RESTful Web Services - API

Vidly, curso ASPNET MVC 5 y Web API

Objetivo y descripción introductoria

El objetivo principal era crear una aplicación que cumpliera con una arquitectura RESTful Web API, incluyendo las funcionalidades principales de la misma.

Sin embargo, **noté que la implementación se estaba llevando en un entorno de desarrollo utilizando .NET Framework**, y por la diferencia de versiones, algunas sentencias, estructuras, librerías o funciones ya no cumplían o se encuentran descontinuadas, **por lo que decidí continuar con la implementación de la aplicación del curso utilizando .NET Core** .

El repositorio en cuestión, se encuentra creado, acondicionado y configurado en una arquitectura **ASP.NET Core Web API**.

De igual manera, este repositorio tiene como fin principal, demostrar el funcionamiento de las operaciones básicas de un servicio web REST, aún no se ha añadido funcionalidad más compleja a la aplicación.

Aspectos, conceptos y conocimientos aplicados

Debido a que la implementación fue realizada en ASP.NET Core, se realizaron algunos ajustes:

Dependency Injection

Algunos paquetes, librerías, namespaces y configuraciónes fueron configuradas dentro del Startup.cs de la siguiente manera:

Configure Services

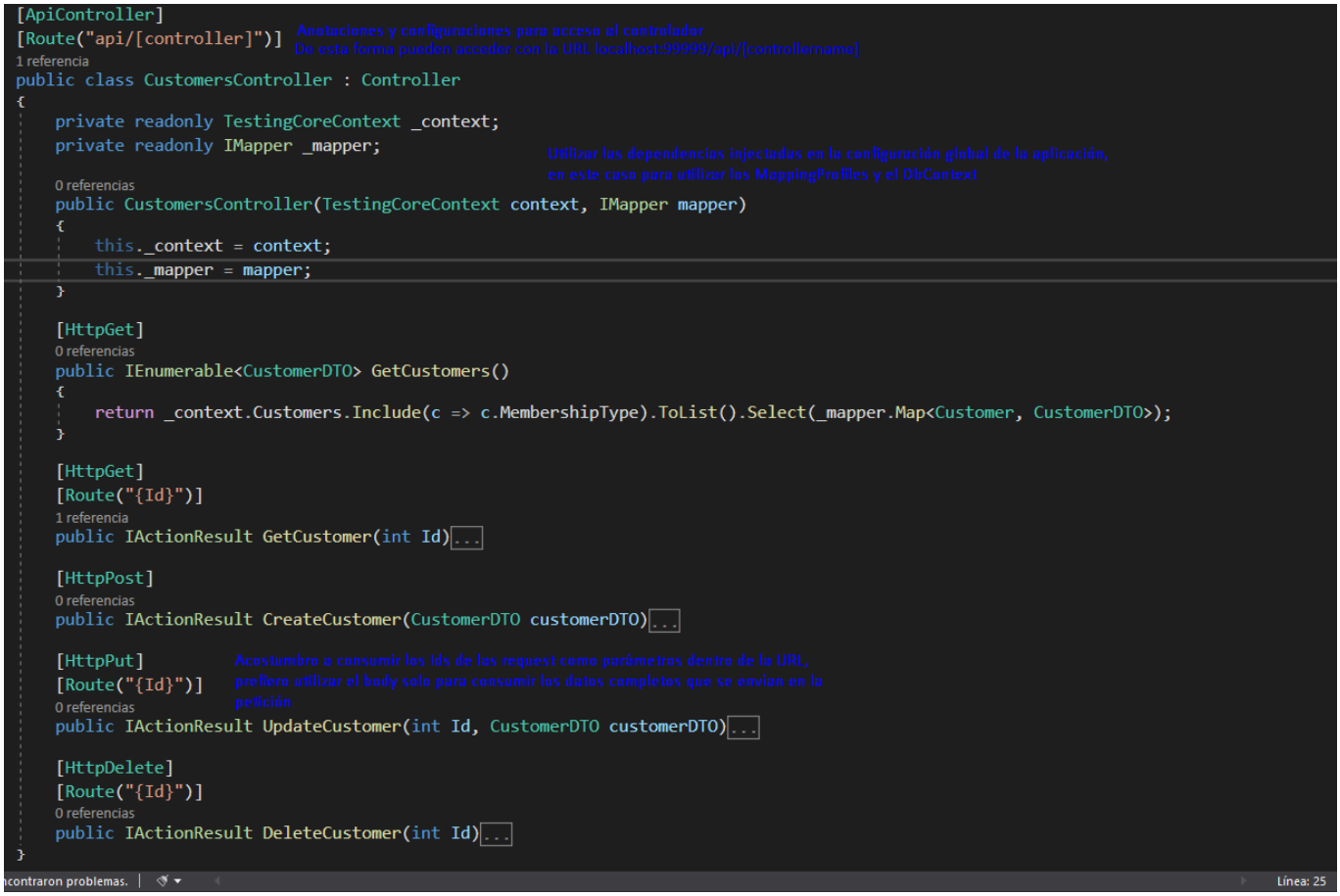
```
// This method gets called by the runtime. Use this method to add services to the container.
0 referencias
public void ConfigureServices(IServiceCollection services)
{
    services.AddAutoMapper(typeof(Startup));   Configuración de clase Mapper vía Dependency Injection

    services.AddControllers().AddNewtonsoftJson(options =>
    {                                          Aplicar formato y estructura camelCase para las respuestas en Json
        options.SerializerSettings.ContractResolver = new CamelCasePropertyNamesContractResolver();
        options.SerializerSettings.Formatting = Newtonsoft.Json.Formatting.Indented;
    });

    services.AddCors(c => c.AddPolicy("TestingPolicy", builder =>
    {
        builder.AllowAnyOrigin()   Configuración de políticas para aceptar o rechazar las solicitudes, metodos y orígenes de los distintos clientes.
        .AllowAnyMethod()          Por motivos de desarollo y prueba, aceptará cualquier origen.
        .AllowAnyHeader();
    }));

    services.AddDbContext<TestingCoreContext>(options =>   Configuración de DbContext para el mapeo de entidades y conexión con EntityFrameworkCore
    {
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConenction"));
    });

    services.AddDefaultIdentity<ApplicationUser>()   Colocar clase/entidad de usuario por default para la autenticación de usuarios utilizando AspNetCore Identity
        .AddEntityFrameworkStores<TestingCoreContext>();
}
```

Configure

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseCors("TestingPolicy");   Hacer el uso de las políticas declaradas en la configuración para aceptar o denegar las request.

    app.UseHttpsRedirection();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

Controllers

Debido a que en esta plantilla o arquitectura de trabajo no se cuenta con vistas o está directamente configurada para funcionar como WebAPI, .NET Core hace más sencillo adaptar las rutas y la declaración de ApiControllers:



ActionResults

De igual forma y gracias al curso, he implementado las convenciones que mejor se adapten a una aplicación tipo RESTful, mandando los resultados correspondientes:

```
[HttpGet]
[Route("{Id}")]
1 referencia
public IActionResult GetCustomer(int Id)
{
    var customerInDb = _context.Customers.SingleOrDefault(c => c.Id == Id);

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

    return Ok(_mapper.Map<Customer, CustomerDTO>(customerInDb));
}

[HttpPost]
0 referencias
public IActionResult CreateCustomer(CustomerDTO customerDTO)
{
    if (!ModelState.IsValid)
    {
        return BadRequest();
    }

    Customer customer = _mapper.Map<CustomerDTO, Customer>(customerDTO);

    _context.Add(customer);
    _context.SaveChanges();

    customerDTO.Id = customer.Id;


    return CreatedAtAction(nameof(GetCustomer), new { Id = customer.Id }, customerDTO);

    //return Ok(customerDTO);
}
```

Siendo sincero y de manera muy personal, yo acostumbraba más a utilizar respuestas tipo JsonResult en la mayoría de mis respuestas, debido a que se me hacía más facil adaptar las respuestas para los clientes de esta forma. Pero entiendo que, **al utilizar una WebAPI como medio de consumo para N cantidad y variedad de clientes, tiene que estar lo mejor adaptada para distintos escenarios.**

Funcionalidad

El proyecto contiene una funcionalidad basada en RESTful Web Services / API, por lo que se encarga de manejar todas las peticiones recibidas por los clientes, utilizando el protocolo HTTP como parte principal y sus métodos, los cuales son:

Método	Función
GET	Obtener, consultar, o leer información/recursos
POST	Envíar datos, información o recursos (normalmente para insertar datos en el servidor)

Método	Función
PUT	Actualizar algún recurso del lado del servidor
DELETE	Eliminar/remover algún recurso u objeto del servidor

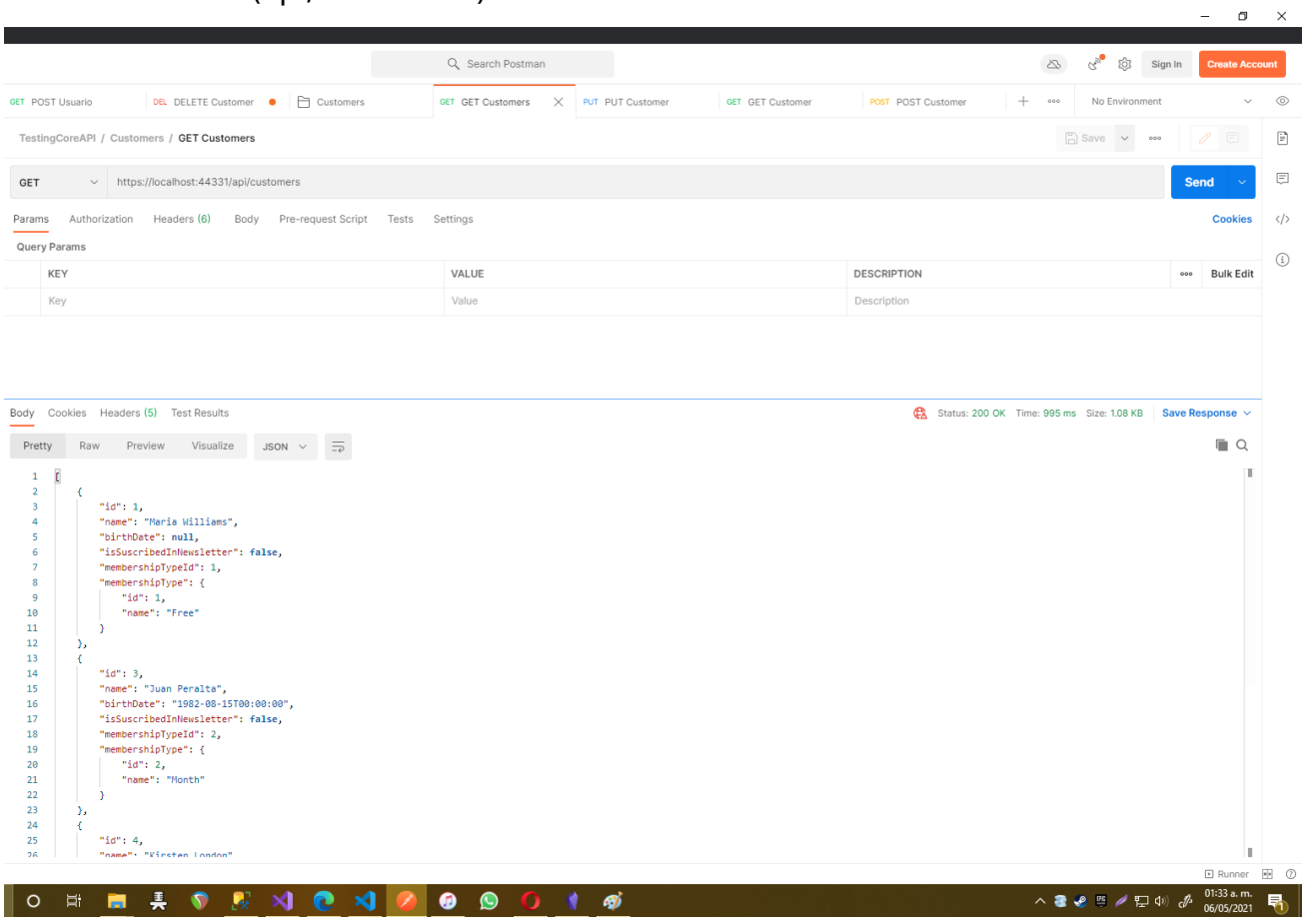
Derivado de esto, la aplicación contiene las funciones para consultar, insertar, actualizar y eliminar registros que estén en una base de datos, siendo o actuando en su mayoría como un CRUD.

Pruebas

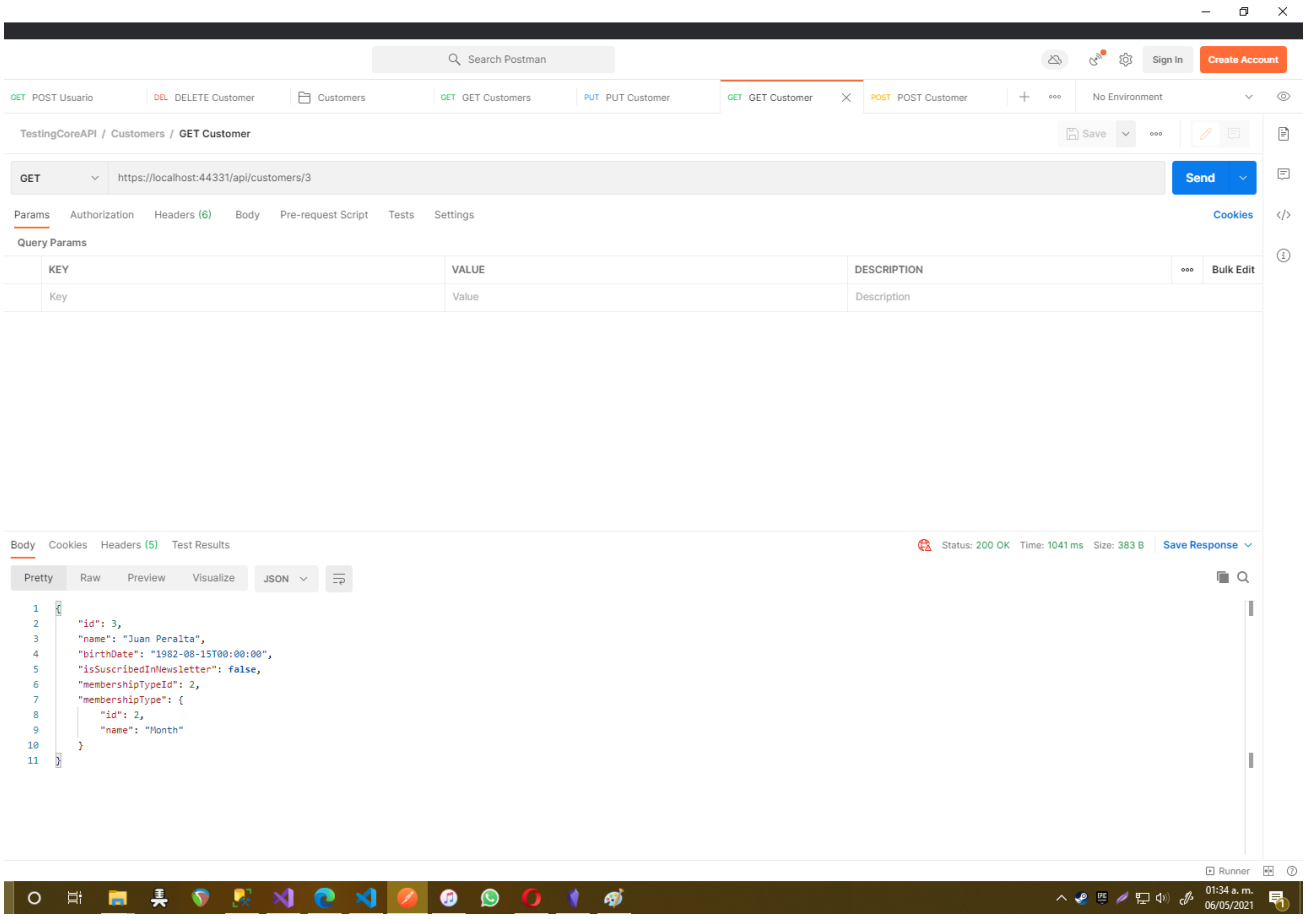
Las funcionalidades de esta aplicación han sido probadas en primera instancia utilizando la herramienta **Postman** para el manejo de solicitudes a los servicios y evaluación de las respuestas.

CustomersController

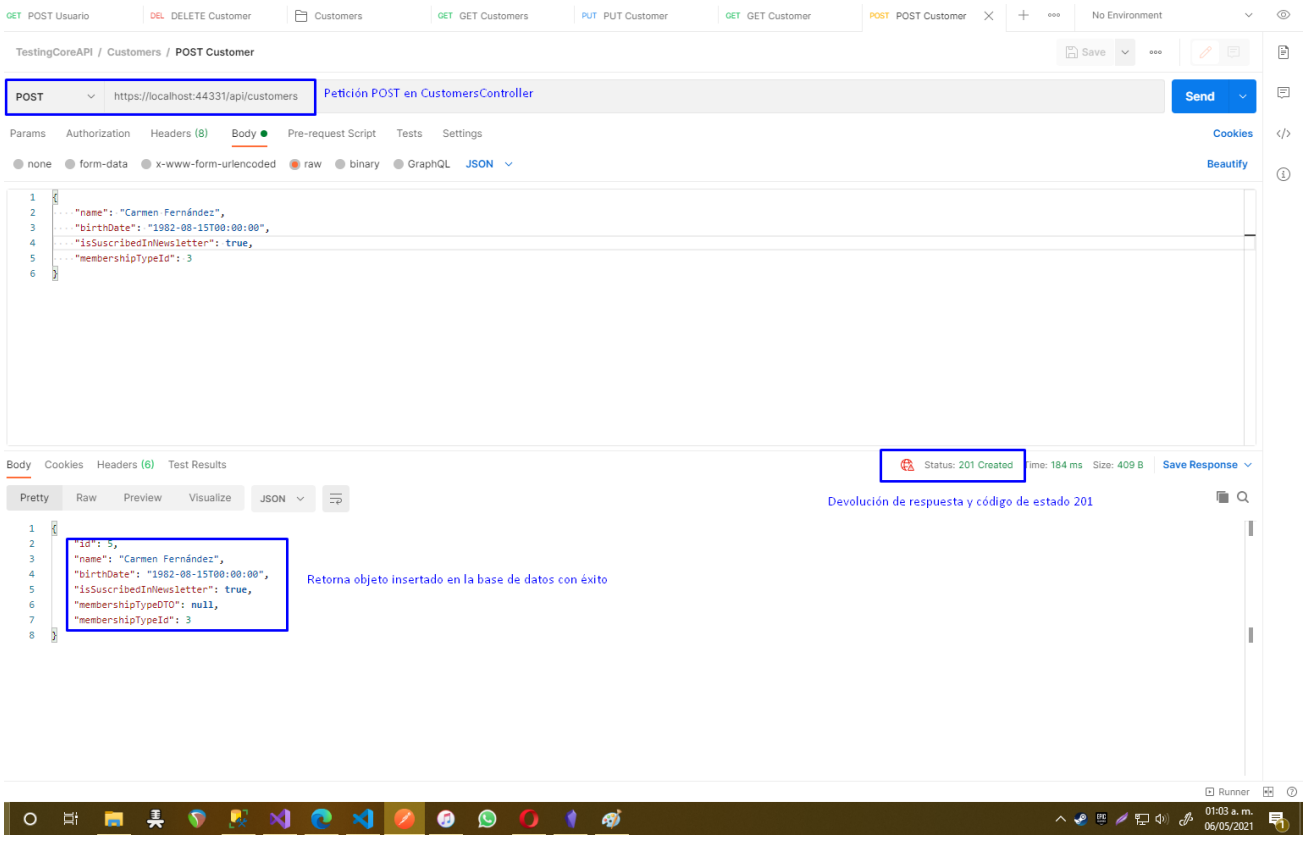
- GET Customers (api/customers)



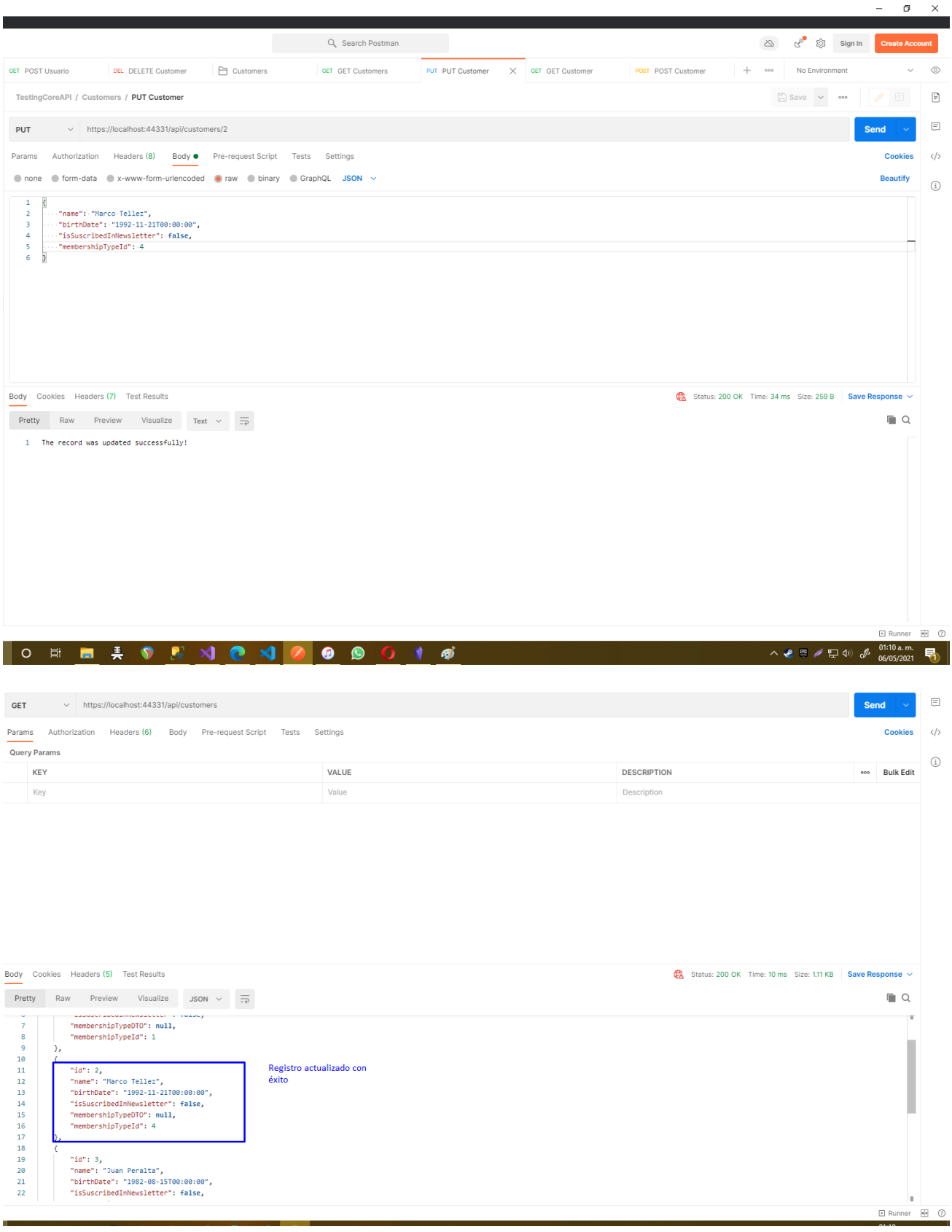
GET Customer (api/customers/Id)



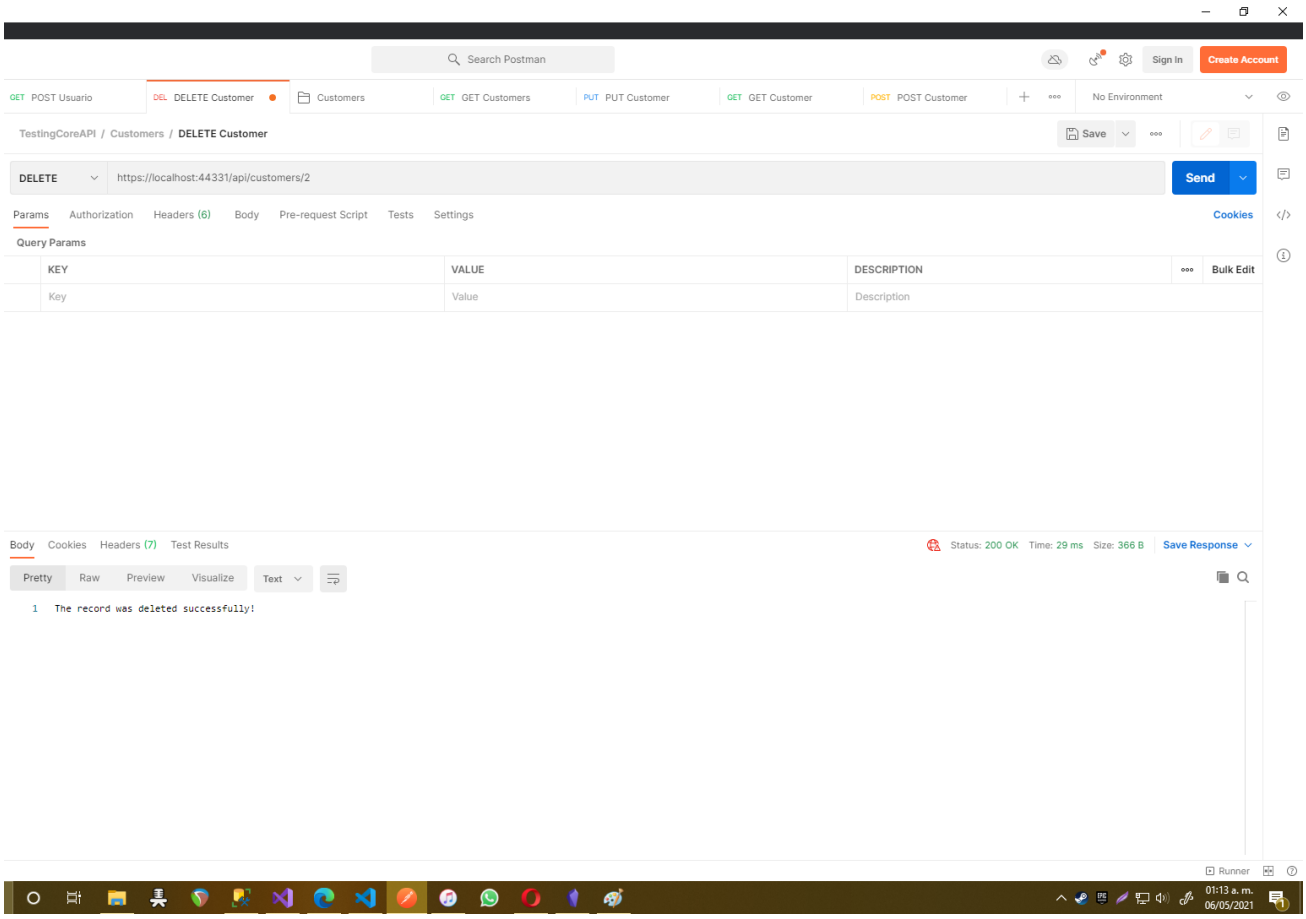
POST Customer (api/customer)



- PUT Customer (api/customer/Id)

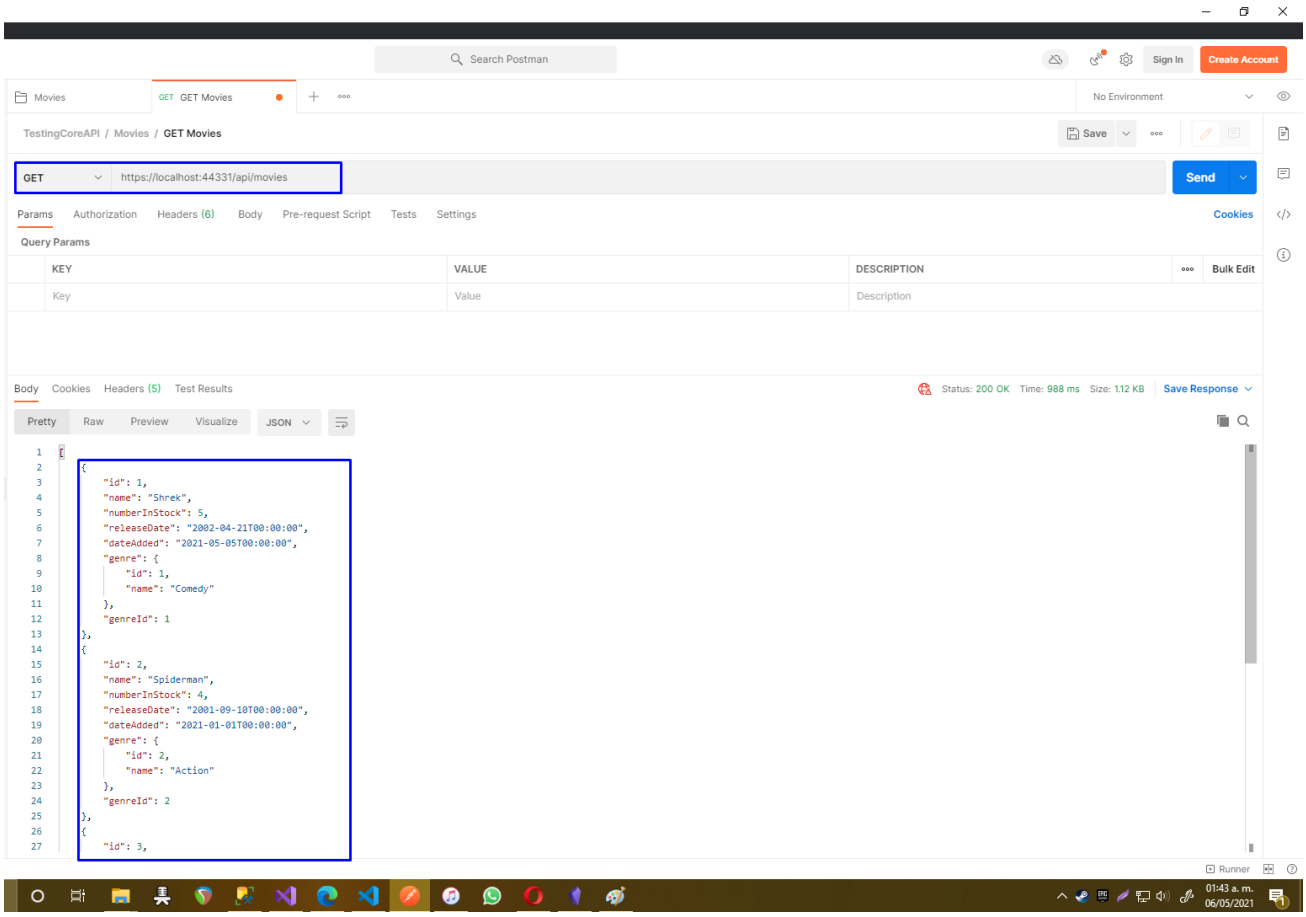


- DELETE Customer (api/delete/Id)

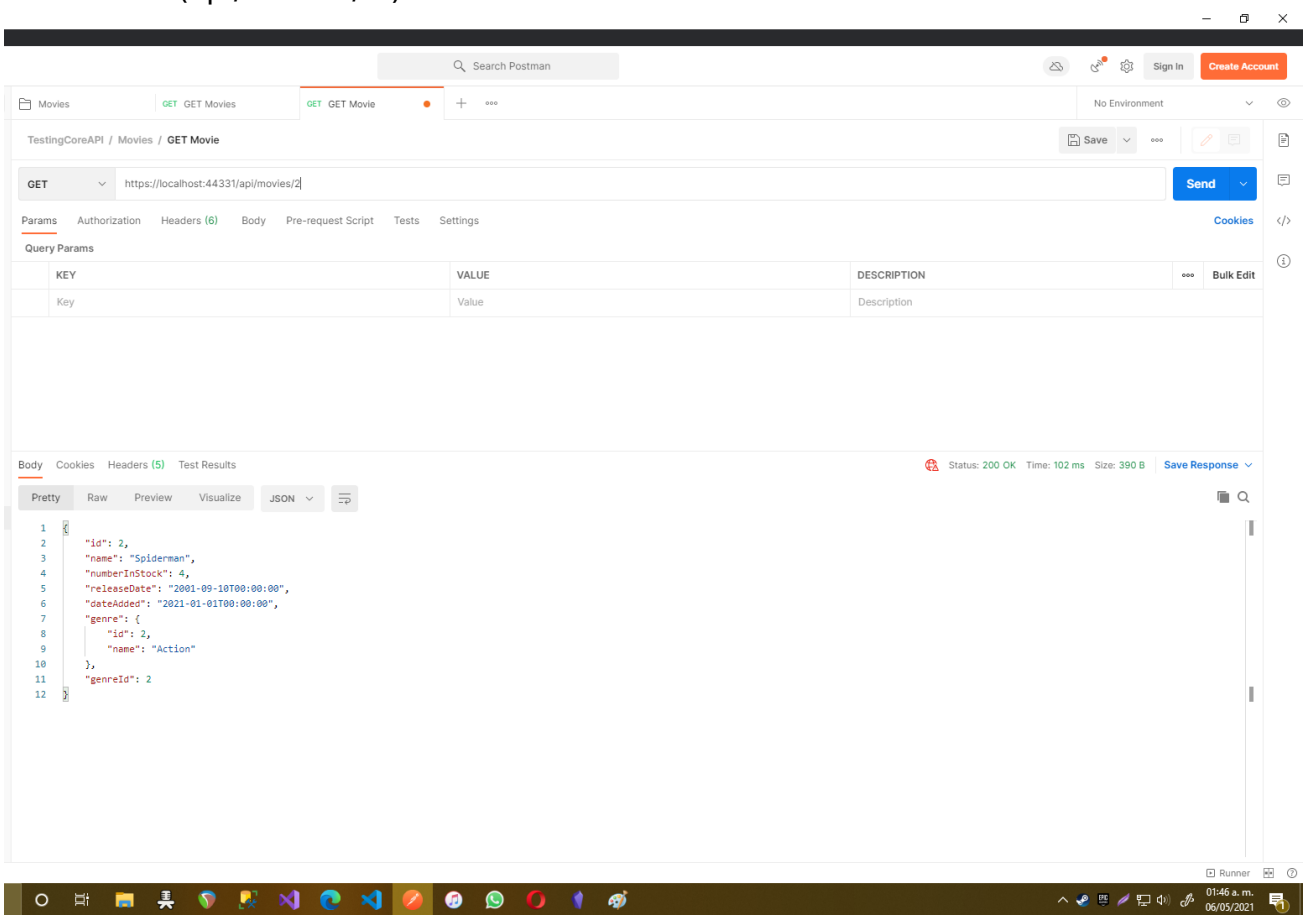


MoviesController

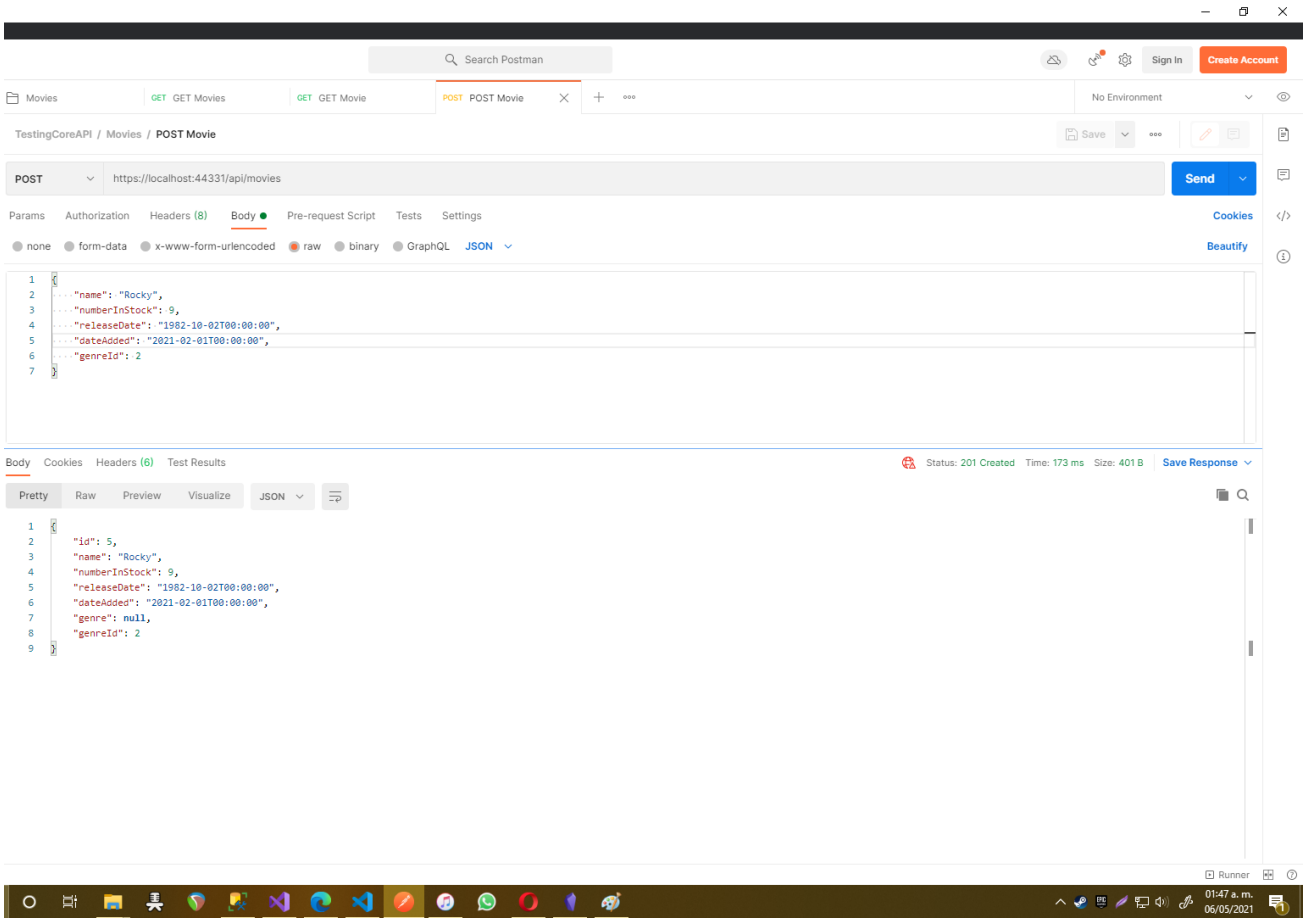
- GET Movies (api/movies)



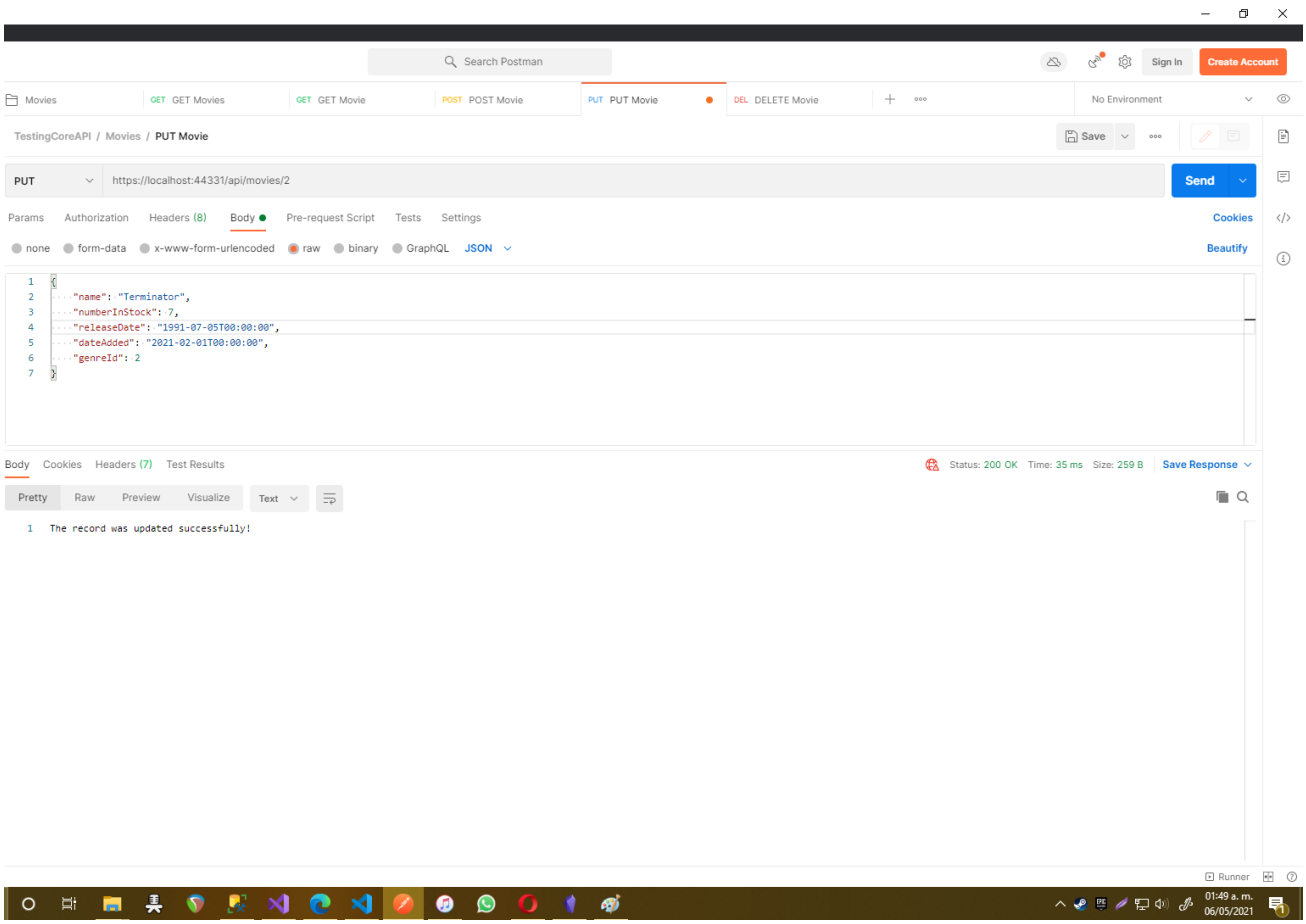
- GET Movie (api/movies/Id)

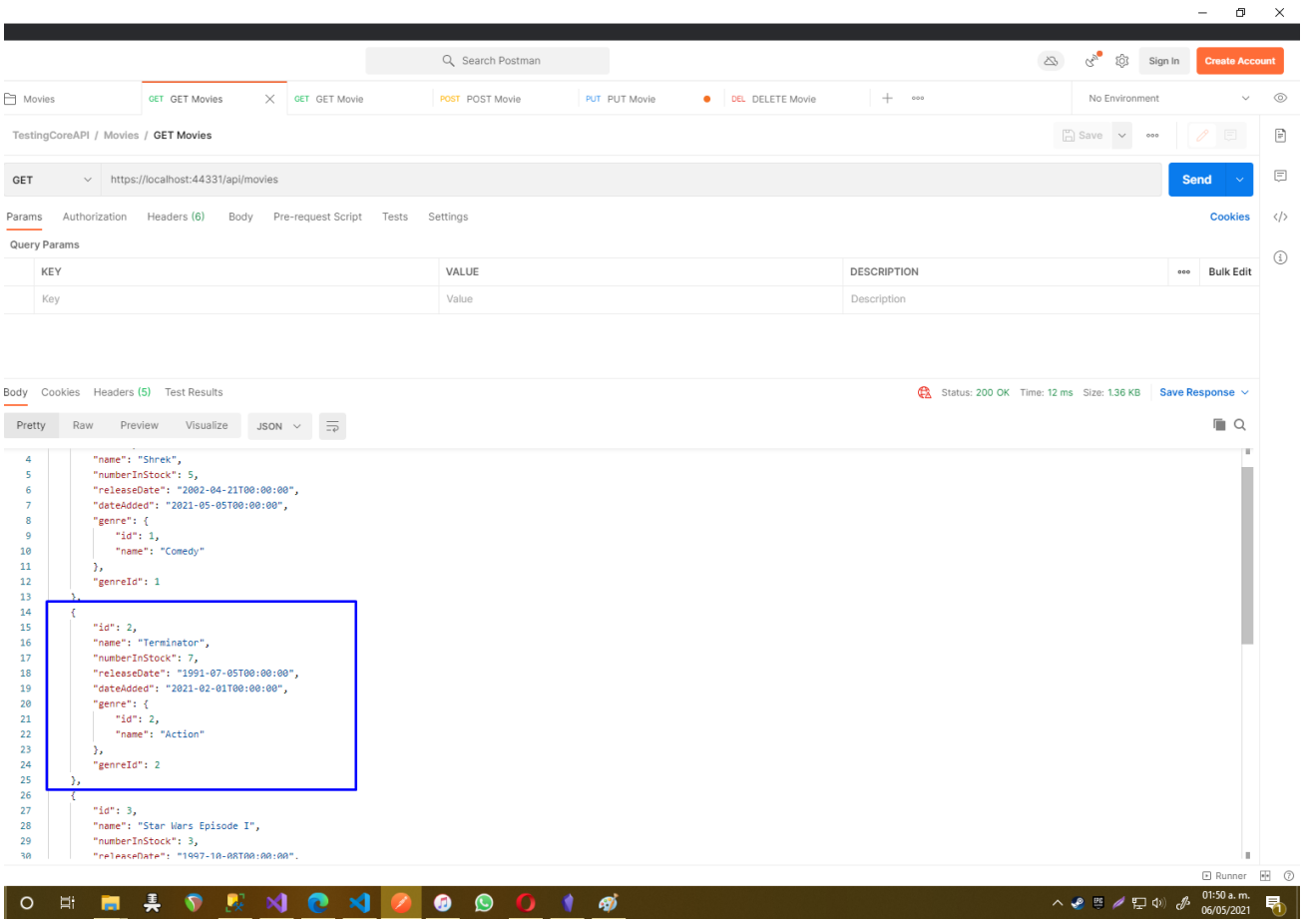


POST Movie (api/movies)

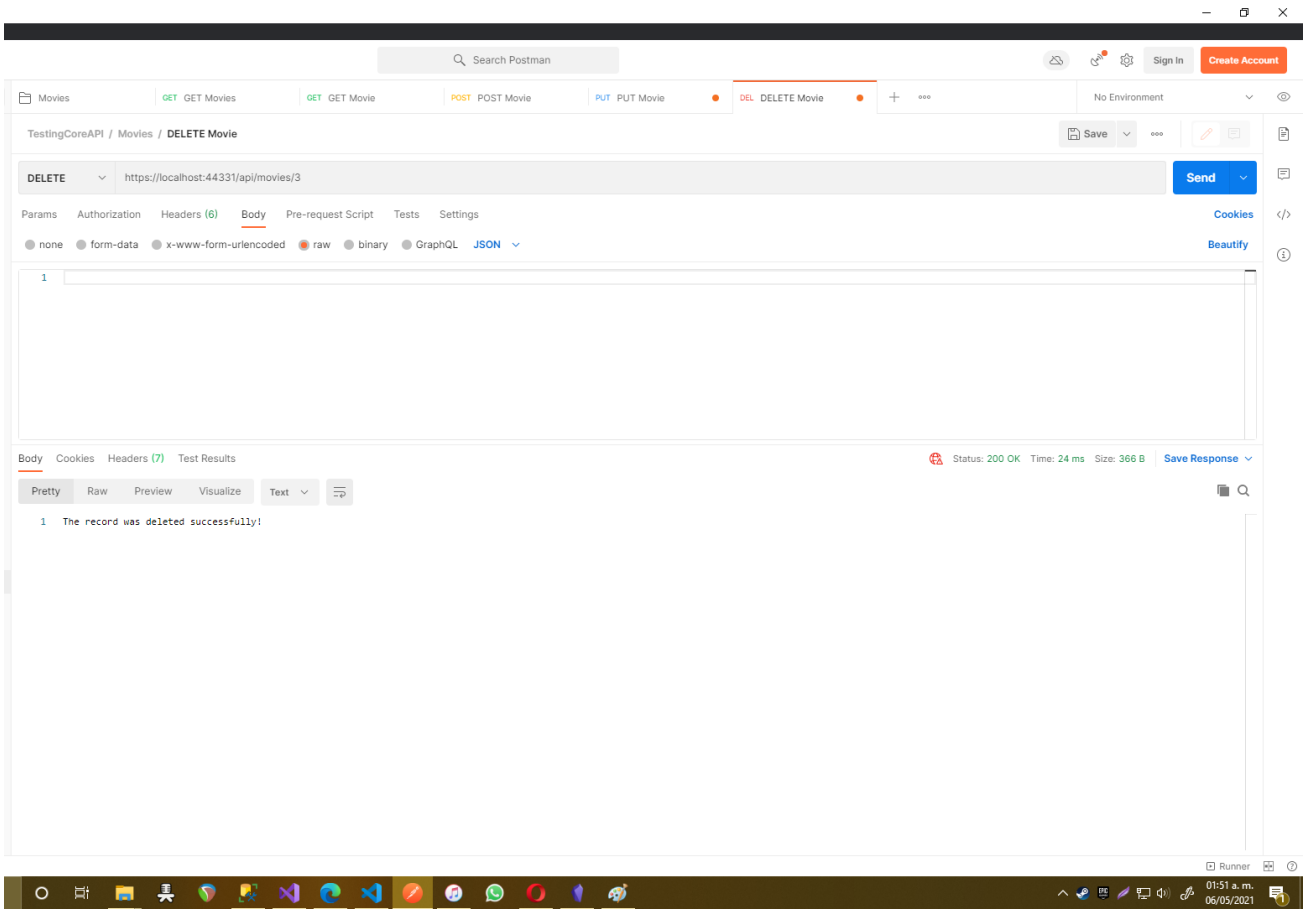


PUT Movie (api/movies/Id)





- DELETE Movie (api/movies/Id)

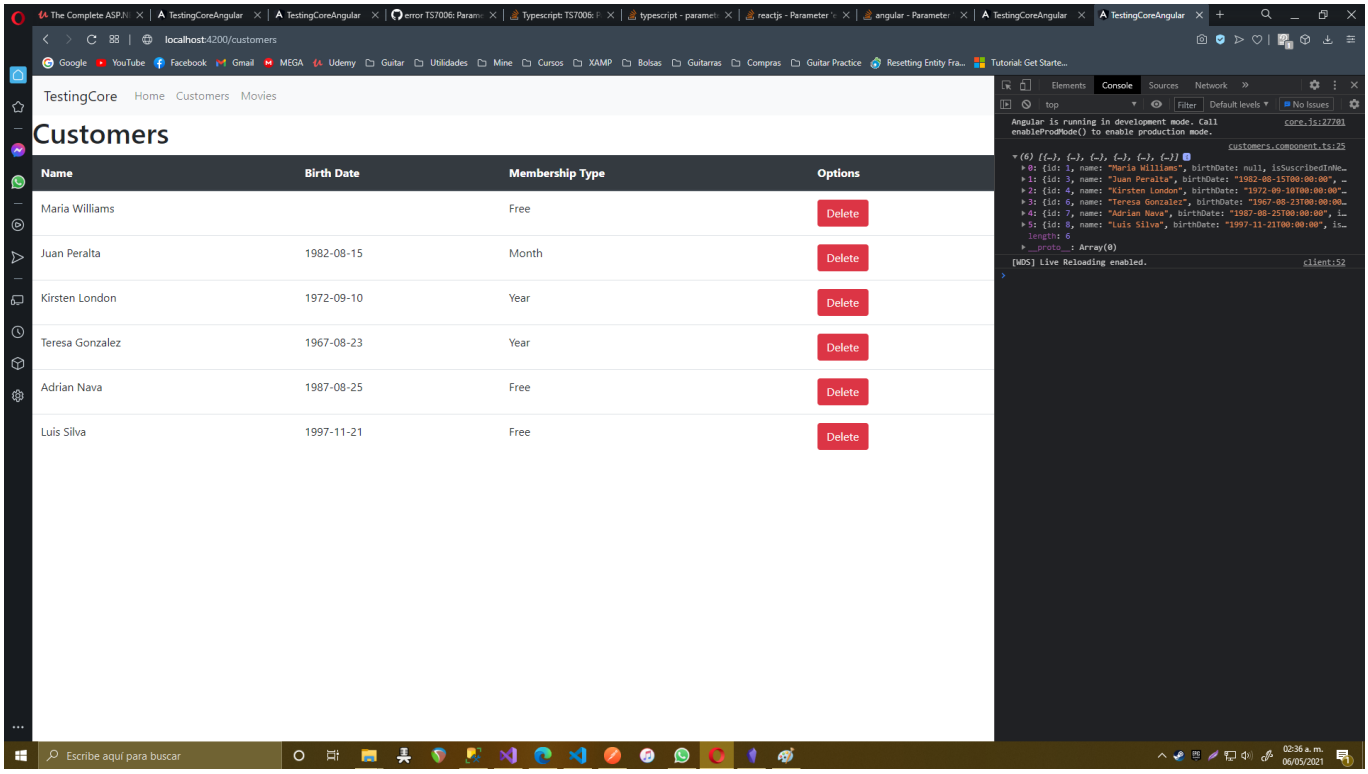


Pruebas con otros Frameworks

De igual manera, he decidido realizar un pequeño proyecto/aplicación en angular para manejar el apartado del cliente, y así, poder realizar las consultas y peticiones a la Web API.

Adjunto una pequeña demostración en donde nuestra aplicación de angular hace un llamado a nuestra Web API mandando una solicitud GET al controlador CustomersController:

Vista Customer en AngularJS



Componente Customer en AngularJS

```typescript
@Component({
  selector: 'app-customers',
  templateUrl: './customers.component.html',
  styleUrls: ['./customers.component.css']
})
export class CustomersComponent implements OnInit {

  constructor(private service: SharedService, private datePipe: DatePipe) { }

  CustomerList: any = [];

  oCustomer: any;

  ngOnInit(): void {
    this.refreshCustomerList();
  }

  refreshCustomerList() {
    this.service.getCustomers().subscribe(data => {
      this.CustomerList = data;
      console.log(data);
    });
  }

  deleteCustomer(customer) {
    console.log(customer);
    this.service.deleteCustomer(customer.id).subscribe(data => {
      console.log(data);
    });
  }
}
```

Obtener la lista de clientes de nuestra WebAPI

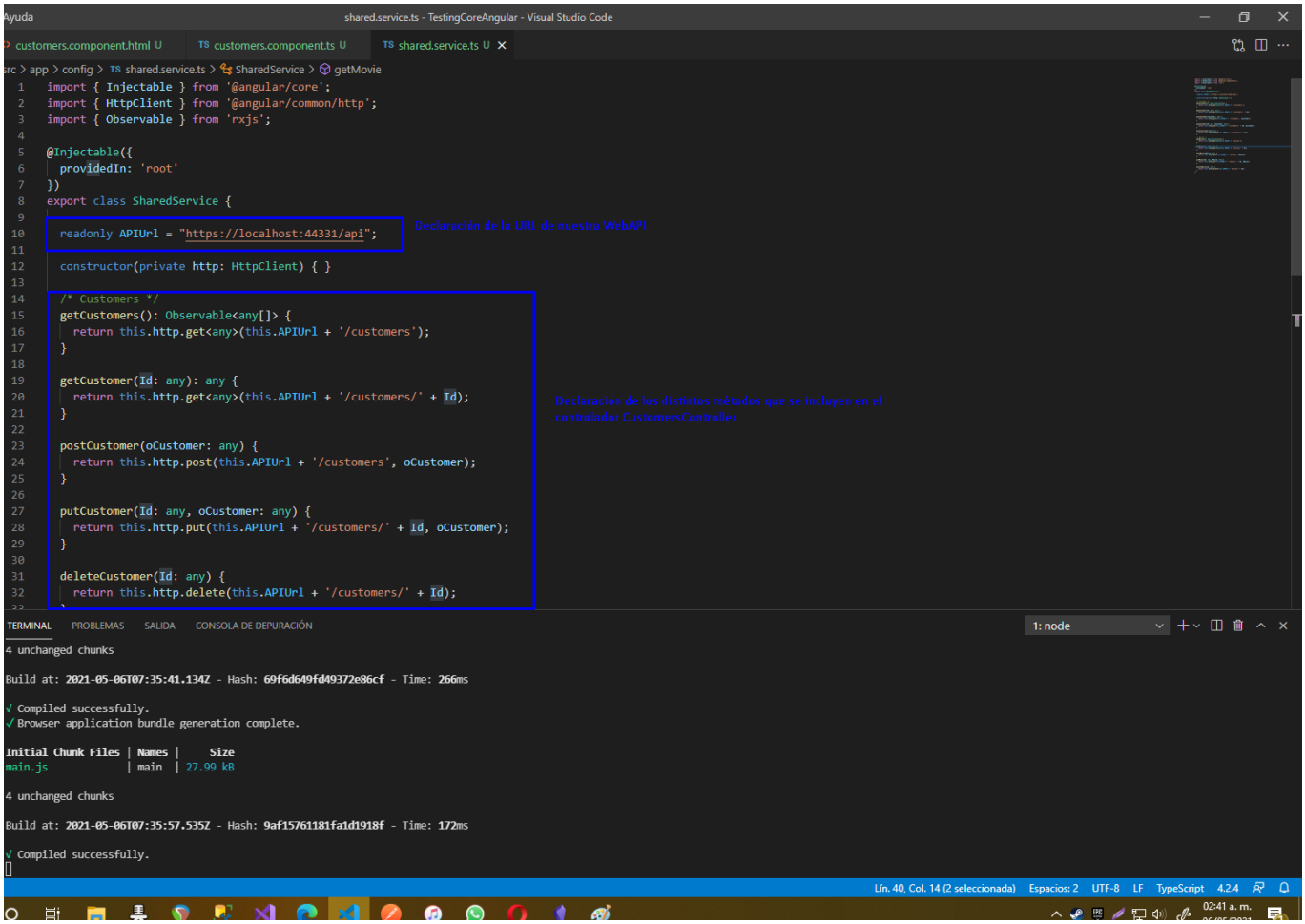
```html
<div>
    <h1>Customers</h1>
</div>

<div>
    <table class="table table-hover">
        <thead class="thead-dark">
            <tr>
                <th>Name</th>
                <th>Birth Date</th>
                <th>Membership Type</th>
                <th>Options</th>
            </tr>
        </thead>
        <tbody>
            <tr *ngFor="let customer of CustomerList">
                <td>{{customer.name}}</td>
                <td>{{customer.birthDate | date: 'yyyy-MM-dd'}}</td>
                <td>{{customer.membershipType.name}}</td>
                <td>
                    <button class="btn btn-danger" (click)="deleteCustomer(customer)">
                        Delete
                    </button>
                </td>
            </tr>
        </tbody>
    </table>
</div>
```

Colección obtenida de nuestra Web API

Parte técnica y declaración de la URL en Angular

Creé un servicio el cual tiene declarada la URL de la API desarrollada en ASP.NET Core, de esta manera, solo es ir agregando los métodos que tenga el API para poder realizar las diferentes acciones:

SharedService AngularJS



Notas Generales/Conclusiones

Aprovecho este espacio para poder comentar mis conclusiones y algunas notas extra.

El objetivo de este proyecto/aplicación era demostrar las funciones básicas/principales de una aplicación RESTful, la cuál normalmente se implementa del lado del servidor y que está orientada a resolver peticiones por parte de los clientes manejando como parte base el protocolo HTTP y todas sus acciones/métodos.

Debido a que era un proyecto de introducción, **aun falta mayor funcionalidad respecto al curso**.
Algunos de esos conceptos faltantes era la implementación de autenticación de usuario, roles y otros aspectos utilizando Identity.

Planeo aplicarlos en este proyecto debido a que. **he escuchado y he visto distintas formas de validar las sesiones y la autenticación, tal como el caso de JWT (JsonWebToken) entre otros**.

Veo que el manejo de un ORM es de bastante ayuda debido a que, al refactorizar o realizar ajustes mayores dentro de una WebAPI, expone menos a los clientes que la consumen, ya que, si son ajustes no controlados, impactaría a todos los clientes enlazados a la misma.

Realmente espero que este trabajo haya sido de su agrado y haya demostrado conceptos que he aprendido tanto de forma autodidácta, y con la ayuda del curso que me compartieron.
Pienso que, uno de los obstáculos de este análisis, estudio e incluso implementación, fue el traspasar ciertos conocimientos en una arquitectura .NET Framework a .NET Core.