UD7 Servicios web.

7.2 Servicio de lista de tareas con ASP.NET Core.

En esta parte vamos a crear un servicio de lista de tareas sencillo y veremos cómo consumir dicho servicio.

Vamos a implementar la siguiente API:

API	Descripción	Petición	Respuesta
GET /api/todo	Obtiene la lista de tareas	-	Lista de tareas
GET /api/todo/{id}	Obtiene una tarea por su ID	-	Tarea
POST /api/todo	Añade una nueva tarea	Tarea	Tarea
PUT /api/todo/{id}	Actualiza una tarea existente	Tarea	-
DELETE /api/todo/{id}	Elimina una tarea	-	-

Creación del proyecto y modelo de datos.

1. Creamos un nuevo proyecto de tipo aplicación web ASP.NET Core y seleccionamos la plantilla API.

2. A continuación pasamos al modelo, que en este caso será muy sencillo, constando sólo de una única clase *TodoItem*, que representa una tarea pendiente. Creamos la clase en una carpeta *Models*:

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

3. Una vez que el modelo está listo añadimos el contexto de datos. Por simplicidad lo añadimos a la carpeta *Models*, y llamaremos a nuestra clase *TodoContext*:

```
using Microsoft.EntityFrameworkCore;

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

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

4. Registramos el contexto de datos en *StartUp.cs*:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<TodoContext>(opt =>
        opt.UseInMemoryDatabase("TodoList"));

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

En este ejemplo vamos a utilizar un contexto de datos en memoria para simplificar el ejercicio. Un contexto de datos en memoria almacena los datos en la memoria, de modo que al cerrar la aplicación estos desaparecen, por lo que en la práctica no tiene más utilidad que para probar la aplicación. Para pasar a una aplicación real deberíamos sustituir este contexto de datos por uno que almacene la información en un almacén de datos real, tal y como hemos hecho en los temas anteriores.

5. Ahora vamos a eliminar el controlador de nuestro proyecto y crear uno nuevo, que será el que implemente nuestra API. El nombre del controlador será TodoController. Para ello hacemos click derecho en nuestra carpeta Controllers, y seleccionando la opción **Agregar | Nuevo elemento...** y luego **seleccionamos Clase de controlador API.**

6. Modificamos el código del controlador y lo cambiamos por el siguiente:

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using TodoApi.Models;

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

        public TodoController(TodoContext context)
        {
            _context = context;

            if (_context.TodoItems.Count() == 0)
            {
                // Create a new TodoItem if collection is empty,
                _context.TodoItems.Add(new TodoItem { Name = "Item1" });
                _context.SaveChanges();
            }
        }
    }

}
```

Lo que hemos hecho hasta ahora es definir un controlador para nuestra API, que de momento no expone ningún método. Únicamente tenemos un constructor que comprueba si nuestro contexto de datos no contiene ninguna tarea y si es así crea una. Esto lo haremos por simplicidad a la hora de probar nuestra API al asegurarnos de que siempre contenga datos que devolver.

7. Vamos a comenzar por implementar los métodos que permiten obtener las tareas pendientes, y siguiendo las normas de REST, estos deben responder a peticiones de tipo GET. Implementamos un método que devuelve todas las tareas y otro que devuelve una tarea concreta, a partir de su ID.

```
// GET: api/Todo
[HttpGet]
public ActionResult<IEnumerable<TodoItem>> GetTodoItems()
{
    return _context.TodoItems.ToList();
}

// GET: api/Todo/5
[HttpGet("{id}")]
public ActionResult<TodoItem> GetTodoItem(long id)
{
    var todoItem = _context.TodoItems.Find(id);

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

    return todoItem;
}
```

El primer método es muy simple y simplemente devuelve la lista completa de tareas. El segundo busca una tarea por su ID y devuelve este o un resultado de no encontrado.

Los métodos devuelven un objeto de tipo ActionResult que encapsula la información real devuelta por el método. Esto es así porque ASP.NET Core se encargará de formatear los datos de respuesta al cliente conforme al formato acordado, por defecto json.

Estos métodos se asociarán a recursos localizados en las siguientes URLs:

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

8. Vamos a probar nuestra API directamente desde nuestro navegador.

Postman.

Aunque hemos visto que podemos probar los primeros métodos de nuestra API con un navegador, esto no será posible para métodos que respondan a diferentes verbos HTTP.

Existen numerosas aplicaciones que permiten realizar peticiones HTTP, y lo que haremos es utilizar una de ellas para probar nuestros servicios. En concreto utilizamos una aplicación llamada Postman, que podremos obtener en https://www.getpostman.com/downloads.

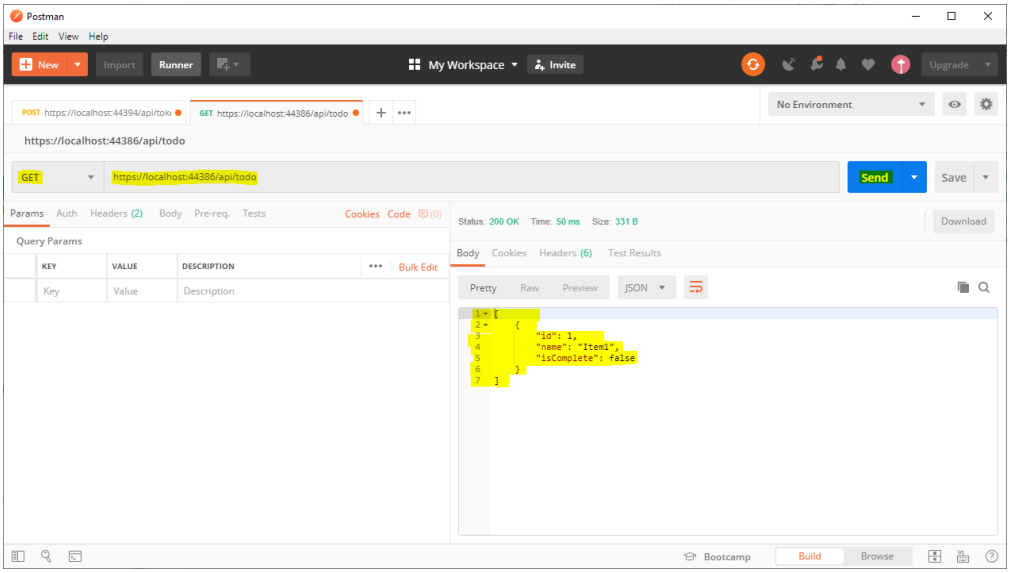
Además de permitirnos hacer peticiones de cualquier tipo veremos que nos ofrece muchas más ventajas, como un mayor detalle de los datos que se pasan al servicio web y son devueltos por este.

1. Comenzamos por instalar la aplicación**.**

2. Arrancamos la aplicación y para probar nuestros servicios deshabilitaremos la opción **SSL Certificate Verification** que se encuentra en **File -> Settings**.

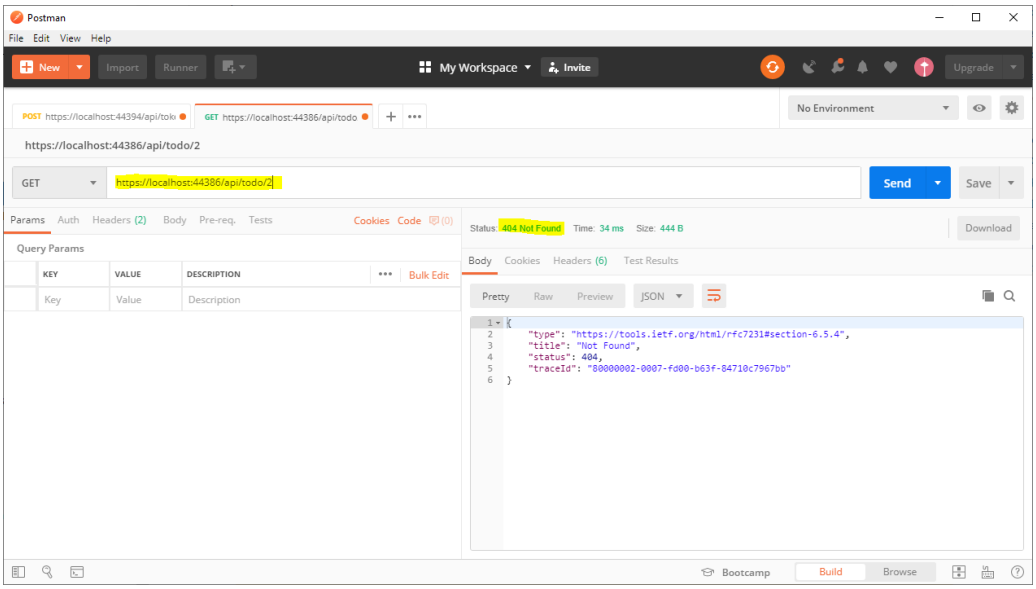
3. Arrancamos nuestro servicio desde Visual Studio.

4. Creamos una nueva petición en Postman, seleccionando el método GET y como URL utilizamos la de nuestro servicio web. A continuación pulsamos el botón **Enviar**.



Postman realiza la petición y muestra el resultado obtenido del servicio web. Como podemos observar el servidor envía un mensaje cuyo código de respuesta es 200, que indica que todo ha sido correcto. Los datos devueltos están en formato json, aunque en nuestro método no hemos tenido que realizar ninguna conversión, ya que ASP.NET Core ha realizado este trabajo por nosotros. Al seguir estas convenciones estándar y utilizar un formato como json, cualquier cliente puede utilizar nuestro servicio.

5. Vamos a hacer una nueva petición, en este caso intentaremos obtener el elemento de la lista de tareas con el ID de valor 2. De nuevo introducimos la URL y pulsamos enviar.



La petición en este caso incluye el id del recurso a obtener, de modo que será el método encargado de buscar una tarea concreta el que se utilice. Al no encontrar ningún elemento con dicho ID, nuestro método devuelve el valor obtenido de llamar al método *NotFound()*.

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

De nuevo, es ASP.NET Core el encargado de devolver un mensaje con un código de respuesta 404, que representa un error por un recurso no encontrado.

Crear una nueva tarea.

Vamos ahora a crear un nuevo método que permita añadir una nueva tarea a nuestra API.

1. Añadimos el siguiente método a nuestro controlador:

```
// POST: api/Todo
[HttpPost]
public ActionResult<TodoItem> PostTodoItem(TodoItem item)
{
    _context.TodoItems.Add(item);
    _context.SaveChanges();

    return CreatedAtAction(nameof(GetTodoItem), new { id = item.Id }, item);
}
```

   Mediante el atributo *HttpPost* indicamos que este método responderá a las peticiones POST a nuestro servicio. El método es sencillo, simplemente añadimos la nueva tarea recibida a nuestro almacén de datos y guardamos los cambios.

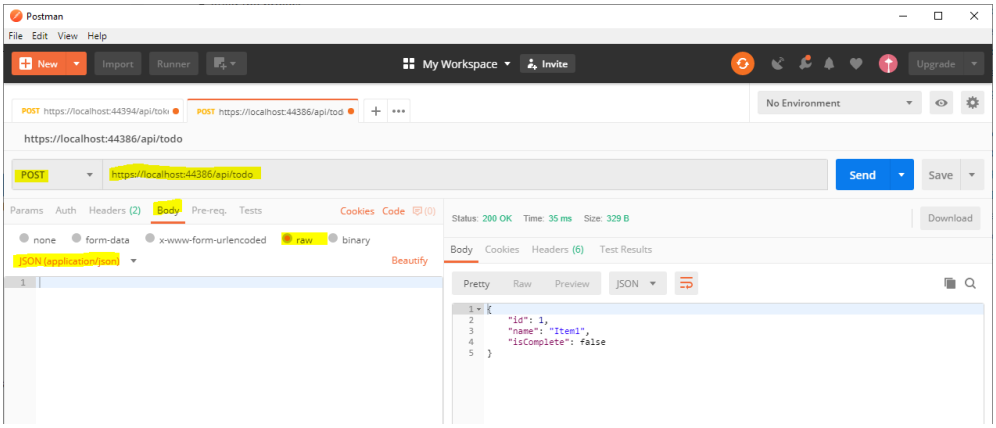
   Los dos aspectos más importantes de este método son los siguientes:

- El método debe recibir los datos de una tarea que debemos guardar en nuestro almacén de datos. Esto supone un objeto completo, que será pasado como argumento en la llamada al servicio web, que irá en dentro del cuerpo de la petición HTTP.

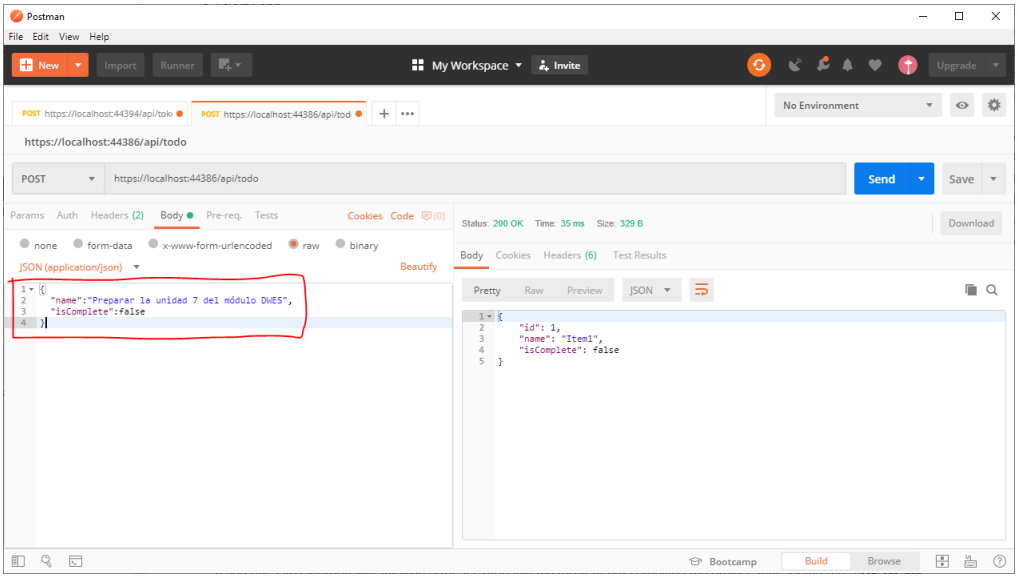
- El método genera su respuesta mediante el método CreatedAtAction(). Este método devuelve el código de respuesta 201 en caso de que el elemento haya sido creado con éxito. En segundo lugar añade una cabecera *Location* en la respuesta, que contendrá la URI del objeto recién creado.

2. Para probar nuestro nuevo método arrancamos nuestro servicio desde Visual Studio.

3. Creamos una nueva petición en Postman, seleccionando el método POST y como URL utilizamos la de nuestro servicio web, de la forma *https://localhost:xxxxxx/api/todo*.
   Como explicamos anteriormente, la tarea que queremos añadir debe ser pasada dentro del cuerpo de la llamada al servicio, para ello seleccionamos la pestaña *Body* y marcamos el botón *Raw* y en el tipo de datos seleccionamos *JSON (application/json)*, que es el formato por defecto.

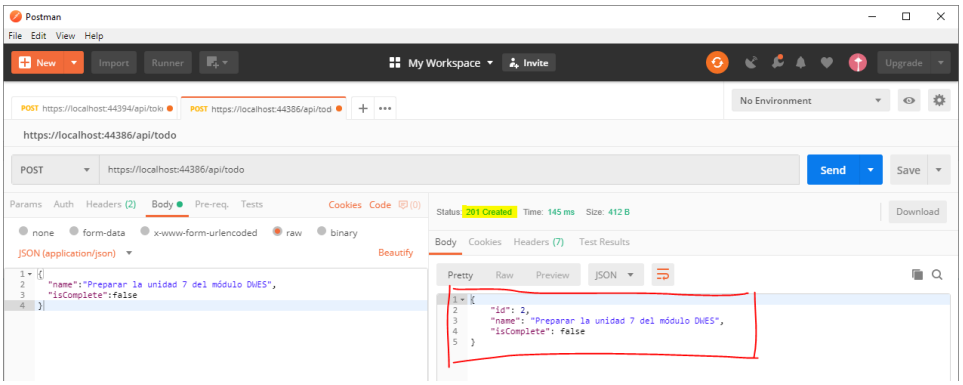


4. Ahora lo que debemos hacer es introducir los datos del objeto que queremos crear, utilizando notación json.



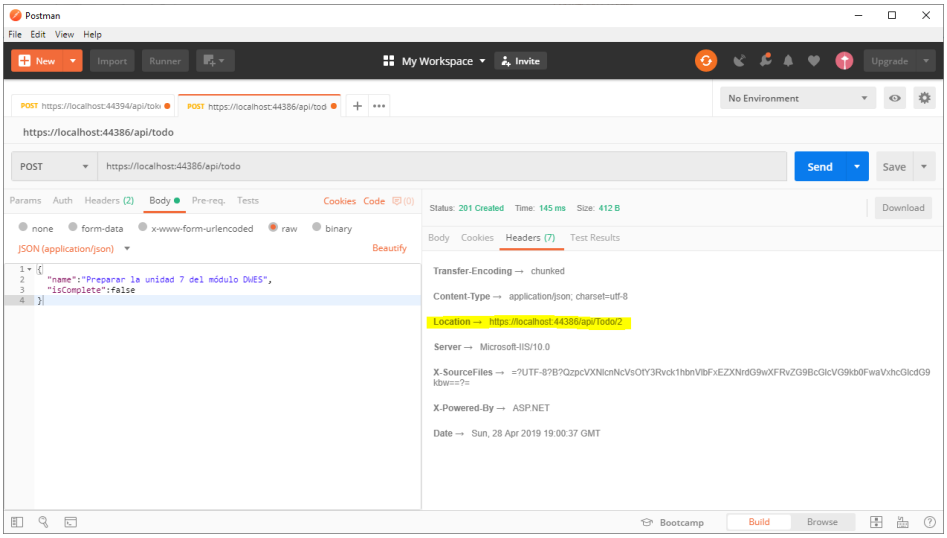
CIFP A Carballeira-Marcos Valcárcel
Rúa A Carballeira s/n
Ourense CP 32002
Tfno. 988788470, Fax 988227935

5. Una vez tenemos preparada nuestra petición pulsamos el botón **Enviar**.

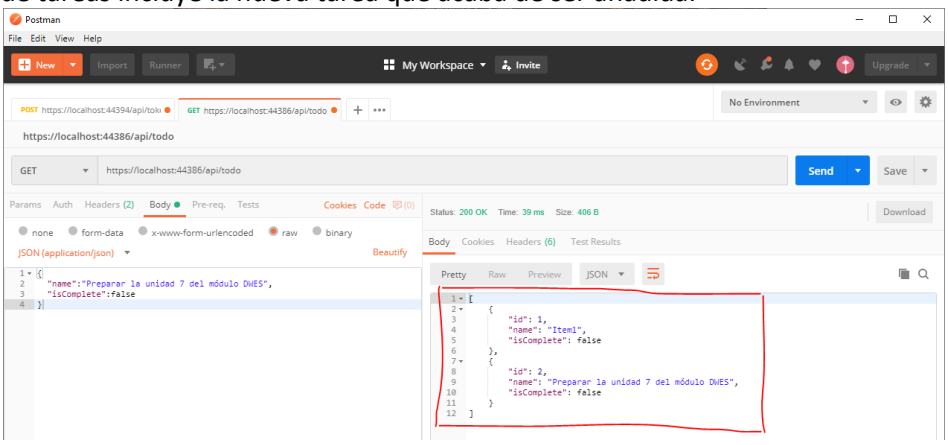


Podemos ver que en la respuesta hemos obtenido un código 201, indicando que la operación se ha completado con éxito. Además, en el cuerpo de la respuesta tenemos el objeto completo que hemos creado, conteniendo ya el ID que se le ha asignado.

Por último, si abrimos la pestaña *Headers*, podemos ver como se ha incluido la información *Location*, con la URL del nuevo elemento que acabamos de crear.



6. Si cambiamos de nuevo nuestra petición para enviar una solicitud GET veremos como en este caso la lista de tareas incluye la nueva tarea que acaba de ser añadida.



Modificar una tarea.

En este caso añadiremos a nuestra API un método que permita modificar una tarea existente.

1. Añadimos el siguiente método a nuestro controlador:

```
// PUT: api/Todo/5
[HttpPut("{id}")]
public  IActionResult PutTodoItem(long id, TodoItem item)
{
    if (id != item.Id)
    {
         return BadRequest();
    }

    _context.Entry(item).State = EntityState.Modified;
    _context.SaveChanges();

    return NoContent();
}
```

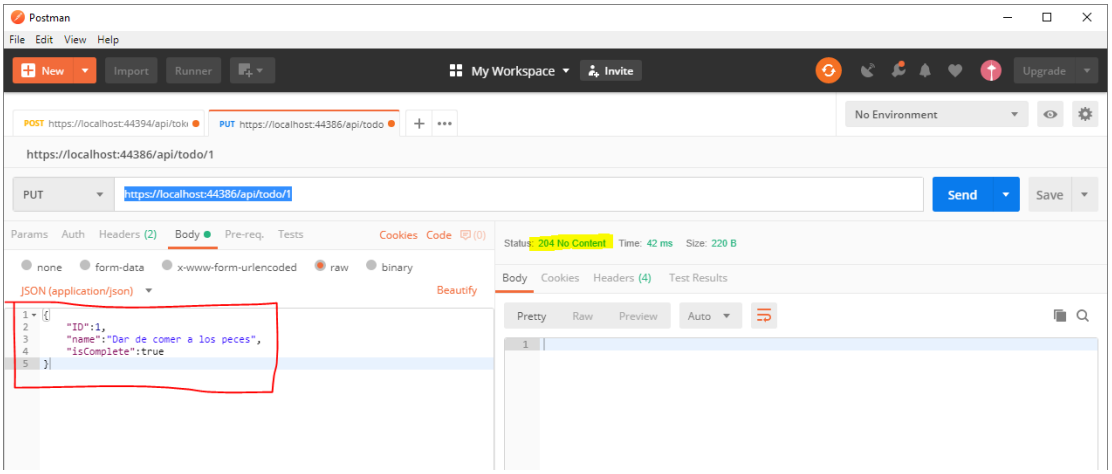
El método PUT es similar al método POST. De acuerdo con la especificación HTTP, requiere que el cliente envíe en objeto completo a modificar, de modo que no sería suficiente con enviar solamente los cambios.

El método comprueba que existe el objeto a modificar, y en caso contrario devuelve un código de error. La especificación HTTP indica que en caso de una operación con éxito se debe devolver un código 204, lo que hacemos con el método *NoContent()*.

Los dos aspectos más importantes de este método son los siguientes:

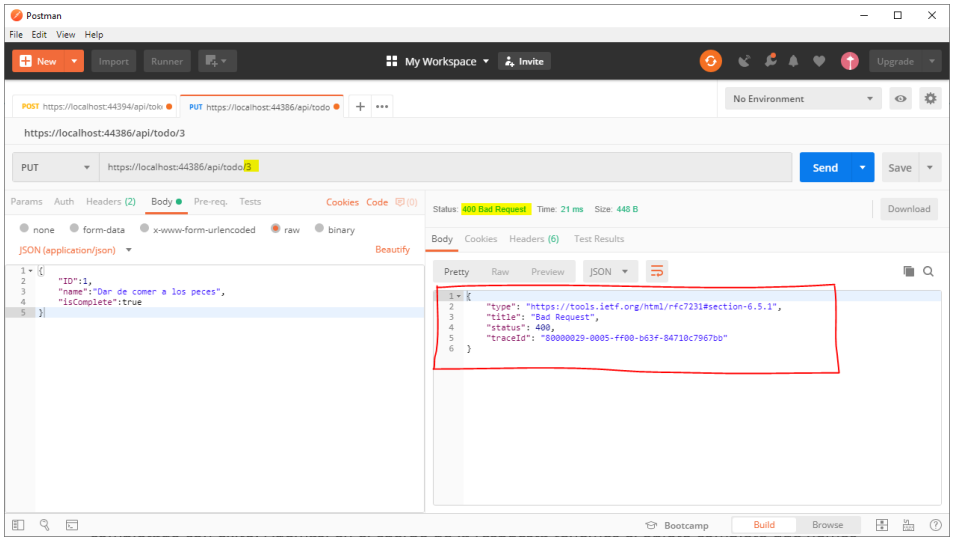
2. Para probar nuestro nuevo método arrancamos una vez más nuestro servicio desde Visual Studio.

3. Creamos una nueva petición en Postman, seleccionando esta vez el método PUT y la URL del objeto que queremos modificar, en nuestro ejemplo https://localhost:xxxxxxx/api/todo/1.
En el cuerpo de la petición introducimos el objeto que vamos a modificar. Por último pulsamos *Enviar*.



Podemos ver cómo se devuelve un código 204 que indica que el método se ha ejecutado correctamente. Si hacemos una petición GET podremos comprobar como el objeto ha cambiado.

4. Vamos a probar ahora a intentar actualizar un objeto inexistente. Para ello simplemente cambiamos la URI por un ID de un objeto que no esté disponible.



En este caso obtenemos un código 400, de petición incorrecta, ya que no existe el objeto que tratamos de modificar.

Eliminar una tarea.

Por último añadiremos el método que falta en nuestra API, que permitiría eliminar una tarea.

1. Añadimos el siguiente código a nuestro controlador:

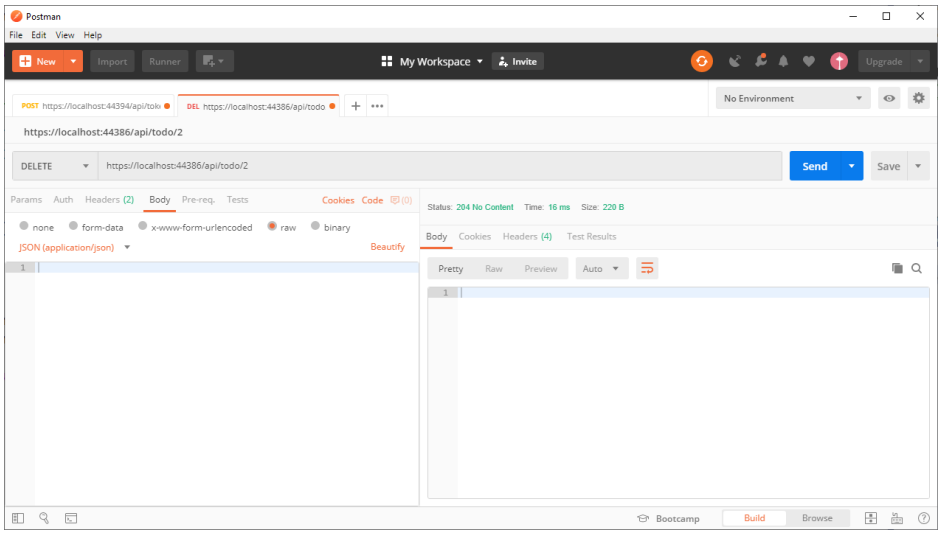
```
// DELETE: api/Todo/5
[HttpDelete("{id}")]
public IActionResult DeleteTodoItem(long id)
{
    var todoItem = _context.TodoItems.Find(id);

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

    _context.TodoItems.Remove(todoItem);
    _context.SaveChanges();

    return NoContent();
}
```

Al igual que ocurría con PUT, se comprueba que existe el objeto a modificar, y en caso contrario devuelve un código de error. Si el objeto se elimina con éxito se devuelve un código 204 (No content).

2. Arrancamos por última vez el servicio desde Visual Studio.

3. Creamos una nueva petición en Postman, seleccionamos el método DELETE y la URL del objeto a eliminar, en nuestro ejemplo https://localhost:xxxxxxx/api/todo/1.

4. Cuando pulsemos Enviar obtendremos un código 204 si se encuentra el objeto que queremos eliminar y un código 404 en caso contrario.

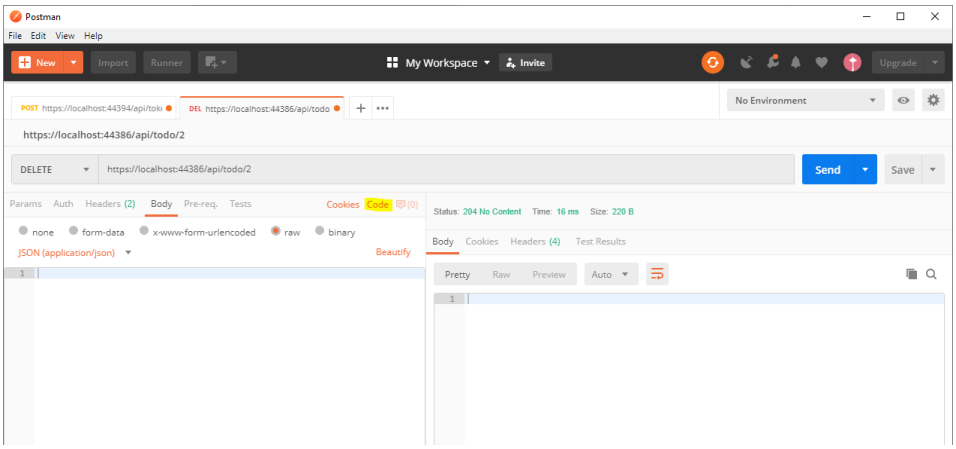


Código cliente.

Como ya hemos comentado en varias ocasiones, una de las ventajas de los servicios web es que estos pueden ser consumidos por clientes desarrollados para cualquier plataforma y en cualquier lenguaje de programación. Una de las opciones más habituales es la de consumir los servicios web utilizando código javascript desde una página web.

Una de las funcionalidades de Postman es la de generar de forma automática código cliente para consumir los servicios, tal y como esta petición está definida en su ventana de solicitud.

Por ejemplo, si quisiéramos incluir código javascript en una página para eliminar una tarea, definiríamos la petición como DELETE e introduciríamos la URL del objeto a eliminar.



Ahora, para ver el código necesario para generar una petición igual que la que haríamos pulsando el botón **Enviar**, pulsaremos en el enlace **Code**.

CIFP A Carballeira-Marcos Valcárcel
Rúa A Carballeira s/n
Ourense CP 32002
Tfno. 988788470, Fax 988227935

A continuación elegimos el lenguaje de programación para el que queremos obtener el código cliente, dentro de los muchos disponibles. En nuestro caso vamos a seleccionar una petición jQuery AJAX.