Tutorial: Creación de una API web con ASP.NET Core

Artículo • 06/10/2022 • Tiempo de lectura: 54 minutos

Por Rick Anderson y Kirk Larkin

En este tutorial se enseñan los conceptos básicos de la compilación de una API web con ASP.NET Core.

En este tutorial aprenderá a:

- ✓ Crear un proyecto de API web.
- ✓ Agregar una clase de modelo y un contexto de base de datos.
- ✓ Aplicar scaffolding a un controlador con métodos CRUD.
- ✓ Configurar el enrutamiento, las rutas de acceso URL y los valores devueltos.
- ✓ Llame a la API web con http-repl.

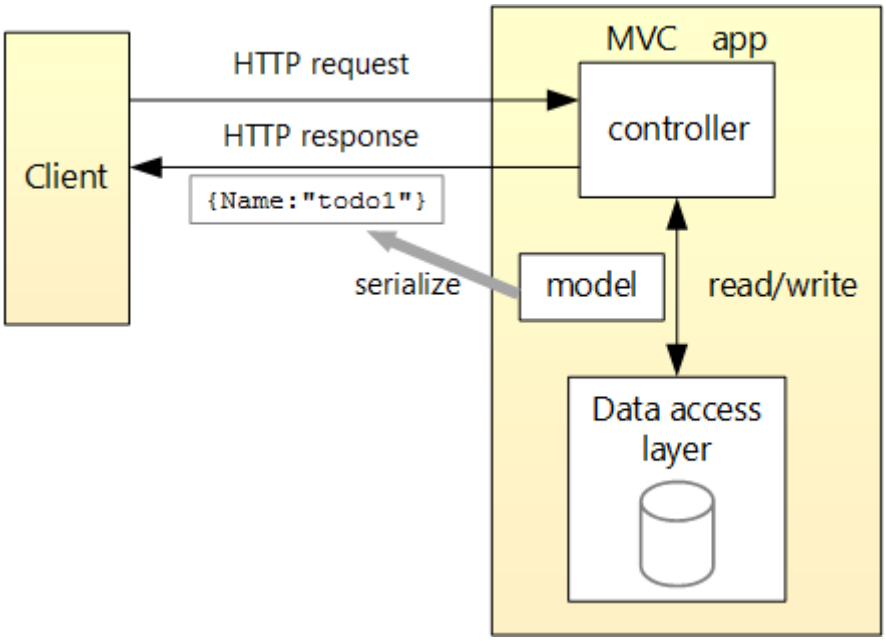
Al final, tendrá una API web que pueda administrar los elementos "to-do" almacenados en una base de datos.

Información general

En este tutorial se crea la siguiente API:

API	Descripción	Cuerpo de la solicitud	Cuerpo de la respuesta
`GET /api/todoitems`	Obtener todas las tareas pendientes	None	Matriz de tareas pendientes
`GET /api/todoitems/{id}`	Obtener un elemento por identificador	None	Tarea pendiente
`POST /api/todoitems`	Incorporación de un nuevo elemento	Tarea pendiente	Tarea pendiente
`PUT /api/todoitems/{id}`	Actualizar un elemento existente	Tarea pendiente	None
`DELETE /api/todoitems/{id}`	Eliminación de un elemento	None	None

En el diagrama siguiente, se muestra el diseño de la aplicación.



Requisitos previos

Visual Studio

- Visual Studio 2022 con la carga de trabajo de **ASP.NET y desarrollo web**

Creación de un proyecto web

Visual Studio

- En el menú **Archivo**, seleccione **Nuevo**>**Proyecto**.
- Escriba *API web* en el cuadro de búsqueda.
- Seleccione la plantilla **API web de ASP.NET Core** y seleccione **Siguiente**.
- En el **cuadro de diálogo Configurar el nuevo proyecto**, asigne al proyecto el nombre *TodoApi* y seleccione **Siguiente**.
- En el cuadro de diálogo **Información adicional**:
  - Confirme que el **Marco** es **.NET 6.0 (Compatibilidad a largo plazo)**.
  - Confirme que la casilla **Use controllers(uncheck to use minimal APIs)** [Usar controladores (desactivar para usar API mínimas)] está activada.
  - Seleccione **Crear**.

> **Nota**

Para obtener instrucciones sobre cómo agregar paquetes a aplicaciones .NET, consulte los artículos de *Instalación y administración de paquetes* en **Flujo de trabajo de consumo de paquetes (NuGet documentación).** Confirme las versiones correctas del paquete en **NuGet.org** .

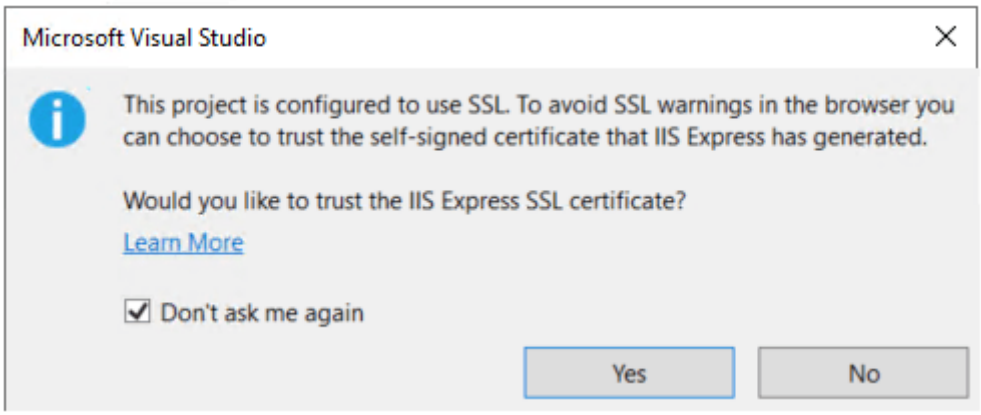
Prueba del proyecto

La plantilla de proyecto crea una API `WeatherForecast` compatible con Swagger.

Visual Studio

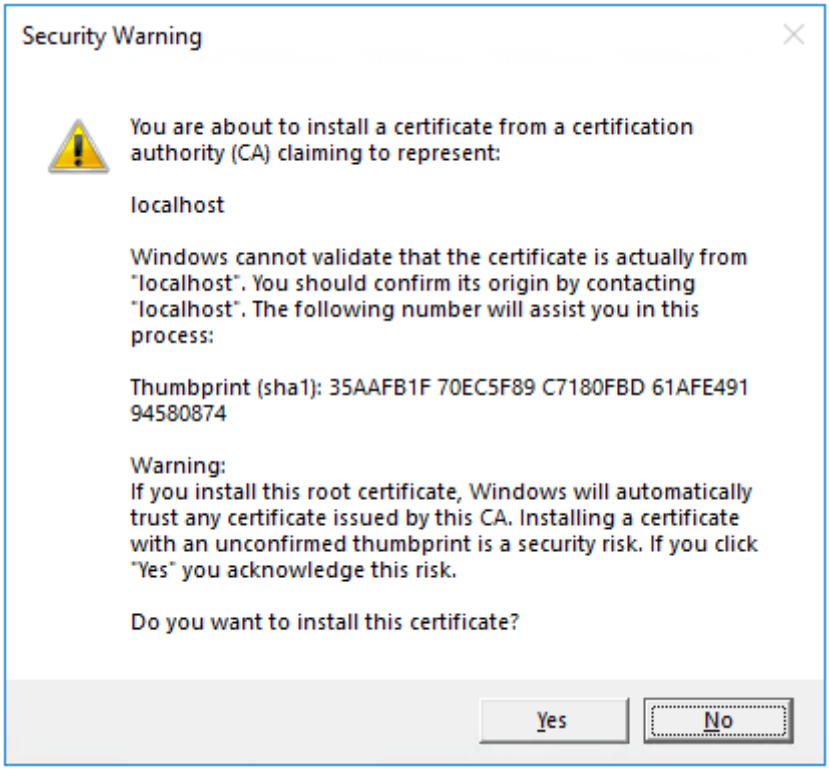
Presione Ctrl+F5 para ejecutarla sin el depurador.

Visual Studio muestra el siguiente cuadro de diálogo cuando un proyecto aún no está configurado para usar SSL:



Haga clic en **Sí** si confía en el certificado SSL de IIS Express.

Se muestra el cuadro de diálogo siguiente:



Si acepta confiar en el certificado de desarrollo, seleccione **Sí**.

Para obtener información sobre cómo confiar en el explorador Firefox, consulte Error de certificado SEC_ERROR_INADEQUATE_KEY_USAGE de Firefox.

Visual Studio inicia el explorador predeterminado y navega hasta `https://localhost:<port>/swagger/index.html`, donde `<port>` es un número de puerto elegido aleatoriamente.

Se abre la página de Swagger `/swagger/index.html`. Seleccione **GET**>**Try it out**>**Execute** (GET > Probar > Ejecutar). La página muestra lo siguiente:

- Comando de Curl para probar la API WeatherForecast
- Dirección URL para probar la API WeatherForecast
- Código de respuesta, cuerpo y encabezados
- Cuadro de lista desplegable con los tipos de medios y el esquema y valor de ejemplo

Si no aparece la página Swagger, consulte este problema de GitHub .

Swagger se usa para generar una documentación útil y páginas de ayuda para API web. Este tutorial se centra en cómo crear una API web. Para más información sobre Swagger, vea Documentación de la API web de ASP.NET Core con Swagger/OpenAPI.

Copie y pegue la **URL de solicitud** en el explorador: `https://localhost:<port>/weatherforecast`.

Se devuelve un JSON similar al siguiente ejemplo:

JSON


```
[
    {
        "date": "2019-07-16T19:04:05.7257911-06:00",
        "temperatureC": 52,
        "temperatureF": 125,
        "summary": "Mild"
    },
    {
        "date": "2019-07-17T19:04:05.7258461-06:00",
        "temperatureC": 36,
        "temperatureF": 96,
        "summary": "Warm"
    },
    {
        "date": "2019-07-18T19:04:05.7258467-06:00",
        "temperatureC": 39,
        "temperatureF": 102,
        "summary": "Cool"
    },
    {
        "date": "2019-07-19T19:04:05.7258471-06:00",
        "temperatureC": 10,
        "temperatureF": 49,
        "summary": "Bracing"
    },
    {
        "date": "2019-07-20T19:04:05.7258474-06:00",
        "temperatureC": -1,
        "temperatureF": 31,
        "summary": "Chilly"
    }
]
```


Actualización de launchUrl

En *Properties\launchSettings.json*, actualice `launchUrl` de `"swagger"` a `"api/todoitems"`:

JSON

```
"launchUrl": "api/todoitems",
```

Dado que Swagger se quitará, el marcado anterior cambia la dirección URL que se inicia con el método GET del controlador agregado en las secciones siguientes.

Incorporación de una clase de modelo

Un *modelo* es un conjunto de clases que representan los datos que la aplicación administra. El modelo para esta aplicación es una clase `TodoItem` única.

Visual Studio

- En el **Explorador de soluciones**, haga clic con el botón derecho en el proyecto. Seleccione **Agregar**>**Nueva carpeta**. Asigne a la carpeta el nombre `Models`.
- Haga clic con el botón derecho en la carpeta `Models` y seleccione **Agregar**>**Clase**. Asigne a la clase el nombre *TodoItem* y seleccione **Agregar**.
- Reemplace el código de plantilla por lo siguiente:

C#

```
namespace TodoApi.Models
{
    public class TodoItem
    {
        public long Id { get; set; }
        public string? Name { get; set; }
        public bool IsComplete { get; set; }
    }
}
```

La propiedad `Id` funciona como clave única en una base de datos relacional.

Las clases de modelo pueden ir en cualquier lugar del proyecto, pero, por convención, se usa la carpeta `Models` .

Incorporación de un contexto de base de datos

El *contexto de base de datos* es la clase principal que coordina la funcionalidad de Entity Framework para un modelo de datos. Esta clase se crea derivándola de la clase Microsoft.EntityFrameworkCore.DbContext.

Visual Studio

Adición de paquetes NuGet

- En el menú **Herramientas**, seleccione **Administrador de paquetes NuGet** > **Administrar paquetes NuGet para la solución**.
- Seleccione la pestaña **Examinar** y escriba `Microsoft.EntityFrameworkCore.InMemory` en el cuadro de búsqueda.
- En el panel izquierdo, seleccione `Microsoft.EntityFrameworkCore.InMemory`.
- Active la casilla **Proyecto** en el panel derecho y, después, seleccione **Instalar**.

Adición del contexto de la base de datos TodoContext

- Haga clic con el botón derecho en la carpeta `Models` y seleccione **Agregar**>**Clase**. Asigne a la clase el nombre *TodoContext* y haga clic en **Agregar**.

- Escriba el siguiente código:

C#

```
using Microsoft.EntityFrameworkCore;
using System.Diagnostics.CodeAnalysis;

namespace TodoApi.Models
{
    public class TodoContext : DbContext
    {
        public TodoContext(DbContextOptions<TodoContext> options)
            : base(options)
        {
        }

        public DbSet<TodoItem> TodoItems { get; set; } = null!;
    }
}
```

Registro del contexto de base de datos

En ASP.NET Core, los servicios (como el contexto de la base de datos) deben registrarse con el contenedor de inserción de dependencias (DI). El contenedor proporciona el servicio a los controladores.

Actualice `Program.cs` con el siguiente código:

C#

```
using Microsoft.EntityFrameworkCore;
using TodoApi.Models;


var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();
builder.Services.AddDbContext<TodoContext>(opt =>
    opt.UseInMemoryDatabase("TodoList"));
//builder.Services.AddSwaggerGen(c =>
//{
//    c.SwaggerDoc("v1", new() { Title = "TodoApi", Version = "v1" });
//});

var app = builder.Build();

// Configure the HTTP request pipeline.
if (builder.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
    //app.UseSwagger();
    //app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json",
"TodoApi v1"));
}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

El código anterior:

- Quita las llamadas de Swagger.
- Quita las directivas `using` sin usar.
- Agrega el contexto de base de datos para el contenedor de DI.
- Especifica que el contexto de base de datos usará una base de datos en memoria.

Scaffolding de un controlador

Visual Studio

- Haga clic con el botón derecho en la carpeta *Controllers*.
- Seleccione **Agregar**>**Nuevo elemento con scaffolding**.
- Seleccione **Controlador de API con acciones mediante Entity Framework** y, después, seleccione **Agregar**.
- En el cuadro de diálogo **Add API Controller with actions, using Entity Framework** (Agregar controlador de API con acciones mediante Entity Framework):
  - Seleccione **TodoItem (TodoApi.Models)** en **Clase de modelo**.
  - Seleccione **TodoContext (TodoApi.Models)** en **Clase de contexto de datos**.
  - Seleccione **Agregar**.

  Si se produce un error en la operación de scaffolding, seleccione **Agregar** para probar el scaffolding una segunda vez.

El código generado:

- Marca la clase con el atributo [ApiController]. Este atributo indica que el controlador responde a las solicitudes de la API web. Para información sobre comportamientos específicos que permite el atributo, vea Creación de una API web con ASP.NET Core.
- Utiliza la inserción de dependencias para insertar el contexto de base de datos (`TodoContext`) en el controlador. El contexto de base de datos se usa en cada uno de los métodos CRUD del controlador.

Las plantillas de ASP.NET Core para:

- Controladores con vistas incluyen `[action]` en la plantilla de ruta.
- Controladores de API no incluyen `[action]` en la plantilla de ruta.

Si el token `[action]` no está en la plantilla de ruta, el nombre de la acción se excluye de la ruta. Es decir, el nombre del método asociado a la acción no se usa en la ruta coincidente.

Actualización del método create de PostTodoItem

Actualice la instrucción "return" en `PostTodoItem` para usar el operador nameof:

C#

```
[HttpPost]
public async Task<ActionResult<TodoItem>> PostTodoItem(TodoItem todoItem)
{
    _context.TodoItems.Add(todoItem);
    await _context.SaveChangesAsync();

    //return CreatedAtAction("GetTodoItem", new { id = todoItem.Id }, todoI-
tem);
    return CreatedAtAction(nameof(GetTodoItem), new { id = todoItem.Id },
todoItem);
}
```

El código anterior es un método HTTP POST, indicado por el atributo [HttpPost]. El método obtiene el valor de tareas pendientes del cuerpo de la solicitud HTTP.

Para más información, vea Enrutamiento mediante atributos con atributos Http[Verb].

El método CreatedAtAction realiza las acciones siguientes:

- Devuelve un código de estado HTTP 201 cuando se ha ejecutado correctamente. HTTP 201 es la respuesta estándar para un método HTTP POST que crea un recurso en el servidor.
- Agrega un encabezado Location a la respuesta. El encabezado `Location` especifica el URI de la tarea pendiente recién creada. Para obtener más información, consulte 10.2.2 201 creado .
- Hace referencia a la acción `GetTodoItem` para crear el identificador URI del encabezado `Location`. La palabra clave `nameof` de C# se usa para evitar que se codifique de forma rígida el nombre de acción en la llamada a `CreatedAtAction`.

Instalación de http-repl

En este tutorial se usa http-repl para probar la API web.

- Ejecute el comando siguiente en el símbolo del sistema:

  CLI de .NET

  ```
  dotnet tool install -g Microsoft.dotnet-httprepl
  ```

- Si no tiene instalados el entorno de ejecución ni el SDK de .NET 6.0, instale el entorno de ejecución de .NET 6.0 .

Prueba de PostTodoItem

- Presione Ctrl+F5 para ejecutar la aplicación.
- Abra una nueva ventana de terminal y ejecute los siguientes comandos. Si la aplicación usa un número de puerto diferente, reemplace 5001 en el comando httpprepl por su número de puerto.

CLI de .NET

```
httprepl https://localhost:5001/api/todoitems
post -h Content-Type=application/json -c "{"name":"walk dog","isCom-
plete":true}"
```

Este es un ejemplo del resultado del comando:

Resultados

```
HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
Date: Tue, 07 Sep 2021 20:39:47 GMT
Location: https://localhost:5001/api/TodoItems/1
Server: Kestrel
Transfer-Encoding: chunked

{
  "id": 1,
  "name": "walk dog",
  "isComplete": true
}
```

Prueba del URI del encabezado de ubicación

Para probar el encabezado de ubicación, cópielo y péguelo en un comando `get` httpprepl.

En el ejemplo siguiente se supone que todavía está en una sesión httpprepl. Si finalizó la sesión httpprepl anterior, reemplace `connect` por `httprepl` en los siguientes comandos:

CLI de .NET

```
connect https://localhost:5001/api/todoitems/1
get
```

Este es un ejemplo del resultado del comando:

Resultados

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Tue, 07 Sep 2021 20:48:10 GMT
Server: Kestrel
Transfer-Encoding: chunked

{
  "id": 1,
  "name": "walk dog",
  "isComplete": true
}
```

Examen de los métodos GET

Se implementan dos puntos de conexión GET:

- `GET /api/todoitems`
- `GET /api/todoitems/{id}`

Acaba de ver un ejemplo de la ruta `/api/todoitems/{id}`. Pruebe la ruta `/api/todoitems`:

CLI de .NET

```
connect https://localhost:5001/api/todoitems
get
```

Este es un ejemplo del resultado del comando:

Resultados

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Tue, 07 Sep 2021 20:59:21 GMT
Server: Kestrel
Transfer-Encoding: chunked

[
  {
    "id": 1,
    "name": "walk dog",
    "isComplete": true
  }
]
```

Esta vez, el JSON devuelto es una matriz de un elemento.

Esta aplicación utiliza una base de datos en memoria. Si la aplicación se detiene y se inicia, la solicitud GET precedente no devolverá ningún dato. Si no se devuelve ningún dato, publique los datos en la aplicación con POST.

Enrutamiento y rutas URL

El atributo [HttpGet] indica un método que responde a una solicitud HTTP GET. La ruta de dirección URL para cada método se construye como sigue:

- Comience por la cadena de plantilla en el atributo `Route` del controlador:

  C#

  ```
  [Route("api/[controller]")]
  [ApiController]
  public class TodoItemsController : ControllerBase
  ```

- Reemplace `[controller]` por el nombre del controlador, que convencionalmente es el nombre de clase de controlador sin el sufijo "Controller". En este ejemplo, el nombre de clase de controlador es **TodoItems**Controller; por tanto, el nombre del controlador es "TodoItems". El enrutamiento en ASP.NET Core no distingue entre mayúsculas y minúsculas.

- Si el atributo `[HttpGet]` tiene una plantilla de ruta (por ejemplo, `[HttpGet("products")]`), anéxela a la ruta de acceso. En este ejemplo no se usa una plantilla. Para más información, vea Enrutamiento mediante atributos con atributos Http[Verb].

En el siguiente método `GetTodoItem`, `"{id}"` es una variable de marcador de posición correspondiente al identificador único de la tarea pendiente. Al invocar a `GetTodoItem`, el valor `"{id}"` de la URL se proporciona al método en su parámetro `id`.

C#

```
[HttpGet("{id}")]
public async Task<ActionResult<TodoItem>> GetTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }
```

```
    return todoItem;
}
```

Valores devueltos

El tipo de valor devuelto de los métodos `GetTodoItems` y `GetTodoItem` es ActionResult<T> type. ASP.NET Core serializa automáticamente el objeto a JSON y escribe el JSON en el cuerpo del mensaje de respuesta. El código de respuesta de este tipo de valor devuelto es 200 OK, suponiendo que no haya ninguna excepción no controlada. Las excepciones no controladas se convierten en errores 5xx.

Los tipos de valores devueltos `ActionResult` pueden representar una gama amplia de códigos de estado HTTP. Por ejemplo, `GetTodoItem` puede devolver dos valores de estado diferentes:

- Si no hay ningún elemento que coincida con el identificador solicitado, el método devolverá un código de error de estado 404 NotFound.
- En caso contrario, el método devuelve 200 con un cuerpo de respuesta JSON. Devolver `item` genera una respuesta HTTP 200.

Método PutTodoItem

Examine el método `PutTodoItem`:

C#

```
[HttpPut("{id}")]
public async Task<IActionResult> PutTodoItem(long id, TodoItem todoItem)
{
    if (id != todoItem.Id)
    {
        return BadRequest();
    }

    _context.Entry(todoItem).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!TodoItemExists(id))
        {
            return NotFound();
```

```
        }
        else
        {
            throw;
        }
    }

    return NoContent();
}
```

`PutTodoItem` es similar a `PostTodoItem`, salvo por el hecho de que usa HTTP PUT. La respuesta es 204 Sin contenido . Según la especificación HTTP, una solicitud PUT requiere que el cliente envíe toda la entidad actualizada, no solo los cambios. Para admitir actualizaciones parciales, use HTTP PATCH.

Si recibe un error al llamar a `PutTodoItem` en la siguiente sección, llame a `GET` para asegurarse de que hay un elemento en la base de datos.

Prueba del método PutTodoItem

En este ejemplo se usa una base de datos en memoria que se debe inicializar cada vez que se inicia la aplicación. Debe haber un elemento en la base de datos antes de que realice una llamada PUT. Llame a GET para asegurarse de que hay un elemento en la base de datos antes de realizar una llamada PUT.

Actualice el elemento to-do que tiene el Id = 1 y establezca su nombre en `"feed fish"`:

CLI de .NET

```
connect https://localhost:5001/api/todoitems/1
put -h Content-Type=application/json -c "{"id":1,"name":"feed fish","isCom-
plete":true}"
```

Este es un ejemplo del resultado del comando:

Resultados

```
HTTP/1.1 204 No Content
Date: Tue, 07 Sep 2021 21:20:47 GMT
Server: Kestrel
```

Método DeleteTodoItem

Examine el método `DeleteTodoItem`:

C#

```
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);
    if (todoItem == null)
    {
        return NotFound();
    }

    _context.TodoItems.Remove(todoItem);
    await _context.SaveChangesAsync();

    return NoContent();
}
```

Prueba del método DeleteTodoItem

Elimine la tarea pendiente con el identificador 1:

CLI de .NET

```
connect https://localhost:5001/api/todoitems/1
delete
```

Este es un ejemplo del resultado del comando:

Resultados

```
HTTP/1.1 204 No Content
Date: Tue, 07 Sep 2021 21:43:00 GMT
Server: Kestrel
```

Prevención del exceso de publicación

Actualmente, la aplicación de ejemplo expone todo el objeto `TodoItem`. Las aplicaciones de producción suelen limitar los datos que se escriben y se devuelven mediante un subconjunto del modelo. Hay varias razones para ello y la seguridad es una de las principales. El subconjunto de un modelo se suele conocer como un objeto de transferencia de datos (DTO), modelo de entrada o modelo de vista. En este tutorial, se usa **DTO**.

Se puede usar un DTO para:

- Evitar el exceso de publicación.
- Ocultar las propiedades que los clientes no deben ver.
- Omitir algunas propiedades para reducir el tamaño de la carga.
- Acoplar los gráficos de objetos que contienen objetos anidados. Los gráficos de objetos acoplados pueden ser más cómodos para los clientes.

Para mostrar el enfoque del DTO, actualice la clase `TodoItem` a fin de que incluya un campo secreto:

C#

```
namespace TodoApi.Models
{
    public class TodoItem
    {
        public long Id { get; set; }
        public string? Name { get; set; }
        public bool IsComplete { get; set; }
        public string? Secret { get; set; }
    }
}
```

El campo secreto debe ocultarse en esta aplicación, pero una aplicación administrativa podría decidir exponerlo.

Compruebe que puede publicar y obtener el campo secreto.

Cree un modelo de DTO:

C#

```
namespace TodoApi.Models
{
    public class TodoItemDTO
    {
        public long Id { get; set; }
        public string? Name { get; set; }
        public bool IsComplete { get; set; }
    }
}
```

Actualice el valor de `TodoItemsController` para usar `TodoItemDTO`:

C#

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using TodoApi.Models;
```

```
namespace TodoApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class TodoItemsController : ControllerBase
    {
        private readonly TodoContext _context;

        public TodoItemsController(TodoContext context)
        {
            _context = context;
        }

        // GET: api/TodoItems
        [HttpGet]
        public async Task<ActionResult<IEnumerable<TodoItemDTO>>> GetTodoI-
tems()
        {
            return await _context.TodoItems
                .Select(x => ItemToDTO(x))
                .ToListAsync();
        }

        // GET: api/TodoItems/5
        [HttpGet("{id}")]
        public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
        {
            var todoItem = await _context.TodoItems.FindAsync(id);

            if (todoItem == null)
            {
                return NotFound();
            }

            return ItemToDTO(todoItem);
        }
        // PUT: api/TodoItems/5
        // To protect from overposting attacks, see
https://go.microsoft.com/fwlink/?linkid=2123754
        [HttpPut("{id}")]
        public async Task<IActionResult> UpdateTodoItem(long id, TodoItemDTO
todoItemDTO)
        {
            if (id != todoItemDTO.Id)
            {
                return BadRequest();
            }

            var todoItem = await _context.TodoItems.FindAsync(id);
            if (todoItem == null)
            {
                return NotFound();
            }
```

```
            todoItem.Name = todoItemDTO.Name;
            todoItem.IsComplete = todoItemDTO.IsComplete;

            try
            {
                await _context.SaveChangesAsync();
            }
            catch (DbUpdateConcurrencyException) when (!TodoItemExists(id))
            {
                return NotFound();
            }

            return NoContent();
        }
        // POST: api/TodoItems
        // To protect from overposting attacks, see
https://go.microsoft.com/fwlink/?linkid=2123754
        [HttpPost]
        public async Task<ActionResult<TodoItemDTO>>
CreateTodoItem(TodoItemDTO todoItemDTO)
        {
            var todoItem = new TodoItem
            {
                IsComplete = todoItemDTO.IsComplete,
                Name = todoItemDTO.Name
            };

            _context.TodoItems.Add(todoItem);
            await _context.SaveChangesAsync();

            return CreatedAtAction(
                nameof(GetTodoItem),
                new { id = todoItem.Id },
                ItemToDTO(todoItem));
        }

        // DELETE: api/TodoItems/5
        [HttpDelete("{id}")]
        public async Task<IActionResult> DeleteTodoItem(long id)
        {
            var todoItem = await _context.TodoItems.FindAsync(id);

            if (todoItem == null)
            {
                return NotFound();
            }

            _context.TodoItems.Remove(todoItem);
            await _context.SaveChangesAsync();

            return NoContent();
        }

        private bool TodoItemExists(long id)
        {
```

```
            return _context.TodoItems.Any(e => e.Id == id);
        }

        private static TodoItemDTO ItemToDTO(TodoItem todoItem) =>
            new TodoItemDTO
            {
                Id = todoItem.Id,
                Name = todoItem.Name,
                IsComplete = todoItem.IsComplete
            };
    }
}
```

Compruebe que no puede publicar ni obtener el campo secreto.

Llamar a la API web con JavaScript

Consulte Tutorial: Llamada a una API web de ASP.NET Core con JavaScript.

Agregar compatibilidad con la autenticación a una API web

ASP.NET Core Identity agrega la funcionalidad de inicio de sesión de la interfaz de usuario (IU) a las aplicaciones web de ASP.NET Core. Para proteger las API web y las SPA, use una de las siguientes opciones:

- Azure Active Directory
- Azure Active Directory B2C (Azure AD B2C)
- Duende Identity Server

Duende Identity Server es un marco de OpenID Connect y OAuth 2.0 para ASP.NET Core. Duende Identity Server permite las siguientes características de seguridad:

- Autenticación como servicio (AaaS)
- Inicio de sesión único (SSO) mediante varios tipos de aplicaciones
- Control de acceso para API
- Federation Gateway

> **ⓘ Importante**
>
> **Duende Software** puede requerir que pague una tarifa de licencia para el uso en producción de Duende Identity Server. Para más información, vea **Migración de**

ASP.NET Core 5.0 a 6.0.

Para más información, consulte la documentación de Duende Identity Server (sitio web de Duende Software) .

Publicar en Azure

Para obtener más información sobre la implementación en Azure, consulte Inicio rápido: Implementación de una aplicación web de ASP.NET.

Recursos adicionales

Vea o descargue el código de ejemplo para este tutorial . Vea cómo descargarlo.

Para obtener más información, vea los siguientes recursos:

- Creación de API web con ASP.NET Core
- Documentación de la API web de ASP.NET Core con Swagger/OpenAPI
- Razor Pages con Entity Framework Core en ASP.NET Core: Tutorial 1 de 8
- Enrutar a acciones de controlador de ASP.NET Core
- Tipos de valor devuelto de acción del controlador en la API web de ASP.NET Core
- Implementar aplicaciones de ASP.NET Core en Azure App Service
- Hospedaje e implementación de ASP.NET Core
- Creación de una API web con ASP.NET Core