

Manual de usuario

GASTROLAB(Documentación_breve)

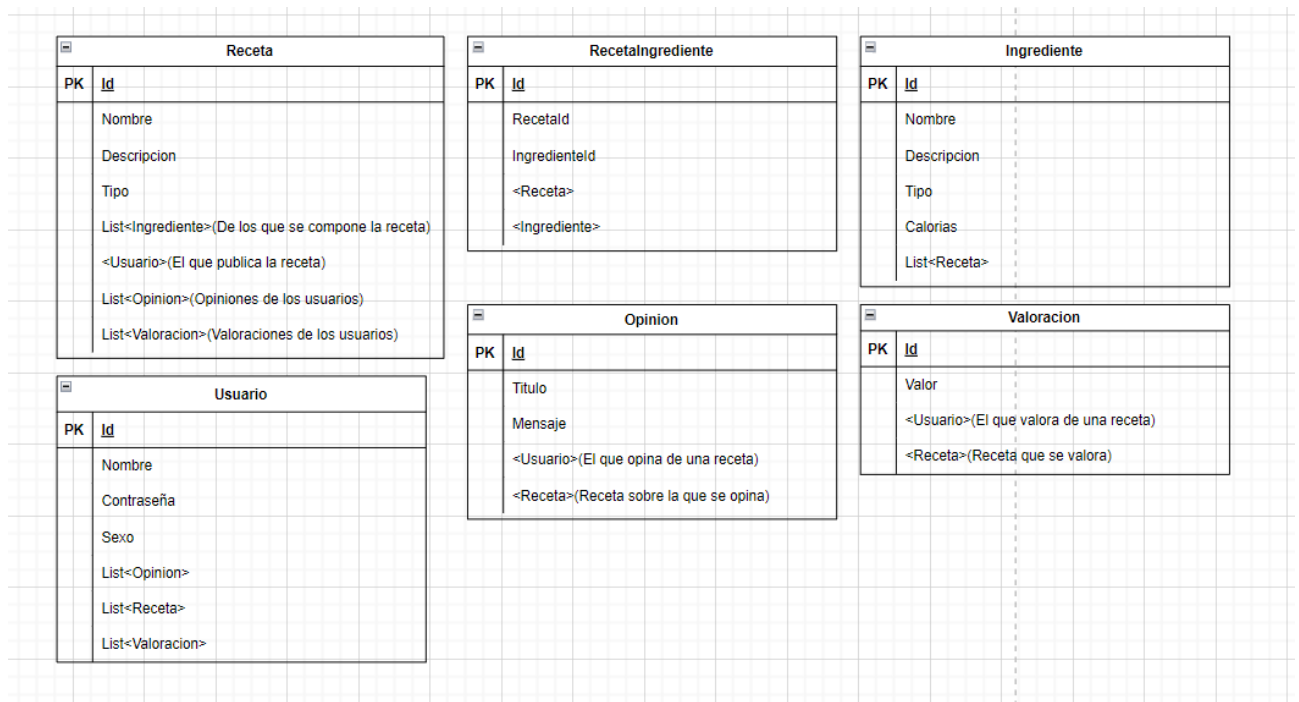
1.Preludio(Introducción al proyecto).

Comenzando desde cero, Gastrolab es una iniciativa conformada por dos proyectos uno de angular y otro de .net que junto con una base de datos conforman un proyecto conjunto visualizable tanto desde la web como desde la api en formato json.

La idea principal de Gastrolab es generar una plataforma donde los usuarios sean capaces de crear recetas, discutir sobre ellas mediante opiniones y votarlas y ver su resultado en la sección principal.

2.Creación del modelo de datos en draw.io.

Este es el modelo final y escueto de lo que son los modelos y relaciones entre las tablas del proyecto durante el desarrollo de este se han ido añadiendo y quitando tablas a veces pecando de ser demasiado ambicioso incluyendo funcionalidades sin tiempo a desarrollar en 40 horas. La idea principal del modelo es que todo gire en torno a recetas, pero que el resto de tablas estén muy interrelacionadas entre ellas y puedas obtener información de unas sobre las otras sin ningún problema, esa es la premisa principal del modelo



3.Creación del proyecto de la api en .net y creación de los modelos.

Se comenzo el proyecto de .net con una pequeña plantilla que proporciona visual studio lo mejor de esta es que ya viene swagger implementado el cual nos permitira probar los metodos de nuestros controladores de una forma mucho mas interactiva que simplemente tirando peticiones web o rezando que funcione ya desde el service de angular, a partir de ahí se creo una carpeta modelos donde se alojarían los diferentes modelos definidos previamente desde draw.io.



ASP.NET Core Web API

Una plantilla de proyecto para crear una aplicación ASP.NET Core con un controlador de ejemplo para un servicio RESTful HTTP. Esta plantilla también puede usarse para controladores y vistas de ASP.NET Core MVC.

[C#](#)[Linux](#)[macOS](#)[Windows](#)[Nube](#)[Servicio](#)[Web](#)[WebAPI](#)

```
namespace GastroLabApp.Models
{
    22 referencias
    public class Receta
    {
        9 referencias
        public int Id { get; set; }
        1 referencia
        public string Nombre { get; set; }
        0 referencias
        public string Descripcion { get; set; }
        0 referencias
        public string Tipo { get; set; }
        0 referencias
        public string Url { get; set; }
        1 referencia
        public ICollection<RecetaIngrediente> IngredientesReceta { get; set; }
        2 referencias
        public ICollection<Opinion>? Opiniones { get; set; }
        2 referencias
        public ICollection<Valoracion>? Valoraciones { get; set; }
        2 referencias
        public Usuario Usuario { get; set; }
    }
}
```

Una vez creados nuestros modelos, crearemos la carpeta data y la clase DataContext.cs donde declararemos los diferentes datasets o tablas que luego irán a la base de datos, también desde appsetting configuramos la conexión a una base de datos vacía creada en sql server management studio

```
using GastroLabApp.Models;
using Microsoft.EntityFrameworkCore;
namespace GastroLabApp.Data
{
    15 referencias
    public class DataContext: DbContext
    {
        0 referencias
        public DataContext(DbContextOptions<DataContext> options) : base(options)
        {
        }
        5 referencias
        public DbSet<Ingrediente> Ingredientes { get; set; }
        7 referencias
        public DbSet<Receta> Recetas { get; set; }
        4 referencias
        public DbSet<Usuario> Usuarios { get; set; }
        4 referencias
        public DbSet<Opinion> Opiniones { get; set; }
        2 referencias
        public DbSet<Valoracion> Valoraciones { get; set; }
        3 referencias
        public DbSet<RecetaIngrediente> RecetasIngredientes { get; set; }



        0 referencias
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<RecetaIngrediente>()
                .HasKey(ri => new { ri.RecetaId, ri.IngredienteId });
            modelBuilder.Entity<RecetaIngrediente>()
                .HasOne(r => r.Receta)
                .WithMany(ri => ri.IngredientesReceta)
                .HasForeignKey(i => i.RecetaId);
        }
    }
}
```

```

{
  "ConnectionStrings": {
    "DefaultConnection": "Data Source=DESKTOP-9Q02QRT;Initial Catalog=GastroLabApp;Integrated Security=True;C
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}

```

Tras haber establecido el data context y la conexión a la base de datos vacía instale desde los paquetes de nuggets de la solución las extensiones:

	Microsoft.EntityFrameworkCore.SqlServer por Microsoft	7.0.5
	Microsoft SQL Server database provider for Entity Framework Core.	7.0.7
	Microsoft.EntityFrameworkCore.Tools por Microsoft	7.0.5
	Entity Framework Core Tools for the NuGet Package Manager Console in Visual Studio.	7.0.7

Estas dos extensiones me ahorrarían un gran tiempo para usar en otras partes del desarrollo ya que desde la consola de paquetes nuggets y con la conexión establecida con la base de datos vacía me permitirían migrar los modelos creados a la base de datos creando las tablas mediante los datasets y los comandos Add-Migration y Update-Database.

Ya con nuestra base de datos creada y con nuestros modelos acabados pasaríamos a la creación de las interfaces donde estableceríamos los métodos que deben crearse obligatoriamente en el repositorio de cada modelo y una vez acabadas las interfaces pasaríamos en si al repositorio el cual interactuar con la base de datos mediante un objeto context.

```

using GastroLabApp.Models;

namespace GastroLabApp.Interfaces
{
    9 referencias
    public interface IRecetaRepository
    {
        3 referencias
        ICollection<Receta> GetRecetas();
        4 referencias
        Receta GetReceta(int id);
        2 referencias
        ICollection<Ingrediente> GetIngredientesByReceta(int id);
        2 referencias
        ICollection<Opinion> GetOpinionesByReceta(int RecetaId);
        2 referencias
        bool CreateReceta(List<int> IngredienteId, Receta receta);
        2 referencias
        bool UpdateReceta(Receta receta, List<int> IngredienteId);
        2 referencias
        bool DeleteReceta(int recetaId);
        6 referencias
        bool RecetaExist(int id);
        4 referencias
        bool Save();
    }
}

```

```

using GastroLabApp.Data;
using GastroLabApp.Dto;
using GastroLabApp.Interfaces;
using GastroLabApp.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Diagnostics;
using Microsoft.OpenApi.Any;

namespace GastroLabApp.Repository
{
    public class RecetaRepository : IRecetaRepository
    {
        private readonly DataContext context;

        public RecetaRepository(DataContext context)
        {
            this.context = context;
        }

        public ICollection<Receta> GetRecetas()
        {
            return context.Recetas.OrderBy(r => r.Id).ToList();
        }

        public Receta GetReceta(int RecetaId)
        {
            return context.Recetas.Where(r => r.Id == RecetaId).FirstOrDefault();
        }
    }
}

```

Creadas una vez todas las interfaces y repositorios con los métodos de adición, obtención, cambio y borrado que se consideraron adecuados se paso al desarrollo de los controles donde ya comenzariamos a manejar protocolos http y el uso del Get, Post, Delete y el Put.

```

using GastroLabApp.Models;
using GastroLabApp.Interfaces;
using Microsoft.AspNetCore.Mvc;
using AutoMapper;
using GastroLabApp.Dto;
using System.Collections.Generic;

namespace GastroLabApp.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class RecetaController : Controller
    {
        private readonly IRecetaRepository recetaRepository;
        private readonly IUserarioRepository usuarioRepository;
        private readonly IMapper mapper;

        public RecetaController(IRecetaRepository recetaRepository, IUserarioRepository usuarioRepository, IMapper mapper)
        {
            this.recetaRepository = recetaRepository;
            this.usuarioRepository = usuarioRepository;
            this.mapper = mapper;
        }

        [HttpGet]
        [ProducesResponseType(200, Type = typeof(IEnumerable<Receta>))]
        public IActionResult GetRecetas()
        {
            var recetas = mapper.Map<List<RecetaDto>>(recetaRepository.GetRecetas());
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }
            return Ok(recetas);
        }
    }
}

```

Acabados los controladores podriamos ya ejecutar nuestra API y comenzar a testear los resultados que se pueden obtener desde swagger sin embargo tras testear un poco por swagger me di cuenta de que a la hora de sobre todo recibir datos con el Get al recibir objetos como por ejemplo una receta tambien te venian los campos que conformaban la relación viéndose estos nulos ya que solo existian en el modelo para crear la relación entre tablas, tras esto investigue un poco porque sucedía y descubri la existencia de los dtos que en resumidas cuentas son objetos de transmisión de datos que solo contienen la información perteneciente a los modelos de forma individual, por lo que tambien cree un

mapping profile el cual serviría para cambiar en cualquier momento un objeto de dto a objeto de la clase.

```
using GastroLabApp.Models;

namespace GastroLabApp.Dto
{
    7 referencias
    public class RecetaDto
    {
        1 referencia
        public int Id { get; set; }
        1 referencia
        public string Nombre { get; set; }
        0 referencias
        public string Descripcion { get; set; }
        0 referencias
        public string Tipo { get; set; }
        0 referencias
        public string Url { get; set; }
    }
}
```

```
using AutoMapper;
using GastroLabApp.Dto;
using GastroLabApp.Models;

namespace GastroLabApp.Helper
{
    1 referencia
    public class MappingProfiles:Profile
    {
        0 referencias
        public MappingProfiles()
        {
            CreateMap<Receta, RecetaDto>();
            CreateMap<RecetaDto, Receta>();
            CreateMap<Ingrediente, IngredienteDto>();
            CreateMap<IngredienteDto, Ingrediente>();
            CreateMap<Usuario, UsuarioDto>();
            CreateMap<UsuarioDto, Usuario>();
            CreateMap<Opinion, OpinionDto>();
            CreateMap<OpinionDto, Opinion>();
            CreateMap<Valoracion, ValoracionDto>();
            CreateMap<ValoracionDto, Valoracion>();
        }
    }
}
```



AutoMapper por Jimmy Bogard
A convention-based object-object mapper.

12.0.1 ✖



AutoMapper.Extensions.Microsoft.DependencyInjection por J
AutoMapper extensions for ASP.NET Core

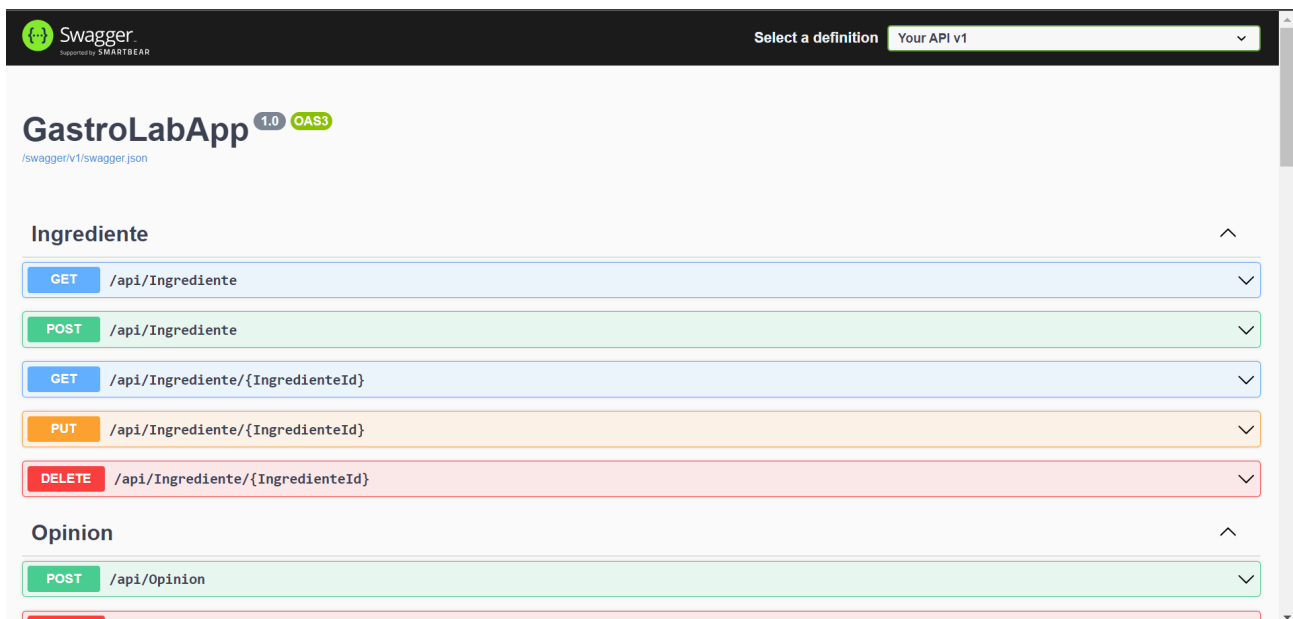
12.0.1

También es importante mencionar los cambios necesarios para el funcionamiento del mapeo y las interfaces desde el appsettings.

```
builder.Services.AddControllers();
builder.Services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies());
builder.Services.AddScoped<IRecetaRepository, RecetaRepository>();
builder.Services.AddScoped<IIngredienteRepository, IngredienteRepository>();
builder.Services.AddScoped<IUsuarioRepository, UsuarioRepository>();
builder.Services.AddScoped<IOpinionRepository, OpinionRepository>();
builder.Services.AddScoped<IValoracionRepository, ValoracionRepository>();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
builder.Services.AddDbContext<DataContext>(options =>
{
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection"));
});

var app = builder.Build();
```

Acabando con la API en .net tan solo nos queda probar los diferentes metodos de los controladores desde swagger para ello vale con ejecutar el proyecto desde el boton de play de la parte superior del entorno de desarrollo.



Pasando ya al proyecto de angular se puede dividir su creación en tres partes para ser más breves la primera es la creación de todos los componentes a utilizar en el proyecto, la segunda es el enrutamiento de todos estos componentes y definir en que dirección estará cada uno de ellos, creación de interfaces para almacenar más cómodamente los objetos traídos desde la base de datos y por último pero de hecho lo más importante el service.ts el cual tiene los metodos que van a establecer conexión con la api mediante uso de http.

Para la creación del proyecto utilice el comando npx desde terminal, ahora una vez finalizado no lo recomiendo ya que viene con demasiadas librerías y ficheros a los que no les daremos uso y por su enorme peso no se puede subir a github al ser mayor de 100MB

es la razón de que solo haya ido subiendo el src con los componentes y todo lo antes mencionado al repositorio.

Lo primero que hice como tal fue crear el service y un componente de prueba para verificar que se pudiera obtener correctamente la información de la base de datos, al usar swagger ya me aseguraba que el camino hasta la API estaba cubierto y que si había algún problema se trataba de la conexión desde angular.

Tuve que hacer algunos cambios en la API ya que me daba errores por el intercambio cruzado o cors, es lo que tiene ser yo el que crea la API, así que tuve que permitir cualquier tipo de intercambio de información entre las cabeceras desde la API.

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace GastroLabApp
{
    1 referencia
    public class Startup
    {
        0 referencias
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        1 referencia
        public IConfiguration Configuration { get; }

        0 referencias
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddCors(options =>
            {
                options.AddDefaultPolicy(builder =>
                {
                    builder.AllowAnyOrigin()
                        .AllowAnyMethod()
                        .AllowAnyHeader();
                });
            });
        }
    }
}
```

Una vez me dejaron de surgir estos errores conseguí acabar una pequeña versión de lo que sería mi service.ts y cree el componente Test-Page que ya se quedo con ese nombre como pagina principal donde mostraría las primeras recetas creadas desde la api y no desde angular ejecutando el proyecto desde terminal mediante ng serve.

```

import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { Usuario } from '../interfaces/usuario.interface';
import { Receta } from '../interfaces/receta.interface';
import { Ingrediente } from '../interfaces/ingrediente.interface';
import { Opinion } from '../interfaces/opinion.interface';
import { Valoracion } from '../interfaces/valoracion.interface';
@Injectable({
  providedIn: 'root'
})
export class apiservice {
  private apiUrl = 'https://localhost:7271/api';

  constructor(private http: HttpClient) {}

  obtenerUsuarios(): Observable<Usuario[]> {
    return this.http.get<Usuario[]>(`${this.apiUrl}/Usuario`);
  }

  obtenerUsuario(UsuarioId:number): Observable<Usuario> {
    return this.http.get<Usuario>(`${this.apiUrl}/Usuario/${UsuarioId}`);
  }

  crearUsuario(usuario: Usuario): Observable<any> {
    return this.http.post(`${this.apiUrl}/Usuario`, usuario);
  }

  obtenerRecetas(): Observable<Receta[]> {
    return this.http.get<Receta[]>(`${this.apiUrl}/Receta`);
  }
}

```

Una vez mostre las primeras recetas quise crear un registro y login de usuario haciendo uso de la tabla usuario y el campo contraseña para esto hice uso de la librería bycriptjs con la que pude encriptar la contraseña en el registro del usuario y compararla mediante el método compare de la librería contra su texto plano en el login por lo que yo nunca llevo a almacenar las contraseñas.

	Id	Nombre	Contraseña	Sexo
1	1	CARLOS12345	\$2a\$10\$OcJWsYPNb2wbKlye4iO.uy.ymezThg4BVLv.XqRXIJA...	Masculino
2	2	Pozo	\$2a\$10\$X4GPdgowNBvC97kJYX8Dv.SI1CCRkVBVcHBc/ZLwAf...	Masculino

Con el register y el login funcionando cree una navbar la cual incluiria en todos los componentes de ahora en adelante mediante la metodologia padre hijo que existe con los componentes de angular en esta navbar muy simple crearia miperfil, ingredientes y recetas donde daría cobertura a la mayoría de métodos de la api.



Cree el componente recetas con un buscador que permite filtrar por nombre y tipo de receta, todos los diseños tras el register y el login los hice responsive a pesar de que soy muy malo maquetando.

Todas las recetas mostradas están almacenadas en cards las cuales al pulsar en ellas te llevan a la pestaña de detalles de la receta que es realmente donde se encuentran la mayoría de características del proyecto.

Recetas

Buscador:

Crear Receta

Borraja con patatas

Tipo: Saludable

Descripción: Patata con borraja cocida



Macarrones con tomate

Tipo: Sencilla

Descripción: Macarrones cocidos con tomate frito



Actualizar Receta

Borrar Receta

Macarrones con tomate

Tipo: Sencilla

Descripción: Macarrones cocidos con tomate frito



Valoración:



Enviar Valoración

Ingredientes:

Nombre: Macarrones

Tipo: Pasta

Calorías: 90

