

# CLASE 6

---

## Trabajo con Recursos

### Introducción

En esta clase veremos cómo trabajar con recursos, es decir las acciones básicas de trabajo con datos: crear, recuperar, actualizar y eliminar registros de una base de datos.

Para lograr este objetivo tenemos que hablar acerca de los **objetos de transferencia de datos** (DTO = *Data Transfer Object*).

DTO es un patrón de diseño de software que se usa para distintos tipos de aplicaciones (no es privativo de APIs) cuya finalidad es ayudar a que los datos/información puedan viajar desde la capa de servicios a las aplicaciones o capa de presentación, ya que muchas veces por desconocimiento o pereza, se usan las clases de entidades para retornar los datos, lo que ocasiona que retornemos más datos de los necesarios o incluso, se deba ir en más de una ocasión a la capa de servicios para recuperar los datos requeridos.

El patrón DTO tiene como finalidad crear un objeto plano (POJO = *Plain Old Java Object* <https://www.martinfowler.com/bliki/POJO.html>), con atributos que puedan ser enviados o recuperados del servidor en **una sola invocación**, de tal forma que un DTO puede contener información de **múltiples fuentes** o tablas y concentrarlas en una única clase simple.

**Nota:** Si desea profundizar en este patrón de diseño, sugerimos este link (aunque el ejemplo esté en Java, puede apreciarse plenamente dada la similitud con C#:  
<https://www.oscarblancarteblog.com/2018/11/30/data-transfer-object-dto-patron-diseno/>

En síntesis no es una práctica aconsejable exponer las entidades de negocio al mundo externo, lo ideal es utilizar objetos de transferencia de datos para acceder a los recursos de la aplicación.

### DTOs y Automapper

Hasta ahora siempre que retornamos recursos de la api usamos utilizamos las entidades de negocio de la aplicación, nos referimos a aquellas clases que sirven para modelar una tabla de la base de datos.

Uno de los inconvenientes de esto es que en ocasiones no queremos mostrar todos los datos contenidos en estas entidades. Es normal que solamente deseemos mostrar un subconjunto de datos de las entidades de negocio, y la forma de resolver es usando **DTOs**.

Un DTO es un objeto que sirve para transportar datos entre procesos. Los utilizaremos para representar los datos que queremos que los clientes del web API reciban. Otro nombre que reciben los n es **View Model**.

Vamos a crear el primer DTO, tenemos la entidad **Autor** con los siguientes campos que agregamos en ejercicios anteriores:

```
public class Autor : IValidatableObject
{
    public int Id { get; set; }

    [Required]
    [PrimeraLetraMayuscula]
    [StringLength(30, ErrorMessage = "El nombre del autor puede tener hasta {1} caracteres")]
    public string Nombre { get; set; }

    public DateTime FechaNacimiento { get; set; }

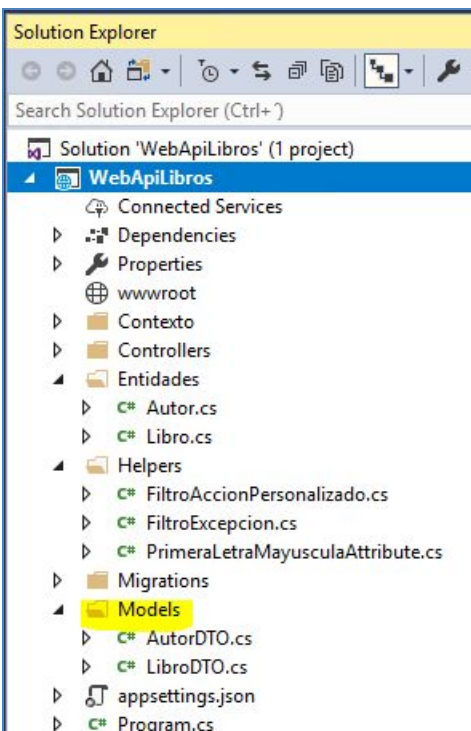
    [CreditCard]
    public string TarjetaCredito { get; set; }

    [Url]
    public string Url { get; set; }

    public List<Libro> Libros { get; set; }

    public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
    {
    }
}
```

Observe que el campo **TarjetaCredito** es considerado información sensible y en determinados procesos sobre todo en una simple consulta de libros de un autor, es un dato que no deberá accederse porque es propio de datos privados del autor y no queremos enviar a todos los clientes del API este dato. Una forma de asegurarnos de no enviar esta información es creando un DTO de autor que solamente tenga los campos que queremos que los clientes de la api vean.



Vamos a crear una clase llamada **AutorDTO** en una nueva carpeta llamada **Models**, con el siguiente código:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.ComponentModel.DataAnnotations;

namespace WebApiLibros.Models
{
    public class AutorDTO
    {
    }
}
```

```

        public int Id { get; set; }
        [Required]
        public string Nombre { get; set; }
        public DateTime FechaNacimiento { get; set; }
        public List<LibroDTO> Libros { get; set; }
    }
}

```

Sólo queremos exponer el Id, nombre, fecha de nacimiento y lista de libros del autor, no queremos exponer el número de tarjeta de crédito ni otros datos.

Creemos también la clase del **LibroDTO** con los siguientes datos:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace WebApiLibros.Models
{
    public class LibroDTO
    {
        public int Id { get; set; }
        public string Titulo { get; set; }
        public int AutorId { get; set; }
    }
}

```

Otra razón por la cual debemos utilizar **DTOs** o **View Models**, es porque ayuda con la evolución de la API. Básicamente si en algún momento queremos enviar una información extra al cliente, no podemos modificar las entidades base de negocio pues no deben verse afectadas dado que se corresponden con la estructura física de almacenamiento de los datos, modificaremos las clases DTO, del mismo modo que si algún se crea un nuevo campo del autor que va a ser para uso interno de la API, esto no va a afectar al autor porque el **AutorDTO** es el que representa la información que los clientes de la API reciben.

En síntesis distintas partes de la aplicación pueden evolucionar de forma independiente sin afectar a los clientes de dichas partes. En programación se llama “**principio de responsabilidad única**” y la responsabilidad de la clase **AutorDTO** es ser la cara de la API a los clientes que la consumen, y la responsabilidad de la clase **Autor** es servir como entidad para guardar información en la base de datos.

.

Ahora en el controlador de autores debemos cambiar el tipo de dato de retorno de **Autor** a **AutorDTO**, vemos un ejemplo con la acción Get con parámetros:

```
public ActionResult<AutorDTO> Get(int id, string param2)
{
    var resultado = context.Autores.Include(x => x.Libros).FirstOrDefault(x => x.Id == id);

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

    return resultado;
}

//[[HttpGet("prime
//public ActionResult<Autor> GetPrimerAutor() //snpnpico
```

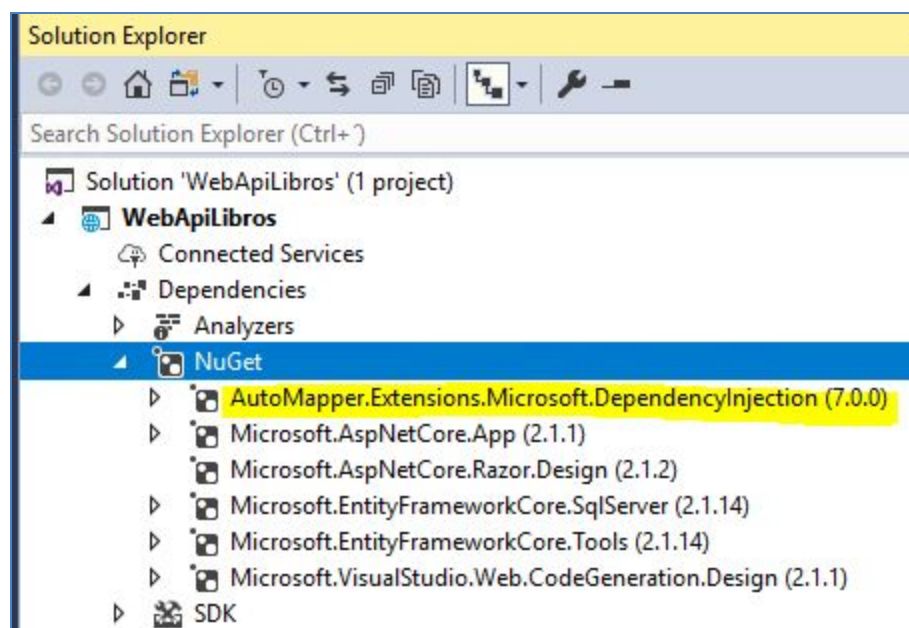
Cannot implicitly convert type 'WebApiLibros.Entidades.Autor' to 'Microsoft.AspNetCore.Mvc.ActionResult<WebApiLibros.Models.AutorDTO>'

Observe que tenemos un error el error porque actualmente estamos intentando devolver un tipo **Autor** cuando lo que queremos devolver es un **AutorDTO** y no se puede hacer esta conversión de tipos de dato en forma implícita.

Una forma de resolver esto es creando una instancia de **AutorDTO** y llenar sus campos con los campos de **Autor** necesarios. Sin embargo hacer esto por cada método y por cada controlador con sus respectivos tipos es realmente molesto, poco productivo y propenso a errores de lógica.

Una solución a esta problemática es utilizar la librería **Automapper** que se encarga de hacer este mapeo de propiedades. Vamos a instalar **Automapper** en el proyecto usando **Package Manager Console**:

```
Package Manager Console
Package source: All
Default project: WebApiLibros
PM> Install-Package AutoMapper.Extensions.Microsoft.DependencyInjection
```



Vamos a configurar **Automapper**, en la clase **startup** método **ConfigureServices** habilitamos el servicio con **AddAutoMapper** (no olvide agregar el namespace `using AutoMapper;`):

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAutoMapper(typeof(Startup));
    services.AddScoped<FiltroAccionPersonalizado>();
    services.AddResponseCaching();

    services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme).AddJwt
    Bearer();

    services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection
String")));

    services.AddMvc(options =>
    {
        Options.Filters.Add(new FiltroExcepcion()); //sin
usar inyección de dependencias
        /*Options.Filters.Add(typeof(FiltroExcepcion);*/
//usando inyección de dependencias

    }).SetCompatibilityVersion(CompatibilityVersion.Version_2_1)
        .AddJsonOptions(options =>
options.SerializerSettings.ReferenceLoopHandling =
Newtonsoft.Json.ReferenceLoopHandling.Ignore);
    }
```

A continuación podemos hacer dos cosas: utilizar perfiles para configurar Automapper o crear el mapeo directamente nosotros. Optaremos por la segunda forma:

```
services.AddAutoMapper(configuration =>
{
    configuration.CreateMap<Autor, AutorDTO>();
}, typeof(Startup));
```

Como queremos configurar un mapeo desde **Autor** a **AutorDTO**, observe que el primer parámetro del método **CreateMap** es el Source y el segundo el Destination.

Luego en la clase **AutorController** podemos hacer el mapeo. Primero necesitamos inyectar el servicio de AutoMapper (no olvide agregar el namespace `using AutoMapper;`):

```
[Route("api/[controller]")]
[ApiController]
public class AutorController : ControllerBase
{
    private readonly ApplicationDbContext context;
    private readonly IMapper mapper;

    public AutorController(ApplicationDbContext context, IMapper
mapper)
    {
        this.context = context;
        this.mapper = mapper;
    }
}
```

Y en la acción **Get** hacer el mapeo, recuerde que buscamos un objeto tipo **Autor** y luego lo mapeamos a un objeto tipo **AutorDTO**, la variable resultado recibe un objeto tipo **Autor**:

```
public ActionResult<AutorDTO> Get(int id, string param2)
{
    var resultado = context.Autores.Include(x =>
x.Libros).FirstOrDefault(x => x.Id == id);

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

    var autorDTO = mapper.Map<AutorDTO>(resultado);

    return autorDTO;
}
```

Hemos mapeado todas las propiedades de **Autor** a **AutorDTO**, y profundizando la explicación el método **Map** mapea por coincidencia en el nombre de las propiedades de ambas clases.

Autmapper también funciona con colecciones de objetos, por ejemplo podemos aplicarlo a la acción **Get** sin parámetros, que retornaba una colección objetos de tipo **Autor** y ahora podemos mapear a una colección de tipo **AutorDTO** sin problema:

```
[HttpGet("/listado")]
[HttpGet("listado")]
[HttpGet]
[ServiceFilter(typeof(FiltroAccionPersonalizado))]
```



```

public ActionResult<IEnumerable<AutorDTO>> Get()
{
    var autores = context.Autores.ToList();
    var autoresDTO = mapper.Map<List<AutorDTO>>(autores);
    return autoresDTO;
}

```

Probamos ambas acciones pero antes dado que la clase **Autor** de esta explicación difiere de las de módulos anteriores, es posible que deba ejecutar una migración de Entity Framework para que se elimine el campo **Edad** y se agregue el campo **FechaNacimiento**, sino al probar el mapeo dará un error indicando que el campo **FechaNacimiento** no existe:

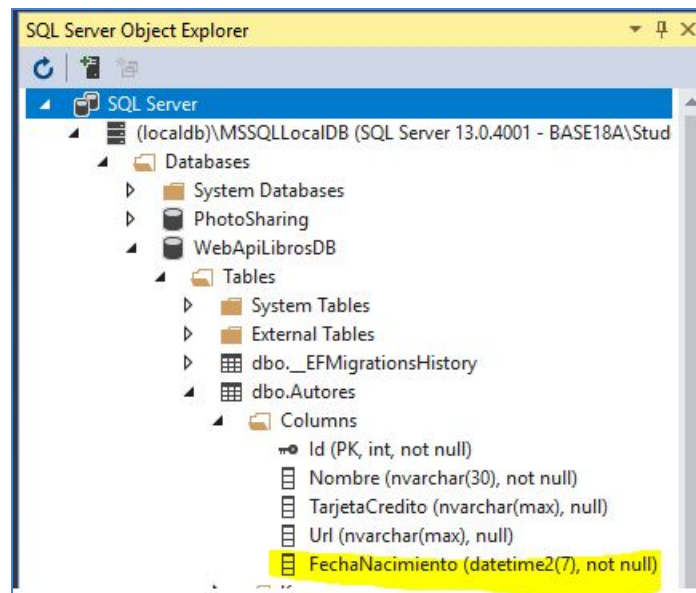
```

PM> Add-Migration dto
Microsoft.EntityFrameworkCore.Infrastructure[10403]
    Entity Framework Core 2.1.14-servicing-32113 initialized

PM> Update-Database
Microsoft.EntityFrameworkCore.Infrastructure[10403]
    Entity Framework Core 2.1.14-servicing-32113 initialized

```

Quedando actualizadas las entidades de negocio en la base de datos.



Y al ejecutar la api, veremos el resultado del mapper sin el campo **TarjetaCredito**:



## Creación de recursos con POST

Vamos a utilizar el método **HTTP Post** para crear recursos; al hacer esto lo correcto es que el web API retorne un código de estado **201 created**. Cuando hacemos esto debemos colocar en la cabecera **Location** de la respuesta HTTP la URL del nuevo recurso creado y además en el cuerpo de la respuesta debemos colocar el nuevo recurso creado.

Esto se logra en ASP.Net ejecutando la función **CreatedAtRouteResult** que hace todo lo antes mencionado.

Un tema a considerar es que a las reglas de ruteo les podemos poner nombres. La idea de ponerle un nombre a una regla de ruteo es poder hacer referencia a dicha regla de ruteo o a dicha ruta o endpoint desde cualquier parte del web API.

Para ponerle un nombre a una regla de ruteo utilizamos el parámetro **Name** de la siguiente manera:

```
[HttpGet("{id}/{param2=hola}", Name = "ObtenerAutor")] //segundo parámetro opcional con v
//public ActionResult<Autor> Get(i
public ActionResult<AutorDTO> Get(int id, string param2)
{
    var resultado = context.Autores.Include(x => x.Libros).FirstOrDefault(x => x.Id == id);

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

    var autorDTO = mapper.Map<AutorDTO>(resultado);

    return autorDTO;
}
```

En el ejemplo anterior Name igual al nombre de la regla de ruteo entonces el nombre de esta regla de ruteo es **ObtenerAutor**. Vamos a utilizar esto para realizar el ejemplo de creación de un recurso.

Vamos al método POST y utilizamos **async** para aprovechar las ventajas de la programación asincrónica en nuestro web API.

```
[HttpPost]
public async Task<ActionResult> Post([FromBody] Autor autor)
{
    context.Autores.Add(autor);
    await context.SaveChangesAsync();
    return new CreatedAtRouteResult("ObtenerAutor", new { id =
autor.Id }, autor);
}
```



```
}
```

Como hemos mencionado al realizar operaciones con la base de datos es buena práctica utilizar programación asíncrona y así poder manejar recursos del servidor web en forma más efectiva.

En el parámetro de la función **Post** indicamos el atributo **FromBody** indicando que la información del autor va a venir en el cuerpo de la petición HTTP. Luego en el cuerpo de esta función utilizamos EF para indicar que queremos agregar el autor a la base de datos y luego guardamos los cambios de manera asíncrona en la base de datos.

Al final utilizamos **CreatedAtRouteResult** para retornar un 201 created:

```
return new CreatedAtRouteResult("ObtenerAutor", new { id = autor.Id },  
autor);
```

Observe que al instanciar el objeto **CreatedAtRouteResult** se pasa el nombre de la ruta **ObtenerAutor** donde se encuentra el recurso para ser consultado de manera individual, y además se le pasa el **id** del autor dado que la acción **Get** espera recibir como parámetro el **id**, y finalmente también pasamos el **objeto Autor** que hemos creado.

Hay una diferencia entre lo retornamos desde **Post** y lo que espera **Get**; **Post** retorna un objeto tipo **Autor** y **Get** retorna un objeto tipo **AutorDTO**, lo correcto sería que ambos retornen el mismo tipo de objeto para ser consistentes con los tipos de datos de retorno de nuestros métodos.

Así que modificaremos la acción **Post** para retornar un **AutorDTO** usando el mapper:

```
[HttpPost]  
public async Task<ActionResult> Post([FromBody] Autor autor)  
{  
    context.Autores.Add(autor);  
    await context.SaveChangesAsync();  
  
    var autorDTO = mapper.Map<AutorDTO>(autor);  
    return new CreatedAtRouteResult("ObtenerAutor", new { id =  
autor.Id }, autorDTO);  
}
```

Continuando con los DTOs vemos que la clase **Autor** tiene un campo **Id** dado que autor es una entidad y representa una tabla en la base de datos. Sin embargo recordamos que mencionamos el “*principio de responsabilidad única*”, **Autor** sirve para guardar información o para representar una tabla en la base de datos y no debería servir como valor de entrada de la función **Post**, tampoco tiene sentido que a través de **Post** se indique el valor del campo **Id**.

Esto representaría realmente un error del cliente de la API y entonces si el cliente nos manda un **Id**, ¿qué deberíamos responderle al cliente?... porque este método es para

crear un nuevo autor y el **Id** se crea internamente en la aplicación y no lo suministra el cliente de la api ni tampoco lo tienen que ver.

Entonces en la acción **Post** usaremos un objeto DTO pero no **AutorDTO**, crearemos otro sin el campo **Id**. En el caso de **AutorDTO** si es necesario que haya un **Id** porque cuando se le devuelva este objeto al cliente del Web API, queremos que conozca el **Id**, por si por ejemplo quiere eliminar un autor en base a ese **Id** siempre y cuando la lógica de negocio de la api quiera eso, pero en el caso de la creación el **Id** no es necesario.

Crearemos una nueva clase en Models llamada **AutorCreacionDTO** con dos propiedades:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.ComponentModel.DataAnnotations;

namespace WebApiLibros.Models
{
    public class AutorCreacionDTO
    {
        [Required]
        public string Nombre { get; set; }
        public DateTime FechaNacimiento { get; set; }
    }
}
```

Y luego en **AutorController** cambiamos el **Autor** por **AutorCreacionDTO**, es decir recibimos un objeto de tipo **AutorCreacionDTO** y lo mapeamos en un objeto **Autor** para luego vía EF guardarlo en la base de datos, EF espera un tipo **Autor**:

```
[HttpPost]
public async Task<ActionResult> Post([FromBody] AutorCreacionDTO
autorCreacion)
{
    //Esto no es necesario desde Asp.Net Core 2.1
    //if (!ModelState.IsValid)
    //{ return BadRequest(ModelState); }

    var autor = mapper.Map<Autor>(autorCreacion);
    context.Add(autor);
    await context.SaveChangesAsync();
    var autorDTO = mapper.Map<AutorDTO>(autor);
    return new CreatedAtRouteResult("ObtenerAutor", new { id =
autor.Id }, autorDTO);
}
```

}

Con **AutoMapper** podemos ejercitar buenas prácticas de construcción de web APIs en forma simple, sencilla y compacta sin necesidad de hacer un mapeo a mano, que insinúa mucho tiempo y propenso a errores de lógica de programación.

Finalmente vamos a la clase **startup** para agregar una configuración a **AutoMapper**, agregaremos la configuración del mapeo de **AutorCreacionDTO** hacia **Autor**:

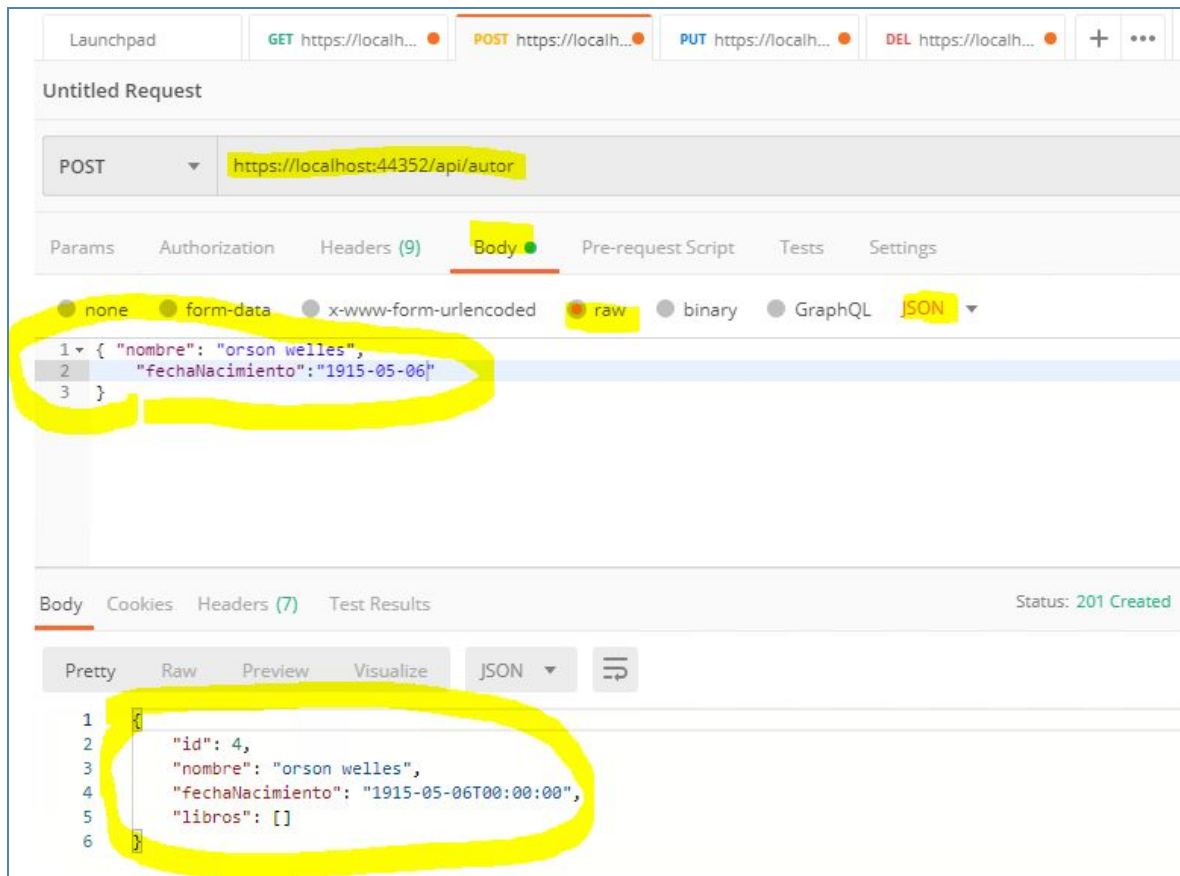
```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAutoMapper(configuration =>
    {
        configuration.CreateMap<Autor, AutorDTO>();
        configuration.CreateMap<AutorCreacionDTO, Autor>();
    }, typeof(Startup));
}
```

Probamos la aplicación con Postman, hacemos primero un **Get** y vemos que retorna los autores desde el objeto **AutorDTO** que tiene id, nombre, fechaNacimiento y lista de libros en este momento vacía:

The screenshot shows the Postman interface with a GET request to `https://localhost:44352/api/autor`. The response status is 200 OK. The response body is a JSON array of two author objects, displayed in a pretty-printed format.

```
[
  {
    "id": 2,
    "nombre": "Arthur Clarke",
    "fechaNacimiento": "0001-01-01T00:00:00",
    "libros": []
  },
  {
    "id": 3,
    "nombre": "Jorge Luis Borges",
    "fechaNacimiento": "0001-01-01T00:00:00",
    "libros": []
  }
]
```

Y probamos **Post** cargando un objeto **AutorCreacionDTO** con nombre y fechaNacimiento, obteniendo un status 201 created y el objeto completo del tipo **AutorDTO** en el body con el **Id** interno que se crea en el registro de la base de datos.



## Actualización completa de recursos con PUT

Para editar un recurso se utiliza un HTTP Put sobre un endpoint. La actualización que trataremos ahora es una **actualización completa** de recursos y luego veremos cómo hacer una **actualización parcial**.

Cuando hacemos una actualización con HTTP Put debe ser completa por lo que esperamos que el cliente envíe todos los datos del recurso. Como respuesta podemos retornar o un **200 ok** o un **204 no content**. Dado que el cliente ya tiene el recurso enviar un 204 no content tiene sentido. Vamos a verlo.

En el controlador de autores vemos la acción **Put**:

```
[HttpPut("{id}")]
public async Task<ActionResult> Put(int id, [FromBody] Autor
value)
{
    if (id != value.Id)
```

```

    {
        BadRequest();
    }

    context.Entry(value).State = EntityState.Modified;
    await context.SaveChangesAsync();
    //return Ok();
    return NoContent();
}

```

Está decorada con el **atributo HTTP** y le indicando que vamos a tener un id en la ruta de la URL. Y los parámetros de la función son **id y el autor que queremos actualizar**. Al igual que en la acción **Post**, el autor viene del body o del cuerpo de la petición HTTP.

Dentro de la función lo primero que hacemos es verificar que el id de la ruta y el id que viene en el objeto **Autor** coincidan, caso contrario retornamos un 400 **BadRequest**. La razón por la que hacemos esto es para evitar que se actualice un recurso incorrecto. Hay otra forma de manejarlo pero por ahora vamos a seguir con esta implementación.

Luego lo que hacemos es utilizar el método **Entry** del contexto de datos para marcar esta entidad como modificada, y luego hacemos un **SaveChanges** para actualizar guardar la actualización en forma asincrónica y finalmente retornar un 200 ok o como no se retorna nada retornar un 204 no content.

La razón por la que estamos usando la verificación del id es porque estamos usando como valor de entrada de la función una entidad, es decir la clase **Autor**. Si quisiéramos evitar esto podemos utilizar un DTO como valor de entrada de la función, podríamos reutilizar el DTO de la creación de autor o crear uno nuevo específicamente para actualización de autores. Como en este caso la actualización y la creación tiene las mismas reglas de negocio podemos usar el DTO de creación de autor. Tenga en cuenta que esto depende de la lógica de negocio de la aplicación.

Entonces modificamos la acción **Put** de la siguiente forma:

```

[HttpPut("{id}")]
public async Task<ActionResult> Put(int id, [FromBody]
AutorCreacionDTO autorActualizacion)
{
    var autor = mapper.Map<Autor>(autorActualizacion);
    autor.Id = id;
    context.Entry(autor).State = EntityState.Modified;
    await context.SaveChangesAsync();
    //return Ok();
    return NoContent();
}

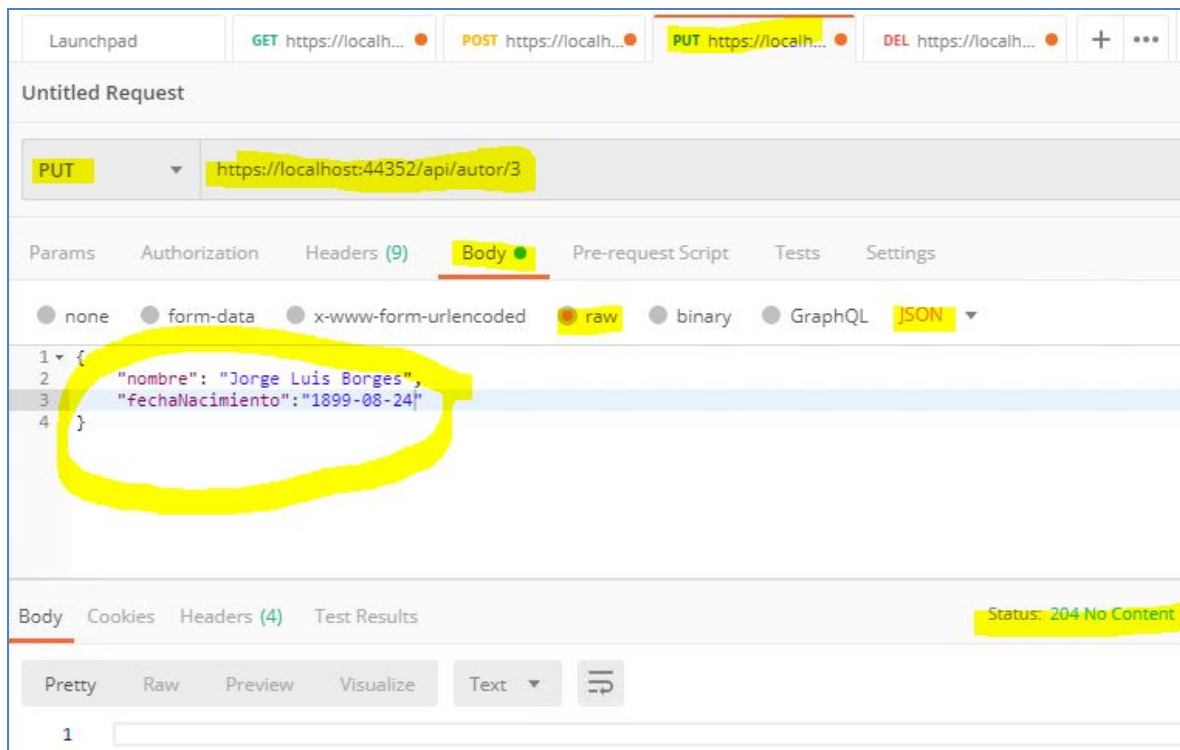
```

Si luego la discrepancia entre los nombres confunde, puede crear un **AutorActualizacionDTO** o cambiarle el nombre a **AutorCreacioncionDTO** por uno más genérico. El código de verificación del id no lo necesitamos dado que

**autorActualizacion** al ser tipo **AutorCreacionDTO** no tiene id, y se simplifica el código. Con Automapper mapeamos el objeto **autorActualizacion** a tipo **Autor** y le agregamos el id dado que no lo tiene y es necesario para que EF pueda hacer la actualización en la base de datos.

Reiteramos que HTTP Put es para actualizaciones completas de un recurso pero no siempre es lo que queremos. Una desventaja de las actualizaciones completas es que necesitamos que el usuario envíe todos los campos del recurso para poder hacer la actualización.

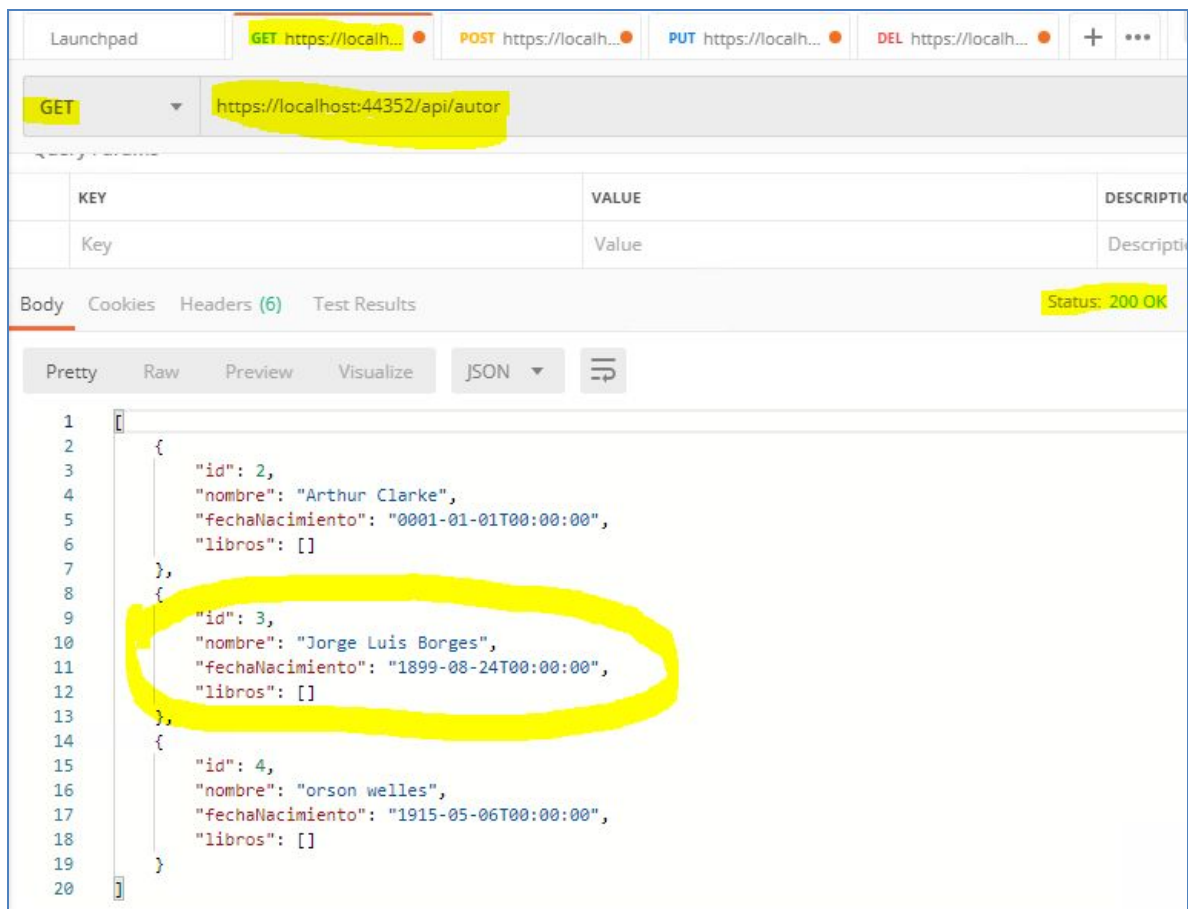
Probamos la acción con Postman:



Observe que en la URL indicamos actualizar el autor con id 3, modificamos la fecha de nacimiento y como resultado obtenemos un 204 no content.

Si luego para verificar consultamos con Get, vemos que la actualización ha sido exitosa.





## Actualización parcial de recursos con PATCH

Utilizamos el método HTTP Patch para aplicar actualizaciones parciales a un recurso, es decir si tenemos un recurso y queremos actualizar solo algunos campos.

El cliente del Web API nos enviará un cuerpo en la petición HTTP que va a indicar los cambios que desea hacer sobre el recurso. En nuestro caso nos interesa saber qué campos desea actualizar el cliente y cuáles van a ser los nuevos valores para esos campos.

¿Pero cómo va el cliente a indicarnos los nuevos valores para los campos? En la norma RFC 6902 existe un standard definido llamado **JSON Patch** que lo podemos utilizar para el HTTP Patch.

Básicamente **JSON Patch** indica cómo debe ser la estructura del cuerpo de la petición HTTP que va a indicar los cambios que el cliente quiere hacer sobre el recurso.

Este estándar define un conjunto de operaciones las cuales son agregar, remover, reemplazar, mover, copiar y probar.

En este momento nos interesa reemplazar para actualizar campos de un recurso.

Un ejemplo del cuerpo de un conjunto de reemplazos con **JSON Patch** es el siguiente:

```
[
  { "op": "replace", "path": "/nombre", "value": "Felipe" },
  { "op": "replace", "path": "/fechaNacimiento", "value": "1905-01-02" }
]
```

**op** es la operación en este caso **replace** (reemplazar), **path** define el campo que queremos actualizar y **value** define el nuevo valor que va a tomar el campo.

En el ejemplo anterior queremos realizar dos reemplazos, uno para el campo **nombre** y otro para el campo **fechaNacimiento**.

Vamos a implementar **JSON Patch** en el proyecto en el controlador de autores, primero sin DTO y luego con:

**Nota:** para implementar esto si tiene ASP.Net Core 3.0 o superior, deberá instalar el paquete **Microsoft.AspNetCore.JsonPatch**

<https://docs.microsoft.com/en-us/aspnet/core/web-api/jsonpatch?view=aspnetcore-3.1>

```
//Patch sin DTO
[HttpPatch("{id}")]
public async Task<ActionResult> Patch(int id, [FromBody]
JsonPatchDocument<Autor> patchDocument)
{
    if (patchDocument == null)
    {
        return BadRequest();
    }

    var autorDeLaDB = await context.Autores.FirstOrDefault(x
=> x.Id == id);
    if (autorDeLaDB == null)
    {
        return NotFound();
    }

    patchDocument.ApplyTo(autorDeLaDB, ModelState);

    var isValid = TryValidateModel(autorDeLaDB);
    if (!isValid)
    {
        return BadRequest(ModelState);
    }
}
```

```

    }

    await context.SaveChangesAsync();
    return NoContent();
}

```

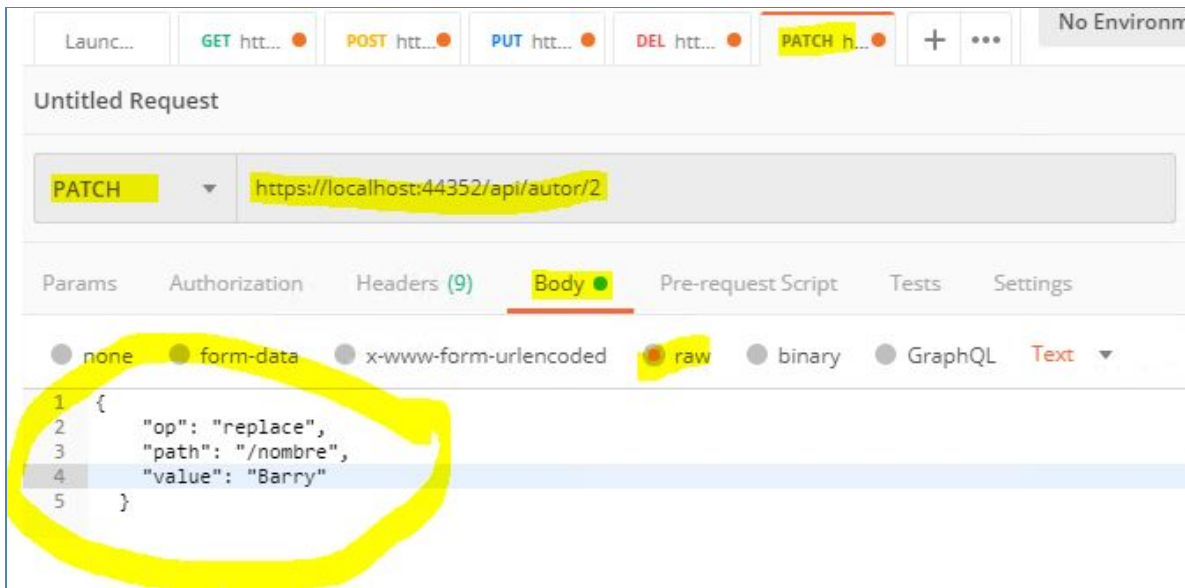
Observe que en el **atributo Http Patch** se espera un id que es el del autor a modificar.

En la acción recibimos **el id y un documento patch en el cuerpo de la petición HTTP**, recuerde que en el cuerpo de la petición HTTP vamos a recibir un conjunto de operaciones dado que en el documento patch se indica en formato JSON las operaciones a realizar sobre los campos específicos, necesitamos una representación de esas operaciones que recibiremos y esa representación en el objeto **JsonPatchDocument<Autor>** de tipo **patchDocument**.

Dentro de la acción procesamos y validamos lo siguiente:

- Si **el documento es nulo** entonces retornamos un BadRequest.
- Buscamos el autor de la base de datos para asegurarnos que el autor a modificar existe, **si no existe retornamos un NotFound**.
- Aplicamos las operaciones que hemos recibido a través del documento **JSONPatch** al objeto **autorDeLaDb** con el **método ApplyTo** y le pasamos el autor de la base de datos y el **ModelState** que aplica los atributos de validación de la clase entidad para controlar que se cumplan las reglas de negocio. Y validamos con el método **TryValidateModel**, si no es válido entonces retornamos un BadRequest con los errores de las reglas que no se cumplen.
- Si todo es correcto guardamos las modificaciones vía EF y retornamos un no content.

Probamos esta modificación parcial con Postman:



Y aquí la actualización parcial con Automapper usando la clase **AutorCreacionDTO**:

```
//Patch con DTO
[HttpPatch("{id}")]
public async Task<ActionResult> Patch(int id, [FromBody]
JsonPatchDocument<AutorCreacionDTO> patchDocument)
{
    if (patchDocument == null)
    {
        return BadRequest();
    }

    var autorDeLaDB = await context.Autores.FirstOrDefaultAsync(x
=> x.Id == id);

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

    var autorDTO = mapper.Map<AutorCreacionDTO>(autorDeLaDB);

    patchDocument.ApplyTo(autorDTO, ModelState);

    var isValid = TryValidateModel(autorDeLaDB);
    if (!isValid)
    {
        return BadRequest(ModelState);
    }
}
```

```

        mapper.Map(autorDTO, autorDeLaDB);
        await context.SaveChangesAsync();
        return NoContent();
    }

```

Y en la clase **startup** en **ConfigureServices** modificar la línea del mapeo.

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddAutoMapper(configuration =>
    {
        configuration.CreateMap<Autor, AutorDTO>();
        configuration.CreateMap<AutorCreacionDTO,
        Autor>().ReverseMap();
    }, typeof(Startup));
}

```

## Eliminación de recursos con DELETE

Para eliminar un recurso se utiliza el método HTTP Delete sobre un endpoint. La función que se va a ejecutar solamente necesita recibir el id del recurso a eliminar. No es necesario recibir la entidad ni un DTO.

Veamos un ejemplo tenemos la acción Delete del controlador de autores:

```

[HttpDelete("{id}")]
public async Task<ActionResult<Autor>> Delete(int id)
{
    var resultado = await context.Autores.Select(x =>
    x.Id).FirstOrDefaultAsync(x => x == id);

    if (resultado == default(int))
    {
        return NotFound();
    }

    context.Autores.Remove(new Autor { Id = resultado });
    await context.SaveChangesAsync();
    return NoContent();
}

```

La función está decorada con el atributo **HTTP Delete** y estamos recibiendo el id del recurso que queremos eliminar.

Lo primero que hacemos es utilizar EF con programación asíncrona **para buscar** la entidad que queremos borrar. Utilizamos la función **Select** de que permite seleccionar sólo las columnas de la tabla que quiero leer, en este caso sólo el **Id** dado que no es necesario leer todos los datos del autor porque lo eliminaremos y sólo leyendo el **Id** de la base de datos nos alcanza, la performance será mejor. El

query que internamente genera EF será mucho más rápido porque estamos solamente solicitando un campo de tamaño pequeño.

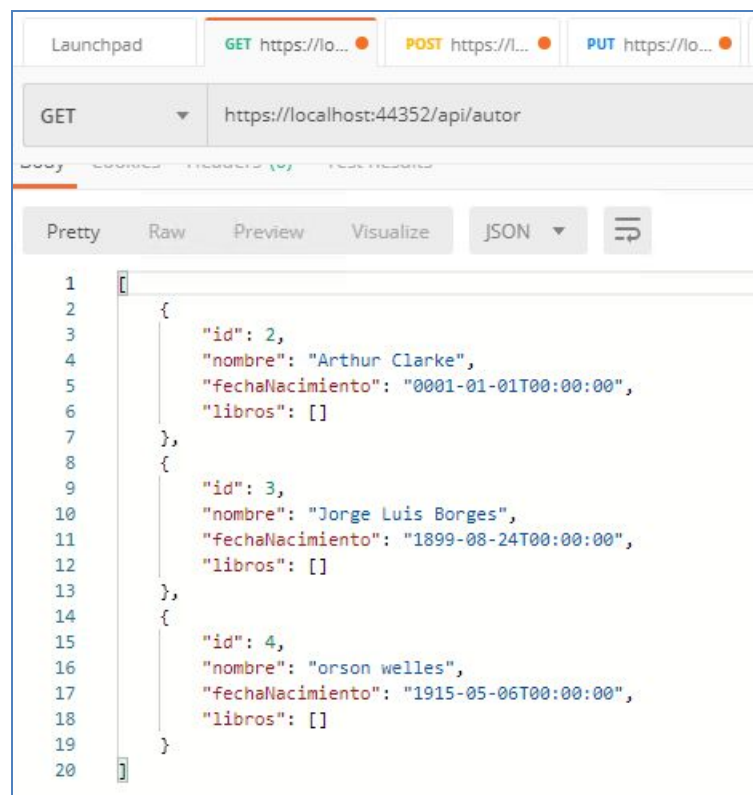
Si esa entidad no existe retornamos un 404 Not found dado que la variable resultado tendrá el valor default del tipo int que es cero, no siendo este un id válido para un autor.

Luego usamos EF para decidir **eliminar el autor** creando una instancia de autor sólo con el valor del id a eliminar. EF es lo bastante inteligente para saber que si queremos eliminar un autor con esta identificación, él va a ir a la base de datos y va a eliminar el registro de la tabla autores que tenga ese Id, no es necesario que le pasemos la entidad completa.

Finalmente guardamos los cambios, y finalmente podemos retornar si queremos **un 200 No content**, o podemos también retornar el autor o la entidad que hemos eliminado.

Si optamos por retornar la entidad deberíamos usar un DTO dado que seguro los datos visibles para el cliente que elimina el recurso, no son todos los modelados en la clase de entidad que interactúa con EF.

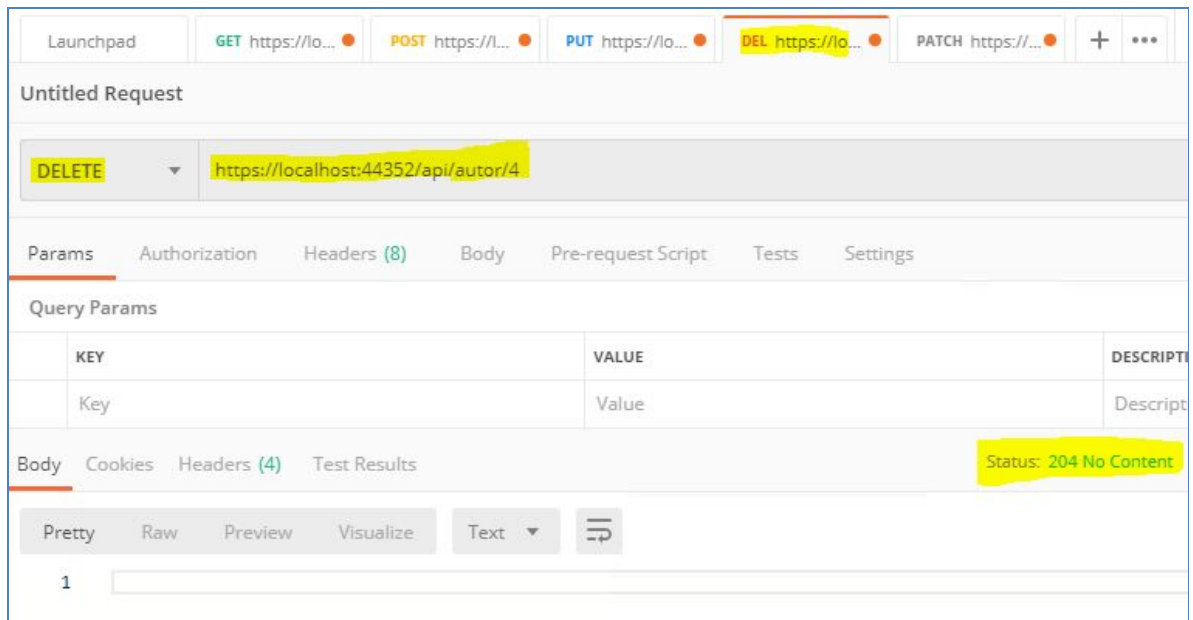
Vamos a probar esta acción con Postman. Actualmente y antes de eliminar tenemos tres autores:



```
1 [
2   {
3     "id": 2,
4     "nombre": "Arthur Clarke",
5     "fechaNacimiento": "0001-01-01T00:00:00",
6     "libros": []
7   },
8   {
9     "id": 3,
10    "nombre": "Jorge Luis Borges",
11    "fechaNacimiento": "1899-08-24T00:00:00",
12    "libros": []
13  },
14  {
15    "id": 4,
16    "nombre": "Orson Welles",
17    "fechaNacimiento": "1915-05-06T00:00:00",
18    "libros": []
19  }
20 ]
```

Ejecutamos la eliminación:





Hemos eliminado el autor con id 4 de la base de datos, lo verificamos nuevamente.

