Crear una web API

Efraín Serna

August 14, 2020

Contents

1 Iniciando con web API

1. Crear un nuevo proyecto de aplicación web con *ASP.NET Core*
2. Elegir una plantilla de *API*
3. Se agrega una carpeta de *Models* y se agregan clases nuevas de modelos de datos.
4. Como se van a utilizar datos desde una Base de Datos, se agrega un contexto de datos de *Entity Framework*
   (a) Abrir el administrador de paquetes NuGet
   (b) Buscar *Microsoft.EntityFrameworkCore*
   (c) Instalar los que terminan en *SqlServer* y *InMemory*
   (d) Dentro de los modelos agregar una clase para el contexto de datos con un nombre terminado en *Context*, por ejemplo: *ProductosContext* o *EscuelaContext*. En verde está lo que se personaliza.

```
using Microsoft.EntityFrameworkCore;

namespace [Proyecto].Models {
  public class [Nombre]Context : DbContext {
    // constructor
```

```
    public [Nombre]Context(
      DbContextOptions<[Nombre]Context> options)
      : base(options) {
    }

    public DbSet<[ClaseModelo]> [Modelo]Items { get; set; }
  }
}
```

(e) Registrar el contexto de datos modificando *Startup.cs* agregando el código en azul

```
public void ConfigureServices(IServiceCollection services) {
  services.AddDbContext<[Nombre]Context>(opt =>
  opt.UseInMemoryDatabase("TodoList"));
  services.AddControllers();
}
```

5. Agregar dentro de la carpeta *Controllers* un controlador (aquí lo pongo como una clase nueva y vacía cuyo nombre termina en *Controller*)

   (a) Agregar una nueva clase

   (b) Agregar los decoradores a la declaración de la clase y hacerla heredar (en verde lo que es personalizable):

   ```
   [ApiController, Route("[Ruta]")]
   public class [Nombre]Controller : ControllerBase
   }
   ```

   (c) Por buena práctica se declara el contexto de manera global a la clase controladora

2 Creación y llamado de métodos

HttpGet Se suele utilizar para hacer petiiciones de información donde los parámetros pueden viajar en la URL. En el decorador se puede especificar la forma de realizar el llamado. Si solo se indican argumentos entre llaves, entonces en la URL de la API no se indica el nombre del método. De manera personal prefiero especificar el nombre de este.

```
[HttpGet("[Identificador de llamada]/{arg1}")]
public async Task<ActionResult> initContext(int arg1) {
// ... código ...
  return Ok(resultado);
}
```

El siguiente ejemplo muestra cómo devolver un dato complejo.

```
[HttpGet("[Identificador]/{id}")]
public async Task<ActionResult<clase modelo>> getItem(int id) {
  var _item = await _context.Colección de entidades.FindAsync(id);
  if (_item == null)
    return NotFound();
  else
    return _item;
}
```

Para devolver una colección de elementos se puede hacer como en el siguiente ejemplo:

```
[HttpGet("[Identificador]")]
public async Task<IEnumerable<ActionResult<clase modelo>>> getItem() {
  return await _context.Colección de entidades.ToListAsync();
}
```

HttpPost Se suele utilizar para operaciones de inserción de registros. La información viaja encapsulada como mecanismo de seguridad, y si se hace por medio del protocolo *https*, se cifran los datos.

```
[HttpPost]
public async Task<ActionResult<clase modelo>> PostItem(
  Iclase modelo item) {
  _context.coleccion de entidades.Add(item);
  await _context.SaveChangesAsync();
  // regresa el elemento recién agregado llamando al método
  // correspondiente al HttpGet
  return CreatedAtAction(nameof(getItem), new {id=item.Id}, item );
}
```

HttpPut Se utiliza para operaciones de actualización. Se le pasan parámetros, por ejemplo el identificador y el objeto a modificar, de aquí se puede hacer una validación de identificadores o algo adicional antes de realizar la actualización. Como ejemplo está el siguiente fragmento de código.

```
[HttpPut("{id}")]
public async Task<ActionResult> PutItem (int id, clase modelo item) {
  if (id != item.Id)
    return BadRequest();
  _context.Entry(item).State = EntityState.Modified;
  try {
    await _context.SaveChangesAsync();
  } catch (DbUpdateConcurrencyException DBUCEx) {
    if (!itemCompraExiste(id))
      return NotFound();
```

```
    else
      throw;
    }
  return NoContent();
}

private bool itemCompraExiste(int id) =>
  _context.colección de entidades.Any(i => i.Id.Equals(id));
```

HttpDelete Se utiliza para el eliminado de información. Se le pasa como argumento el identificador del elemento a borrar. El siguiente fragmento es un ejemplo:

```
[HttpDelete("{id}")]
public async Task<ActionResult<clase modelo>> deleteItem(int id) {
  var item = await _context.colección de entidades.FindAsync(id);
  if (item == null)
    return NotFound();

  _context.colección de entidades.Remove(item);
  await _context.SaveChangesAsync();

  return item;
}
```

3 Objetos de Transferencia de Datos (DTO)

A manera de vistas parciales y derivadas, permiten trasladar los datos a otras estructuras. Para ello se crean nuevos modelos y métodos estáticos que transformen de los modelos primarios a los DTOs, y viceversa.

4 Invocaciones desde Javascript

4.1 jQuery

4.2 Angular