
      "rel":"customer",
      "href":"https://adventure-works.com/customers/3",
      "action":"PUT",
      "types":["application/x-www-form-urlencoded"]
    },
    {
      "rel":"customer",
      "href":"https://adventure-works.com/customers/3",
      "action":"DELETE",
      "types":[]
    },
    {
      "rel":"self",
      "href":"https://adventure-works.com/orders/3",
      "action":"GET",
      "types":["text/xml","application/json"]
    },
    {
      "rel":"self",
      "href":"https://adventure-works.com/orders/3",
      "action":"PUT",
      "types":["application/x-www-form-urlencoded"]
    },
    {
      "rel":"self",
      "href":"https://adventure-works.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://adventure-works.com/customers/3`), the HTTP method, and the supported MIME types. This is all the information that a client application needs to be able to invoke the operation.

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

The set of links that are returned may change, depending on the state of the resource. This is what is meant by hypertext being the "engine of application state."

Versioning a RESTful web API

It is highly unlikely that a web API will remain static. As business requirements change new collections of resources may be added, the relationships between resources might change, and the structure of the data in resources might be amended. While updating a web API to handle new or differing requirements is a relatively straightforward process, you must consider the effects that such changes will have on client applications consuming the web API. The issue is that although the developer designing and implementing a web API has full control over that API, the developer does not have the same degree of control over client applications, which may be built by third-party organizations operating remotely. The primary imperative is to enable existing client applications to continue functioning unchanged while allowing new client applications to take advantage of new features and resources.

Versioning enables a web API to 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 is the simplest approach, and may be acceptable for some internal APIs. Significant changes could be represented as new resources or new links. Adding content to existing

resources might not present a breaking change as client applications that are not expecting to see this content will ignore it.

For example, a request to the URI `https://adventure-works.com/customers/3` should return the details of a single customer containing `id`, `name`, and `address` fields expected by the client application:

```
HTTP

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

{"id":3,"name":"Contoso LLC","address":"1 Microsoft Way Redmond WA 98053"}
```

> **Note**
>
> For simplicity, the example responses shown in this section do not include HATEOAS links.

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

```
HTTP

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

{"id":3,"name":"Contoso LLC","dateCreated":"2014-09-04T12:11:38.0376089Z","ad-
dress":"1 Microsoft Way Redmond WA 98053"}
```

Existing client applications might continue functioning correctly if they are capable of ignoring unrecognized fields, while new client applications can be designed to handle this new field. However, if more radical changes to the schema of resources occur (such as removing or renaming fields) or the relationships between resources change then these may constitute breaking changes that prevent existing client applications from functioning correctly. In these situations, you should consider one of the following approaches.

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 as before, returning resources that conform to their original schema.

Extending the previous example, if the `address` field is restructured into subfields containing each constituent part of the address (such as `streetAddress`, `city`, `state`, and `zipCode`), this version of the resource could be exposed through a URI containing a version number, such as `https://adventure-works.com/v2/customers/3`:

HTTP

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

{"id":3,"name":"Contoso LLC","dateCreated":"2014-09-04T12:11:38.0376089Z","ad-
dress":{"streetAddress":"1 Microsoft Way","city":"Redmond","state":"WA","zip-
Code":98053}}
```

This versioning mechanism is very simple but depends on the server routing 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 a number of different versions. Also, from a purist's point of view, in all cases the client applications are fetching the same data (customer 3), so the URI should not really be different depending on the version. This scheme also complicates implementation of HATEOAS as all links will need to include the version number in their URIs.

Query string versioning

Rather than providing multiple URIs, you can specify the version of the resource by using a parameter within the query string appended to the HTTP request, such as `https://adventure-works.com/customers/3?version=2`. The version parameter should default to a meaningful value such as 1 if it is omitted by older client applications.

This approach has the semantic advantage that the same resource is always retrieved from the same URI, but it depends on the code that handles the request to parse the query string and send back the appropriate HTTP response. This approach also suffers from the same complications for implementing HATEOAS as the URI versioning mechanism.

> **Note**

Some older web browsers and web proxies will not cache responses for requests that include a query string in the URI. This can degrade performance for web applications that use a web API and that run from within such a web browser.

Header versioning

Rather than appending the version number as a query string parameter, you could 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, although the code handling the client request could use a default value (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://adventure-works.com/customers/3 HTTP/1.1
Custom-Header: api-version=1
```

```
HTTP

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

{"id":3,"name":"Contoso LLC","address":"1 Microsoft Way Redmond WA 98053"}
```

Version 2:

```
HTTP

GET https://adventure-works.com/customers/3 HTTP/1.1
Custom-Header: api-version=2
```

```
HTTP

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

{"id":3,"name":"Contoso LLC","dateCreated":"2014-09-04T12:11:38.0376089Z","ad-
```

```
dress":{"streetAddress":"1 Microsoft Way","city":"Redmond","state":"WA","zip-
Code":98053}}
```

As with the previous two approaches, implementing HATEOAS requires including the appropriate custom header in any links.

Media type versioning

When a client application sends an HTTP GET request to a web server it should stipulate the format of the content that it can handle by using an Accept header, as described earlier in this guidance. Frequently 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 is possible to define custom media types that include information enabling the client application to indicate which version of a resource it is expecting.

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

HTTP

```
GET https://adventure-works.com/customers/3 HTTP/1.1
Accept: application/vnd.adventure-works.v1+json
```

The code handling the request is responsible for processing the *Accept* header and honoring it as far as possible (the client application may specify multiple formats in the *Accept* header, in which case the web server can 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/1.1 200 OK
Content-Type: application/vnd.adventure-works.v1+json; charset=utf-8

{"id":3,"name":"Contoso LLC","address":"1 Microsoft Way Redmond WA 98053"}
```

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

This approach is arguably the purest of the versioning mechanisms and lends itself naturally to HATEOAS, which can include the MIME type of related data in resource links.

> **Note**
>
> When you select a versioning strategy, you should also consider the implications on performance, especially caching on the web server. The URI versioning and Query String versioning schemes are cache-friendly inasmuch as the same URI/query string combination refers to the same data each time.
>
> The Header versioning and Media Type versioning mechanisms typically require additional 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 issue can become acute if a client application communicates with a web server through a proxy that implements caching, and that only forwards a request to the web server if it does not currently hold a copy of the requested data in its cache.

Open API Initiative

The Open API Initiative was created by an industry consortium to standardize REST API descriptions across vendors. As part of this initiative, the Swagger 2.0 specification was renamed the OpenAPI Specification (OAS) and brought under the Open API Initiative.

You may want to adopt OpenAPI for your web APIs. Some points to consider:

- The OpenAPI Specification comes with a set of opinionated guidelines on how a REST API should be designed. That has advantages for interoperability, but requires more care when designing your API to conform to the specification.
- OpenAPI promotes a contract-first approach, rather than an implementation-first approach. Contract-first means you design the API contract (the interface) first and then write code that implements the contract.

- Tools like Swagger can generate client libraries or documentation from API contracts. For example, see ASP.NET Web API help pages using Swagger.

Next steps

- Microsoft REST API guidelines . Detailed recommendations for designing public REST APIs.
- Azure REST API guidelines . Design guidelines for Azure REST APIs.
- Web API checklist . A useful list of items to consider when designing and implementing a web API.
- Open API Initiative . Documentation and implementation details on Open API.