Implementação da API Web

Artigo • 12/06/2023

Uma API Web RESTful cuidadosamente projetada define os recursos, as relações e os esquemas de navegação que podem ser acessados por aplicativos cliente. Ao implementar e implantar uma API da Web, você deve considerar os requisitos físicos do ambiente de hospedagem dessa API e o modo como ela é construída, em vez da estrutura lógica dos dados. Estas diretrizes abordam as práticas recomendadas para implementar uma API da Web e publicá-la para torná-la disponível para aplicativos cliente. Para obter informações detalhadas sobre o design de API Web, confira Design de API Web.

Processamento de solicitações

Considere os pontos a seguir ao implementar o código para processar solicitações.

As ações GET, PUT, DELETE, HEAD e PATCH devem ser idempotentes

O código que implementa essas solicitações não deve impor nenhum efeito colateral. A mesma solicitação repetida sobre o mesmo recurso deve resultar no mesmo estado. Por exemplo, enviar várias solicitações DELETE para o mesmo URI deve ter o mesmo efeito, embora o código de status de HTTP nas mensagens de resposta possa ser diferente. A primeira solicitação DELETE pode retornar o código de status 204 (Sem conteúdo), enquanto uma solicitação DELETE subsequente pode retornar um código de status 404 (Não encontrado).

> **Observação**
>
> O artigo Idempotency Patterns (Padrões de idempotência), no blog de Jonathan Oliver, fornece uma visão geral da idempotência e de como ela se relaciona às operações de gerenciamento de dados.

Ações POST que criam novos recursos não devem ter efeitos colaterais não relacionados

Se uma solicitação POST se destina a criar um novo recurso, os efeitos da solicitação devem ser limitados ao novo recurso (e possivelmente a quaisquer recursos diretamente

relacionados, se houver algum tipo de vinculação envolvida). Por exemplo, em um sistema de comércio eletrônico, uma solicitação POST que cria um novo pedido para um cliente também pode alterar os níveis de estoque e gerar informações de faturamento, mas não deve modificar informações não diretamente relacionadas ao pedido ou ter quaisquer outros efeitos colaterais no estado geral do sistema.

Evite implementar operações POST, PUT e DELETE verborrágicas

Evite projetar sua API com base em *operações* de conversação. Cada solicitação vem com sobrecarga de protocolo, rede e computação. Por exemplo, a execução de 100 solicitações menores em vez de uma solicitação em lote maior incorre em sobrecarga adicional no cliente, na rede e no servidor de recursos. Sempre que possível, forneça suporte ao verbo HTTP para coleções de recursos em vez de apenas recursos individuais.

- Uma solicitação GET para uma coleção pode recuperar vários recursos ao mesmo tempo.
- Uma solicitação POST pode conter os detalhes de vários novos recursos e adicioná-los todos à mesma coleção.
- Uma solicitação PUT pode substituir todo o conjunto de recursos em uma coleção.
- Uma solicitação DELETE pode remover uma coleção inteira.

O suporte a OData (Open Data Protocol) incluído na ASP.NET Web API 2 oferece a capacidade de realizar solicitações em lote. Um aplicativo cliente pode agrupar várias solicitações de API da Web e enviá-las ao servidor em uma única solicitação HTTP e receber uma única resposta HTTP que contém as respostas a cada solicitação. Para obter mais informações, consulte Habilitar o lote no serviço OData da API Web.

Siga a especificação de HTTP ao enviar uma resposta

Uma API da Web deve retornar mensagens contendo o código de status HTTP correto (para permitir que o cliente determine como tratar o resultado), os cabeçalhos HTTP apropriados (para que o cliente entenda a natureza do resultado) e um corpo formatado adequadamente (para permitir que o cliente analise o resultado).

Por exemplo, uma operação POST deve retornar o código de status 201 (Criado), e a mensagem de resposta deve incluir o URI do recurso recém-criado no cabeçalho Location da mensagem de resposta.

Ofereça suporte à negociação de conteúdo

O corpo de uma mensagem de resposta pode conter dados em uma variedade de formatos. Por exemplo, uma solicitação HTTP GET poderia retornar dados em formato JSON ou XML. Quando o cliente envia uma solicitação, ele pode incluir um cabeçalho Accept que especifica os formatos de dados que pode processar. Esses formatos são especificados como tipos de mídia. Por exemplo, um cliente que emite uma solicitação GET que recupera uma imagem pode especificar um cabeçalho Accept listando os tipos de mídia que o cliente pode processar, como `image/jpeg, image/gif, image/png`. Quando a API da Web retorna o resultado, ela deve formatar os dados usando um destes tipos de mídia e especificar o formato no cabeçalho Content-Type da resposta.

Se o cliente não especificar um cabeçalho Accept, use um formato padrão adequado para o corpo da resposta. Por exemplo, a estrutura da ASP.NET Web API utiliza JSON como formato padrão para dados baseados em texto.

Forneça links para dar suporte à descoberta de recursos e navegação estilo HATEOAS

A abordagem HATEOAS permite que um cliente navegue e descubra recursos por meio de um ponto de partida inicial. Isso é realizado por meio de links contendo URIs; quando um cliente emite uma solicitação HTTP GET para obter um recurso, a resposta deve conter URIs que permitam que um aplicativo cliente localize rapidamente quaisquer recursos diretamente relacionados. Por exemplo, em uma API da Web que oferece suporte a uma solução de comércio eletrônico, um cliente pode ter realizado muitos pedidos. Quando um aplicativo cliente recupera os detalhes de um cliente, a resposta deve incluir links que permitam que o aplicativo cliente envie solicitações HTTP GET capazes de recuperar esses pedidos. Além disso, links estilo HATEOAS devem, para executar cada solicitação, descrever as outras operações (POST, PUT, DELETE e assim por diante) para as quais há suporte em cada recurso vinculado, junto com o URI correspondente. Essa abordagem é descrita mais detalhadamente em Design de API.

Atualmente não há nenhum padrão que rege a implementação de HATEOAS, mas o exemplo a seguir ilustra uma abordagem possível. Neste exemplo, uma solicitação HTTP GET que localiza os detalhes de um cliente retorna uma resposta que inclui links HATEOAS que referenciam os pedidos desse cliente:

HTTP

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

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"]}
]}
```

Neste exemplo, os dados do cliente são representados pela classe `Customer` mostrada no snippet de código a seguir. Os links HATEOAS são mantidos na propriedade de coleção `Links` :

C#

```
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; }
}
```

A operação HTTP GET recupera os dados do cliente por meio do armazenamento, constrói um objeto `Customer` e, em seguida, preenche a coleção `Links`. O resultado é formatado como uma mensagem de resposta JSON. Cada link inclui os seguintes campos:

- A relação (`Rel`) entre o objeto que está sendo retornado e o objeto descrito pelo link. Nesse caso, `self` indica que o link é uma referência para o objeto em si (semelhante a um ponteiro `this` em muitas linguagens orientadas a objeto), enquanto `orders` é o nome de uma coleção que contém as informações de pedido relacionadas.
- O hiperlink (`Href`) para o objeto sendo descrito pelo link na forma de um URI.
- O tipo de solicitação HTTP (`Action`) que pode ser enviada para esse URI.
- O formato de quaisquer dados (`Types`) que devam ser fornecidos na solicitação HTTP ou que possam ser retornados na resposta, dependendo do tipo da solicitação.

Os links HATEOAS mostrados no exemplo de resposta HTTP indicam que um aplicativo cliente pode executar as seguintes operações:

- Uma solicitação HTTP GET para o URI `https://adventure-works.com/customers/2` para coletar os detalhes do cliente (novamente). Os dados podem ser retornados como XML ou JSON.
- Uma solicitação HTTP PUT para o URI `https://adventure-works.com/customers/2` para modificar os detalhes do cliente. Os novos dados devem ser fornecidos na mensagem de solicitação no formato x-www-form-urlencoded.
- Uma solicitação HTTP DELETE para o URI `https://adventure-works.com/customers/2` para excluir o cliente. A solicitação não espera nenhuma informação adicional nem dados de retorno no corpo da mensagem de resposta.
- Uma solicitação HTTP GET para o URI `https://adventure-works.com/customers/2/orders` para localizar todos os pedidos do cliente. Os dados podem ser retornados como XML ou JSON.
- Uma solicitação HTTP POST para o URI `https://adventure-works.com/customers/2/orders` para criar um pedido para esse cliente. Os dados devem ser fornecidos na mensagem de solicitação no formato x-www-form-urlencoded.

Tratamento de exceções

Leve em consideração os pontos a seguir se uma operação lançar uma exceção não capturada.

Capture exceções e retorne uma resposta significativa para os clientes

O código que implementa uma operação HTTP deve oferecer tratamento abrangente de exceções, em vez de permitir que exceções não percebidas se propaguem para a estrutura. Se uma exceção torna impossível concluir a operação com êxito, a exceção pode ser passada de volta na mensagem de resposta, mas ela deve incluir uma descrição significativa do erro que causou a exceção. A exceção também deve incluir o código de status HTTP adequado em vez de simplesmente retornar o código de status 500 para todas as situações. Por exemplo, se uma solicitação de usuário faz com que uma atualização de banco de dados que viola uma restrição (por exemplo, a tentativa de excluir um cliente que tenha pedidos pendentes), você deve retornar o código de status 409 (Conflito) e um corpo de mensagem que indique o motivo para o conflito. Se outra condição torna a solicitação inexequível, você pode retornar o código de status 400 (Solicitação Incorreta). Encontre uma lista completa dos códigos de status HTTP na página Definições de código de status, no site do W3C.

O exemplo de código intercepta diferentes condições e retorna uma resposta apropriada.

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();
    }
}
```

> **Dica**
>
> Não inclua informações que podem ser úteis para um invasor tentando invadir sua API.

Muitos servidores Web interceptam as condições de erro antes que elas atinjam a API da Web. Por exemplo, se você configura a autenticação para um site da Web e o usuário não fornece as informações de autenticação corretas, o servidor Web deve responder com o código de status 401 (Não Autorizado). Depois de um cliente ter sido autenticado, seu código poderá executar suas próprias verificações para conferir se o cliente é capaz de acessar o recurso solicitado. Se essa autorização falhar, você deve retornar o código de status 403 (Proibido).

Trate exceções de modo consistente e registre em log as informações referentes a erros

Para tratar exceções de maneira consistente, considere a implementação de uma estratégia global para tratamento de erro, em toda a API da Web. Você também deve incorporar o log de erros, que captura os detalhes completos de cada exceção; esse log pode conter informações detalhadas, desde que elas não estejam acessíveis aos clientes pela Web.

Diferencie erros no lado do cliente de erros no lado do servidor

O protocolo HTTP faz distinção entre os erros que ocorrem devido ao aplicativo cliente (os códigos de status HTTP 4xx) e erros que são causados por um problema no servidor (os códigos de status HTTP 5xx). Certifique-se de respeitar essa convenção em todas as mensagens de resposta de erro.

Otimizando o acesso a dados do lado do cliente

Em um ambiente distribuído, como aquele envolvendo um servidor Web e aplicativos cliente, um dos principais motivos de preocupação é a rede. Ela pode atuar como um gargalo considerável, especialmente se um aplicativo cliente enviar solicitações ou receber dados com frequência. Portanto, você deve ter como objetivo minimizar a quantidade de tráfego que flui pela rede. Considere os pontos a seguir ao implementar o código para recuperar e manter dados:

Ofereça suporte a cache no lado do cliente

O protocolo HTTP 1.1 oferece suporte a caching em clientes e servidores intermediários por meio dos quais uma solicitação é roteada pelo uso do cabeçalho Cache-Control. Quando um aplicativo cliente envia uma solicitação HTTP GET para a API da Web, a resposta pode incluir um cabeçalho Cache-Control que indica se os dados no corpo da resposta podem ou não ser armazenados com segurança pelo cliente ou um servidor intermediário por meio do qual a solicitação foi encaminhada, além de indicar quanto tempo resta até que esses dados expirem e sejam considerados desatualizados.

O exemplo a seguir mostra uma solicitação HTTP GET e a resposta correspondente, que inclui um cabeçalho Cache-Control:

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}
```

Neste exemplo, o cabeçalho Cache-Control especifica que os dados retornados devem expirar após 600 segundos e ser adequados para um único cliente; além disso, esses dados não devem ser armazenados em um cache compartilhado usado por outros clientes (eles são *particulares*). O cabeçalho Cache-Control pode especificar *public* em vez de *private* caso os dados possam ser armazenados em um cache compartilhado, ou então pode especificar *no-store* caso os dados **não** possam ser armazenados em cache

pelo cliente. O exemplo de código a seguir mostra como construir um cabeçalho Cache-Control em uma mensagem de resposta:

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;
    }
    ...
}
```

Esse código usa uma classe personalizada `IHttpActionResult` chamada `OkResultWithCaching`. Essa classe habilita o controlador a definir o conteúdo do cabeçalho de cache:

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(Cancellati-
onToken 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) {Re-
asonPhrase = "Operation was cancelled"};
        }
        return response;
    }
}
```

> ⓘ **Observação**
>
> O protocolo HTTP também define a diretiva *no-cache* para o cabeçalho Cache-Control. Confusamente, essa diretiva não significa "não armazenar em cache", mas sim "revalidar as informações em cache com o servidor antes de retorná-las"; os dados ainda podem ser armazenados em cache, mas eles são verificados cada vez que são usados para assegurar que ainda estão atualizados.

O gerenciamento de cache é responsabilidade do aplicativo cliente ou do servidor intermediário, mas se corretamente implementado ele pode economizar largura de banda e melhorar o desempenho, eliminando a necessidade de buscar os dados que já foram recuperados recentemente.

O valor *max-age* no cabeçalho Cache-Control é apenas um guia, não uma garantia de que os dados correspondentes não serão alterados durante o período especificado. A API da Web deve definir o parâmetro max-age para um valor adequado, que depende da volatilidade esperada dos dados. Quando esse período expirar, o cliente deve descartar o objeto do cache.

> ⓘ **Observação**
>
> Navegadores da Web mais modernos dão suporte a arquivos offline, adicionando os cabeçalhos cache-control apropriados a solicitações e examinando os cabeçalhos dos resultados, conforme descrito. No entanto, alguns navegadores mais antigos não armazenarão em cache os valores retornados de uma URL que inclua uma cadeia de consulta. Isso geralmente não é um problema para

aplicativos cliente personalizados que implementam sua própria estratégia de gerenciamento de cache, baseada no protocolo discutido aqui.

Alguns proxies mais antigos exibem o mesmo comportamento e podem não armazenar em cache solicitações baseadas em URLs com cadeias de consulta. Isso pode ser um problema para os aplicativos cliente personalizados que se conectam a um servidor Web por meio de um proxy desse tipo.

Forneça ETags para otimizar o processamento de consulta

Quando um aplicativo cliente recupera um objeto, a mensagem de resposta também pode incluir uma *marca da entidade (ETag)*. Uma Etag é uma cadeia de caracteres opaca que indica a versão de um recurso. Cada vez que um recurso for alterado, a Etag também será modificada. Essa ETag deve ser armazenada em cache pelo aplicativo cliente como parte dos dados. O exemplo de código a seguir mostra como adicionar uma ETag como parte da resposta a uma solicitação HTTP GET. Esse código usa o método `GetHashCode` de um objeto para gerar um valor numérico que identifica o objeto (você pode substituir esse método se necessário e gerar seu próprio hash usando um algoritmo, como MD5):

C#

```
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;
    }
```

```
    ...
}
```

A mensagem de resposta publicada pela API da Web tem essa aparência:

```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}
```

> **Dica**
>
> Por motivos de segurança, não permita que dados confidenciais ou dados retornados por uma conexão autenticada (HTTPS) sejam armazenados em cache.

Um aplicativo cliente pode emitir uma solicitação GET subsequente para recuperar o mesmo recurso a qualquer momento e, se o recurso foi alterado (se ele tem uma ETag diferente), a versão armazenada em cache deve ser descartada e a nova versão deve ser adicionada ao cache. Se um recurso é grande e exige uma quantidade significativa de largura de banda para ser transmitido de volta ao cliente, repetidas solicitações para buscar os mesmos dados podem tornar-se ineficientes. Para combater isso, o protocolo HTTP define o processo a seguir para otimizar as solicitações GET às quais você deve oferecer suporte em uma API da Web:

- O cliente cria uma solicitação GET contendo a ETag para a versão atualmente armazenada em cache do recurso referenciado em um cabeçalho HTTP If-None-Match:

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

- A operação GET na API da Web obtém a ETag atual para os dados solicitados (pedido 2 no exemplo acima) e compara-a ao valor do cabeçalho If-None-Match.
- Se a ETag atual para os dados solicitados corresponde à ETag fornecida pela solicitação, isso significa que o recurso não foi alterado e a API da Web deve

retornar uma resposta HTTP com um corpo de mensagem vazio, além de um código de status 304 (Não Modificado).

- Se a ETag atual para os dados solicitados não corresponde à ETag fornecida pela solicitação, isso significa que os dados mudaram e a API da Web deve retornar uma resposta HTTP com os novos dados no corpo da mensagem, além de um código de status 200 (OK).
- Se os dados solicitados não existem mais, a API da Web deve retornar uma resposta HTTP com o código de status 404 (Não Encontrado).
- O cliente usa o código de status para manter o cache. Se os dados não mudaram (código de status 304), o objeto pode permanecer armazenado em cache e o aplicativo cliente deve continuar a usar essa versão do objeto. Se os dados mudaram (código de status 200), o objeto armazenado em cache deve ser descartado e o novo objeto deve ser inserido. Se os dados não estão mais disponíveis (código de status 404), o objeto deve ser removido do cache.

> ⓘ **Observação**
>
> Se o cabeçalho de resposta contém o cabeçalho Cache-Control com valor no-store, o objeto deve ser sempre removido do cache, independentemente do código de status HTTP.

O código a seguir mostra o método `FindOrderByID` estendido para oferecer suporte ao cabeçalho If-None-Match. Observe que, se o cabeçalho If-None-Match for omitido, o pedido especificado sempre será recuperado:

C#

```
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
exists)
            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, hashedOrde-
rEtag) == 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();
        }
    }
...
}
```

Este exemplo incorpora uma classe personalizada adicional `IHttpActionResult`, denominada `EmptyResultWithCaching`. Essa classe age simplesmente como um wrapper

em torno de um objeto `HttpResponseMessage` , que não contém um corpo de resposta:

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;
    }
}
```

Dica

Neste exemplo, a ETag para os dados é gerada pelo hash dos dados recuperados da fonte de dados subjacente. Se a ETag pode ser computada de algum outro modo, o processo pode ser otimizado ainda mais e os dados só precisarão ser coletados da fonte de dados se tiverem sofrido modificações. Essa abordagem é especialmente útil se os dados forem grandes ou se acessar a fonte de dados puder resultar em latência significativa (por exemplo, se a fonte de dados for um banco de dados remoto).

Use ETags para oferecer suporte à simultaneidade otimista

Para habilitar atualizações nos dados armazenados em cache anteriormente, o protocolo HTTP oferece suporte a uma estratégia de simultaneidade otimista. Se, após coletar um recurso e armazená-lo em cache, o aplicativo cliente subsequentemente envia uma solicitação PUT ou DELETE para alterar ou remover o recurso, essa solicitação deve incluir o cabeçalho If-Match, que faz referência à ETag. A API da Web pode usar essas informações para determinar se o recurso já foi alterado por outro usuário desde que foi recuperado e enviar uma resposta apropriada para o aplicativo cliente, conforme demonstrado a seguir:

- O cliente cria uma solicitação PUT contendo os novos detalhes para o recurso e a ETag para a versão atualmente armazenada em cache do recurso referenciado em um cabeçalho HTTP If-Match. O exemplo a seguir mostra uma solicitação PUT que atualiza um pedido:

  ```
  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
  ```

- A operação PUT na API da Web obtém a ETag atual para os dados solicitados (pedido 1 no exemplo acima) e compara-a ao valor do cabeçalho If-Match.

- Se a ETag atual para os dados solicitados corresponde à ETag fornecida pela solicitação, isso significa que o recurso não foi alterado e a API da Web deve executar a atualização, retornando uma mensagem com código de status HTTP 204 (Sem Conteúdo) se for bem-sucedida. A resposta pode incluir os cabeçalhos Cache-Control e ETag para a versão atualizada do recurso. A resposta sempre deve incluir o cabeçalho de localização que referencia o URI do recurso recém-atualizado.

- Se a ETag atual para os dados solicitados não corresponde à ETag fornecida pela solicitação, isso significa que os dados foram modificados por outro usuários desde que foram coletados, e a API da Web deve retornar uma resposta HTTP com um corpo de mensagem vazio e um código de status 412 (Falha na Pré-condição).

- Se o recurso a ser atualizado não existe mais, a API da Web deve retornar uma resposta HTTP com o código de status 404 (Não Encontrado).

- O cliente usa o código de status e os cabeçalhos de resposta para manter o cache. Se os dados tiverem sido atualizados (código de status 204), o objeto poderá permanecer armazenado em cache (contanto que o cabeçalho Cache-Control não esteja definido como no-store), mas a ETag deverá ser atualizada. Se os dados foram alterados por outro usuário (código de status 412) ou não foram encontrados (código de status 404), o objeto armazenado em cache deve ser descartado.

O exemplo de código a seguir mostra uma implementação da operação PUT para o controlador Orders:

```
C#
```

```
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();
        }
    }
    ...
}
```

> **Dica**
>
> O uso do cabeçalho If-Match é totalmente opcional e, se ele for omitido, a API da Web sempre tentará atualizar o pedido especificado, possivelmente substituindo às cegas uma atualização feita por outro usuário. Para evitar problemas devido a atualizações perdidas, forneça sempre um cabeçalho If-Match.

Tratamento de grandes solicitações e respostas

Pode haver ocasiões em que um aplicativo cliente precise emitir solicitações que enviam ou recebem dados, as quais podem atingir um tamanho de vários megabytes (ou ainda maior). Aguardar enquanto essa quantidade de dados é transmitida pode fazer com que o aplicativo cliente pare de responder. Quando você precisar processar solicitações que incluem uma quantidade significativa de dados, considere os seguintes pontos:

Otimize as solicitações e respostas que envolvem objetos grandes

Alguns recursos podem ser objetos grandes ou incluir campos grandes, como imagens de gráficos ou outros tipos de dados binários. Uma API da Web deve oferecer suporte a streaming, para habilitar carregamento e download otimizados desses recursos.

O protocolo HTTP fornece o mecanismo de codificação para transferência em partes, para transmitir grandes objetos de dados para um cliente. Quando o cliente envia uma solicitação HTTP GET para um objeto grande, a API da Web pode enviar a resposta em *partes* fragmentadas por uma conexão HTTP. O comprimento dos dados na resposta

pode não ser conhecido inicialmente (podem ser gerados), por isso o servidor que hospeda a API Web deve enviar uma mensagem de resposta com cada parte que especifique o cabeçalho `Transfer-Encoding: Chunked`, em vez de um cabeçalho Content-Length. O aplicativo cliente pode receber cada uma das partes por vez, para criar a resposta completa. A transferência de dados é concluída quando o servidor envia de volta uma parte final com tamanho zero.

Uma única solicitação pode perfeitamente resultar em um objeto massivo que consome recursos consideráveis. Se, durante o processo de streaming, a API Web determina que o volume de dados em uma solicitação excedeu alguns limites aceitáveis, ela pode anular a operação e retornar uma mensagem de resposta com o código de status 413 (Entidade de Solicitação Muito Grande).

Você pode minimizar o tamanho de objetos grandes transmitidos pela rede pelo uso de compactação HTTP. Essa abordagem ajuda a reduzir a quantidade de tráfego de rede e a latência de rede associada, mas para isso exigem processamento adicional no cliente e no servidor que hospeda a API da Web. Por exemplo, um aplicativo cliente que espera receber dados compactados pode incluir um cabeçalho de solicitação `Accept-Encoding: gzip` (outros algoritmos de compactação de dados também podem ser especificados). Se o servidor oferece suporte à compactação, ele deverá responder com o conteúdo mantido em formato gzip no corpo da mensagem e o cabeçalho de resposta `Content-Encoding: gzip`.

Você pode combinar compactação codificada com streaming; compacte os dados antes de enviá-los por streaming, então especifique a codificação de conteúdo gzip e codificação de transferência em partes nos cabeçalhos das mensagens. Observe também que alguns servidores Web (como o Internet Information Server) podem ser configurados para compactar automaticamente respostas HTTP, independentemente de a API da Web compactar os dados ou não.

Implemente respostas parciais para clientes que não oferecem suporte a operações assíncronas

Como alternativa para streaming assíncrono, um aplicativo cliente pode solicitar explicitamente que dados para objetos grandes sejam transmitidos em partes, conhecidas como respostas parciais. O aplicativo cliente envia uma solicitação HTTP HEAD para obter informações sobre o objeto. Se a API Web oferece suporte a respostas parciais, ela deve responder à solicitação HEAD com uma mensagem de resposta que contém um cabeçalho `Accept-Ranges` e um cabeçalho `Content-Length` que indica o tamanho total do objeto, mas o corpo da mensagem deve estar vazio. O aplicativo cliente pode usar essas informações para construir uma série de operações GET que

especificam um intervalo de bytes a receber. A API da Web deve retornar uma mensagem de resposta com status HTTP 206 (Conteúdo Parcial), um cabeçalho Content-Length, que especifica a quantidade real de dados incluídos no corpo da mensagem de resposta e, por fim, um cabeçalho de Content-Range, que indica qual parte (como os bytes `4000` a `8000`) do objeto esses dados representam.

As solicitações HTTP HEAD e as respostas parciais são descritas mais detalhadamente em Design de API.

Evite enviar mensagens de status 100-Continue desnecessárias em aplicativos cliente

Um aplicativo cliente que está prestes a enviar uma grande quantidade de dados para um servidor pode determinar primeiro se o servidor está ou não realmente disposto a aceitar a solicitação. Antes de enviar os dados, o aplicativo cliente pode enviar uma solicitação HTTP com um cabeçalho Expect: 100-Continue, um cabeçalho Content-Length que indica o tamanho dos dados e, além disso, um corpo de mensagem vazio. Se o servidor estiver disposto a processar a solicitação, ele deve responder com uma mensagem que especifica o status HTTP 100 (Continuar). O aplicativo cliente pode, em seguida, prosseguir e enviar a solicitação completa, incluindo os dados no corpo da mensagem.

Se você hospeda um serviço usando o Serviços de informações da Internet (IIS), o driver HTTP.sys detecta e processa automaticamente cabeçalhos Expect: 100-Continue antes de passar as solicitações ao seu aplicativo Web. Isso significa que é improvável que você veja esses cabeçalhos no código do aplicativo, e você pode presumir que o IIS já filtrou todas as mensagens que classifica como impróprias ou muito grandes.

Se você está criando aplicativos cliente usando o .NET Framework, todas as mensagens POST e PUT enviarão primeiro, por padrão, mensagens com cabeçalhos Expect: 100-Continue. Assim como acontece com o lado do servidor, o processo é tratado de modo transparente pelo .NET Framework. No entanto, como resultado desse processo, cada solicitação POST e PUT causa duas viagens de ida e volta ao servidor, mesmo para solicitações pequenas. Se seu aplicativo não está enviando solicitações com grandes quantidades de dados, você pode desabilitar esse recurso usando a classe `ServicePointManager` para criar objetos `ServicePoint` no aplicativo cliente. Um objeto `ServicePoint` processa as conexões que o cliente faz a um servidor com base no esquema nos fragmentos de host de URIs que identificam recursos no servidor. Então, você pode definir a propriedade `Expect100Continue` do objeto `ServicePoint` como false. Todas as solicitações POST e PUT subsequentes feitas pelo cliente por meio de um URI que corresponda ao esquema e aos fragmentos de host do objeto `ServicePoint` serão

enviadas sem cabeçalhos Expect: 100-Continue. O código a seguir mostra como configurar um `ServicePoint` que configura todas as solicitações enviadas para URIs com um esquema `http` e um host de `www.contoso.com`.

C#

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

Defina também a propriedade estática `Expect100Continue` da classe `ServicePointManager` para especificar o valor padrão dessa propriedade em todos os objetos ServicePoint criados posteriormente.

Ofereça suporte a paginação para solicitações que podem retornar um grande número de objetos

Se uma coleção contém um grande número de recursos, emitir uma solicitação GET para o URI correspondente pode resultar em processamento significativo no servidor que hospeda a API da Web, afetando o desempenho, além de gerar uma quantidade significativa de tráfego de rede resultando em aumento da latência.

Para tratar desses casos, a API da Web deve oferecer suporte a cadeias de consulta que habilitam o aplicativo cliente a refinar solicitações ou coletar dados em blocos (ou páginas) mais gerenciáveis e discretos. O código a seguir mostra o método `GetAllOrders` no controlador `Orders`. Esse método obtém os detalhes de pedidos. Se esse método fosse irrestrito, ele poderia perfeitamente retornar uma grande quantidade de dados. Os parâmetros `limit` e `offset` têm a finalidade de reduzir o volume de dados a um subconjunto menor; nesse caso, por padrão, somente os primeiros 10 pedidos:

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;
    }
```

```
    ...
}
```

Usando o URI `https://www.adventure-works.com/api/orders?limit=30&offset=50`, um aplicativo cliente pode emitir uma solicitação para recuperar 30 pedidos começando com deslocamento 50.

> **Dica**
>
> Evite habilitar aplicativos cliente a especificar cadeias de consulta que resultem em um URI com mais de 2.000 caracteres. Muitos servidores e clientes da Web não podem processar URIs tão longos.

Manutenção da capacidade de resposta, escalabilidade e disponibilidade

A mesma API da Web pode ser utilizada por muitos aplicativos cliente em execução em qualquer lugar do mundo. É importante garantir que a API da Web seja implementada para manter a capacidade de resposta sob uma carga pesada, para ser dimensionável de modo a oferecer suporte a uma carga de trabalho altamente variável e, por fim, para garantir a disponibilidade para clientes que executam operações críticas para os negócios. Considere os pontos a seguir ao determinar como atender a esses requisitos:

Forneça suporte assíncrono para solicitações de execução longa

Uma solicitação que pode levar muito tempo para ser processada deve ser realizada sem bloquear o cliente que enviou a solicitação. A API da Web pode executar algumas verificações iniciais para validar a solicitação, iniciar uma tarefa separada para executar o trabalho e, em seguida, retornar uma mensagem de resposta com o código HTTP 202 (Aceito). A tarefa pode ser executada de maneira assíncrona como parte da API de Web em processamento ou pode ser descarregada para uma tarefa em segundo plano.

A API da Web também deve fornecer um mecanismo para retornar os resultados do processamento para o aplicativo cliente. Você pode fazer isso fornecendo um mecanismo de pesquisa para que aplicativos cliente consultem periodicamente se o processamento foi concluído e obtenham o resultado, ou então habilitando a API da Web para enviar uma notificação quando a operação tiver sido concluída.

Você pode implementar um mecanismo de pesquisa simples, fornecendo um URI de *sondagem* que atue como um recurso virtual usando a seguinte abordagem:

1. O aplicativo cliente envia a solicitação inicial para a API da Web.
2. A API da Web armazena informações sobre a solicitação em uma tabela mantida no armazenamento de tabela ou no Cache do Microsoft Azure e gera uma chave exclusiva para essa entrada, possivelmente na forma de um GUID (identificador global exclusivo). Como alternativa, uma mensagem contendo informações sobre a solicitação e a chave exclusiva também pode ser enviada por meio do Barramento de Serviço do Azure.
3. A API da Web inicia o processamento como uma tarefa separada ou com uma biblioteca como o Hangfire. A API da Web registra o estado da tarefa na tabela como *Em execução*.

    - Se você usar o Barramento de Serviço do Azure, o processamento de mensagens será feito separadamente da API, possivelmente usando o Azure Functions ou o AKS.

4. A API da Web retorna uma mensagem de resposta com o código de status HTTP 202 (Aceito) e um URI contendo a chave exclusiva gerada – algo como */polling/{guid}*.
5. Quando a tarefa for concluída, a API da Web armazenará os resultados na tabela e definirá o estado da tarefa como *Concluída*. Observe que, se a tarefa falhar, a API da Web poderá também armazenar informações sobre a falha e definir o status como *Falha*.

    - Considere a aplicação de técnicas de repetição para resolver possíveis falhas transitórias.

6. Enquanto a tarefa é executada, o cliente pode continuar a executar seu próprio processamento. Ele pode enviar periodicamente uma solicitação para o URI recebido anteriormente.
7. A API da Web no URI consulta o estado da tarefa correspondente na tabela e retorna uma mensagem de resposta, com código de status HTTP 200 (OK) e contendo esse estado (*Running*, *Complete* ou *Failed*). Se a tarefa foi concluída ou falhou, a mensagem de resposta também pode incluir os resultados de processamento ou quaisquer informações disponíveis sobre o motivo da falha.

    - Se o processo de execução longa tiver mais estados intermediários, é melhor usar uma biblioteca que ofereça suporte ao padrão saga, como NServiceBus ou MassTransit.

Entre as opções para implementar notificações, temos:

- Uso de um hub de notificações para enviar respostas assíncronas aos aplicativos cliente. Para obter mais informações, confira Enviar notificações para usuários específicos usando os Hubs de Notificações do Azure.
- Usar o modelo Comet para manter uma conexão de rede persistente entre o cliente e o servidor que hospeda a API da Web, e usar essa conexão para enviar mensagens do servidor por push de volta para o cliente. O artigo da revista MSDN Criando um Aplicativo Comet Simples no Microsoft .NET Framework descreve um exemplo de solução.
- Uso do SignalR para enviar dados em tempo real do servidor Web ao cliente em uma conexão de rede persistente. O SignalR está disponível para aplicativos Web ASP.NET como um pacote do NuGet. Você pode encontrar mais informações no site SignalR ASP.NET .

Verifique se cada solicitação é sem estado

Cada solicitação deve ser considerada atômica. Não deve haver nenhuma dependência entre uma solicitação feita por um aplicativo cliente e as solicitações subsequentes emitidas pelo mesmo cliente. Essa abordagem ajuda na escalabilidade; instâncias do serviço Web podem ser implantadas em vários servidores. Solicitações de cliente podem ser direcionadas para qualquer uma dessas instâncias e os resultados devem ser sempre os mesmos. Ele também melhora a disponibilidade pelo mesmo motivo; se um servidor Web falhar, as solicitações poderão ser roteadas para outra instância (usando o Gerenciador de Tráfego do Azure), enquanto o servidor será reiniciado sem nenhum dano aos aplicativos cliente.

Acompanhe os clientes e implemente limitação para reduzir as chances de ataques DoS

Se um cliente específico faz um grande número de solicitações em um determinado período de tempo, ele pode monopolizar o serviço e afetar o desempenho de outros clientes. Para atenuar esse problema, uma API da Web pode monitorar chamadas de aplicativos cliente acompanhando o endereço IP de todas as solicitações em entrada ou registrando cada acesso autenticado em log. Você pode usar essas informações para limitar o acesso aos recursos. Se um cliente exceder um limite definido, a API da Web pode retornar uma mensagem de resposta com status 503 (Serviço Indisponível) e incluir um cabeçalho Retry-After, que especifica quando o cliente pode enviar a próxima solicitação sem que esta seja recusada. Essa estratégia pode ajudar a reduzir as chances de que um ataque DoS (Negação de Serviço) de um conjunto de clientes paralise o sistema.

Gerencie as conexões HTTP persistentes cuidadosamente

O protocolo HTTP oferece suporte a conexões HTTP persistentes nos casos em que elas estão disponíveis. A especificação HTTP 1.0 adicionou a conexão: cabeçalho Keep-Alive que permite que um aplicativo cliente indique ao servidor que ele pode usar a mesma conexão para enviar solicitações posteriores em vez de abrir novas. A conexão será fechada automaticamente se o cliente não reutilizar a conexão dentro de um período definido pelo host. Esse comportamento é o padrão no HTTP 1.1 conforme usado por serviços do Azure, portanto, não é necessário incluir cabeçalhos Keep-Alive nas mensagens.

Manter uma conexão aberta pode ajudar a melhorar a capacidade de resposta, reduzindo a latência e o congestionamento da rede, mas pode ser prejudicial para escalabilidade ao manter conexões desnecessárias abertas por mais tempo que o necessário, limitando a capacidade de conexão de outros clientes simultâneos. Isso também pode afetar vida útil da bateria, se o aplicativo cliente for executado em um dispositivo móvel; se o aplicativo realizará apenas solicitações ocasionais ao servidor, manter uma conexão aberta pode descarregar a bateria mais rapidamente. Para garantir que uma conexão não seja tornada persistente com o HTTP 1.1, o cliente pode incluir um cabeçalho Connection:Close às mensagens, para substituir o comportamento padrão. Do mesmo modo, se um servidor está lidando com um grande número de clientes, ele pode incluir um cabeçalho Connection:Close nas mensagens de resposta que, consequentemente, devem fechar a conexão e poupar os recursos do servidor.

> ⓘ **Observação**
>
> As conexões HTTP persistentes são um recurso puramente opcional para reduzir a sobrecarga de rede associada à repetição no estabelecimento de um canal de comunicação. Nem a API da Web, tampouco o aplicativo cliente dependem de uma conexão HTTP persistente estar disponível. Não use conexões HTTP persistentes para implementar sistemas de notificação de estilo Comet. Em vez disso, você deve usar soquetes (ou soquetes da Web, se disponível) na camada TCP. Finalmente, observe que os cabeçalhos Keep-Alive são de uso limitado se um aplicativo cliente comunica-se com um servidor por meio de um proxy, já que apenas a conexão com o cliente e o proxy será persistente.

Publicação e gerenciamento de uma API da Web

Para disponibilizar uma API da Web para aplicativos cliente, essa API deve ser implantada em um ambiente de host. Esse ambiente é normalmente um servidor Web, embora possa ser algum outro tipo de processo de host. Ao publicar uma API da Web, você deve considerar os seguintes pontos:

- Todas as solicitações devem ser autenticadas e autorizadas; o nível apropriado de controle de acesso deve ser imposto.
- Uma API da Web comercial pode estar sujeita a diversas garantias de qualidade relacionadas a tempos de resposta. Se a carga puder variar significativamente ao longo do tempo, certifique-se de que esse ambiente de host seja escalável.
- Pode ser necessário fazer a medição das solicitações para fins de monetização.
- Talvez seja necessário controlar o fluxo do tráfego para a API da Web, além de implementar limitação para clientes específicos que tenham esgotado suas cotas.
- Requisitos normativos podem obrigar o registro e auditoria de todas as solicitações e respostas.
- Para garantir a disponibilidade, pode ser necessário monitorar a integridade do servidor que hospeda a API da Web e reiniciá-lo se necessário.

É útil poder separar esses problemas dos problemas técnicos relacionados à implementação da API da Web. Por esse motivo, considere a possibilidade de criar uma fachada, em execução como um processo separado e que encaminhe as solicitações para a API da Web. A fachada pode fornecer as operações de gerenciamento e encaminhar solicitações validadas para a API da Web. O uso de uma fachada também pode oferecer muitas vantagens funcionais, incluindo:

- Agir como um ponto de integração para várias APIs da Web.
- Transformar mensagens e converter os protocolos de comunicação para clientes criados por meio de tecnologias diferentes.
- Armazenar solicitações e respostas em cache para reduzir a carga no servidor que hospeda a API da Web.

Teste de uma API da Web

Uma API da web Deve ser testada tão cuidadosamente quanto qualquer outro software. Você deve considerar a criação de testes de unidade para validar a funcionalidade.

A natureza de uma API da Web apresenta seus próprios requisitos adicionais para verificar se ela está funcionando corretamente. Você deve prestar atenção especial aos aspectos a seguir:

- Teste todas as rotas para verificar que elas invocam as operações corretas. Esteja ciente principalmente do código de status HTTP 405 (Método Não Permitido) que

está sendo retornado inesperadamente, já que isso pode indicar uma incompatibilidade entre uma rota e os métodos HTTP (GET, POST, PUT e DELETE) que podem ser enviados para essa rota.

Envie solicitações HTTP para as rotas que não oferecem suporte a elas, como enviar uma solicitação POST para um recurso específico (solicitações POST só devem ser enviadas para coleções de recursos). Nesses casos, a única resposta válida *deve* ser o código de status 405 (Não Permitido).

- Verifique se todas as rotas são protegidas adequadamente e estão sujeitas às verificações apropriadas de autenticação e autorização.

  > ⓘ **Observação**
  >
  > Alguns aspectos de segurança, como autenticação de usuário, muito provavelmente são de responsabilidade do ambiente de host, e não da API da Web; ainda assim, é necessário incluir testes de segurança como parte do processo de implantação.

- Teste o processamento de exceção realizado por cada operação e verifique se uma resposta HTTP apropriada e significativa é passada de volta para o aplicativo cliente.

- Verifique se as mensagens de solicitação e resposta são bem formadas. Por exemplo, se uma solicitação HTTP POST contém os dados para um novo recurso no formato x-www-form-urlencoded, confirme que a operação correspondente analisa corretamente os dados, cria os recursos e retorna uma resposta que contém os detalhes do novo recurso, incluindo o cabeçalho Location correto.

- Verifique todos os links e URIs nas mensagens de resposta. Por exemplo, uma mensagem HTTP POST deve retornar o URI do recurso recém-criado. Todos os links HATEOAS devem ser válidos.

- Certifique-se de que cada operação retorna os códigos de status corretos para diferentes combinações de entrada. Por exemplo:
  - Se uma consulta for bem-sucedida, ela deverá retornar o código de status 200 (OK)
  - Se um recurso não for encontrado, a operação deverá retornar o código de status HTTP 404 (Não Encontrado).
  - Se o cliente envia uma solicitação que exclui um recurso com êxito, o código de status deve ser 204 (Sem Conteúdo).

    - Se o cliente envia uma solicitação que cria um novo recurso, o código de status deve ser 201 (Criado).

Cuidado com códigos de status de resposta inesperados do intervalo 5xx. Normalmente, essas mensagens são relatadas pelo servidor de host para indicar que não foi possível atender a uma solicitação válida.

- Teste as diferentes combinações de cabeçalho de solicitação que um aplicativo cliente pode especificar e certifique-se de que a API da Web retorna as informações esperadas nas mensagens de resposta.

- Teste as cadeias de consulta. Se uma operação aceita parâmetros opcionais (por exemplo, solicitações de paginação), teste as diferentes combinações e ordens dos parâmetros.

- Verifique se as operações assíncronas são concluídas com êxito. Se a API da Web oferece suporte a streaming para solicitações que retornam objetos binários grandes (como vídeo ou áudio), certifique-se de que as solicitações de cliente não são bloqueadas enquanto os dados são transmitidos. Se a API Web implementar a sondagem para operações de modificação de dados de execução longa, verifique se as operações relatam o status corretamente durante o processo.

Você também deve criar e executar testes de desempenho para verificar que a API da Web funciona satisfatoriamente durante emergências. Você pode criar um projeto de teste de carga e desempenho da Web usando o Visual Studio Ultimate.

Uso do Gerenciamento de API do Azure

No Azure, considere o uso do Gerenciamento de API do Azure para publicar e gerenciar uma API Web. Usando esse recurso, é possível gerar um serviço que atua como uma fachada para uma ou mais APIs da Web. O serviço em si é um serviço Web escalonável que você pode criar e configurar usando o portal do Azure. Você pode usar esse serviço para publicar e gerenciar uma API da Web conforme descrito a seguir:

1. Implante a API da Web em um site, Serviço de Nuvem do Azure ou máquina virtual do Azure.

2. Conecte o Serviço de Gerenciamento de API à API da Web. Solicitações enviadas para a URL da API de gerenciamento são mapeadas para URIs na API da Web. O mesmo serviço de Gerenciamento de API pode rotear solicitações para mais de uma API da Web. Isso permite que você agregue várias APIs da Web em um único serviço de gerenciamento. Do mesmo modo, a mesma API da Web pode ser

referenciada de mais de um serviço de Gerenciamento de API, se for necessário restringir ou particionar a funcionalidade disponível para aplicativos diferentes.

> ⓘ **Observação**
>
> Os URIs nos links HATEOAS gerados como parte da resposta para solicitações HTTP GET devem referenciar a URL do serviço de gerenciamento de API, e não o servidor Web hospedando a API da Web.

3. Para cada API da Web, especifique as operações HTTP expostas por essa API, juntamente com quaisquer parâmetros opcionais que uma operação possa utilizar como entrada. Você também pode configurar se o serviço de gerenciamento de API deve ou não armazenar em cache a resposta recebida da API da Web, para reduzir a ocorrência de solicitações repetidas para os mesmos dados. Registre os detalhes das respostas HTTP que cada operação pode gerar. Essas informações são usadas para gerar a documentação para desenvolvedores, portanto, é importante que sejam completas e precisas.

   Você pode definir operações manualmente usando os assistentes fornecidos pelo portal do Azure ou importá-las de um arquivo que contenha as definições no formato WADL ou Swagger.

4. Defina as configurações de segurança para comunicações entre o serviço de gerenciamento de API e o servidor Web que hospeda a API da Web. O serviço de gerenciamento de API atualmente oferece suporte a autenticação Básica e autenticação mútua pelo uso de certificados, além de autorização do usuário OAuth (Open Authorization) 2.0.

5. Criar um produto. Um produto é a unidade de publicação: você adiciona ao produto as APIs da Web que conectou anteriormente ao serviço de gerenciamento. Quando o produto é publicado, APIs da web são disponibilizadas para os desenvolvedores.

   > ⓘ **Observação**
   >
   > Antes de publicar um produto, você também pode definir grupos de usuários que podem acessar o produto e adicionar usuários a esses grupos. Isso lhe confere controle sobre os desenvolvedores e aplicativos que podem usar a API da Web. Se uma API da Web está sujeita a aprovação, um desenvolvedor, antes de poder acessá-la, deve enviar uma solicitação ao administrador do produto. O administrador pode conceder ou negar acesso ao desenvolvedor.

Os desenvolvedores existentes também poderão ser bloqueados se as circunstâncias mudarem.

6. Configure políticas para cada API da Web. Políticas controlam aspectos como permitir ou não chamadas entre domínios, como autenticar clientes, converter ou não de modo transparente entre os formatos de dados XML e JSON, restringir ou não chamadas de um determinado intervalo de IPs, cotas de uso e, por fim, limitar ou não a taxa de chamada. As políticas podem ser aplicadas globalmente no produto inteiro, para uma única API da Web em um produto ou então para operações individuais em uma API da Web.

Para obter mais informações, confira a Documentação do Gerenciamento de API.

> **Dica**
>
> O Azure oferece o Gerenciador de Tráfego do Azure, que permite implementar failover e balanceamento de carga, além de reduzir a latência entre várias instâncias de um site hospedado em diferentes locais geográficos. Você pode usar o Gerenciador de Tráfego do Azure em conjunto com o Serviço de Gerenciamento de API; o Serviço de Gerenciamento de API pode rotear solicitações para instâncias de um site por meio do Gerenciador de Tráfego do Azure. Para obter mais informações, confira **Traffic Manager routing methods** (Métodos de roteamento do Gerenciador de Tráfego).
>
> Nessa estrutura, se você estiver usando nomes DNS personalizados para sites, você deve configurar o registro CNAME apropriado para que cada site aponte para o nome DNS do site do Gerenciador de Tráfego do Azure.

Suporte a desenvolvedores do lado do cliente

Os desenvolvedores construindo aplicativos cliente normalmente exigem informações sobre como acessar a API da Web e documentação sobre os parâmetros, tipos de dados, tipos de retorno e códigos de retorno que descrevem as diferentes solicitações e respostas entre o serviço Web e o aplicativo cliente.

Documentar as operações REST de uma API da Web

O Serviço de Gerenciamento de API do Azure inclui um portal do desenvolvedor que descreve as operações REST expostas por uma API da Web. Se um produto já foi publicado, ele é exibido nesse portal. Os desenvolvedores podem usar esse portal para

inscrever-se para obter acesso; o administrador pode aprovar ou negar a solicitação. Se o desenvolvedor for aprovado, ele recebe uma chave de assinatura que é usada para autenticar as chamadas dos aplicativos cliente que ele desenvolve. Essa chave deve ser fornecida com cada chamada à API da Web; caso contrário, essa chamada será rejeitada.

Esse portal também fornece:

- Documentação do produto, listando as operações que ele expõe, os parâmetros necessários e as diferentes respostas que podem ser retornadas. Observe que essas informações são geradas de detalhes fornecidos na etapa 3 da lista na seção Publicando uma API da Web usando o Serviço de Gerenciamento de API do Microsoft Azure.
- Snippets de código que mostram como invocar operações em várias linguagens, inclusive JavaScript, C#, Java, Ruby, Python e PHP.
- Um console do desenvolvedor, que permite que um desenvolvedor envie uma solicitação HTTP para testar cada operação no produto e visualize os resultados.
- Uma página em que o desenvolvedor pode relatar quaisquer questões ou problemas encontrados.

O portal do Azure permite que você personalize o portal do desenvolvedor para alterar a definição de estilo e o layout de acordo com a identidade visual da sua organização.

Implementar um SDK de cliente

Criar um aplicativo cliente que chama solicitações REST para acessar uma API da Web requer a criação de uma quantidade significativa de código para construir cada solicitação e formatá-la adequadamente, enviar a solicitação ao servidor que hospeda o serviço Web e analisar a resposta para descobrir se a solicitação foi bem-sucedida ou falhou, além de extrair os dados retornados. Para isolar o aplicativo cliente dessas preocupações, você pode fornecer um SDK que encapsula a interface REST e abstrai esses detalhes de baixo nível dentro de um conjunto de métodos mais funcional. Um aplicativo cliente usa esses métodos, os quais convertem transparentemente chamadas em solicitações REST e, em seguida, convertem as respostas de volta em valores de retorno do método. Essa é uma técnica comum implementada por muitos serviços, incluindo o SDK do Azure.

Criar um SDK no lado do cliente é uma tarefa considerável, já que ele precisa ser implementado de modo consistente e testado com cuidado. No entanto, grande parte desse processo pode ser tornado mecânico, e muitos fornecedores oferecem ferramentas que podem automatizar muitas dessas tarefas.

Monitoramento de uma API da Web

Dependendo de como você publicou e implantou a API da Web, você pode monitorá-la diretamente, ou então coletar informações de uso e integridade analisando o tráfego que passa pelo Serviço de Gerenciamento de API.

Monitorando uma API da Web diretamente

Se você tiver implementado sua API da Web usando o modelo de ASP.NET Web API (seja como um projeto de API da Web, ou como uma função web em um serviço de nuvem do Azure) e o Visual Studio 2013, você pode coletar dados de uso, desempenho e disponibilidade por meio do Application Insights para ASP.NET. O Application Insights é um pacote que rastreia e registra de modo transparente as informações sobre solicitações e respostas quando a API da Web é implantada na nuvem; uma vez que o pacote está instalado e configurado, você não precisa modificar nenhum código em sua API da Web para usá-la. Quando você implanta a API da Web em um site do Azure, todo o tráfego é examinado e as estatísticas a seguir são coletadas:

- Tempo de resposta do servidor.
- Número de solicitações do servidor e os detalhes de cada solicitação.
- As solicitações mais lentas em termos de tempo médio de resposta.
- Os detalhes de quaisquer eventuais solicitações com falha.
- O número de sessões iniciadas por agentes de usuário e navegadores diferentes.
- As páginas exibidas mais frequentemente (útil principalmente para aplicativos da Web em vez de APIs Web).
- As diferentes funções de usuário ao acessar a API da Web.

Veja esses dados em tempo real no portal do Azure. Crie também testes na Web que monitorem a integridade da API Web. Um teste na Web envia uma solicitação periódica para um URI especificado na API Web e captura a resposta. Você pode especificar a definição de uma resposta bem-sucedida (como o código de status HTTP 200) e, se a solicitação não retornar essa resposta, você pode fazer com que um alerta seja enviado ao administrador. Se necessário, o administrador pode reiniciar o servidor que hospeda a API da Web caso ele tenha falhado.

Para obter mais informações, confira Application Insights - Introdução ao ASP.NET.

Monitoramento de uma API da Web por meio do Serviço de Gerenciamento de API

Se você publicou sua API Web usando o serviço Gerenciamento de API, a página Gerenciamento de API no portal do Azure contém um painel que permite ver o desempenho geral do serviço. A página Análise permite aprofundar-se nos detalhes de como o produto está sendo usado. Essa página contém as seguintes guias:

- **Uso**. Essa guia fornece informações sobre o número de chamadas de API feitas e a largura de banda usada para lidar com essas chamadas ao longo do tempo. Você pode filtrar os detalhes de uso por produto, API e operação.
- **Integridade**. Essa guia permite que você visualize o resultado de solicitações de API (os códigos de status HTTP retornados), a eficácia da política de cache, o tempo de resposta de API e o tempo de resposta do serviço. Novamente, você pode filtrar os detalhes de integridade por produto, API e operação.
- **Atividade**. Essa guia fornece um resumo de texto dos números de chamadas bem-sucedidas, chamadas com falha, chamadas bloqueadas, tempo médio de resposta e tempos de resposta para cada produto, API da Web e operação. Esta página também lista o número de chamadas feitas por cada desenvolvedor.
- **Visão rápida**. Essa guia exibe um resumo dos dados de desempenho, incluindo os desenvolvedores responsáveis por fazer a maioria das chamadas de API, além dos produtos, APIs da Web e operações que receberam essas chamadas.

Você pode usar essas informações para determinar se uma operação ou API da Web específica está agindo como um gargalo e, se necessário, dimensionar o ambiente de host e adicionar mais servidores. Você também pode verificar se um ou mais aplicativos estão usando um volume desproporcional de recursos e aplicam as políticas adequadas para definir cotas e limitar as taxas de chamada.

> ⓘ **Observação**
>
> Você pode alterar os detalhes de um produto publicado, as alterações serão aplicadas imediatamente. Por exemplo, você pode adicionar ou remover uma operação de uma API da Web sem precisar publicar novamente o produto que contém essa API.

Próximas etapas

- OData para ASP.NET Web API : contém exemplos e mais informações sobre como implementar uma API da Web com protocolo OData usando o ASP.NET.
- Apresentando o suporte a lotes na API da Web e OData para API da Web : descreve como implementar operações em lote em uma API da Web usando o protocolo OData.

- O artigo Idempotency patterns (Padrões de idempotência), no blog de Jonathan Oliver, fornece uma visão geral da idempotência e de como ela se relaciona às operações de gerenciamento de dados.
- Definições de código de status, no site do W3C, contém uma lista completa dos códigos de status HTTP e as respectivas descrições.
- Executar tarefas em segundo plano com WebJobs: fornece informações e exemplos sobre como usar os WebJobs para realizar operações em segundo plano.
- Notificação de usuários nos Hubs de Notificação do Azure mostra como usar um Hub de notificação do Azure para enviar respostas assíncronas a aplicativos cliente.
- Gerenciamento de API: descreve como publicar um produto que fornece acesso controlado e seguro a uma API da Web.
- Referência de API REST de Gerenciamento de API do Azure descreve como usar a API REST de Gerenciamento de API para criar aplicativos de gerenciamento personalizados.
- Métodos de roteamento do Gerenciador de Tráfego resume como o Gerenciador de Tráfego do Azure pode ser usado para realizar o balanceamento de carga de solicitações entre várias instâncias de um site que hospeda uma API da Web.
- Application Insights - Introdução ao ASP.NET: fornece informações detalhadas sobre como instalar e configurar o Application Insights em um projeto de ASP.NET Web API.

Comentários