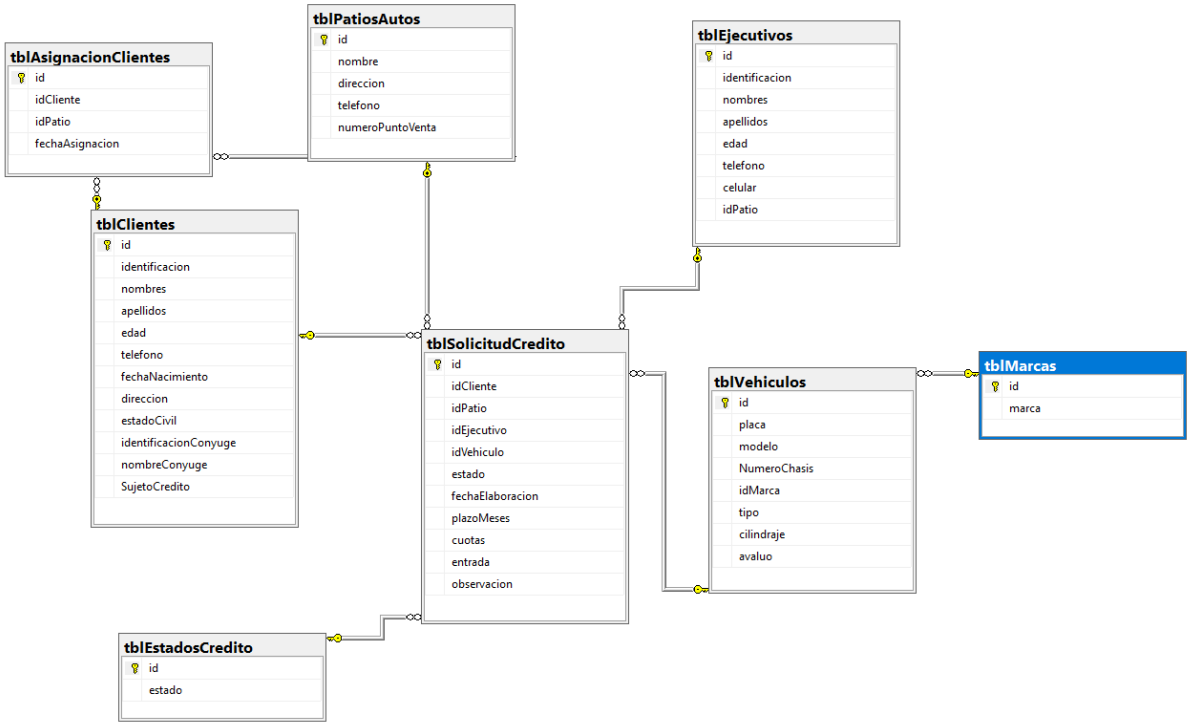
Carlos Alberto Camargo Ochoa
ccamargo@pichincha.com

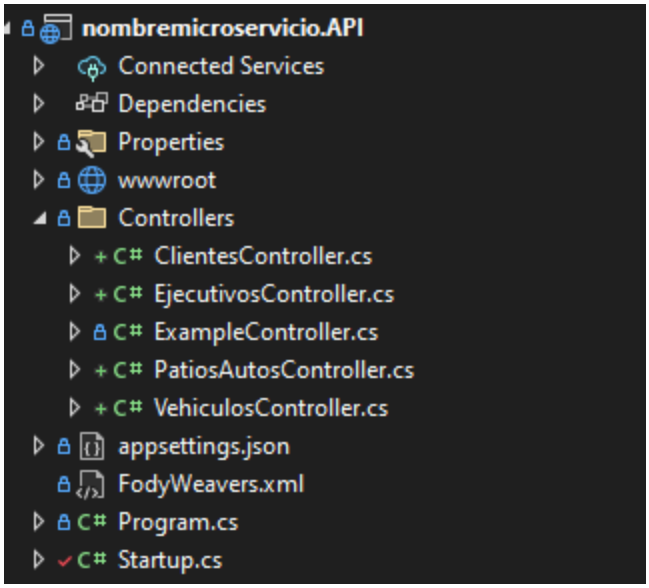
Informe creación de API vehículos

Generación de la estructura de base de datos

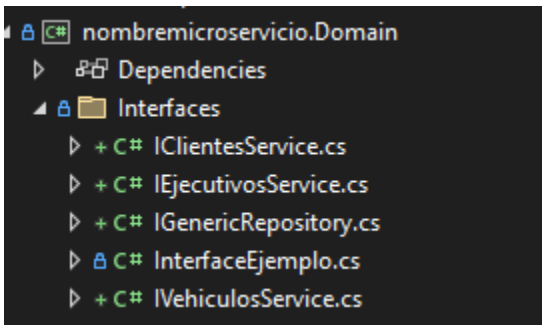
Se realiza la estructura y posterior creación de la base de datos basada en las historias de usuarios cargadas en el tablero de trello que me fue asignado



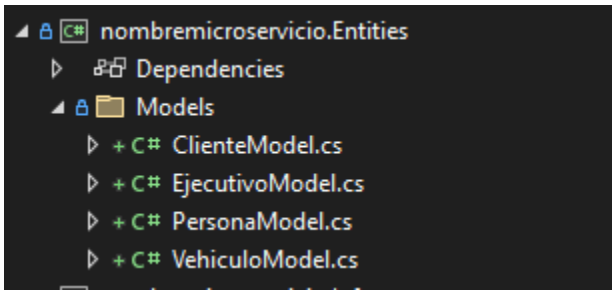
Creación de los endpoints que se van a utilizar (Orientación a realización de los CRUD), para este caso se crean los endpoints de las entidades Clientes, Ejecutivos, Patios y Vehículos.



Creación de las Interfaces que se van a utilizar para inyectar los diferentes servicios de infraestructura, en estas de definen los diferentes métodos que se van a utilizar en el desarrollo de las pruebas



Definición de modelos, se crean las clases que reflejan las entidades creadas en la base de datos.



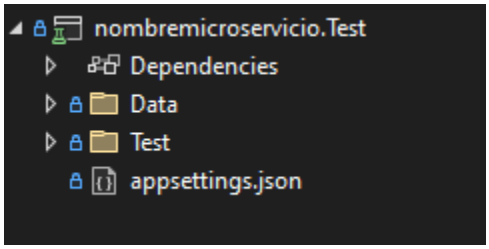
Detalle de los modelos que se trabajarán en la creación de las pruebas

```
public interface IClientes
{
    7 references | 2/2 passing
    ClienteModel Get(int id);
    6 references | 2/2 passing
    IEnumerable<ClienteModel> GetAll();
    7 references | 2/2 passing
    ClienteModel Post(ClienteModel cliente);
    6 references | 2/2 passing
    bool Put(int id, ClienteModel cliente);
    6 references | 2/2 passing
    bool Delete(int id);

}
```

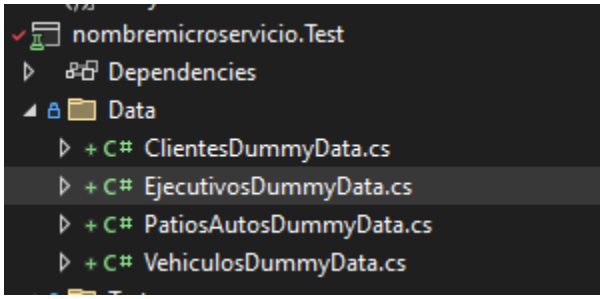
Una vez que se crean los modelos se definen sus atributos y se indica a qué tabla pertenece en la base de datos

```
[Table("tblVehiculos")]
39 references
public class VehiculoModel
{
    7 references
    public int Id { get; set; }
    3 references
    public string Placa { get; set; }
    3 references
    public int Modelo { get; set; }
    3 references
    public string NumeroChasis { get; set; }
    3 references
    public string Marca { get; set; }
    3 references
    public string Tipo { get; set; }
    3 references
    public string Cilindraje { get; set; }
    3 references
    public decimal Avaluo { get; set; }

}
```

Se crea el proyecto para la implementación de las pruebas basado en la librería XUnit, se crean también 2 directorios en los cuales se almacenarán los objetos falsos para probar la implementación y otro en el que se almacenarán las clases de pruebas.



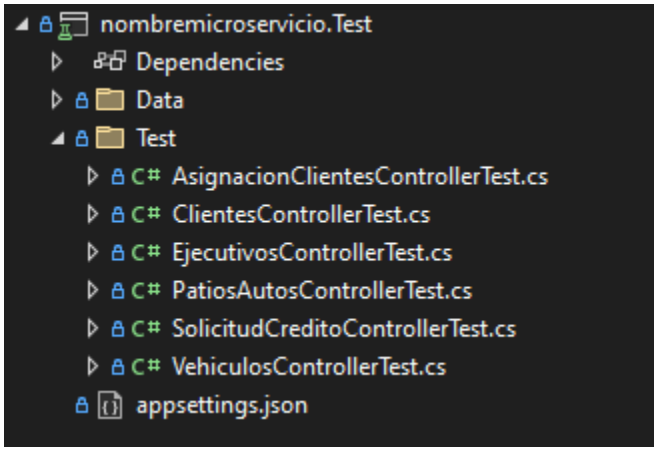
Definición de las clases de datos falsos para generar las pruebas de los verbos en los endpoints.



Ejemplo de datos creados para simular la data

```
public static List<ClienteModel> Clientes = new List<ClienteModel>()
    {
        new ClienteModel()
        {
            Id = 1,
            Identificación = "123456",
            Nombres = "Andres Alberto",
            Apellidos = "Ejemplo1 Ejemplo2",
            Edad = 30,
            Teléfono = "+573003174328",
            FechaNacimiento = "24/08/1991",
            Direccion = "Direccion de ejemplo",
            EstadoCivil = "Soltero",
            IdentificacionConyugue = "00000000000",
            NombreConyugue = "Ejemplo3"


        },
    new ClienteModel()
        {
            Id = 2,
            Identificación = "123456",
            Nombres = "Andres Alberto",
            Apellidos = "Ejemplo1 Ejemplo2",
            Edad = 45,
            Teléfono = "+573003174328",
            FechaNacimiento = "24/08/1991",
            Direccion = "Direccion de ejemplo",
            EstadoCivil = "Soltero",
            IdentificacionConyugue = "00000000000",
            NombreConyugue = "Ejemplo3"


        }
    };
```

Creación de las clases para la definición de las pruebas basadas en los controladores para los endpoints que se crearon previamente.



Para la definición de los objetos como instancias en las clases de pruebas se utilizará la librería “Mock”. La forma en que se nombran los métodos de las pruebas será “nombre método que se prueba” “resultado esperado en la prueba” “Objeto que se espera como resultado”, como ejemplo se utilizará el método “GetAll” del controlador “ClienteController”, del cual se espera una respuesta positiva.

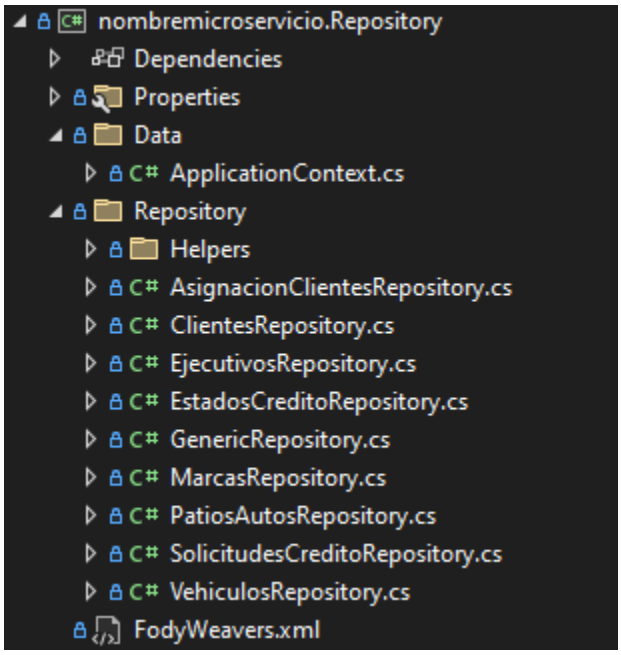
```
[Fact]
0 references
public void GetAll_OnSuccess_ListOfClients()
{
    //Arrange
    var mockService = new Mock<IClientesService>();
    mockService
        .Setup(service => service.GetAll())
        .Returns(ClientesDummyData.Clientes);
    var ObjController = new ClientesController(mockService.Object);
    //Act
    var result = ObjController.GetAll();
    //Assert
    result.Should().BeOfType<OkObjectResult>();
    var objResult = (OkObjectResult)ObjController.GetAll();
    objResult.Value.Should().BeOfType<List<ClienteModel>>();
}
```

Una vez se crea la prueba se continúa con la definición del método en el controlador sin implementación, esto con el objetivo de que se pueda implementar en el desarrollo.

```
4 references | 0/3 passing
public async Task<IActionResult> GetAll()
{
    return null;
}
```

Una vez realizada la creación de los métodos que prueban las funciones del endpoint se desarrolla la capa de repositorio de la cual se entenderá como la librería que se comunicará con la base de datos, para esto se utilizaran las librerías de la herramienta Entity Framework.

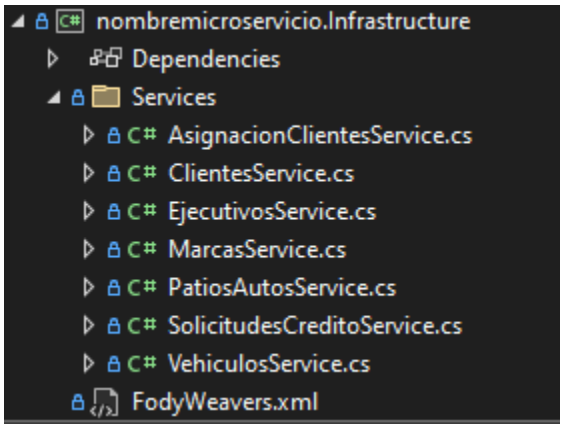
Se crean las carpetas Data y Repository. En la carpeta Data se almacena la librería que realizará la comunicación entre la aplicación y la base de datos.



En las clases de repositorio de crean los métodos de acceso a los datos para su actualización basándose en las interfaces que se crearon previamente.

```
2 references
public class ClientesRepository : IClientes
{
    private readonly IGenericRepository<ClienteModel> dataAdapter;
    0 references
    public ClientesRepository(IGenericRepository<ClienteModel> _data)...
    5 references | 2/2 passing
    public bool Delete(int id)...

    6 references | 2/2 passing
    public ClienteModel Get(int id) => dataAdapter.Get(id);

    5 references | 2/2 passing
    public IEnumerable<ClienteModel> GetAll() => dataAdapter.Get().ToList();

    6 references | 2/2 passing
    public ClienteModel Post(ClienteModel cliente)...

    5 references | 2/2 passing
    public bool Put(int id, ClienteModel cliente)...
}
```

Posteriormente se crea la librería de infraestructura donde se almacenan las clases de servicio donde quedará almacenada la lógica de negocio.



Se crean las clases que almacenan la lógica de negocio independientes por cada entidad de negocio establecida en las demás capas y la base de datos.

```
2 references
public class ClientesService : IClientesService
{
    public readonly IClientes objRepository;
    0 references
    public ClientesService(IClientes obj)...

    2 references
    public void ChargeCsv(string rutaArchivo)...

    5 references | 2/2 passing
    public bool Delete(int id)...

    6 references | 2/2 passing
    public ClienteModel Get(int id)...

    5 references | 2/2 passing
    public IEnumerable<ClienteModel> GetAll()...

    6 references | 2/2 passing
    public ClienteModel Post(ClienteModel data)...

    5 references | 2/2 passing
    public bool Put(int id, ClienteModel data)...
}
```

Finalmente se procede a desarrollar el conjunto de métodos de los endpoints inyectando directamente en el constructor de los controladores las interfaces que se crearon previamente para la lógica de negocio.

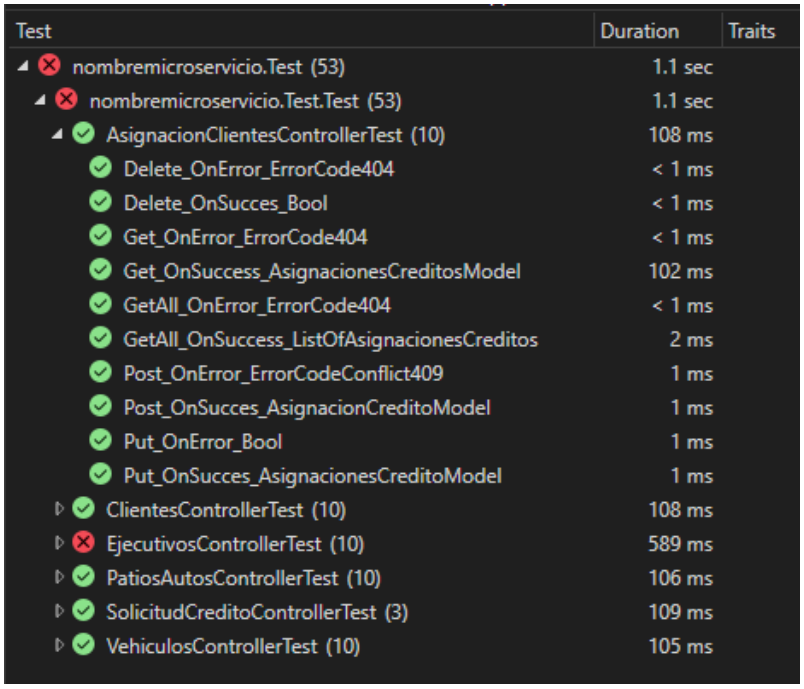
```
private readonly IClientesService _clientesService;

10 references | 10/10 passing
public ClientesController(IClientesService clientesService)
{
    _clientesService = clientesService;
    _clientesService.ChargeCsv(@"../nombremicroservicio.API/CsvFiles/Clientes.csv");
}
```

```
private readonly IClientesService _clientesService;

10 references | 10/10 passing
public ClientesController(IClientesService clientesService)...

3 references | 2/2 passing
public IActionResult Get(int id)...

[HttpGet]
3 references | 2/2 passing
public IActionResult GetAll()...

[HttpPost]
3 references | 2/2 passing
public IActionResult Post([FromBody] ClienteModel data)...
[HttpPut("{id}")]
3 references | 2/2 passing
public IActionResult Put(int id, [FromBody] ClienteModel data)...

[HttpDelete("{id}")]
2 references | 2/2 passing
public IActionResult Delete(int id)...
```

Se realiza la configuración de las interfaces de inyección en el archivo de configuración de la aplicación web.

```
services.AddScoped(typeof(IGenericRepository<>), typeof(GenericRepository<>));
services.AddScoped(typeof(IClientes), typeof(ClientesRepository));
services.AddScoped(typeof(IClientesService), typeof(ClientesService));
services.AddScoped(typeof(IVehiculos), typeof(VehiculosRepository));
services.AddScoped(typeof(IVehiculosService), typeof(VehiculosService));
services.AddScoped(typeof(IPatiosAutos), typeof(PatiosAutosRepository));
services.AddScoped(typeof(IPatiosAutosService), typeof(PatiosAutosService));
services.AddScoped(typeof(ISolicitudesCredito), typeof(SolicitudesCreditoRepository));
services.AddScoped(typeof(ISolicitudesCreditoService), typeof(SolicitudesCreditoService));
services.AddScoped(typeof(IAsignacionClientes), typeof(AsignacionClientesRepository));
services.AddScoped(typeof(IAsignacionClientesService), typeof(AsignacionClientesService));
```

Se realiza la depuración de las pruebas y se comprueba la cobertura de las pruebas en el proyecto.


Test
Duration
Traits
nombremicroservicio.Test (53) 1.1 sec
nombremicroservicio.Test.Test (53) 1.1 sec
AsignacionClientesControllerTest (10) 108 ms
Delete_OnError_ErrorCode404 < 1 ms
Delete_OnSucces_Bool < 1 ms
Get_OnError_ErrorCode404 < 1 ms
Get_OnSuccess_AsignacionesCreditosModel 102 ms
GetAll_OnError_ErrorCode404 < 1 ms
GetAll_OnSuccess_ListOfAsignacionesCreditos 2 ms
Post_OnError_ErrorCodeConflict409 1 ms
Post_OnSucces_AsignacionCreditoModel 1 ms
Put_OnError_Bool 1 ms
Put_OnSucces_AsignacionesCreditoModel 1 ms
ClientesControllerTest (10) 108 ms
EjecutivosControllerTest (10) 589 ms
PatiosAutosControllerTest (10) 106 ms
SolicitudCreditoControllerTest (3) 109 ms
VehiculosControllerTest (10) 105 ms