Web API implementation

06/13/2023

A carefully designed RESTful web API defines the resources, relationships, and navigation schemes that are accessible to client applications. When you implement and deploy a web API, you should consider the physical requirements of the environment hosting the web API and the way in which the web API is constructed rather than the logical structure of the data. This guidance focuses on best practices for implementing a web API and publishing it to make it available to client applications. For detailed information about web API design, see Web API design.

Processing requests

Consider the following points when you implement the code to handle requests.

GET, PUT, DELETE, HEAD, and PATCH actions should be idempotent

The code that implements these requests should not impose any side-effects. The same request repeated over the same resource should result in the same state. For example, sending multiple DELETE requests to the same URI should have the same effect, although the HTTP status code in the response messages might be different. The first DELETE request might return status code 204 (No Content), while a subsequent DELETE request might return status code 404 (Not Found).

> **Note**
>
> The article Idempotency Patterns on Jonathan Oliver's blog provides an overview of idempotency and how it relates to data management operations.

POST actions that create new resources should not have unrelated side-effects

If a POST request is intended to create a new resource, the effects of the request should be limited to the new resource (and possibly any directly related resources if there's some sort of linkage involved). For example, in an e-commerce system, a POST request that creates a new order for a customer might also amend inventory levels and generate billing information, but it should not modify information not directly related to the order or have any other side-effects on the overall state of the system.

Avoid implementing chatty POST, PUT, and DELETE operations

Avoid designing your API based on *chatty* operations. Every request comes with protocol, network, and compute overhead. For example, executing 100 smaller requests instead of one larger batch request incurs added overhead in the client, on the network, and at the resource server. Whenever possible, provide HTTP verb support for resource collections instead of just individual resources.

- A GET request to a collection can retrieve multiple resources at the same time.
- A POST request can contain the details for multiple new resources and add them all to the same collection.
- A PUT request can replace the entire set of resources in a collection.
- A DELETE request can remove an entire collection.

The Open Data Protocol (OData) support included in ASP.NET Web API 2 provides the ability to batch requests. A client application can package up several web API requests and send them to the server in a single HTTP request and receive a single HTTP response that contains the replies to each request. For more information, see Enable Batch in Web API OData Service.

Follow the HTTP specification when sending a response

A web API must return messages that contain the correct HTTP status code to enable the client to determine how to handle the result, the appropriate HTTP headers so that the client understands the nature of the result, and a suitably formatted body to enable the client to parse the result.

For example, a POST operation should return status code 201 (Created) and the response message should include the URI of the newly created resource in the Location header of the response message.

Support content negotiation

The body of a response message can contain data in a variety of formats. For example, an HTTP GET request could return data in JSON or XML format. When the client submits a request, it can include an Accept header that specifies the data formats that it can handle. These formats are specified as media types. For example, a client that issues a GET request that retrieves an image can specify an Accept header that lists the media types that the client can handle, such as `image/jpeg, image/gif, image/png`. When the web API returns the result, it should format the data by using one of these media types and specify the format in the Content-Type header of the response.

If the client doesn't specify an Accept header, then use a sensible default format for the response body. As an example, the ASP.NET Web API framework defaults to JSON for text-based data.

Provide links to support HATEOAS-style navigation and discovery of resources

The HATEOAS approach enables a client to navigate and discover resources from an initial starting point. This is achieved by using links containing URIs; when a client issues an HTTP GET request to obtain a resource, the response should contain URIs that enable a client application to quickly locate any directly related resources. For example, in a web API that supports an e-commerce solution, a customer might place many orders. When a client application retrieves the details for a customer, the response should include links that enable the client application to send HTTP GET requests that can retrieve these orders. Additionally, HATEOAS-style links should describe the other operations (POST, PUT, DELETE, and so on) that each linked resource supports together with the corresponding URI to perform each request. This approach is described in more detail in API design.

Currently there are no standards that govern the implementation of HATEOAS, but the following example illustrates one possible approach. In this example, an HTTP GET request that finds the details for a customer returns a response that includes HATEOAS links that reference the orders for that customer:

HTTP

```http
GET https://adventure-works.com/customers/2 HTTP/1.1
Accept: text/json
...
```

HTTP

```http
HTTP/1.1 200 OK
...
Content-Type: application/json; charset=utf-8
...
Content-Length: ...
{"CustomerID":2,"CustomerName":"Bert","Links":[
    {"rel":"self",
    "href":"https://adventure-works.com/customers/2",
    "action":"GET",
    "types":["text/xml","application/json"]},
    {"rel":"self",
    "href":"https://adventure-works.com/customers/2",
    "action":"PUT",
    "types":["application/x-www-form-urlencoded"]},
    {"rel":"self",
    "href":"https://adventure-works.com/customers/2",
    "action":"DELETE",
    "types":[]},
    {"rel":"orders",
    "href":"https://adventure-works.com/customers/2/orders",
    "action":"GET",
    "types":["text/xml","application/json"]},
    {"rel":"orders",
    "href":"https://adventure-works.com/customers/2/orders",
    "action":"POST",
    "types":["application/x-www-form-urlencoded"]}
]}
```

In this example, the customer data is represented by the `Customer` class shown in the following code snippet. The HATEOAS links are held in the `Links` collection property:

C#

```csharp
public class Customer
{
    public int CustomerID { get; set; }
    public string CustomerName { get; set; }
    public List<Link> Links { get; set; }
    ...
}

public class Link
{
    public string Rel { get; set; }
    public string Href { get; set; }
    public string Action { get; set; }
    public string [] Types { get; set; }
}
```

The HTTP GET operation retrieves the customer data from storage and constructs a `Customer` object, and then populates the `Links` collection. The result is formatted as a JSON response message. Each link includes the following fields:

- The relationship (`Rel`) between the object being returned and the object described by the link. In this case `self` indicates that the link is a reference back to the object itself (similar to a `this` pointer in many object-oriented languages), and `orders` is the name of a collection containing the related order information.
- The hyperlink (`Href`) for the object being described by the link in the form of a URI.
- The type of HTTP request (`Action`) that can be sent to this URI.

- The format of any data (`Types`) that should be provided in the HTTP request or that can be returned in the response, depending on the type of the request.

The HATEOAS links shown in the example HTTP response indicate that a client application can perform the following operations:

- An HTTP GET request to the URI `https://adventure-works.com/customers/2` to fetch the details of the customer (again). The data can be returned as XML or JSON.
- An HTTP PUT request to the URI `https://adventure-works.com/customers/2` to modify the details of the customer. The new data must be provided in the request message in x-www-form-urlencoded format.
- An HTTP DELETE request to the URI `https://adventure-works.com/customers/2` to delete the customer. The request doesn't expect any additional information or return data in the response message body.
- An HTTP GET request to the URI `https://adventure-works.com/customers/2/orders` to find all the orders for the customer. The data can be returned as XML or JSON.
- An HTTP POST request to the URI `https://adventure-works.com/customers/2/orders` to create a new order for this customer. The data must be provided in the request message in x-www-form-urlencoded format.

Handling exceptions

Consider the following points if an operation throws an uncaught exception.

Capture exceptions and return a meaningful response to clients

The code that implements an HTTP operation should provide comprehensive exception handling rather than letting uncaught exceptions propagate to the framework. If an exception makes it impossible to complete the operation successfully, the exception can be passed back in the response message, but it should include a meaningful description of the error that caused the exception. The exception should also include the appropriate HTTP status code rather than simply returning status code 500 for every situation. For example, if a user request causes a database update that violates a constraint (such as attempting to delete a customer that has outstanding orders), you should return status code 409 (Conflict) and a message body indicating the reason for the conflict. If some other condition renders the request unachievable, you can return status code 400 (Bad Request). You can find a full list of HTTP status codes on the Status code definitions page on the W3C website.

The code example traps different conditions and returns an appropriate response.

C#

```
[HttpDelete]
[Route("customers/{id:int}")]
public IHttpActionResult DeleteCustomer(int id)
{
    try
    {
        // Find the customer to be deleted in the repository
        var customerToDelete = repository.GetCustomer(id);

        // If there is no such customer, return an error response
        // with status code 404 (Not Found)
        if (customerToDelete == null)
        {
            return NotFound();
        }

        // Remove the customer from the repository
        // The DeleteCustomer method returns true if the customer
        // was successfully deleted
        if (repository.DeleteCustomer(id))
        {
            // Return a response message with status code 204 (No Content)
            // To indicate that the operation was successful
            return StatusCode(HttpStatusCode.NoContent);
        }
        else
        {
            // Otherwise return a 400 (Bad Request) error response
            return BadRequest(Strings.CustomerNotDeleted);
        }
    }
    catch
    {
        // If an uncaught exception occurs, return an error response
        // with status code 500 (Internal Server Error)
        return InternalServerError();
    }
}
```

Tip

Don't include information that could be useful to an attacker attempting to penetrate your API.

Many web servers trap error conditions themselves before they reach the web API. For example, if you configure authentication for a web site and the user fails to provide the correct authentication information, the web server should respond with status code 401 (Unauthorized). Once a client has been authenticated, your code can perform its own checks to verify that the client should be able access the requested resource. If this authorization fails, you should return status code 403 (Forbidden).

Handle exceptions consistently and log information about errors

To handle exceptions in a consistent manner, consider implementing a global error handling strategy across the entire web API. You should also incorporate error logging which captures the full details of each exception; this error log can contain detailed information as long as it's not made accessible over the web to clients.

Distinguish between client-side errors and server-side errors

The HTTP protocol distinguishes between errors that occur due to the client application (the HTTP 4xx status codes), and errors that are caused by a mishap on the server (the HTTP 5xx status codes). Make sure that you respect this convention in any error response messages.

Optimizing client-side data access

In a distributed environment such as that involving a web server and client applications, one of the primary sources of concern is the network. This can act as a considerable bottleneck, especially if a client application is frequently sending requests or receiving data. Therefore you should aim to minimize the amount of traffic that flows across the network. Consider the following points when you implement the code to retrieve and maintain data:

Support client-side caching

The HTTP 1.1 protocol supports caching in clients and intermediate servers through which a request is routed by the use of the Cache-Control header. When a client application sends an HTTP GET request to the web API, the response can include a Cache-Control header that indicates whether the data in the body of the response can be safely cached by the client or an intermediate server through which the request has been routed, and for how long before it should expire and be considered out-of-date.

The following example shows an HTTP GET request and the corresponding response that includes a Cache-Control header:

HTTP

```http
GET https://adventure-works.com/orders/2 HTTP/1.1
```

HTTP

```
HTTP/1.1 200 OK
...
Cache-Control: max-age=600, private
Content-Type: text/json; charset=utf-8
Content-Length: ...
{"orderID":2,"productID":4,"quantity":2,"orderValue":10.00}
```

In this example, the Cache-Control header specifies that the data returned should be expired after 600 seconds, and is only suitable for a single client and must not be stored in a shared cache used by other clients (it's *private*). The Cache-Control header could specify *public* rather than *private* in which case the data can be stored in a shared cache, or it could specify *no-store* in which case the data must **not** be cached by the client. The following code example shows how to construct a Cache-Control header in a response message:

C#

```
public class OrdersController : ApiController
{
    ...
    [Route("api/orders/{id:int:min(0)}")]
    [HttpGet]
    public IHttpActionResult FindOrderByID(int id)
    {
        // Find the matching order
        Order order = ...;
        ...
        // Create a Cache-Control header for the response
        var cacheControlHeader = new CacheControlHeaderValue();
        cacheControlHeader.Private = true;
        cacheControlHeader.MaxAge = new TimeSpan(0, 10, 0);
        ...

        // Return a response message containing the order and the cache con-
trol header
        OkResultWithCaching<Order> response = new OkResultWithCaching<Order>
(order, this)
        {
            CacheControlHeader = cacheControlHeader
        };
        return response;
    }
    ...
}
```

This code uses a custom `IHttpActionResult` class named `OkResultWithCaching`. This class enables the controller to set the cache header contents:

C#

```
public class OkResultWithCaching<T> : OkNegotiatedContentResult<T>
{
    public OkResultWithCaching(T content, ApiController controller)
        : base(content, controller) { }

    public OkResultWithCaching(T content, IContentNegotiator contentNegotia-
tor, HttpRequestMessage request, IEnumerable<MediaTypeFormatter> formatters)
        : base(content, contentNegotiator, request, formatters) { }

    public CacheControlHeaderValue CacheControlHeader { get; set; }
    public EntityTagHeaderValue ETag { get; set; }

    public override async Task<HttpResponseMessage>
ExecuteAsync(CancellationToken cancellationToken)
    {
        HttpResponseMessage response;
        try
        {
            response = await base.ExecuteAsync(cancellationToken);
            response.Headers.CacheControl = this.CacheControlHeader;
            response.Headers.ETag = ETag;
        }
        catch (OperationCanceledException)
        {
            response = new HttpResponseMessage(HttpStatusCode.Conflict) {Rea-
sonPhrase = "Operation was cancelled"};
        }
        return response;
    }
}
```

ⓘ **Note**

The HTTP protocol also defines the *no-cache* directive for the Cache-Control header. Rather confusingly, this directive doesn't mean "do not cache" but rather "revalidate the cached information with the server before returning it"; the data can still be cached, but it's checked each time it's used to ensure that it's still current.

Cache management is the responsibility of the client application or intermediate server, but if properly implemented it can save bandwidth and improve performance by removing the need to fetch data that has already been recently retrieved.

The *max-age* value in the Cache-Control header is only a guide and not a guarantee that the corresponding data won't change during the specified time. The web API should set the max-age to a suitable value depending on the expected volatility of the data. When this period expires, the client should discard the object from the cache.

ⓘ Note

Most modern web browsers support client-side caching by adding the appropriate cache-control headers to requests and examining the headers of the results, as described. However, some older browsers will not cache the values returned from a URL that includes a query string. This isn't usually an issue for custom client applications which implement their own cache management strategy based on the protocol discussed here.

Some older proxies exhibit the same behavior and might not cache requests based on URLs with query strings. This could be an issue for custom client applications that connect to a web server through such a proxy.

Provide ETags to optimize query processing

When a client application retrieves an object, the response message can also include an *entity tag (ETag)*. An ETag is an opaque string that indicates the version of a resource; each time a resource changes the ETag is also modified. This ETag should be cached as part of the data by the client application. The following code example shows how to add an ETag as part of the response to an HTTP GET request. This code uses the `GetHashCode` method of an object to generate a numeric value that identifies the object (you can override this method if necessary and generate your own hash using an algorithm such as MD5) :

C#

```csharp
public class OrdersController : ApiController
{
    ...
    public IHttpActionResult FindOrderByID(int id)
    {
        // Find the matching order
        Order order = ...;
        ...

        var hashedOrder = order.GetHashCode();
        string hashedOrderEtag = $"\"{hashedOrder}\"";
        var eTag = new EntityTagHeaderValue(hashedOrderEtag);

        // Return a response message containing the order and the cache con-
trol header
        OkResultWithCaching<Order> response = new OkResultWithCaching<Order>
(order, this)
        {
            ...,
            ETag = eTag
        };
        return response;
    }
    ...
}
```

The response message posted by the web API looks like this:

```HTTP
HTTP/1.1 200 OK
...
Cache-Control: max-age=600, private
Content-Type: text/json; charset=utf-8
ETag: "2147483648"
Content-Length: ...
{"orderID":2,"productID":4,"quantity":2,"orderValue":10.00}
```

> **Tip**
>
> For security reasons, don't allow sensitive data or data returned over an authenticated (HTTPS) connection to be cached.

A client application can issue a subsequent GET request to retrieve the same resource at any time, and if the resource has changed (it has a different ETag) the cached version should be discarded and the new version added to the cache. If a resource is large and requires a significant amount of bandwidth to transmit back to the client, repeated requests to fetch the same data can become inefficient. To combat this, the HTTP protocol defines the following process for optimizing GET requests that you should support in a web API:

- The client constructs a GET request containing the ETag for the currently cached version of the resource referenced in an If-None-Match HTTP header:

  ```HTTP
  GET https://adventure-works.com/orders/2 HTTP/1.1
  If-None-Match: "2147483648"
  ```

- The GET operation in the web API obtains the current ETag for the requested data (order 2 in the above example), and compares it to the value in the If-None-Match header.

- If the current ETag for the requested data matches the ETag provided by the request, the resource has not changed and the web API should return an HTTP response with an empty message body and a status code of 304 (Not Modified).

- If the current ETag for the requested data doesn't match the ETag provided by the request, then the data has changed and the web API should return an HTTP response with the new data in the message body and a status code of 200 (OK).

- If the requested data no longer exists then the web API should return an HTTP response with the status code of 404 (Not Found).

- The client uses the status code to maintain the cache. If the data has not changed (status code 304) then the object can remain cached and the client application should continue to use this version of the object. If the data has changed (status code 200) then the cached object should be discarded and the new one inserted. If the data is no longer available (status code 404) then the object should be removed from the cache.

> ⓘ **Note**
>
> If the response header contains the Cache-Control header no-store then the object should always be removed from the cache regardless of the HTTP status code.

The following code shows the `FindOrderByID` method extended to support the If-None-Match header. Notice that if the If-None-Match header is omitted, the specified order is always retrieved:

C#

```csharp
public class OrdersController : ApiController
{
    [Route("api/orders/{id:int:min(0)}")]
    [HttpGet]
    public IHttpActionResult FindOrderByID(int id)
    {
        try
        {
            // Find the matching order
            Order order = ...;

            // If there is no such order then return NotFound
            if (order == null)
            {
                return NotFound();
            }

            // Generate the ETag for the order
            var hashedOrder = order.GetHashCode();
            string hashedOrderEtag = $"\"{hashedOrder}\"";

            // Create the Cache-Control and ETag headers for the response
            IHttpActionResult response;
            var cacheControlHeader = new CacheControlHeaderValue();
            cacheControlHeader.Public = true;
            cacheControlHeader.MaxAge = new TimeSpan(0, 10, 0);
            var eTag = new EntityTagHeaderValue(hashedOrderEtag);

            // Retrieve the If-None-Match header from the request (if it ex-
ists)
            var nonMatchEtags = Request.Headers.IfNoneMatch;

            // If there is an ETag in the If-None-Match header and
            // this ETag matches that of the order just retrieved,
            // then create a Not Modified response message
            if (nonMatchEtags.Count > 0 &&
                String.CompareOrdinal(nonMatchEtags.First().Tag, hashedOrder-
Etag) == 0)
            {
                response = new EmptyResultWithCaching()
                {
                    StatusCode = HttpStatusCode.NotModified,
                    CacheControlHeader = cacheControlHeader,
                    ETag = eTag
                };
            }
            // Otherwise create a response message that contains the order
details
            else
            {
                response = new OkResultWithCaching<Order>(order, this)
                {
                    CacheControlHeader = cacheControlHeader,
                    ETag = eTag
                };
            }

            return response;
        }
        catch
        {
            return InternalServerError();
        }
    }
...
}
```

This example incorporates an additional custom `IHttpActionResult` class named `EmptyResultWithCaching`. This class simply acts as a wrapper around an `HttpResponseMessage` object that doesn't contain a response body:

C#

```
public class EmptyResultWithCaching : IHttpActionResult
{
    public CacheControlHeaderValue CacheControlHeader { get; set; }
    public EntityTagHeaderValue ETag { get; set; }
    public HttpStatusCode StatusCode { get; set; }
    public Uri Location { get; set; }
    public async Task<HttpResponseMessage> ExecuteAsync(CancellationToken
cancellationToken)
    {
        HttpResponseMessage response = new HttpResponseMessage(StatusCode);
        response.Headers.CacheControl = this.CacheControlHeader;
        response.Headers.ETag = this.ETag;
        response.Headers.Location = this.Location;
        return response;
    }
}
```

Tip

In this example, the ETag for the data is generated by hashing the data retrieved from the underlying data source. If the ETag can be computed in some other way, then the process can be optimized further and the data only needs to be fetched from the data source if it has changed. This approach is especially useful if the data is large or accessing the data source can result in significant latency (for example, if the data source is a remote database).

Use ETags to Support Optimistic Concurrency

To enable updates over previously cached data, the HTTP protocol supports an optimistic concurrency strategy. If, after fetching and caching a resource, the client application subsequently sends a PUT or DELETE request to change or remove the resource, it should include an If-Match header that references the ETag. The web API can then use this information to determine whether the resource has already been changed by another user since it was retrieved and send an appropriate response back to the client application as follows:

- The client constructs a PUT request containing the new details for the resource and the ETag for the currently cached version of the resource referenced in an If-Match HTTP header. The following example shows a PUT request that updates an order:

  HTTP

  ```
  PUT https://adventure-works.com/orders/1 HTTP/1.1
  If-Match: "2282343857"
  Content-Type: application/x-www-form-urlencoded
  Content-Length: ...
  productID=3&quantity=5&orderValue=250
  ```

- The PUT operation in the web API obtains the current ETag for the requested data (order 1 in the above example), and compares it to the value in the If-Match header.

- If the current ETag for the requested data matches the ETag provided by the request, the resource has not changed and the web API should perform the update, returning a message with HTTP status code 204 (No Content) if it's successful. The response can include Cache-Control and ETag headers for the updated version of the resource. The response should always include the Location header that references the URI of the newly updated resource.

- If the current ETag for the requested data doesn't match the ETag provided by the request, then the data has been changed by another user since it was fetched and the web API should return an HTTP response with an empty message body and a status code of 412 (Precondition Failed).

- If the resource to be updated no longer exists then the web API should return an HTTP response with the status code of 404 (Not Found).

- The client uses the status code and response headers to maintain the cache. If the data has been updated (status code 204) then the object can remain cached (as long as the Cache-Control header doesn't specify no-store) but the ETag should be updated. If the data was changed by another user (status code 412) or not found (status code 404) then the cached object should be discarded.

The next code example shows an implementation of the PUT operation for the Orders controller:

C#

```csharp
public class OrdersController : ApiController
{
    [HttpPut]
    [Route("api/orders/{id:int}")]
    public IHttpActionResult UpdateExistingOrder(int id, DTOOrder order)
    {
        try
        {
            var baseUri = Constants.GetUriFromConfig();
            var orderToUpdate = this.ordersRepository.GetOrder(id);
            if (orderToUpdate == null)
            {
                return NotFound();
            }

            var hashedOrder = orderToUpdate.GetHashCode();
            string hashedOrderEtag = $"\"{hashedOrder}\"";

            // Retrieve the If-Match header from the request (if it exists)
            var matchEtags = Request.Headers.IfMatch;

            // If there is an ETag in the If-Match header and
            // this ETag matches that of the order just retrieved,
            // or if there is no ETag, then update the Order
            if (((matchEtags.Count > 0 &&
                String.CompareOrdinal(matchEtags.First().Tag, hashedOrderE-
tag) == 0)) ||
                matchEtags.Count == 0)
            {
                // Modify the order
                orderToUpdate.OrderValue = order.OrderValue;
                orderToUpdate.ProductID = order.ProductID;
                orderToUpdate.Quantity = order.Quantity;

                // Save the order back to the data store
                // ...

                // Create the No Content response with Cache-Control, ETag,
and Location headers
                var cacheControlHeader = new CacheControlHeaderValue();
                cacheControlHeader.Private = true;
                cacheControlHeader.MaxAge = new TimeSpan(0, 10, 0);

                hashedOrder = order.GetHashCode();
                hashedOrderEtag = $"\"{hashedOrder}\"";
                var eTag = new EntityTagHeaderValue(hashedOrderEtag);

                var location = new Uri($"{baseUri}/{Constants.ORDERS}/{id}");
                var response = new EmptyResultWithCaching()
                {
                    StatusCode = HttpStatusCode.NoContent,
                    CacheControlHeader = cacheControlHeader,
                    ETag = eTag,
                    Location = location
                };

                return response;
            }

            // Otherwise return a Precondition Failed response
            return StatusCode(HttpStatusCode.PreconditionFailed);
        }
        catch
        {
            return InternalServerError();
        }
    }
    ...
}
```

Tip

Use of the If-Match header is entirely optional, and if it's omitted the web API always attempts to update the specified order, possibly blindly overwriting an update made by another user. To avoid problems due to lost updates, always provide an If-Match header.

Handling large requests and responses

Occasionally, a client application needs to issue requests that send or receive data that might be several megabytes (or bigger) in size. Waiting while this amount of data is transmitted could cause the client application to become unresponsive. Consider the following points when you need to handle requests that include significant amounts of data:

Optimize requests and responses that involve large objects

Some resources might be large objects or include large fields, such as graphics images or other types of binary data. A web API should support streaming to enable optimized uploading and downloading of these resources.

The HTTP protocol provides the chunked transfer encoding mechanism to stream large data objects back to a client. When the client sends an HTTP GET request for a large object, the web API can send the reply back in piecemeal *chunks* over an HTTP connection. The length of the data in the reply might not be known initially (it might be generated), so the server hosting the web API should send a response message with each chunk that specifies the `Transfer-Encoding: Chunked` header rather than a Content-Length header. The client application can receive each chunk in turn to build up the complete response. The data transfer completes when the server sends back a final chunk with zero size.

A single request could conceivably result in a massive object that consumes considerable resources. If during the streaming process the web API determines that the amount of data in a request has exceeded some acceptable bounds, it can abort the operation and return a response message with status code 413 (Request Entity Too Large).

You can minimize the size of large objects transmitted over the network by using HTTP compression. This approach helps to reduce the amount of network traffic and the associated network latency, but at the cost of requiring additional processing at the client and the server hosting the web API. For example, a client application that expects to receive compressed data can include an `Accept-Encoding: gzip` request header (other data compression algorithms can also be specified). If the server supports compression it should respond with the content held in gzip format in the message body and the `Content-Encoding: gzip` response header.

You can combine encoded compression with streaming; compress the data first before streaming it, and specify the gzip content encoding and chunked transfer encoding in the message headers. Also note that some web servers (such as Internet Information Server) can be configured to automatically compress HTTP responses regardless of whether the web API compresses the data or not.

Implement partial responses for clients that don't support asynchronous operations

As an alternative to asynchronous streaming, a client application can explicitly request data for large objects in chunks, known as partial responses. The client application sends an HTTP HEAD request to obtain information about the object. If the web API supports partial responses it should respond to the HEAD request with a response message that contains an `Accept-Ranges` header and a `Content-Length` header that indicates the total size of the object, but the body of the message should be empty. The client application can use this information to construct a series of GET requests that specify a range of bytes to receive. The web API should return a response message with HTTP status 206 (Partial Content), a Content-Length header that specifies the actual amount of data included in the body of the response message, and a Content-Range header that indicates which part (such as bytes `4000 to 8000`) of the object this data represents.

HTTP HEAD requests and partial responses are described in more detail in API design.

Avoid sending unnecessary 100-Continue status messages in client applications

A client application that is about to send a large amount of data to a server might determine first whether the server is actually willing to accept the request. Prior to sending the data, the client application can submit an HTTP request with an Expect: 100-Continue header, a Content-Length header that indicates the size of the data, but an empty message body. If the server is willing to handle the request, it should respond with a message that specifies the HTTP status 100 (Continue). The client application can then proceed and send the complete request including the data in the message body.

If you host a service by using Internet Information Services (IIS), the HTTP.sys driver automatically detects and handles Expect: 100-Continue headers before passing requests to your web application. This means that you are unlikely to see these headers in your application code, and you can assume that IIS has already filtered any messages that it deems to be unfit or too large.

If you build client applications by using the .NET Framework, then all POST and PUT messages will first send messages with Expect: 100-Continue headers by default. As with the server-side, the process is handled transparently by the .NET Framework. However, this process results in each POST and PUT request causing two round-trips to the server, even for small requests. If your application isn't sending requests with large amounts of data, you can disable this feature by using the `ServicePointManager` class to create `ServicePoint` objects in the client application. A `ServicePoint` object handles the connections that the client makes to a server based on the scheme and host fragments of URIs that identify resources on the server. You can then set the `Expect100Continue` property of the `ServicePoint` object to false. All subsequent POST and PUT requests made by the client through a URI that matches the scheme and host fragments of the `ServicePoint` object will be sent without Expect: 100-Continue headers. The following code shows how to configure a `ServicePoint` object that configures all requests sent to URIs with a scheme of `http` and a host of `www.contoso.com`.

```C#
Uri uri = new Uri("https://www.contoso.com/");
ServicePoint sp = ServicePointManager.FindServicePoint(uri);
sp.Expect100Continue = false;
```

You can also set the static `Expect100Continue` property of the `ServicePointManager` class to specify the default value of this property for all subsequently created ServicePoint objects.

Support pagination for requests that might return large numbers of objects

If a collection contains a large number of resources, issuing a GET request to the corresponding URI could result in significant processing on the server hosting the web API affecting performance, and generate a significant amount of network traffic resulting in increased latency.

To handle these cases, the web API should support query strings that enable the client application to refine requests or fetch data in more manageable, discrete blocks (or pages). The following code shows the `GetAllOrders` method in the `Orders` controller. This method retrieves the details of orders. If this method was unconstrained, it could conceivably return a large amount of data. The `limit` and `offset` parameters are intended to reduce the volume of data to a smaller subset, in this case only the first 10 orders by default:

```C#
public class OrdersController : ApiController
{
    ...
    [Route("api/orders")]
    [HttpGet]
    public IEnumerable<Order> GetAllOrders(int limit=10, int offset=0)
    {
        // Find the number of orders specified by the limit parameter
        // starting with the order specified by the offset parameter
        var orders = ...
        return orders;
    }
    ...
}
```

A client application can issue a request to retrieve 30 orders starting at offset 50 by using the URI `https://www.adventure-works.com/api/orders?limit=30&offset=50`.

Tip

Avoid enabling client applications to specify query strings that result in a URI that is more than 2000 characters long. Many web clients and servers can't handle URIs that are this long.

Maintaining responsiveness, scalability, and availability

The same web API might be used by many client applications running anywhere in the world. It is important to ensure that the web API is implemented to maintain responsiveness under a heavy load, to be scalable to support a highly varying workload, and to guarantee availability for clients that perform business-critical operations. Consider the following points when determining how to meet these requirements:

Provide asynchronous support for long-running requests

A request that might take a long time to process should be performed without blocking the client that submitted the request. The web API can perform some initial checking to validate the request, initiate a separate task to perform the work, and then return a response message with HTTP code 202 (Accepted). The task could run asynchronously as part of the web API processing, or it could be offloaded to a background task.

The web API should also provide a mechanism to return the results of the processing to the client application. You can achieve this by providing a polling mechanism for client applications to periodically query whether the processing has finished and obtain the result, or enabling the web API to send a notification when the operation has completed.

You can implement a simple polling mechanism by providing a *polling* URI that acts as a virtual resource using the following approach:

1. The client application sends the initial request to the web API.

2. The web API stores information about the request in a table held in Azure Table Storage or Azure Managed Redis and generates a unique key for this entry, possibly in the form of a globally unique identifier (GUID). Alternatively, a message containing information about the request and the unique key could be sent via Azure Service Bus as well.
3. The web API initiates the processing as a separate task or with a library like Hangfire. The web API records the state of the task in the table as *Running*.

    - If you use Azure Service Bus, the message processing would be done separately from the API, possibly by using Azure Functions or AKS.

4. The web API returns a response message with HTTP status code 202 (Accepted), and a URI containing the unique key generated - something like */polling/{guid}*.
5. When the task has completed, the web API stores the results in the table, and it sets the state of the task to *Complete*. Note that if the task fails, the web API could also store information about the failure and set the status to *Failed*.

    - Consider applying retry techniques to resolve possibly transient failures.

6. While the task is running, the client can continue performing its own processing. It can periodically send a request to the URI it received earlier.
7. The web API at the URI queries the state of the corresponding task in the table and returns a response message with HTTP status code 200 (OK) containing this state (*Running*, *Complete*, or *Failed*). If the task has completed or failed, the response message can also include the results of the processing or any information available about the reason for the failure.

    - If the long-running process has more intermediate states, it's better to use a library that supports the saga pattern, like NServiceBus or MassTransit.

Options for implementing notifications include:

- Using a notification hub to push asynchronous responses to client applications. For more information, see Send notifications to specific users by using Azure Notification Hubs.
- Using the Comet model to retain a persistent network connection between the client and the server hosting the web API, and using this connection to push messages from the server back to the client. The MSDN magazine article Building a Simple Comet Application in the Microsoft .NET Framework describes an example solution.
- Using SignalR to push data in real time from the web server to the client over a persistent network connection. SignalR is available for ASP.NET web applications as a NuGet package. You can find more information on the ASP.NET SignalR website.

Ensure that each request is stateless

Each request should be considered atomic. There should be no dependencies between one request made by a client application and any subsequent requests submitted by the same client. This approach assists in scalability; instances of the web service can be deployed on a number of servers. Client requests can be directed at any of these instances and the results should always be the same. It also improves availability for a similar reason; if a web server fails requests can be routed to another instance (by using Azure Traffic Manager) while the server is restarted with no ill effects on client applications.

Track clients and implement throttling to reduce the chances of DoS attacks

If a specific client makes a large number of requests within a given period of time it might monopolize the service and affect the performance of other clients. To mitigate this issue, a web API can monitor calls from client applications either by tracking the IP address of all incoming requests or by logging each authenticated access. You can use this information to limit resource access. If a client exceeds a defined limit, the web API can return a response message with status 503 (Service Unavailable) and include a Retry-After header that specifies when the client can send the next request without it being declined. This strategy can help to reduce the chances of a Denial Of Service (DoS) attack from a set of clients stalling the system.

Manage persistent HTTP connections carefully

The HTTP protocol supports persistent HTTP connections where they are available. The HTTP 1.0 specification added the Connection:Keep-Alive header that enables a client application to indicate to the server that it can use the same connection to send subsequent requests rather than opening new ones. The connection closes automatically if the client doesn't reuse the connection within a period defined by the host. This behavior is the default in HTTP 1.1 as used by Azure services, so there's no need to include Keep-Alive headers in messages.

Keeping a connection open can help to improve responsiveness by reducing latency and network congestion, but it can be detrimental to scalability by keeping unnecessary connections open for longer than required, limiting the ability of other concurrent clients to connect. It can also affect battery life if the client application is running on a mobile device; if the application only makes occasional requests to the server, maintaining an open connection can cause the battery to drain more quickly. To ensure that a connection isn't made persistent with HTTP 1.1, the client can include a Connection:Close header with messages to override the default behavior. Similarly, if a server is handling a very large number of clients it can include a Connection:Close header in response messages which should close the connection and save server resources.

ⓘ Note

Persistent HTTP connections are an optional feature that you can use to reduce network overhead by avoiding the repeated establishment of a communication channel. However, neither the web API nor the client application should depend on the availability of a persistent HTTP connection. Don't use persistent HTTP connections to implement Comet-style notification systems. Use sockets instead, or WebSockets if available, at the Transmission Control Protocol layer. Keep-Alive headers have limited usefulness when a client application communicates with a server via a proxy. Only the connection between the client and the proxy remains persistent.

Publishing and managing a web API

To make a web API available for client applications, the web API must be deployed to a host environment. This environment is typically a web server, although it might be some other type of host process. You should consider the following points when publishing a web API:

- All requests must be authenticated and authorized, and the appropriate level of access control must be enforced.
- A commercial web API might be subject to various quality guarantees concerning response times. It's important to ensure that host environment is scalable if the load can vary significantly over time.
- It might be necessary to meter requests for monetization purposes.
- It might be necessary to regulate the flow of traffic to the web API, and implement throttling for specific clients that have exhausted their quotas.
- Regulatory requirements might mandate logging and auditing of all requests and responses.
- To ensure availability, it might be necessary to monitor the health of the server hosting the web API and restart it if necessary.

It's useful to be able to decouple these issues from the technical issues concerning the implementation of the web API. For this reason, consider creating a façade, running as a separate process and that routes requests to the web API. The façade can provide the management operations and forward validated requests to the web API. Using a façade can also bring many functional advantages, including:

- Acting as an integration point for multiple web APIs.
- Transforming messages and translating communications protocols for clients built by using varying technologies.
- Caching requests and responses to reduce load on the server hosting the web API.

Testing a web API

A web API should be tested as thoroughly as any other piece of software. You should consider creating unit tests to validate the functionality.

The nature of a web API brings its own additional requirements to verify that it operates correctly. You should pay particular attention to the following aspects:

- Test all routes to verify that they invoke the correct operations. Be especially aware of HTTP status code 405 (Method Not Allowed) being returned unexpectedly as this can indicate a mismatch between a route and the HTTP methods (GET, POST, PUT, DELETE) that can be dispatched to that route.

  Send HTTP requests to routes that don't support them, such as submitting a POST request to a specific resource (POST requests should only be sent to resource collections). In these cases, the only valid response *should* be status code 405 (Not Allowed).

- Verify that all routes are protected properly and are subject to the appropriate authentication and authorization checks.

  > **Note**
  >
  > Some aspects of security such as user authentication are most likely to be the responsibility of the host environment rather than the web API, but it's still necessary to include security tests as part of the deployment process.

- Test the exception handling performed by each operation and verify that an appropriate and meaningful HTTP response is passed back to the client application.

- Verify that request and response messages are well-formed. For example, if an HTTP POST request contains the data for a new resource in x-www-form-urlencoded format, confirm that the corresponding operation correctly parses the data, creates the resources, and returns a response containing the details of the new resource, including the correct Location header.

- Verify all links and URIs in response messages. For example, an HTTP POST message should return the URI of the newly created resource. All HATEOAS links should be valid.

- Ensure that each operation returns the correct status codes for different combinations of input. For example:
  - If a query is successful, it should return status code 200 (OK)
  - If a resource isn't found, the operation should return HTTP status code 404 (Not Found).

   - If the client sends a request that successfully deletes a resource, the status code should be 204 (No Content).
   - If the client sends a request that creates a new resource, the status code should be 201 (Created).

Watch out for unexpected response status codes in the 5xx range. These messages are usually reported by the host server to indicate that it was unable to fulfill a valid request.

- Test the different request header combinations that a client application can specify and ensure that the web API returns the expected information in response messages.

- Test query strings. If an operation can take optional parameters (such as pagination requests), test the different combinations and order of parameters.

- Verify that asynchronous operations complete successfully. If the web API supports streaming for requests that return large binary objects (such as video or audio), ensure that client requests aren't blocked while the data is streamed. If the web API implements polling for long-running data modification operations, verify that the operations report their status correctly as they proceed.

You should also create and run performance tests to check that the web API operates satisfactorily under duress. You can build a web performance and load test project by using Visual Studio Ultimate.

Using Azure API Management

On Azure, consider using Azure API Management to publish and manage a web API. Using this facility, you can generate a service that acts as a façade for one or more web APIs. The service is itself a scalable web service that you can create and configure by using the Azure portal. You can use this service to publish and manage a web API as follows:

1. Deploy the web API to a website, Azure cloud service, or Azure virtual machine.

2. Connect the API management service to the web API. Requests sent to the URL of the management API are mapped to URIs in the web API. The same API management service can route requests to more than one web API. This enables you to aggregate multiple web APIs into a single management service. Similarly, the same web API can be referenced from more than one API management service if you need to restrict or partition the functionality available to different applications.

   Note

The URIs, in the HATEOAS links that are generated as part of the response for HTTP GET requests, should reference the URL of the API management service and not the web server that's hosting the web API.

3. For each web API, specify the HTTP operations that the web API exposes together with any optional parameters that an operation can take as input. You can also configure whether the API management service should cache the response received from the web API to optimize repeated requests for the same data. Record the details of the HTTP responses that each operation can generate. This information is used to generate documentation for developers, so it's important that it's accurate and complete.

   You can either define operations manually using the wizards provided by the Azure portal, or you can import them from a file containing the definitions in WADL or Swagger format.

4. Configure the security settings for communications between the API management service and the web server hosting the web API. The API management service currently supports Basic authentication and mutual authentication using certificates, and Open Authorization (OAuth) 2.0 user authorization.

5. Create a product. A product is the unit of publication; you add the web APIs that you previously connected to the management service to the product. When the product is published, the web APIs become available to developers.

   Note

   Prior to publishing a product, you can also define user-groups that can access the product and add users to these groups. This gives you control over the developers and applications that can use the web API. If a web API is subject to approval, prior to being able to access it a developer must send a request to the product administrator. The administrator can grant or deny access to the developer. Existing developers can also be blocked if circumstances change.

6. Configure policies for each web API. Policies govern aspects such as whether cross-domain calls should be allowed, how to authenticate clients, whether to convert between XML and JSON data formats transparently, whether to restrict calls from a given IP range, usage quotas, and whether to limit the call rate. Policies can be applied globally across the entire product, for a single web API in a product, or for individual operations in a web API.

For more information, see the API Management documentation.

Tip

Azure provides the Azure Traffic Manager which enables you to implement failover and load-balancing, and reduce latency across multiple instances of a web site hosted in different geographic locations. You can use Azure Traffic Manager in conjunction with the API Management Service; the API Management Service can route requests to instances of a web site through Azure Traffic Manager. For more information, see Traffic Manager routing methods.

In this structure, if you use custom DNS names for your web sites, you should configure the appropriate CNAME record for each web site to point to the DNS name of the Azure Traffic Manager web site.

Supporting client-side developers

Developers constructing client applications typically require information on how to access the web API, and documentation concerning the parameters, data types, return types, and return codes that describe the different requests and responses between the web service and the client application.

Document the REST operations for a web API

The Azure API Management Service includes a developer portal that describes the REST operations exposed by a web API. When a product has been published it appears on this portal. Developers can use this portal to sign up for access; the administrator can then approve or deny the request. If the developer is approved, they are assigned a subscription key that is used to authenticate calls from the client applications that they develop. This key must be provided with each web API call otherwise it will be rejected.

This portal also provides:

- Documentation for the product, listing the operations that it exposes, the parameters required, and the different responses that can be returned. Note that this information is generated from the details provided in step 3 in the list in the Publishing a web API by using the Microsoft Azure API Management Service section.
- Code snippets that show how to invoke operations from several languages, including JavaScript, C#, Java, Ruby, Python, and PHP.
- A developers' console that enables a developer to send an HTTP request to test each operation in the product and view the results.
- A page where the developer can report any issues or problems found.

The Azure portal enables you to customize the developer portal to change the styling and layout to match the branding of your organization.

Implement a client SDK

Building a client application that invokes REST requests to access a web API requires writing a significant amount of code to construct each request and format it appropriately, send the request to the server hosting the web service, and parse the response to work out whether the request succeeded or failed and extract any data returned. To insulate the client application from these concerns, you can provide an SDK that wraps the REST interface and abstracts these low-level details inside a more functional set of methods. A client application uses these methods, which transparently convert calls into REST requests and then convert the responses back into method return values. This is a common technique that is implemented by many services, including the Azure SDK.

Creating a client-side SDK is a considerable undertaking as it has to be implemented consistently and tested carefully. However, much of this process can be made mechanical, and many vendors supply tools that can automate many of these tasks.

Monitoring a web API

Depending on how you have published and deployed your web API you can monitor the web API directly, or you can gather usage and health information by analyzing the traffic that passes through the API Management service.

Monitoring a web API directly

If you have implemented your web API by using the ASP.NET Web API template (either as a Web API project or as a Web role in an Azure cloud service) and Visual Studio 2013, you can gather availability, performance, and usage data by using ASP.NET Application Insights. Application Insights is a package that transparently tracks and records information about requests and responses when the web API is deployed to the cloud; once the package is installed and configured, you don't need to amend any code in your web API to use it. When you deploy the web API to an Azure web site, all traffic is examined and the following statistics are gathered:

- Server response time.
- Number of server requests and the details of each request.
- The top slowest requests in terms of average response time.
- The details of any failed requests.
- The number of sessions initiated by different browsers and user agents.

- The most frequently viewed pages (primarily useful for web applications rather than web APIs).
- The different user roles accessing the web API.

You can view this data in real time in the Azure portal. You can also create web tests that monitor the health of the web API. A web test sends a periodic request to a specified URI in the web API and captures the response. You can specify the definition of a successful response (such as HTTP status code 200), and if the request doesn't return this response you can arrange for an alert to be sent to an administrator. If necessary, the administrator can restart the server hosting the web API if it has failed.

For more information, see Application Insights - Get started with ASP.NET.

Monitoring a web API through the API Management Service

If you have published your web API by using the API Management service, the API Management page on the Azure portal contains a dashboard that enables you to view the overall performance of the service. The Analytics page enables you to drill down into the details of how the product is being used. This page contains the following tabs:

- **Usage**. This tab provides information about the number of API calls made and the bandwidth used to handle these calls over time. You can filter usage details by product, API, and operation.
- **Health**. This tab enables you to view the outcome of API requests (the HTTP status codes returned), the effectiveness of the caching policy, the API response time, and the service response time. Again, you can filter health data by product, API, and operation.
- **Activity**. This tab provides a text summary of the numbers of successful calls, failed calls, blocked calls, average response time, and response times for each product, web API, and operation. This page also lists the number of calls made by each developer.
- **At a glance**. This tab displays a summary of the performance data, including the developers responsible for making the most API calls, and the products, web APIs, and operations that received these calls.

You can use this information to determine whether a particular web API or operation is causing a bottleneck, and if necessary scale the host environment and add more servers. You can also ascertain whether one or more applications are using a disproportionate volume of resources and apply the appropriate policies to set quotas and limit call rates.

> ⓘ **Note**
>
> You can change the details for a published product, and the changes are applied immediately. For example, you can add or remove an operation from a web API without

requiring that you republish the product that contains the web API.

Next steps

- ASP.NET Web API OData contains examples and further information on implementing an OData web API by using ASP.NET.
- Introducing batch support in Web API and Web API OData describes how to implement batch operations in a web API by using OData.
- Idempotency patterns on Jonathan Oliver's blog provides an overview of idempotency and how it relates to data management operations.
- Status code definitions on the W3C website contains a full list of HTTP status codes and their descriptions.
- Run background tasks with WebJobs provides information and examples on using WebJobs to perform background operations.
- Azure Notification Hubs notify users shows how to use an Azure Notification Hub to push asynchronous responses to client applications.
- API Management describes how to publish a product that provides controlled and secure access to a web API.
- Azure API Management REST API reference describes how to use the API Management REST API to build custom management applications.
- Traffic Manager routing methods summarizes how Azure Traffic Manager can be used to load-balance requests across multiple instances of a website hosting a web API.
- Application Insights - Get started with ASP.NET provides detailed information on installing and configuring Application Insights in an ASP.NET Web API project.