

Pontificia Universidad
JAVERIANA
Colombia

Laboratorio 1

Arquitectura de Software

Autores:

Juan Felipe González Quintero
Oscar Alejandro Rodríguez Gómez
Andrés Felipe Ruge Passito

PONTIFICIA UNIVERSIDAD JAVERIANA
FACULTAD DE INGENIERÍA
PROGRAMA DE INGENIERÍA DE SISTEMAS
BOGOTÁ, D.C.

9 de mayo de 2025

Contenido

1.	Introducción	3
1.1.	Arquitectura Monolítica.....	3
1.2.	Patrón arquitectónico MVC	3
1.3.	Patrón arquitectónico DAO.....	4
1.4.	Visual Studio	4
1.5.	Microsoft SQL Server.....	4
1.6.	Swagger 3.....	5
2.	Modelado de la aplicación	5
2.1.	Diagrama de alto nivel.....	5
2.2.	Diagrama de contexto	6
2.3.	Diagrama de contenedores	7
2.4.	Diagrama de componentes	8
2.5.	Diagrama de Despliegue	8
3.	Procedimiento	9
3.1.	Instalaciones	9
3.2.	Configuración SQL Server.....	9
3.3.	Implementación	10
4.	Conclusiones	27
5.	Referencias.....	27

1. Introducción

Para este laboratorio se plantea la implementación de una aplicación con estilo arquitectónico monolítico haciendo uso de los patrones MVC y DAO. El lenguaje seleccionado para la codificación será ASP.NET y las herramientas Visual Studio (Community 2022) y SQL Server 2019 Express. El objetivo del laboratorio es ampliar nuestro conocimiento en base a diferentes tecnologías, lenguajes y estilos arquitectónicos complementando el aprendizaje que hemos tenido en la materia hasta el momento.

La aplicación se compone de cuatro entidades (profesión, estudios, persona y teléfono), de las cuáles se debe realizar el CRUD y luego probar los endpoints correspondientes. A continuación, se explican a detalle los componentes del stack tecnológico que integra la aplicación.

1.1. Arquitectura Monolítica

Una arquitectura monolítica es un enfoque de diseño arquitectónico y desarrollo de software tradicional centrado en que la aplicación se compile como una unidad unificada e independiente de otras aplicaciones. En la estructura de una aplicación monolítica, todos los componentes de software son interdependientes debido a la forma que tienen de comunicarse los datos dentro del sistema.

La arquitectura monolítica es ideal para implementar proyectos pequeños como startups, aplicaciones internas en organizaciones o para aplicativos con requerimientos simples en escalado. Sin embargo, no se recomienda aplicar este estilo arquitectónico en aplicaciones grandes ya que una de las dificultades esenciales de implementar un monolito es su complejidad de modificación, pues los desarrolladores necesitarían conocer a la perfección los componentes y las interconexiones entre los mismos, llevando a que la escalabilidad del proyecto sea muy difícil de implementar.

1.2. Patrón arquitectónico MVC

El MVC es un patrón arquitectónico que a través de la separación en 3 componentes (Modelo, Vista y Controlador) separa la lógica y la vista de una aplicación. La ventaja principal de aplicar este patrón es que, al separar los diferentes componentes dependiendo de la responsabilidad que se les asigna, cuando se hace un cambio en alguno de los componentes los demás no se ven afectados, lo que nos permite tener un mayor control y facilidad de implementación a la hora de realizar ajustes.

Los componentes del MVC se encargan de diferentes responsabilidades que están diferenciadas de la siguiente forma:

Modelo: Representa los datos y las reglas del negocio de la aplicación. Su tarea más común es manejar la lógica de negocio y el acceso a datos.

Vista: Representa la interfaz gráfica o presentación. Su responsabilidad es mostrar los datos al usuario que está haciendo uso de la aplicación.

Controlador: Recibe las solicitudes del usuario, las procesa, interactúa con el modelo y actualiza la vista. Actúa como puente entre la Vista y el Modelo.

1.3. Patrón arquitectónico DAO

El patrón de arquitectura DAO (Data Access Object) se encarga de abstraer y encapsular el acceso a los datos de una aplicación, separando esta responsabilidad de la lógica de negocio. Este patrón define una interfaz común para interactuar con la fuente de datos (como una base de datos SQL, NoSQL o incluso una API), y su implementación contiene el código específico necesario para realizar operaciones CRUD sobre una entidad.

La ventaja de utilizar este patrón arquitectónico el resto de la aplicación no necesita saber cómo se acceden o almacenan los datos, mejorando el mantenimiento y modularidad del código.

1.4. Visual Studio

Visual Studio es un entorno de desarrollo integrado (IDE) creado por Microsoft, diseñado para facilitar la creación de aplicaciones de escritorio, móviles, web y servicios en la nube. Soporta múltiples lenguajes de programación como C#, VB.NET, C++, JavaScript, Python, entre otros, y ofrece herramientas avanzadas para escribir, depurar, compilar y probar código en un solo lugar. Está especialmente optimizado para el desarrollo en .NET y se integra bien con servicios como Azure y bases de datos SQL Server.

Visual Studio incluye características como autocompletado inteligente (IntelliSense), depuración paso a paso, diseño visual de interfaces, control de versiones (integración con Git), y pruebas automatizadas. Además, cuenta con una amplia comunidad y una galería de extensiones que permite personalizar el entorno según las necesidades del proyecto. Es ampliamente utilizado tanto por desarrolladores individuales como por grandes equipos en empresas que trabajan con tecnologías de Microsoft.

1.5. Microsoft SQL Server

Microsoft SQL Server es un sistema de gestión de bases de datos relacional (RDBMS) desarrollado por Microsoft, diseñado para almacenar, organizar y acceder a grandes cantidades de datos de forma eficiente. Utiliza el lenguaje SQL (Structured Query Language) para realizar consultas, insertar, actualizar o eliminar información en las bases de datos. Está pensado para aplicaciones empresariales que requieren alta disponibilidad, seguridad y rendimiento, y puede ejecutarse tanto en servidores locales como en la nube a través de Azure SQL.

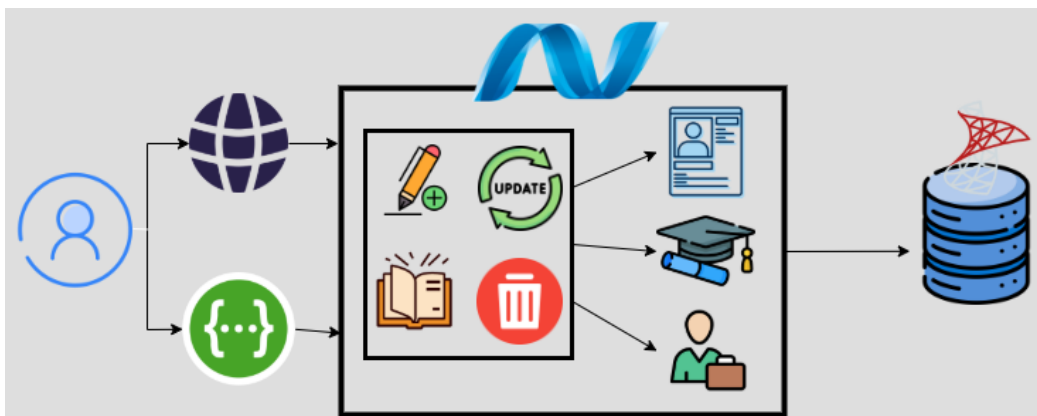
SQL Server ofrece herramientas integradas para administración, análisis de datos, replicación, generación de informes y respaldo automático. También proporciona soporte para transacciones, procedimientos almacenados, integridad referencial y funciones avanzadas como inteligencia empresarial (BI), procesamiento analítico en línea (OLAP) y análisis en tiempo real. Es ampliamente utilizado en entornos corporativos gracias a su escalabilidad, robustez y facilidad de integración con otros productos de Microsoft.

1.6. Swagger 3

Swagger es una herramienta de código abierto utilizada para documentar, diseñar y probar APIs RESTful de forma interactiva. Su núcleo es el uso de un formato estándar que describe los endpoints, parámetros, respuestas, y otros detalles de una API en un archivo legible tanto por humanos como por máquinas (generalmente en formato JSON o YAML). Swagger genera automáticamente una interfaz web que permite explorar y probar las operaciones de la API sin necesidad de escribir código adicional o usar herramientas externas como Postman.

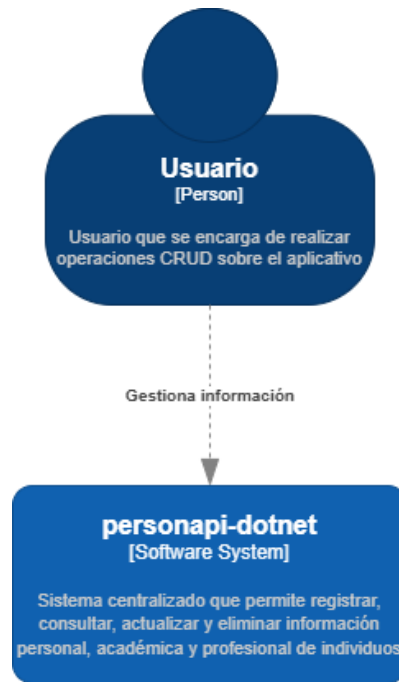
2. Modelado de la aplicación

2.1. Diagrama de alto nivel



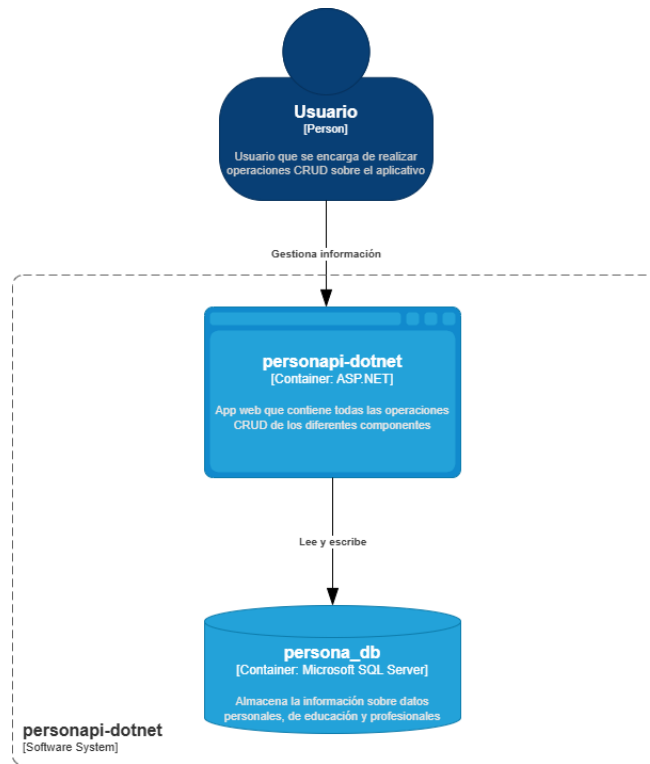
En este diagrama podemos ver como el usuario puede acceder a la aplicación desarrollada en ASP.NET directamente desde su navegador web o desde swagger, ambas le permitirán realizar operaciones CRUD sobre datos personales, de educación o profesionales los cuales se consultan y guardan en una base de datos de Microsoft SQL Server.

2.2. Diagrama de contexto



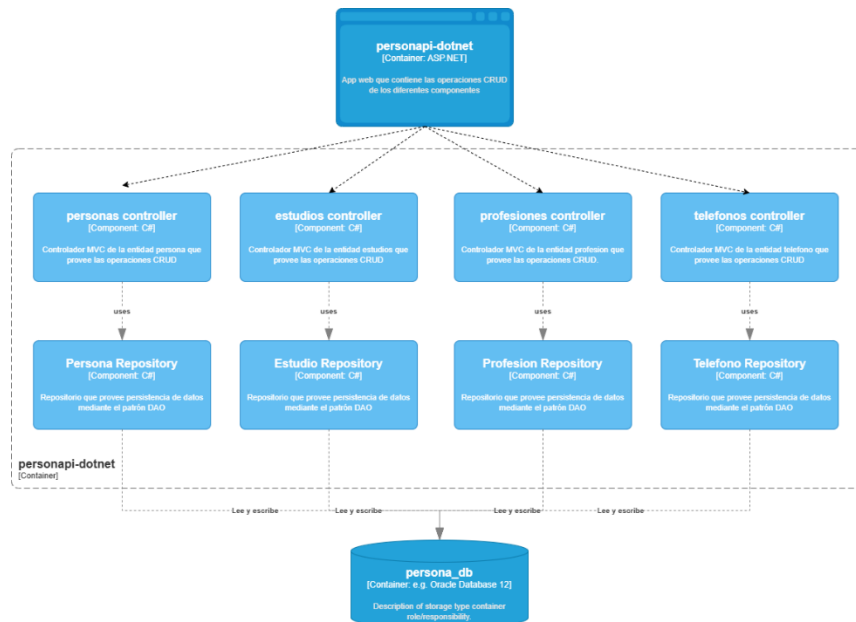
En el diagrama de contexto encontramos información acerca de cómo el “Usuario” que representa a cualquier persona que acceda a la aplicación, puede gestionar información a través de “personapi-dotnet”, que es el nombre que se le da al aplicativo web que contiene los diferentes endpoints que permiten realizar las operaciones CRUD de acuerdo con el modelado de datos presentado.

2.3. Diagrama de contenedores



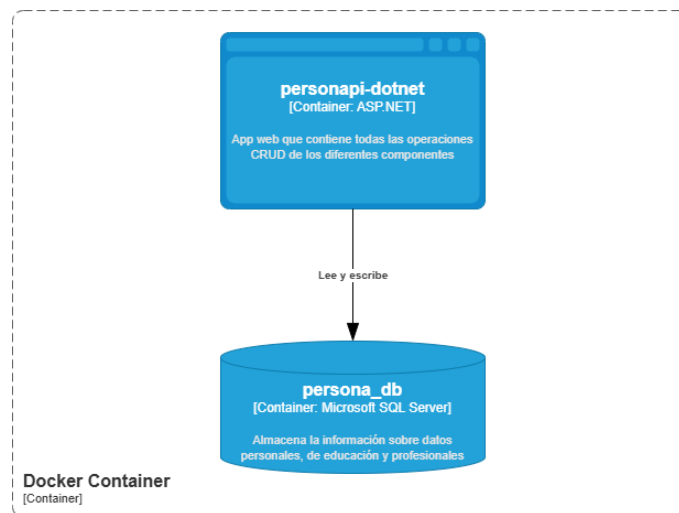
En este diagrama observamos como el usuario interactúa con la aplicación a través de un navegador web. La aplicación le permite al usuario realizar operaciones CRUD de datos personales, de educación y profesionales que se almacenan en la base de datos “persona_db” de Microsoft SQL Server 2019 Express.

2.4. Diagrama de componentes



Este diagrama representa la arquitectura de una aplicación monolítica ASP.NET llamada **personapi-dotnet**, organizada bajo el patrón MVC. Cada entidad del dominio (persona, estudio, profesión, teléfono) tiene su propio controlador que gestiona las operaciones CRUD y se comunica directamente con su respectivo repositorio. Los **repositories** encapsulan la lógica de acceso a datos y son responsables de leer y escribir en la base de datos **persona_db**, eliminando la necesidad de una capa DAO explícita.

2.5. Diagrama de Despliegue



En este diagrama vemos como la aplicación y base de datos están en un mismo contenedor, que para este caso es de Docker cumpliendo la característica principal de una

aplicación monolítica que tiene todos sus componentes interconectados en el mismo espacio físico.

3. Procedimiento

3.1. Instalaciones

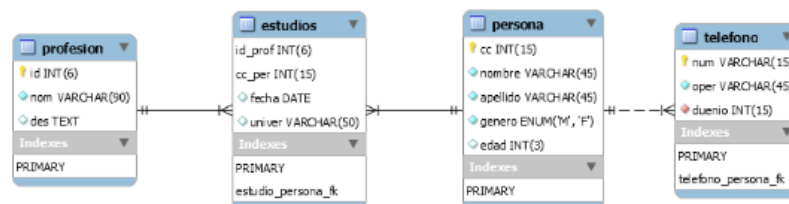
Inicialmente, se procedió con la instalación del sistema gestor de bases de datos [Microsoft SQL Server 2019 Express](#), una edición gratuita y liviana que ofrece las funcionalidades esenciales para desarrollo y pruebas de aplicaciones. Su uso es ampliamente útil debido a su compatibilidad con entornos .NET y su facilidad de integración con herramientas de desarrollo de Microsoft. Paralelamente, se utilizó [Visual Studio 2022](#), una de las versiones más recientes del entorno de desarrollo integrado (IDE) de Microsoft, que ofrece soporte completo para proyectos en .NET 6/7, herramientas avanzadas de depuración, integración con control de versiones (como Git), y una interfaz amigable para el diseño y construcción de aplicaciones monolíticas basadas en el patrón MVC y DAO.

Por último, se procedió con la instalación de [SQL Server Management Studio \(SSMS\) versión 20](#), la cual ofrece una interfaz moderna y mejorada para la administración de bases de datos SQL Server. Esta versión incluye mejoras en la estabilidad y compatibilidad con las versiones más recientes de SQL Server.

La decisión de utilizar SSMS 20 se debió a que la versión 18 presentaba problemas de funcionamiento. En particular, al intentar abrir una nueva consulta, la aplicación se cerraba inesperadamente. Este comportamiento se ha documentado en versiones como la 18.6 y 18.12.1, donde al hacer clic en "Nueva consulta", SSMS se bloquea y se reinicia. Estos problemas parecen estar relacionados con incompatibilidades con versiones de Windows o conflictos con componentes del .NET Framework [1].

3.2. Configuración SQL Server

Para preparar el entorno de desarrollo lo primero que se realizó fue crear la base de datos llamada `persona_db`, la cual se construyó con un script basado en el siguiente modelo de datos:



```

-- Creación de la base de datos
CREATE DATABASE persona_db;
GO

USE persona_db;
GO

-- Tabla persona
CREATE TABLE persona (
    cc INT NOT NULL PRIMARY KEY,
    nombre VARCHAR(45) NOT NULL,
    apellido VARCHAR(45) NOT NULL,
    genero CHAR(1) CHECK (genero IN ('M', 'F')) NOT NULL,
    edad INT NULL
);
GO

-- Tabla profesion
CREATE TABLE profesion (
    id INT NOT NULL PRIMARY KEY,
    nom VARCHAR(90) NOT NULL,
    des TEXT NULL
);
GO

-- Tabla estudios
CREATE TABLE estudios (
    id_prof INT NOT NULL,
    cc_per INT NOT NULL,
    fecha DATE NULL,
    univer VARCHAR(50) NULL,
    PRIMARY KEY (id_prof, cc_per),
    FOREIGN KEY (cc_per) REFERENCES persona(cc),
    FOREIGN KEY (id_prof) REFERENCES profesion(id)
);
GO

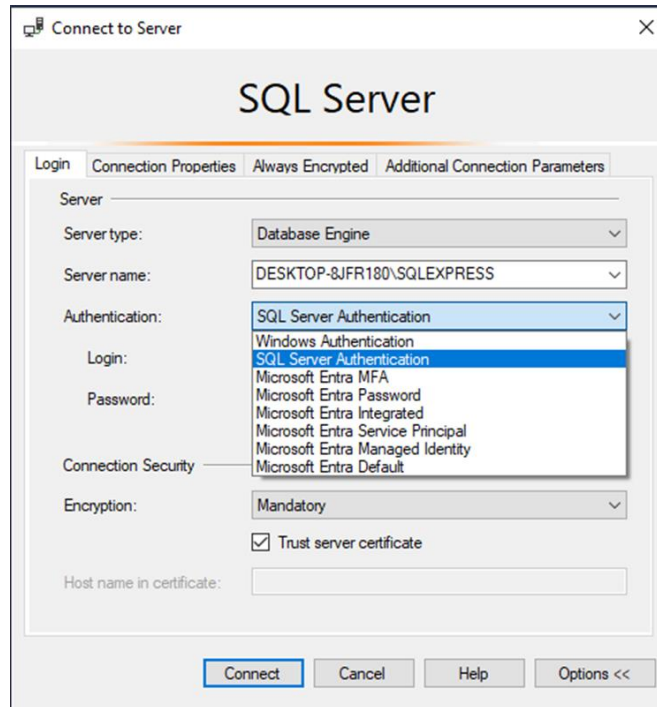
-- Tabla telefono
CREATE TABLE telefono (
    num VARCHAR(15) NOT NULL PRIMARY KEY,
    oper VARCHAR(45) NOT NULL,
    duenio INT NOT NULL,
    FOREIGN KEY (duenio) REFERENCES persona(cc)
);
GO

```

El modelo destaca principalmente por las relaciones que conectan las entidades: una relación uno a muchos entre persona y teléfono (una persona puede tener varios teléfonos) y una relación muchos a muchos entre persona y profesión, resuelta mediante la tabla intermedia estudios (una persona puede haber estudiado varias profesiones y una profesión puede haber sido estudiada por varias personas). Estas relaciones están definidas mediante claves foráneas que aseguran la integridad referencial, lo que permite representar de forma estructurada y coherente los datos personales, académicos y de contacto de los individuos.

3.3. Implementación

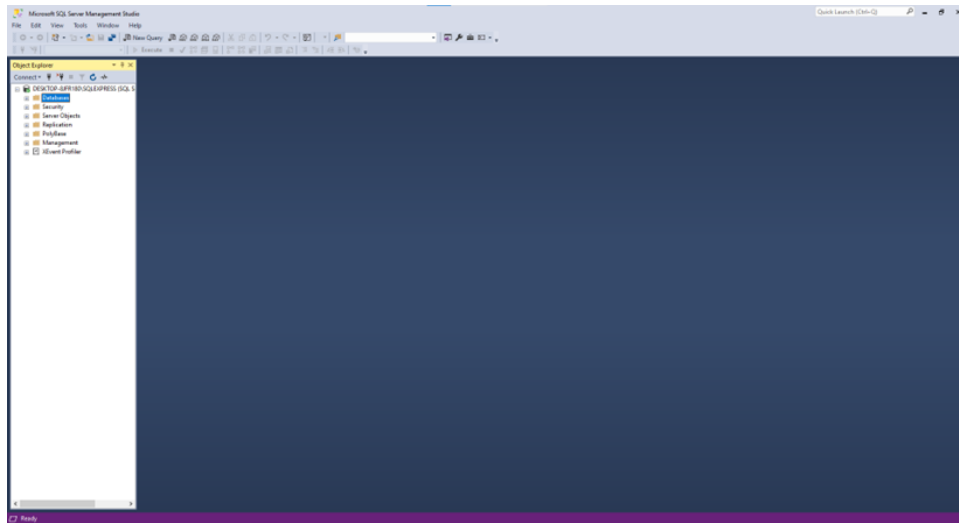
Lo primero que haremos será abrir el **SQL Server Management Studio**.



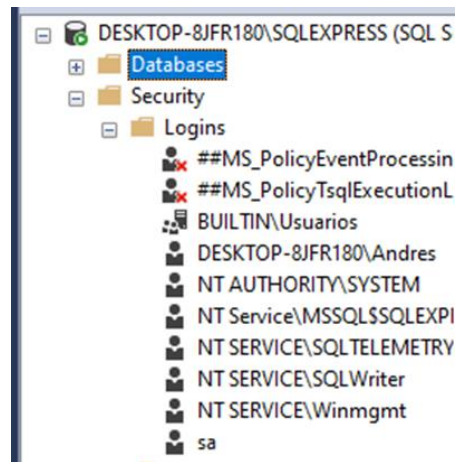
Al abrirlo encontraremos lo que se ve en la imagen, lo primero que se debe hacer es cambiar en **Authentication** la opción de “Windows Authentication” por “SQL Server Authentication”.

Luego en **Login** poner el usuario “sa”, (se verá más adelante en el informe por qué se coloca esto) y en la sección **Password** poner una contraseña cualquiera que deseemos.

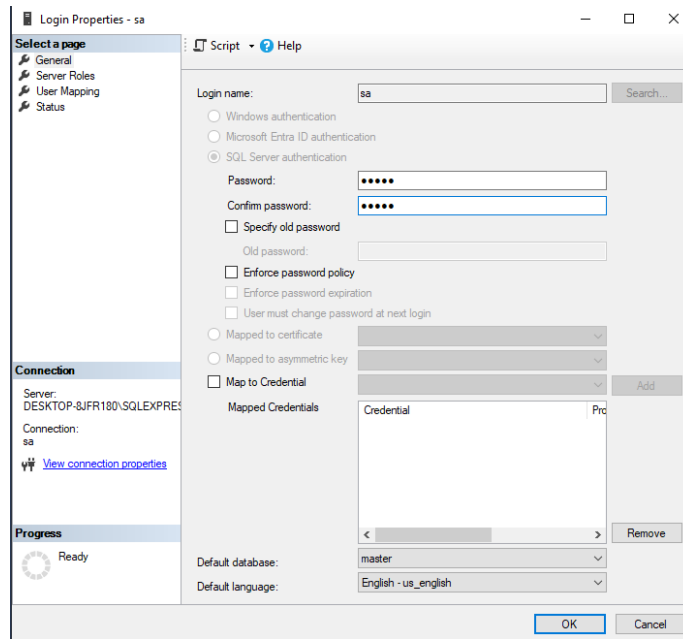
Posteriormente veremos la siguiente interfaz:



En la parte de **Object Explorer** veremos varias carpetas:

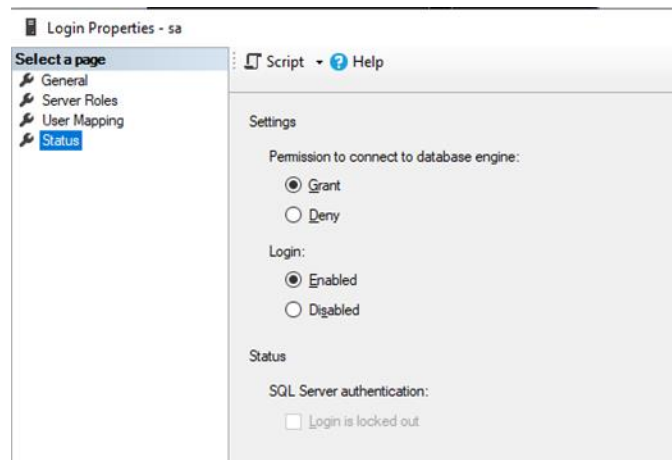


En la carpeta **Security/Logins** encontrara el usuario sa, hacer clic derecho y seleccionar “properties”.



Al abrir se ve la siguiente pantalla, en **Password** poner la contraseña que escogio anteriormente y en **Confirm Password** la misma.

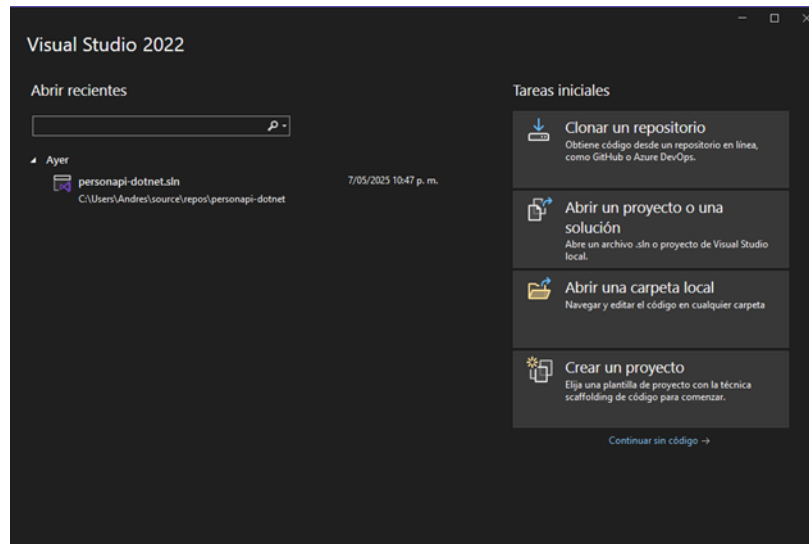
Luego en la parte izquierda en **Select a page**, seleccionar el apartado de “Status”.



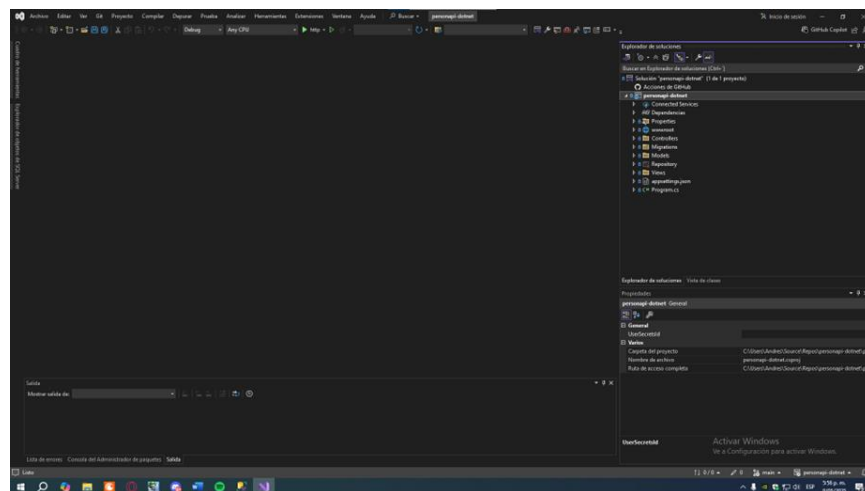
Le damos “Grant” en **Permission to connect to database engine:** y le damos “Enabled” en **Login**. Le damos ok en la parte de inferior derecha y se cerrara la pestaña.

1. Visual Studio

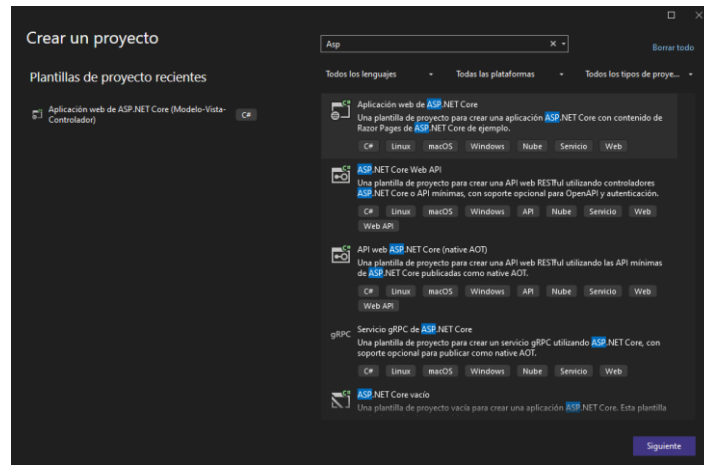
Ahora toca abrir **Visual Studio**, se verá lo siguiente:



Oprimir en **clonar un repositorio** y pedirá la ubicación del repositorio git, allí pondremos la url del repositorio y luego dar clic en clonar. Esto abrirá la siguiente pantalla:



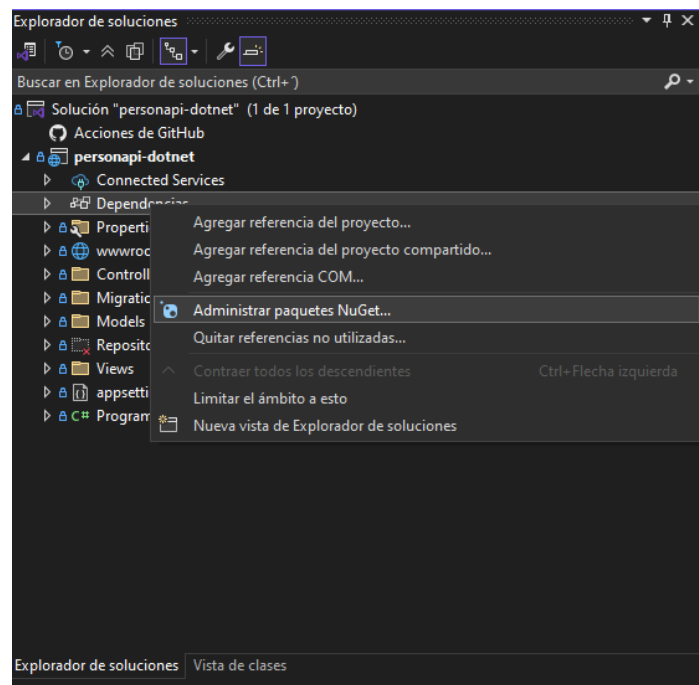
En la sección de **Archivo** oprimir en **Nuevo -> Crear Proyecto**



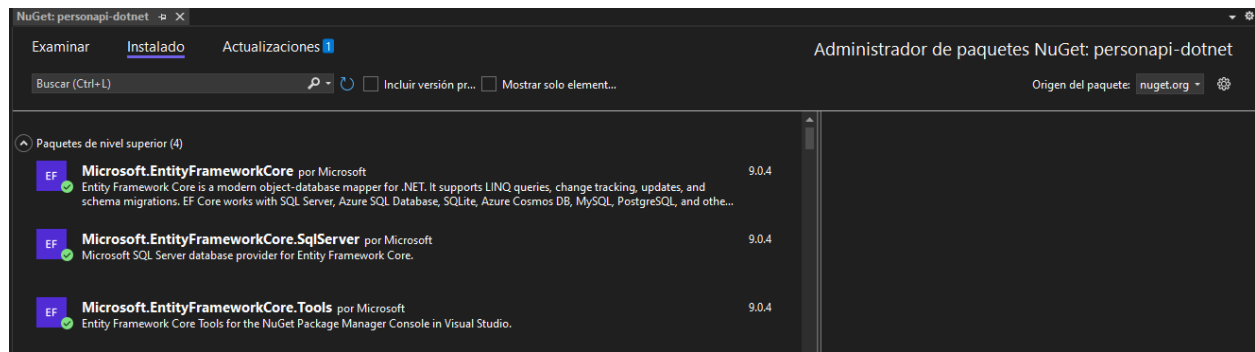
Buscar la plantilla “Aplicación web de ASP.NET Core (Modelo-Vista-Controlador)”, seleccionarla y oprimir en “siguiente”.

Luego, se nombra el proyecto y darle en “continuar”, en este caso, lo nombramos **personapi-dotnet** como se solicita en la guía de laboratorio.

En la pantalla anterior dirigirse al **Explorador de Soluciones** que está en la parte superior derecha:



Hacemos clic derecho en las **Dependencias** y seleccionar “Administrar paquetes NuGet”:

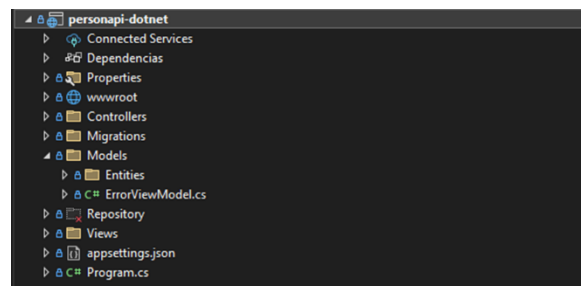


En la sección **Examinar**, haciendo uso de la barra de búsqueda, buscar los siguientes paquetes para instalarlos:

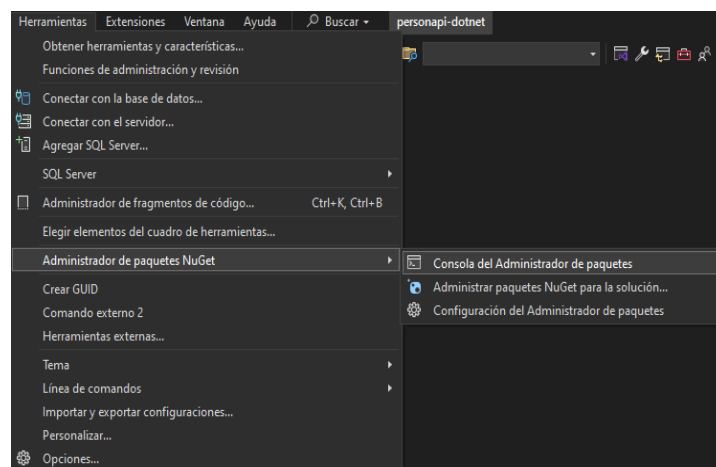
- Microsoft.EntityFrameworkCore
- Microsoft.EntityFrameworkCore.SqlServer
- Microsoft.EntityFrameworkCore.Tools

Como se ve en la imagen anterior, ya hemos realizado la instalación de dichos paquetes.

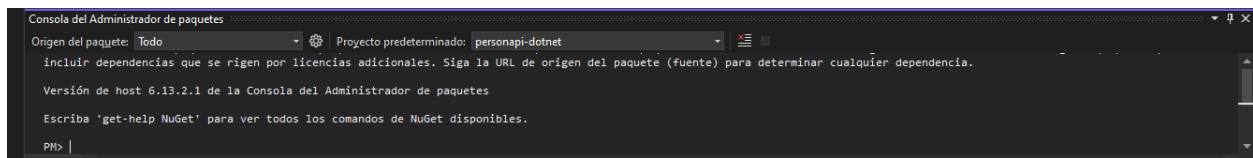
Volvemos al **Explorador de soluciones** y vamos a crear una carpeta llamada **Entities** dentro de la carpeta de **Models** como se muestra en la siguiente imagen:



Ahora en el menú de arriba en la sección de Herramientas seleccionar la opción “Consola de Administrador de paquetes”



En la parte inferior se desplegará el componente anteriormente mencionado de la siguiente forma:

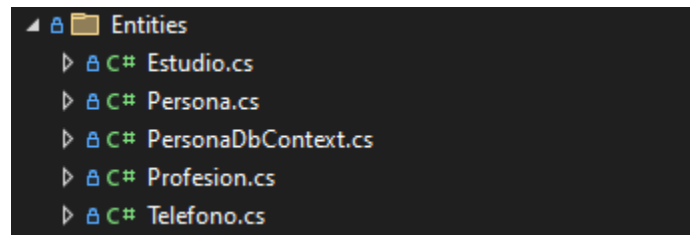


Luego, en la Consola del Administrador de paquetes escribir lo siguiente:

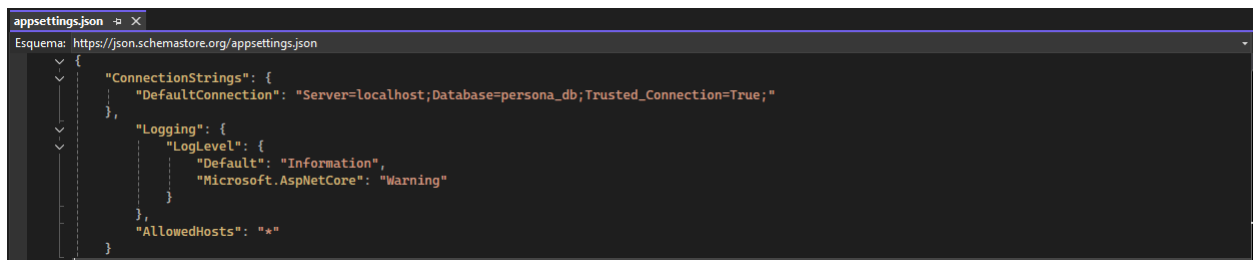
Scaffold-DbContext

```
"Server=localhost\SQLEXPRESS;Database=persona_db;Trusted_Connection=True;TrustServerCertificate=true" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models/Entities
```

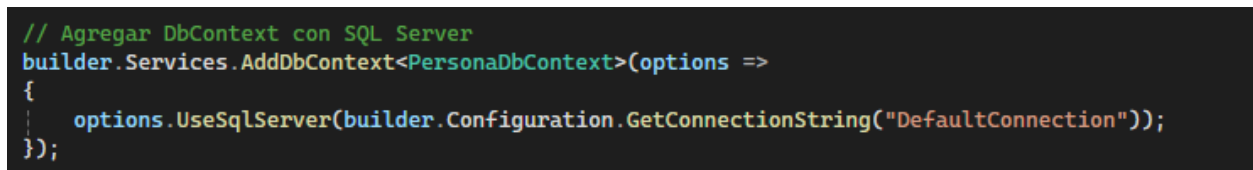
Con este comando se crean las clases entidad a partir de las tablas existentes de la base de datos y el contexto como se observa en la siguiente imagen:



Ahora en el archivo “**appsettings.json**” que se encuentra dentro de nuestro proyecto de **personaapi-dotnet** vamos a agregar la cadena de conexión, para ello abrimos el appsettings.json y en **ConnectionStrings** en el apartado de **DefaultConnection** agregaremos el servidor en el que se encuentra nuestra database y el nombre de la database que creamos como se ve en la siguiente imagen.

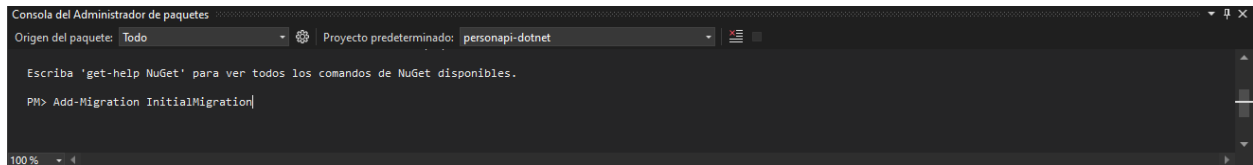


Después de ello configuraremos el **Program.cs** para obtener la cadena de conexión con el siguiente fragmento de código:



Aquí hacemos uso del paquete EF para SQL Server, y obtenemos la cadena de conexión del **appsettings.json** para poder conectarnos a nuestro servidor y poder crear y posteriormente interactuar con la base de datos.

Ahora nuevamente dirigirse a la consola de paquetes y ejecutar el comando **Add-Migration InitialMigration**

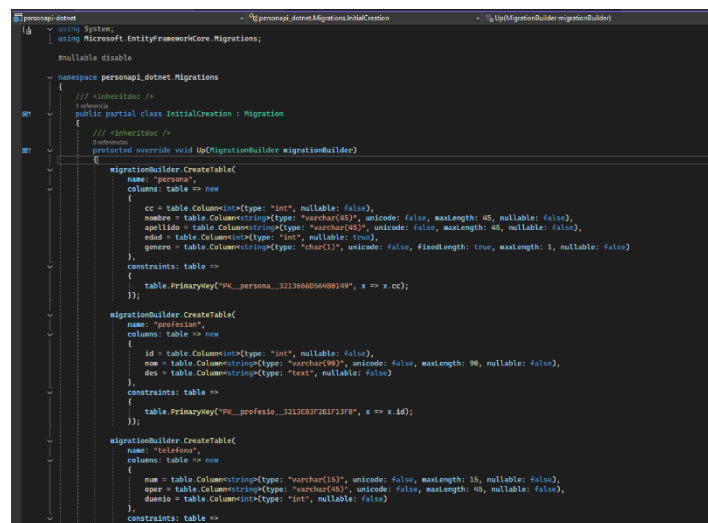
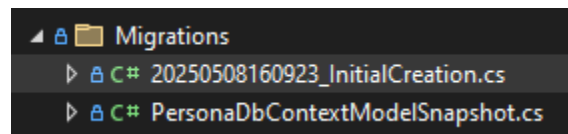


```
Consola del Administrador de paquetes
Origen del paquete: Todo
Proyecto predeterminado: personapi-dotnet

Escriba 'get-help NuGet' para ver todos los comandos de NuGet disponibles.

PM> Add-Migration InitialMigration
```

Lo que realizara este comando es crear una carpeta que tiene un archivo con el código necesario para crear las tablas y relaciones en la base de datos. guiándose del modelo de datos actual. En las siguientes imagenes se puede ver el archivo generado.



```
using System;
using Microsoft.EntityFrameworkCore.Migrations;

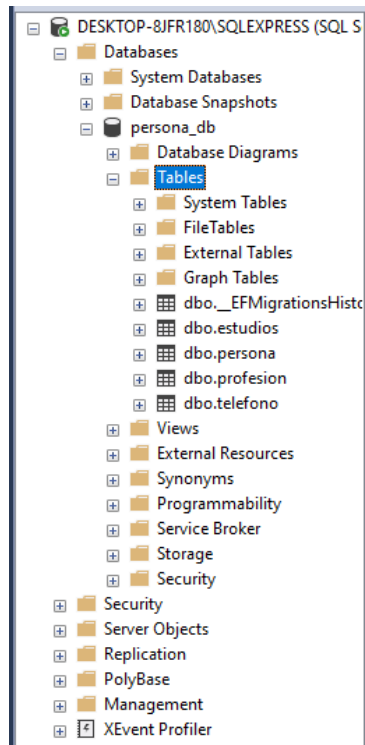
#nullable disable

namespace personapi_dotnet.Migrations
{
    /// <summary>
    ///
    /// </summary>
    public partial class InitialCreation : Migration
    {
        /// <summary>
        ///
        /// </summary>
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "persona",
                columns: table => new
                {
                    cc = table.Column<int>(type: "int", nullable: false),
                    nombre = table.Column<string>(type: "varchar(40)", unicode: false, maxLength: 40, nullable: false),
                    apellido = table.Column<string>(type: "varchar(40)", unicode: false, maxLength: 40, nullable: false),
                    edad = table.Column<int>(type: "int", nullable: true),
                    genero = table.Column<string>(type: "char(1)", unicode: false, fixedLength: true, maxLength: 1, nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_persona_3213666050408149", x => x.cc);
                });

            migrationBuilder.CreateTable(
                name: "profesion",
                columns: table => new
                {
                    id = table.Column<int>(type: "int", nullable: false),
                    nom = table.Column<string>(type: "varchar(90)", unicode: false, maxLength: 90, nullable: false),
                    des = table.Column<string>(type: "text", nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_profesia_3212083720171370", x => x.id);
                });

            migrationBuilder.CreateTable(
                name: "telefono",
                columns: table => new
                {
                    num = table.Column<string>(type: "varchar(10)", unicode: false, maxLength: 10, nullable: false),
                    oper = table.Column<string>(type: "varchar(40)", unicode: false, maxLength: 40, nullable: false),
                    dueño = table.Column<int>(type: "int", nullable: false)
                },
                constraints: table =>
```

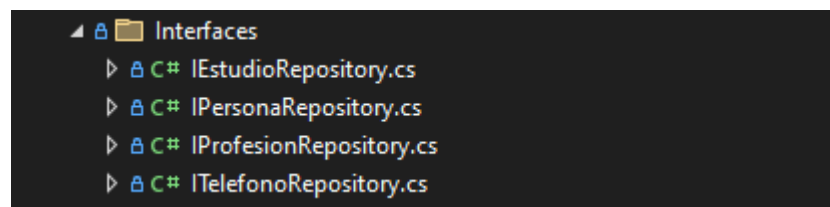
Posteriormente ejecutaremos **Update-Database** para enviar todos estos cambios a la base de datos, si todo está correcto al ir a **SQL Server Management Studio** se puede verificar que se creó la base de datos con sus tablas.



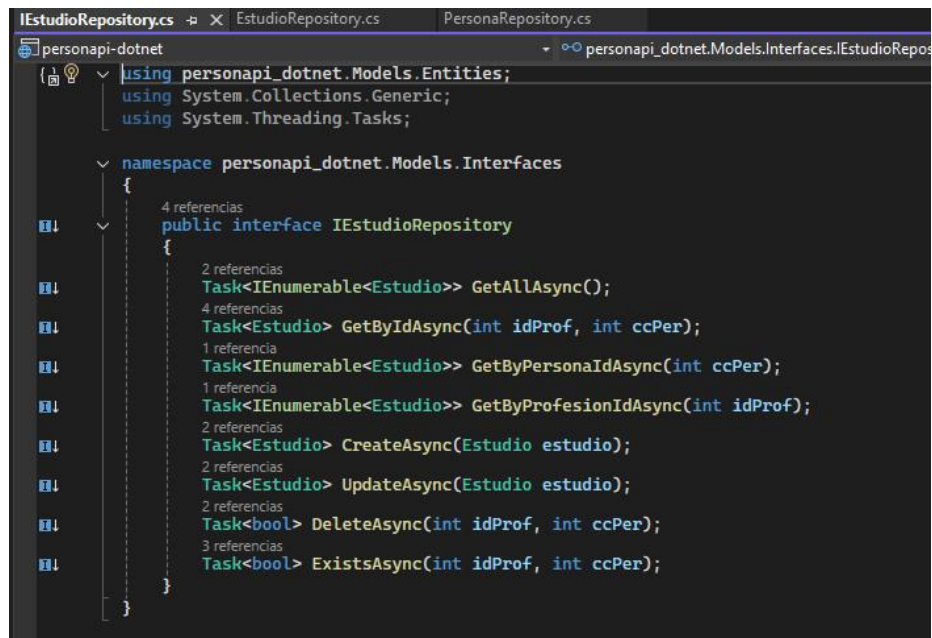
Como se observa se creó correctamente la base de datos y sus tablas.

Posterior a esto ya se puede empezar a realizar los **métodos CRUD** de la API, se realizarán repositorios, interfaces y controladores.

Para las interfaces se creará una carpeta llamada **Interfaces** dentro de la carpeta **Models** y se crearan los métodos que tendrá el CRUD para cada una de las entidades.



En la siguiente imagen podremos ver un ejemplo de implementación de interfaz.



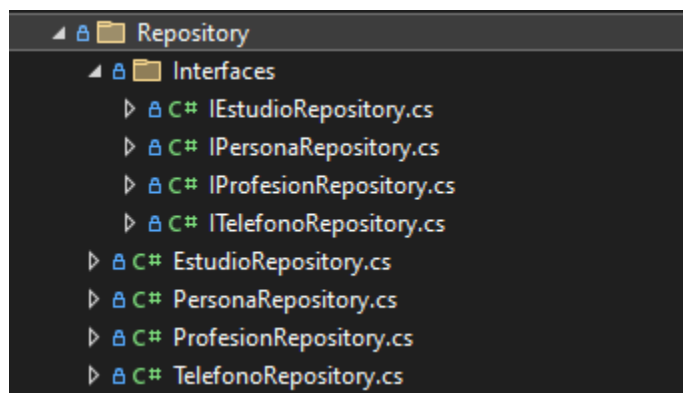
Para los repositorios se creará una carpeta llamada **Repositories** dentro de la carpeta **Models**

En esta se implementaron **cuatro repositorios**, uno por cada entidad principal: Persona, Teléfono, Profesión y Estudios. Estos repositorios siguen el patrón **DAO** y permiten realizar las operaciones CRUD sobre cada entidad.

Cada repositorio está compuesto por:

- Una **interfaz** con los métodos CRUD (IPersonaRepository, etc.).
- Una **clase de implementación** que usa Entity Framework para acceder a la base de datos.

A continuación, veremos como quedaron distribuidas las carpetas.



En la siguientes imagenes podremos ver un ejemplo de implementación de repositorio y su interfaz.

Interfaz

```

IPersonaRepository.cs  X  IEstudioRepository.cs  PersonasApiController.cs  Progra
personapi-dotnet
using personapi_dotnet.Models.Entities;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace personapi_dotnet.Repository.Interfaces
{
    public interface IPersonaRepository
    {
        Task<IEnumerable<Persona>> GetAllAsync();
        Task<Persona> GetByIdAsync(int id);
        Task<Persona> CreateAsync(Persona persona);
        Task<Persona> UpdateAsync(Persona persona);
        Task<bool> DeleteAsync(int id);
        Task<bool> ExistsAsync(int id);
    }
}

```

Repositorio

```

PersonaRepository.cs  X  IPersonaRepository.cs  IEstudioRepository.cs  PersonasApiController.cs
personapi-dotnet
using Microsoft.EntityFrameworkCore;
using personapi_dotnet.Models.Entities;
using personapi_dotnet.Repository.Interfaces;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace personapi_dotnet.Repository
{
    public class PersonaRepository : IPersonaRepository
    {
        private readonly PersonaDbContext _context;

        public PersonaRepository(PersonaDbContext context)
        {
            _context = context;
        }

        public async Task<IEnumerable<Persona>> GetAllAsync()
        {
            return await _context.Personas
                .Include(p => p.Telefonos)
                .Include(p => p.Estudios)
                .ThenInclude(e => e.Profesion)
                .ToListAsync();
        }

        public async Task<Persona> GetByIdAsync(int id)
        {
            return await _context.Personas
                .Include(p => p.Telefonos)
                .Include(p => p.Estudios)
                .ThenInclude(e => e.Profesion)
                .FirstOrDefaultAsync(p => p.Id == id);
        }

        public async Task<Persona> CreateAsync(Persona persona)
        {
            _context.Personas.Add(persona);
            await _context.SaveChangesAsync();
            return persona;
        }

        public async Task<Persona> UpdateAsync(Persona persona)
        {
            _context.Entry(persona).State = EntityState.Modified;
            await _context.SaveChangesAsync();
            return persona;
        }
    }
}

```

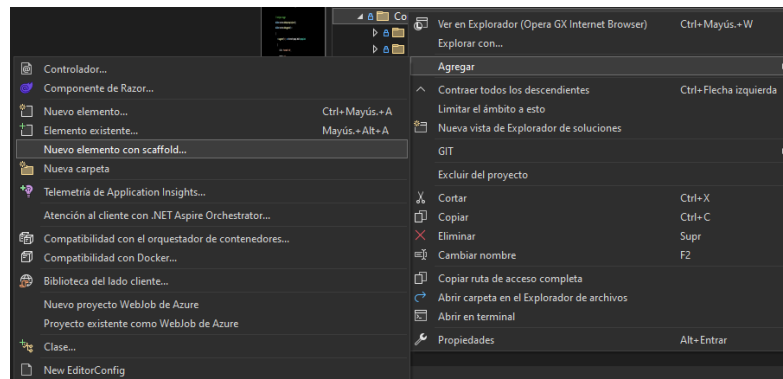
El siguiente paso es ir al Program.cs y agregar un código que permita utilizar cada una de las interfaces desde los controladores.

```
// Registro de repositorios
builder.Services.AddScoped<IPersonaRepository, PersonaRepository>();
builder.Services.AddScoped<IProfesionRepository, ProfesionRepository>();
builder.Services.AddScoped<IEstudioRepository, EstudioRepository>();
builder.Services.AddScoped<ITelefonoRepository, TelefonoRepository>();
```

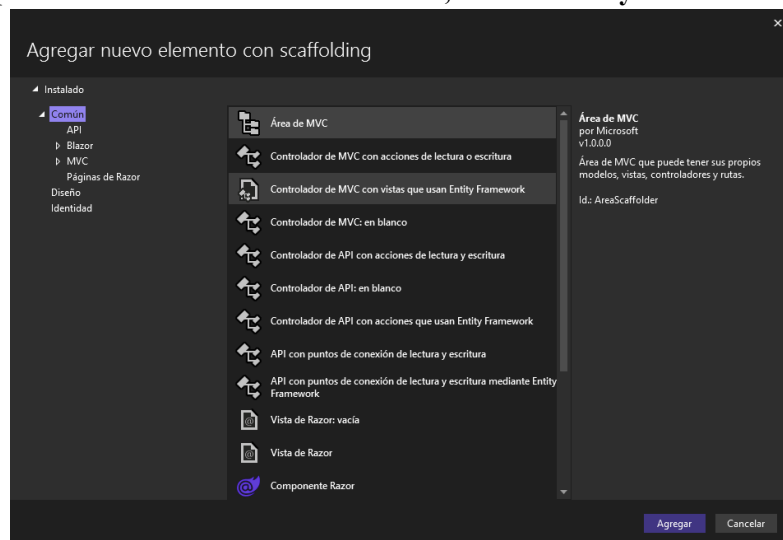
Para permitir la interacción entre el usuario, la vista y la lógica de negocio, se utilizaron controladores que siguen el patrón arquitectónico MVC. Estos se encargan de procesar las solicitudes HTTP, interactuar con los repositorios e invocar las vistas necesarias o devolver respuestas en formato JSON en el caso de las APIs.

Para facilitar el desarrollo y evitar escribir código repetitivo, los controladores junto con sus vistas fueron **generados automáticamente** mediante la herramienta **Scaffold** de Visual Studio. Este proceso se realiza a través del siguiente procedimiento:

- En Visual Studio, se hace clic derecho sobre la carpeta Controllers y se selecciona la opción **“Agregar > Nuevo elemento con scaffold”**.



- Se escoge la opción **“Controlador MVC con vistas, usando Entity Framework”**.



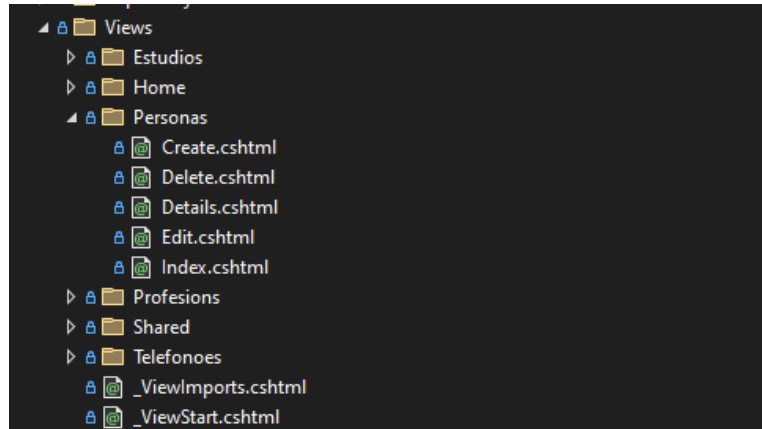
- En la configuración, se selecciona:
- La clase del modelo (por ejemplo, Persona).
- El contexto de la base de datos (PersonaDbContext).

- El nombre del controlador.

Al ejecutar el proceso, Visual Studio genera automáticamente:

- El **controlador MVC** (por ejemplo, PersonaController.cs).
- Las **vistas asociadas** en la carpeta Views/Persona, con los formularios Index, Create, Edit, Details, Delete.

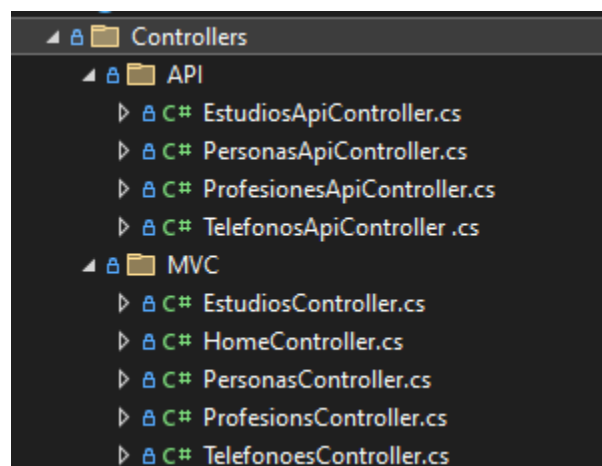
En la siguiente imagen podemos evidenciar la creacion de las Views con sus formularios



Respecto a los controladores, se organizaron en dos carpetas distintas para separar responsabilidades:

- MVC: Contiene los controladores que retornan vistas y están pensados para ser consumidos desde el navegador.
- API: Contiene los controladores de tipo REST que exponen endpoints para ser utilizados (por ejemplo, desde Swagger).

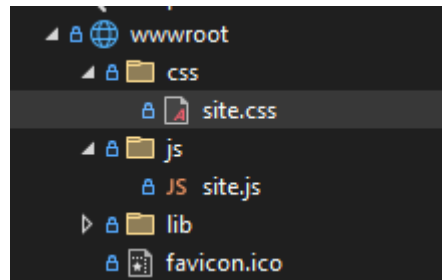
Ambos tipos de controladores hacen las mismas operaciones CRUD, pero su formato y su uso son diferentes, por ello se considera practica la separación.



Como se mencionó anteriormente, las vistas asociadas al MVC se generaron automáticamente con el scaffold, pero es importante resaltar que después se aplicaron estilos utilizando **Bootstrap** para mejorar la presentación visual.

También se creó un archivo site.css ubicado en la carpeta wwwroot, el cual contiene reglas generales de estilo que se aplican a nivel global para mejorar el diseño.

A continuación, veremos el contenido de la carpeta y el site.css



```
site.css u X | PersonalRepository.cs | PersonalRepository.cs | EstudioRepository.cs | PersonasApiController.cs | Program.cs
/* Variables globales */
:root {
  --primary-color: #3498db;
  --secondary-color: #2c3e50;
  --accent-color: #74c3c3;
  --success-color: #2ecc71;
  --warning-color: #f39c12;
  --danger-color: #e74c3c;
  --info-color: #3498db;
  --light-color: #ecf8f1;
  --dark-color: #2c3e50;
  --transition-speed: 0.3s;
}

/* Estilos generales */
body {
  font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
  color: #333;
  background-color: #f5f7fa;
}

a {
  color: var(--primary-color);
  transition: color var(--transition-speed);
}

a:hover {
  color: var(--accent-color);
  text-decoration: none;
}

/* Mejoras de diseño para tablas */
.table {
  border-collapse: separate;
  border-spacing: 0;
}

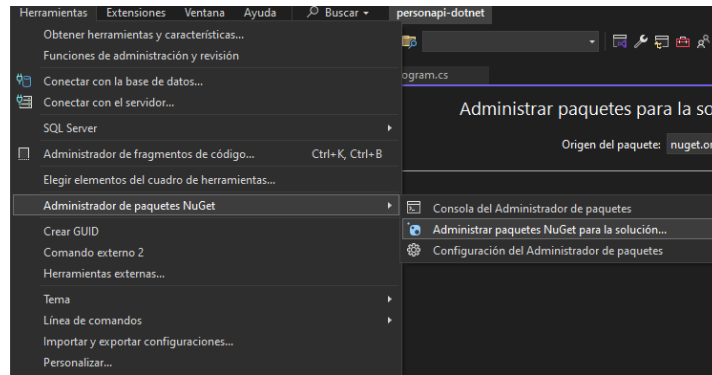
.table thead th {
  background-color: #f8f9fa;
  border-bottom: 2px solid #dee2e6;
  color: #6c757d;
  font-weight: 500;
}

.table-hover tbody tr:hover {
  background-color: rgba(52, 152, 219, 0.05);
}

/* Estilos de tarjetas */
.card {
  transition: all var(--transition-speed);
  border-radius: 8px;
  box-shadow: 4px 4px 0px #ccc;
}
```

Después de esto se realizó la integración con **Swagger** para facilitar la exploración y prueba de los endpoints en el proyecto.

Para instalar **Swagger** lo realizamos desde el **Administrador de paquetes NuGet**, y buscamos **Swashbuckle.AspNetCore** para instalarlo, luego de instalarlo debemos configurarlo en el Program.cs.



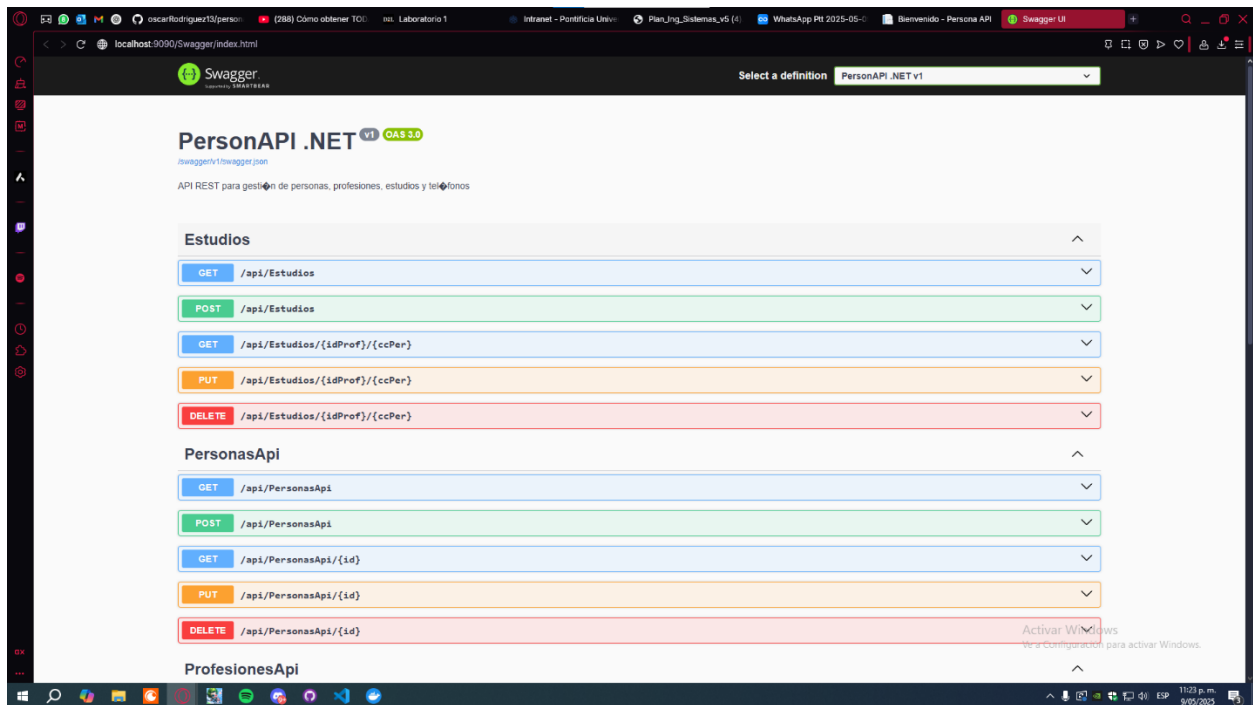
Otra opción que encontramos para la instalación es realizarla en **la Consola del Administrador de Paquetes NuGet** con el siguiente comando “Install-Package Swashbuckle.AspNetCore”

Después de configurar todo, se accedió a la documentación interactiva desde el navegador a través de la ruta /swagger. Allí se visualizan todas las entidades del sistema en la sección de Schemas, como se observa en la siguiente imagen:



En esta interfaz se identifican claramente los modelos: Persona, Telefono, Profesion y Estudio. Cada uno muestra las propiedades definidas en el modelo correspondiente. Por ejemplo, el objeto Persona incluye campos como cc, nombre, apellido, edad y género.

Swagger también permite probar los endpoints en tiempo real desde el navegador.



Por último, para facilitar el despliegue del proyecto, se creó un Dockerfile en la raíz del repositorio, el cual permite construir una imagen de la aplicación ASP.NET Core. Junto a este, se implementó un archivo docker-compose.yml que permite levantar tanto la aplicación como sus servicios relacionados (como SQL Server) con un solo comando.

```
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
WORKDIR /app
EXPOSE 8080

FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
ARG BUILD_CONFIGURATION=Release
WORKDIR /src

COPY ["personapi-dotnet.csproj", "."]
RUN dotnet restore "./personapi-dotnet.csproj" --disable-parallel --verbosity minimal

COPY . .
RUN dotnet build "./personapi-dotnet.csproj" -c $BUILD_CONFIGURATION -o /app/build

FROM build AS publish
RUN dotnet publish "./personapi-dotnet.csproj" -c $BUILD_CONFIGURATION -o /app/publish /p:UseAppHost=false

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "personapi-dotnet.dll"]
```

Esta configuración hace posible ejecutar todo el entorno de desarrollo de forma rápida, portátil y sin depender de configuraciones locales.

```

services:
  webapi:
    build:
      context: .
      dockerfile: Dockerfile
    image: personal-dotnet
    ports:
      - "9090:8080"
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ConnectionStrings__DefaultConnection=Server=sqlserver-db;Database=person_db;User Id=sa;Password=SaPassword!;TrustServerCertificate=True;
    depends_on:
      - db
    networks:
      - my_network

  db:
    image: mcr.microsoft.com/mssql/server:2022-latest
    container_name: sqlserver-db
    environment:
      - SA_PASSWORD=SaPassword!
      - ACCEPT_EULA=Y
    ports:
      - "1433:1433"
    volumes:
      - mssql_data:/var/opt/mssql
    networks:
      - my_network

  db-init:
    image: mcr.microsoft.com/mssql-tools
    container_name: db-init
    depends_on:
      - db
    volumes:
      - /init-db:/scripts
    entrypoint: >
      /bin/bash -c "
        sleep 20;
        /opt/mssql-tools/bin/sqlcmd -S sqlserver-db -U sa -P SaPassword! -d master -i /scripts/init.sql;
      "
    networks:
      - my_network

networks:
  my_network:
    driver: bridge
    volumes:
      mssql_data:

```

4. Conclusiones

1. El proyecto demuestra una correcta implementación del patrón **Modelo-Vista-Controlador (MVC)** usando **ASP.NET Core**, separando de forma clara las responsabilidades de presentación, lógica y acceso a datos. Esta separación permite mayor facilidad de mantenimiento, reutilización de componentes y escalabilidad dentro del entorno monolítico.
2. Aunque se adopta el enfoque Repository de Entity Framework, se mantiene el espíritu del patrón **DAO** al crear interfaces como `IPersonaRepository` y clases concretas que encapsulan las operaciones CRUD para cada entidad. Esto mejora la modularidad y desacopla la lógica de negocio del acceso a datos.
3. La aplicación sigue una **arquitectura monolítica tradicional**, donde todos los componentes (controladores, vistas, repositorios, base de datos) están contenidos dentro de un mismo entorno. Esta estructura se refuerza con el uso de **Docker**, que permite desplegar todo el sistema (API y base de datos) de forma unificada y portátil.
4. El laboratorio saca provecho del ecosistema .NET al integrar herramientas como **Visual Studio 2022**, **Entity Framework Core**, **Swagger**, y **SQL Server**. La automatización mediante scaffolding, migraciones y pruebas con Swagger evidencia un enfoque moderno y profesional en el desarrollo de aplicaciones web bajo la plataforma de Microsoft.

5. Referencias

- [1] <https://trycatchdebug.net/news/1421050/ssms-query-crash>
- [2] <https://www.ibm-com.translate.goog/think/topics/monolithic-architecture? x tr sl=en& x tr tl=es& x tr hl=es& x tr pto=tc>

- [3] <https://reactiveprogramming.io/blog/es/estilos-arquitectonicos/monolitico>
- [4] <https://www.atlassian.com/es/microservices/microservices-architecture/microservices-vs-monolith>
- [5] <https://es.linkedin.com/pulse/arquitectura-monol%C3%ADtica-ariel-alejandro-wagner>
- [6] <https://www.ilimit.com/es/blog/tecnologico-2/arquitecturas-monoliticas-o-arquitectura-de-microservicios-ventajas-e-inconvenientes-15>
- [7] https://www.initiumsoft.com/blog_initium/arquitectura-monolitica/
- [8] <https://reactiveprogramming.io/blog/es/patrones-arquitectonicos/dao>
- [9] <https://www.oscarblancarteblog.com/2018/12/10/data-access-object-dao-pattern/>
- [10] <https://datascientest.com/es/que-es>
- [11] <https://jossjack.wordpress.com/2014/06/22/patron-de-diseno-mvc-modelo-vista-controlador-y-dao-data-access-object/>
- [12] [https://mcazorla.gitbooks.io/programacion-en-el-servidor/content/patrones de disen o en php i/dao data access object.html](https://mcazorla.gitbooks.io/programacion-en-el-servidor/content/patrones%20de%20dise%C3%B1o%20en%20php%20i/dao%20data%20access%20object.html)
- [13] <https://platzi.com/cursos/java-sql/patron-dao-y-repository/>
- [14] <https://www.arquitecturajava.com/dao-vs-repository-y-sus-diferencias/>
- [15] [https://swagger.io.translate.google/docs/specification/v2_0/what-is-swagger/? x tr sl=en& x tr tl=es& x tr hl=es& x tr pto=tc](https://swagger.io.translate.google/docs/specification/v2_0/what-is-swagger/?x_tr_sl=en&x_tr_tl=es&x_tr_hl=es&x_tr_pto=tc)
- [16] <https://www.ibm.com/docs/es/integration-bus/10.0?topic=apis-swagger>
- [17] <https://visualstudio.microsoft.com/es/vs/community/>
- [18] <https://www.microsoft.com/es-es/download/details.aspx?id=101064>