

Laboratorio 1



Pontificia Universidad Javeriana
Arquitectura de Software

Integrantes:
Laura Ovalle
José Manuel Rodríguez
Juan David González

Bogotá D.C
5 de mayo 2025

MARCO CONCEPTUAL.....	3
Monolito.....	3
Patrón Modelo-Vista-Controlador (MVC).....	4
Patrón Data Access Object (DAO).....	5
Rest.....	5
STACK TECNOLÓGICO.....	6
.NET 8.....	6
MS SQL Server 2022.....	6
Swagger 3.....	7
Integración de conceptos y tecnologías.....	7
DISEÑO.....	8
Arquitectura de alto nivel.....	8
Diagrama de contexto C4.....	9
Diagrama de contenedores C4.....	10
Diagrama de componentes C4.....	11
Diagrama de despliegue C4.....	12
PROCEDIMIENTO.....	12
Crear el repositorio git.....	12
Instalación SQL Server 2019 Express modo Básico.....	13
Instalación SQL Server Management Studio 18.....	13
Creación la base de datos.....	14
Creación las tablas según el modelo.....	15
Instalación Visual Studio Community 2022 con los complementos.....	15
Clonar el repositorio local git a partir del remoto creado previamente.....	16
En Visual Studio Community 2022.....	17
Realizar push al repositorio.....	22
Realizar pruebas con Swagger.....	22
CONCLUSIONES.....	24
LECCIONES APRENDIDAS.....	25
REFERENCIAS.....	25

MARCO CONCEPTUAL

Para el desarrollo de este laboratorio, se realizó una investigación detallada con el objetivo de comprender a profundidad los conceptos fundamentales involucrados en la construcción de una aplicación web bajo una arquitectura monolítica. Esta comprensión resulta esencial para estructurar y organizar el software de manera coherente, modular y mantenable.

En este contexto, se abordarán tres conceptos clave que orientan el diseño de la solución: la arquitectura Monolítica, el patrón de diseño Modelo-Vista-Controlador (MVC) y el patrón Data Access Object (DAO). Cada uno de estos elementos cumple un rol específico en la separación de responsabilidades, la estructuración del código y la interacción con los datos.

Adicionalmente, se explicarán herramientas y tecnologías que permiten implementar dichos patrones de manera efectiva. Entre ellas se encuentran Entity Framework Core, utilizado como ORM para el acceso a datos; ASP.NET Core MVC, como framework principal para el desarrollo de la aplicación; API REST y Swagger, para la exposición y documentación de servicios web; y finalmente, SQL Server 2022 junto con SQL Server Management Studio, que proveen la infraestructura necesaria para la gestión y manipulación de los datos persistentes.

Monolito

Un sistema monolítico es un enfoque de diseño de software en el que todos los componentes esenciales de una aplicación (como la interfaz de usuario, la lógica de negocio y el acceso a datos) están integrados en una única unidad indivisible. Esta estructura resulta especialmente adecuada para proyectos pequeños o de mediana escala, ya que permite simplificar tanto el desarrollo como el despliegue. Al tener un único código base, los equipos pueden trabajar con mayor rapidez en las etapas iniciales del ciclo de vida del software.

Entre las principales ventajas de este tipo de arquitectura se encuentra la facilidad para desarrollar y desplegar la aplicación. El hecho de trabajar con un solo directorio centraliza las operaciones y disminuye la complejidad del sistema. Asimismo, al contar con un control centralizado, se facilita la gestión del ciclo de vida del software, incluyendo actualizaciones, monitoreo y pruebas. En términos de rendimiento, una arquitectura monolítica puede resultar más eficiente en entornos con bajo volumen de usuarios, ya que no requiere múltiples llamadas entre servicios distribuidos. Además, la depuración y las pruebas también se ven beneficiadas, dado que todos los componentes se encuentran en el mismo entorno, lo que facilita el rastreo de errores y el diagnóstico de fallos.

Sin embargo, este enfoque también presenta algunas limitaciones significativas. A medida que la aplicación crece, escalar de manera individual los componentes se vuelve una tarea compleja, ya que se debe escalar el sistema completo en conjunto. Esto afecta la eficiencia y el uso de recursos. Por otro lado, cualquier cambio, incluso menor, requiere volver a compilar y desplegar toda la aplicación, lo que implica un mayor costo en tiempo y esfuerzo. Además, un fallo en un solo módulo puede comprometer la disponibilidad de todo el sistema, generando una alta dependencia entre los componentes. También se vuelve más difícil incorporar nuevas tecnologías, ya que cualquier cambio de framework o lenguaje afecta la totalidad del monolito. Esta rigidez tecnológica puede convertirse en una barrera importante para la evolución del sistema.

Desde una perspectiva práctica, muchas empresas líderes como Netflix, Shopify o Atlassian comenzaron utilizando arquitecturas monolíticas debido a su simplicidad y velocidad de implementación en las fases iniciales. No obstante, con el tiempo, estas organizaciones migraron hacia arquitecturas de microservicios para enfrentar los desafíos de escalabilidad, flexibilidad y mantenimiento. A pesar de ello, la arquitectura monolítica sigue siendo una opción válida y eficiente para sistemas acoplados o aplicaciones con alcance limitado.

Patrón Modelo-Vista-Controlador (MVC)

El patrón de diseño Modelo-Vista-Controlador (MVC) es una arquitectura que organiza las aplicaciones en tres componentes principales e interconectados: el Modelo, la Vista y el Controlador. Esta separación de responsabilidades promueve una estructura más limpia y mantenible, facilitando tanto el desarrollo como la escalabilidad del sistema. Al dividir las tareas según su naturaleza, el patrón MVC permite que los cambios en una parte del sistema tengan un impacto mínimo en las demás.

El Modelo se encarga de gestionar los datos y la lógica de negocio. Este componente es responsable de mantener la integridad de la información, procesar las reglas del negocio y comunicarse con las fuentes de datos, como bases de datos. Por su parte, la Vista representa la interfaz gráfica del usuario; su función es mostrar la información proporcionada por el Modelo sin realizar ningún tipo de procesamiento lógico. Finalmente, el Controlador actúa como intermediario entre el usuario y el sistema, interpretando las acciones del usuario, actualizando el Modelo según sea necesario y seleccionando la Vista adecuada para mostrar la respuesta. En la *ilustración 1*, se presenta el modelo general MVC y su flujo de datos correspondiente.

Entre los principales beneficios del uso de MVC se encuentra la separación de responsabilidades, que permite modificar o reemplazar uno de los componentes sin afectar directamente a los demás. Esto favorece la reutilización del código, especialmente en el caso de las Vistas y los Controladores, los cuales pueden adaptarse fácilmente a distintas plataformas o interfaces de usuario. Además, el patrón contribuye a una mejor escalabilidad, ya que nuevas funcionalidades pueden integrarse sin reestructurar el sistema completo.

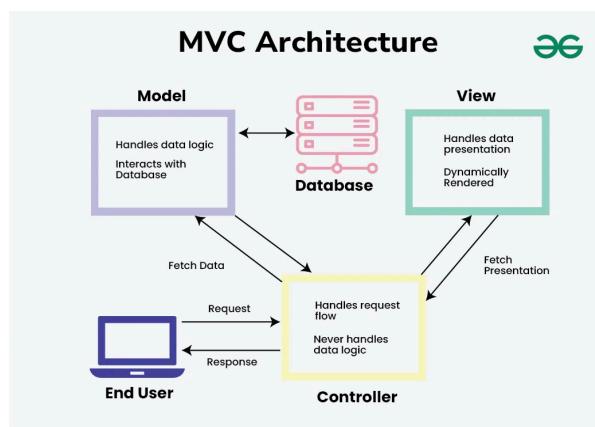


Ilustración 1. Diagrama Modelo-Vista-Controlador (MVC)

Patrón Data Access Object (DAO)

El patrón Data Access Object (DAO) es una solución de diseño que permite abstraer y encapsular la lógica de acceso a datos en una aplicación. Su principal propósito es separar esta lógica de la lógica de negocio, de modo que la aplicación pueda comunicarse con la base de datos sin conocer los detalles técnicos de dicha interacción. Esto mejora significativamente la modularidad del sistema, facilita su mantenimiento y permite que los cambios en la capa de datos no afecten directamente al resto de la aplicación.

Este patrón está compuesto por tres elementos principales. En primer lugar, la interfaz DAO, que define un conjunto estándar de operaciones, comúnmente conocidas como CRUD (Crear, Leer, Actualizar y Eliminar). En segundo lugar, la implementación DAO, que contiene el código específico para interactuar con la base de datos. Y por último, el objeto de modelo o valor, que representa las entidades o estructuras de datos utilizadas dentro de la aplicación, como por ejemplo una clase Estudiante.

Entre sus principales ventajas, destaca el desacoplamiento que genera entre la lógica de negocio y la de acceso a datos, permitiendo que ambas evolucionen de forma independiente. Además, proporciona abstracción, ya que oculta los detalles del sistema de almacenamiento, lo que facilita futuras migraciones o cambios en el motor de base de datos. Otra fortaleza es su testabilidad, dado que el uso de interfaces permite simular implementaciones para pruebas unitarias. Asimismo, los DAOs promueven la reutilización del código, ya que un mismo componente puede ser utilizado por diferentes partes del sistema.

Sin embargo, este patrón también tiene algunas desventajas. Una de ellas es la posible sobrecarga, ya que si no se diseña e implementa correctamente, puede generar consultas inefficientes o redundantes. Además, existe una curva de aprendizaje asociada, pues requiere un buen entendimiento tanto de la lógica de negocio como de las operaciones sobre la base de datos.

Rest

REST (Representational State Transfer) es un estilo arquitectónico ampliamente adoptado para el diseño de servicios web que permite la comunicación entre sistemas distribuidos mediante el protocolo HTTP. Su simplicidad, escalabilidad y facilidad de integración lo han convertido en un estándar de facto para la creación de interfaces entre el cliente y el servidor en aplicaciones modernas.

REST se basa en varios principios fundamentales que guían su implementación. El primero es la arquitectura cliente-servidor, que promueve la separación de responsabilidades entre la interfaz de usuario y la lógica de negocio, permitiendo que ambas evolucionen de forma independiente. Otro principio esencial es la ausencia de estado (statelessness), lo que implica que cada solicitud enviada desde el cliente al servidor debe contener toda la información necesaria para procesarla, sin depender de sesiones almacenadas en el servidor.

Además, REST promueve el uso de una interfaz uniforme basada en métodos estándar de HTTP, como GET, POST, PUT y DELETE, lo cual estandariza las operaciones sobre los recursos. También puede incorporar el principio de HATEOAS (Hypermedia As The Engine Of Application State), en el

cual los recursos devueltos incluyen hipervínculos hacia otras operaciones o recursos relacionados, facilitando la navegación del cliente por la API.

STACK TECNOLÓGICO

El desarrollo de aplicaciones modernas requiere un conjunto de tecnologías que garanticen eficiencia, escalabilidad, seguridad y facilidad de mantenimiento. En este proyecto se ha adoptado un stack tecnológico compuesto por .NET 8, Microsoft SQL Server 2022 y Swagger 3. Esta combinación permite implementar arquitecturas limpias, con separación clara de responsabilidades, acceso optimizado a datos y una interfaz de programación bien documentada. A continuación, se describen las principales características y el rol que desempeña cada una en el proyecto.

.NET 8

Es la versión más reciente del framework unificado y de código abierto desarrollado por Microsoft. Diseñado para el desarrollo de aplicaciones modernas, de alto rendimiento y multiplataforma, .NET 8 proporciona una base robusta para la creación de soluciones web, de escritorio, móviles, en la nube y embebidas. Esta versión continúa con la evolución de la plataforma introduciendo mejoras clave tanto en rendimiento como en experiencia de desarrollo.

Entre sus características más destacadas se encuentran importantes mejoras en la eficiencia de ejecución y en la recolección de basura (GC), lo cual reduce el consumo de memoria y acelera los tiempos de respuesta de las aplicaciones. También se incorpora un nuevo modo de globalización optimizado para dispositivos móviles, que permite reducir el tamaño de las aplicaciones manteniendo compatibilidad con diversos idiomas y culturas. Además, .NET 8 introduce generadores de código que facilitan la interoperabilidad con tecnologías COM y la vinculación directa con configuraciones, incrementando así la productividad y reduciendo errores comunes.

En el ámbito del desarrollo web, .NET 8 extiende el soporte de ASP.NET Core, incluyendo tecnologías como Blazor, SignalR y APIs mínimas, lo cual permite construir aplicaciones interactivas y en tiempo real con mayor simplicidad.

MS SQL Server 2022

Microsoft SQL Server 2022 es un sistema de gestión de bases de datos relacional que ofrece una plataforma sólida, segura y de alto rendimiento para el almacenamiento, consulta y administración de datos estructurados. Diseñado para entornos empresariales, SQL Server 2022 incorpora capacidades avanzadas que mejoran tanto la eficiencia operativa como la seguridad de los datos, posicionándolo como una solución confiable para aplicaciones críticas.

Una de sus principales innovaciones es el procesamiento inteligente de consultas, que optimiza automáticamente el rendimiento de las consultas sin requerir ajustes manuales complejos. Esto se traduce en respuestas más rápidas y una mayor eficiencia en el uso de recursos, especialmente en escenarios con grandes volúmenes de datos. Además, se han reforzado los mecanismos de seguridad mediante características como Always Encrypted, que protege los datos sensibles incluso durante su uso, y seguridad a nivel de fila, que permite definir políticas de acceso detalladas según el perfil del usuario.

En términos de disponibilidad, SQL Server 2022 introduce mejoras en el modelo de alta disponibilidad a través de Always On Failover Clustering, asegurando la continuidad del servicio incluso ante fallos del sistema o del hardware.

Swagger 3

Swagger, actualmente conocido como la especificación OpenAPI, es un conjunto de herramientas y normas que permiten definir, documentar y consumir APIs RESTful de manera estandarizada y comprensible. Su propósito principal es facilitar tanto a desarrolladores como a usuarios finales la comprensión de cómo interactuar con una API, brindando una interfaz clara y funcional que describe los recursos disponibles y cómo acceder a ellos.

Entre sus principales características se destaca la documentación interactiva, que permite a los usuarios probar directamente los endpoints desde una interfaz web sin necesidad de herramientas externas. Esto agiliza las pruebas, mejora la experiencia del desarrollador y contribuye a detectar errores tempranamente. Además, Swagger admite la generación automática de código cliente, lo que facilita la integración de APIs en aplicaciones escritas en distintos lenguajes de programación.

Otra funcionalidad importante es el soporte para versionado de APIs y la capacidad de definir esquemas de seguridad como autenticación por tokens o API keys, lo cual es esencial para garantizar que los servicios estén protegidos y bien estructurados.

Integración de conceptos y tecnologías

La arquitectura del proyecto se fundamenta en la integración coherente de patrones de diseño y herramientas tecnológicas que, permiten construir una aplicación modular, mantenible y escalable. Uno de los pilares principales es el patrón Modelo-Vista-Controlador (MVC), implementado mediante ASP.NET Core MVC, el cual proporciona una estructura robusta para separar la lógica de presentación, negocio y acceso a datos. Esta separación facilita la evolución de la aplicación y mejora la legibilidad del código.

En cuanto a la persistencia de datos, se adopta el patrón Data Access Object (DAO), el cual es gestionado eficientemente a través de Entity Framework Core, una herramienta ORM que permite mapear objetos de la aplicación a tablas de la base de datos. Esta lógica DAO se integra directamente con Microsoft SQL Server 2022, que actúa como sistema gestor de base de datos, garantizando rendimiento, seguridad y confiabilidad en el almacenamiento de la información.

Finalmente, para la comunicación entre el cliente y el servidor, se emplea una arquitectura basada en APIs REST, desarrolladas con ASP.NET Core. Estas APIs siguen principios como la separación cliente-servidor, el uso de métodos HTTP estándar y la ausencia de estado. Su documentación e interacción son facilitadas por Swagger 3, una herramienta que genera una interfaz interactiva para probar y explorar los endpoints, mejorando la comprensión del sistema tanto para desarrolladores como para usuarios técnicos.

DISEÑO

El presente apartado describe el diseño arquitectónico y estructural de la aplicación *personapi-dotnet*, desarrollada bajo un enfoque monolítico empleando el patrón Modelo-Vista-Controlador (MVC) y el patrón Data Access Object (DAO). El objetivo principal del diseño es garantizar una separación clara de responsabilidades, facilitando el mantenimiento, la escalabilidad y la reutilización del código. Se adoptan tecnologías modernas como .NET 7, SQL Server 2022 y Entity Framework Core para el acceso a datos, junto con una interfaz REST documentada mediante Swagger 3. A través de este diseño, se define la estructura lógica del sistema, la organización de las capas, los flujos de interacción entre componentes, y las decisiones técnicas que soportan la implementación de un CRUD completo sobre el modelo de datos proporcionado.

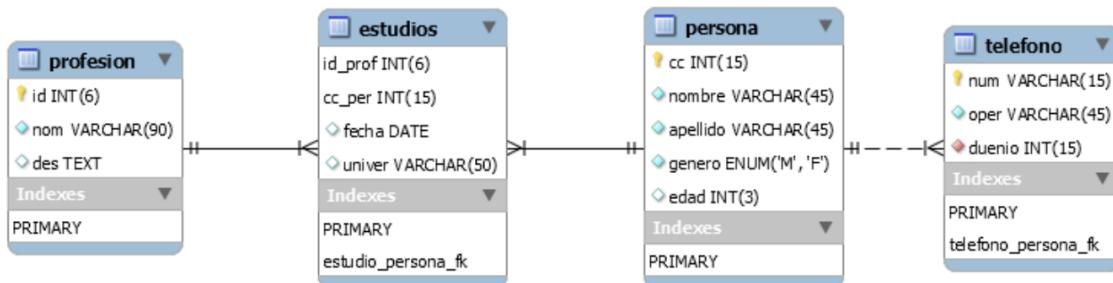


Diagrama 1. Diagrama de Entidad-Relación de la base de datos

Para el laboratorio fue proporcionado un diagrama de entidad-relación que describe la estructura de la base de datos utilizada en el sistema. Este fue diseñado para gestionar la información de las personas, sus profesiones, estudios y teléfonos. La entidad principal es persona, que contiene campos como cédula (clave primaria), nombre, apellido, género y edad. Cada persona puede tener varios teléfonos asociados, lo cual se refleja en la entidad teléfono, donde el número de teléfono es la clave primaria y el campo duenio actúa como clave foránea que referencia a la cédula de la persona propietaria. A su vez, la relación entre las personas y las profesiones se modela a través de la tabla intermedia estudios, que contiene información adicional como la fecha en que se cursó y la universidad. Esta tabla vincula múltiples personas con múltiples profesiones, estableciendo así una relación muchos a muchos entre persona y profesión. Finalmente, la entidad profesión almacena un identificador único, el nombre de la profesión y una descripción. En conjunto, el modelo permite representar de forma estructurada qué personas han estudiado qué profesiones, en qué universidad, y qué teléfonos tienen registrados.

Arquitectura de alto nivel

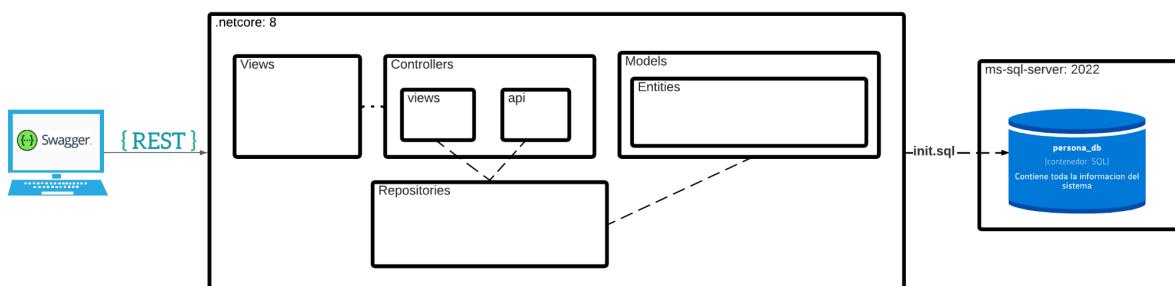


Diagrama 2. Diagrama de arquitectura de alto nivel.

La arquitectura de alto nivel de la aplicación se basa en un patrón monolítico desarrollado en .NET 8, utilizando el patrón MVC (Modelo-Vista-Controlador) para estructurar sus componentes. Este patrón organiza las responsabilidades en tres elementos fundamentales: los modelos, las vistas y los controladores. Además, la aplicación se integra con una base de datos alojada en un servidor SQL Server 2022, la cual almacena toda la información del sistema, específicamente en la base de datos denominada persona_db, inicializada mediante el script init.sql.

El diagrama adjunto ilustra de manera clara esta arquitectura. En la parte izquierda, se observa al cliente interactuando con el sistema a través de Swagger, para probar los endpoints REST del API. Estos endpoints son gestionados por los controladores de API, que forman parte de la capa de controladores, ubicada en el centro del diagrama. Dicha capa se divide en dos secciones: los controladores de vistas, que se encargan de la renderización de la interfaz gráfica (mostrada en la sección superior etiquetada como "Views"), y los controladores de API, que facilitan las interacciones programáticas mediante el protocolo REST.

En la parte derecha del diagrama, se encuentra la capa de modelos, que incluye las entidades del dominio (etiquetadas como "Entities") y los repositorios. Las entidades representan las estructuras de datos del sistema, mientras que los repositorios actúan como intermediarios entre los modelos y la base de datos. La interacción entre los repositorios y la base de datos se realiza directamente, como se indica mediante la conexión.

Diagrama de contexto C4

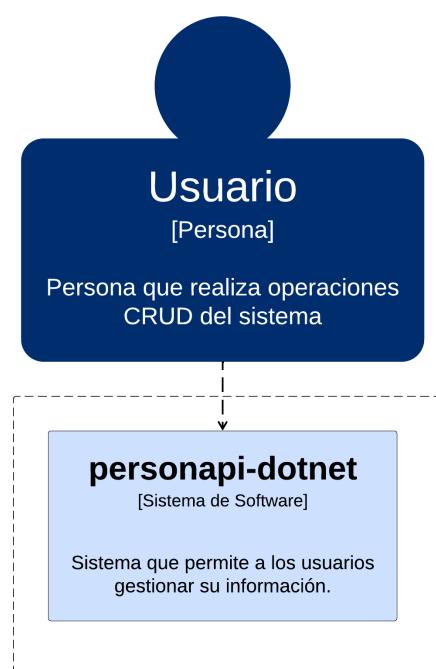


Diagrama 3. Diagrama de contexto C4.

El *Diagrama 3* representa el diagrama de contexto C4 del sistema persona-dotnet, el cual ilustra las interacciones de alto nivel entre los actores externos y el sistema. En este diagrama, se identifica al Usuario como el actor principal, representado mediante la figura de una persona. Este usuario interactúa directamente con el sistema persona-dotnet.

Este diseño permite establecer una comprensión clara del alcance del sistema y su relación con los usuarios, sentando las bases para los niveles posteriores del modelo C4, como el diagrama de contenedores y de componentes, que profundizan en la estructura interna y las interacciones más específicas del sistema.

Diagrama de contenedores C4

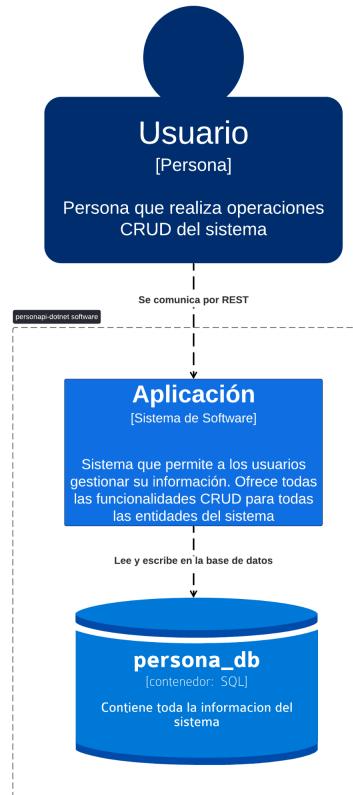


Diagrama 4. Diagrama de contenedores C4.

El diagrama de contenedores, ofrece una visión clara de la arquitectura del sistema personapi-dotnet. En la parte superior, se identifica al Usuario como actor principal, quien interactúa con la aplicación mediante una interfaz REST para realizar operaciones CRUD sobre las entidades del sistema.

En el centro del diagrama se encuentra el contenedor Aplicación, desarrollado en .NET bajo el patrón MVC. Este contenedor procesa las solicitudes del usuario y expone una API RESTful que permite una interacción eficiente, gestionando la lógica de negocio y coordinando el acceso a los datos.

Finalmente, en la parte inferior, se muestra el contenedor persona_db, una base de datos en SQL Server 2022 que almacena toda la información del sistema. La aplicación accede a esta base de datos, utilizando un script de inicialización para configurar su estructura. Esta disposición refleja una arquitectura robusta y bien organizada, con separación clara de responsabilidades.

Diagrama de componentes C4

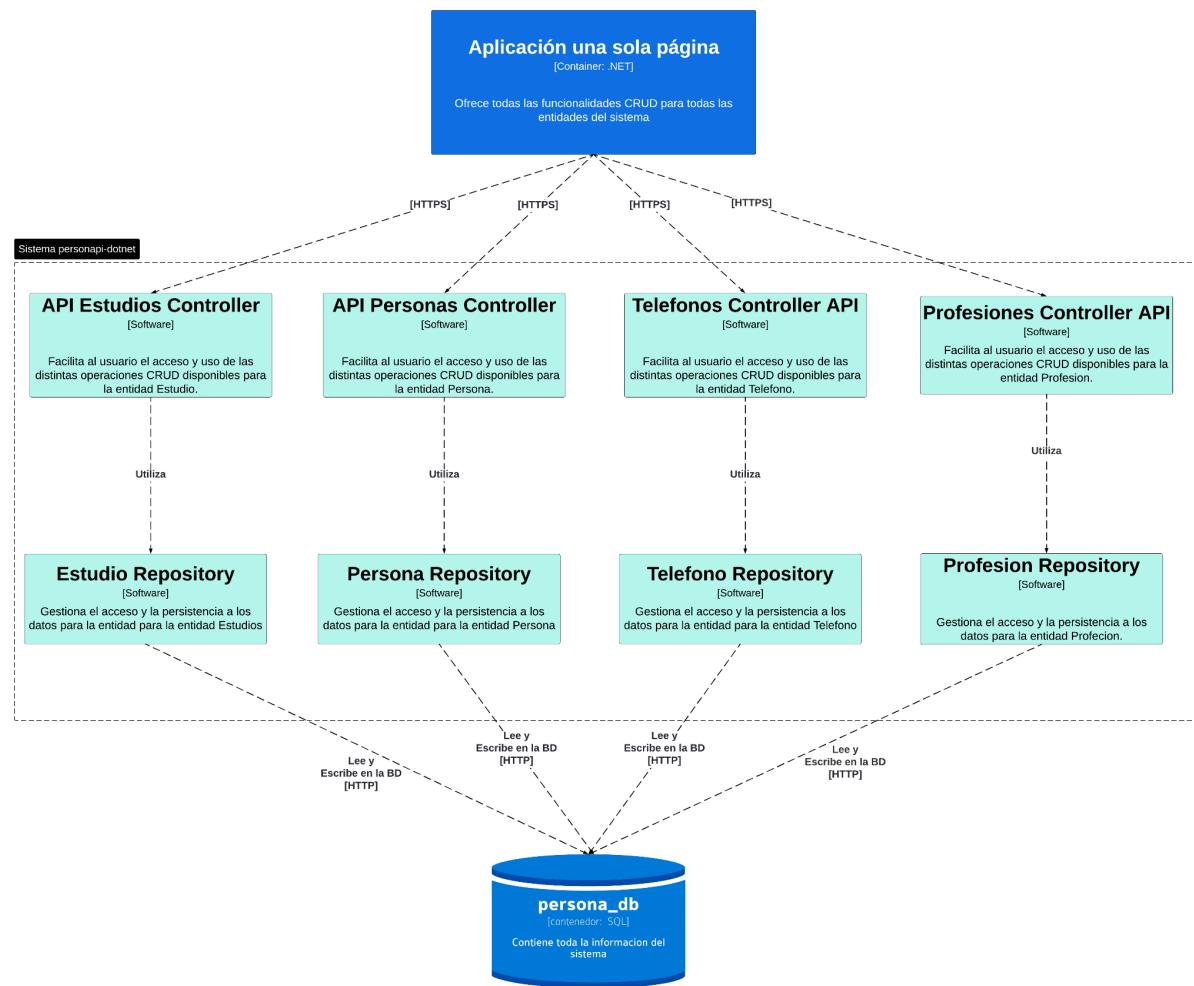


Diagrama 5. Diagrama de componentes C4.

El diagrama de componentes presentado corresponde al nivel 3 del modelo C4 y describe en detalle la estructura interna del sistema personapi-dotnet. En la parte superior se ubica el contenedor principal, una aplicación desarrollada en .NET, que actúa como punto de acceso central para que el usuario realice operaciones CRUD sobre las distintas entidades del sistema, a través de solicitudes HTTP.

La aplicación está compuesta por cuatro controladores principales: API Estudios Controller, API Personas Controller, Telefonos Controller API y Profesiones Controller API. Cada uno de estos componentes expone funcionalidades específicas para gestionar las respectivas entidades, facilitando la interacción del usuario mediante una API REST.

En la capa inferior se encuentran los repositorios correspondientes, responsables de acceder y persistir los datos en la base de datos persona_db. Estos componentes encapsulan la lógica de acceso a datos y se comunican con la base de datos SQL Server mediante Entity Framework Core.

Diagrama de despliegue C4

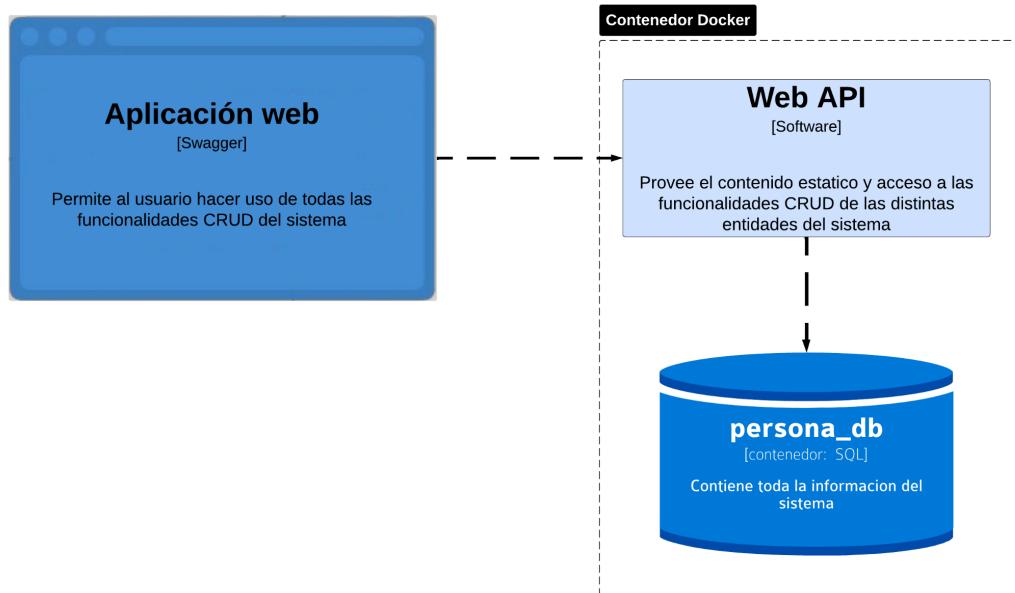


Diagrama 6. Diagrama de despliegue C4.

El *Diagrama 6* presentado anteriormente, corresponde al diagrama de despliegue del sistema personapi-dotnet, muestra cómo los componentes se ejecutan en un entorno basado en contenedores Docker. El sistema se compone de dos contenedores principales: uno que aloja la aplicación ASP.NET Core, responsable de exponer la API REST y procesar las operaciones CRUD, y otro que ejecuta SQL Server 2022, donde se almacena la base de datos persona_db. Ambos contenedores se orquestan mediante un archivo docker-compose.yml, que define sus configuraciones, redes internas y volúmenes persistentes. La comunicación entre la aplicación y la base de datos se realiza a través de una red privada de Docker, mientras que el usuario accede al sistema desde un navegador o Swagger mediante el puerto publicado del contenedor de la API. Este enfoque facilita el despliegue reproducible, el aislamiento de servicios y la portabilidad del sistema en distintos entornos.

PROCEDIMIENTO

A continuación se presenta el procedimiento llevado a cabo para la implementación inicial de una aplicación web personapi-dotnet, utilizando el stack tecnológico y los estilos y patrones arquitectónicos presentados anteriormente. El desarrollo se apoya en una base de datos relacional SQL Server y se gestiona mediante herramientas como Visual Studio 2022 y GitHub.

Crear el repositorio git

En primer lugar, se crea el repositorio en github con el nombre de personapi-dotnet como se presenta en la Ilustración 2. Este repositorio fue configurado de forma que se pueda acceder al público.

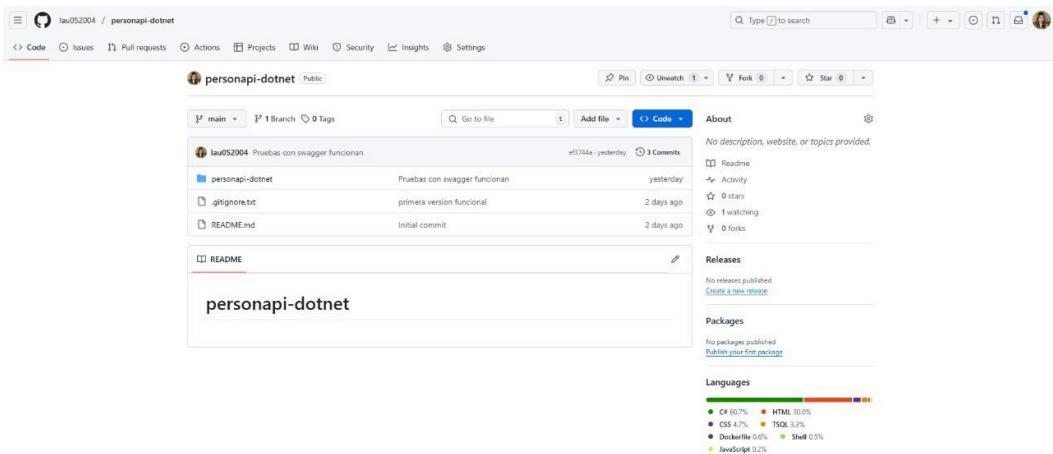


Ilustración 2. Creación de repositorio en Github.

Instalación SQL Server 2019 Express modo Básico

Posteriormente, se realizó la instalación de SQL Server 2019, en modo básico, que va a ser el motor de base de datos relacional para almacenar la información persistente del sistema.

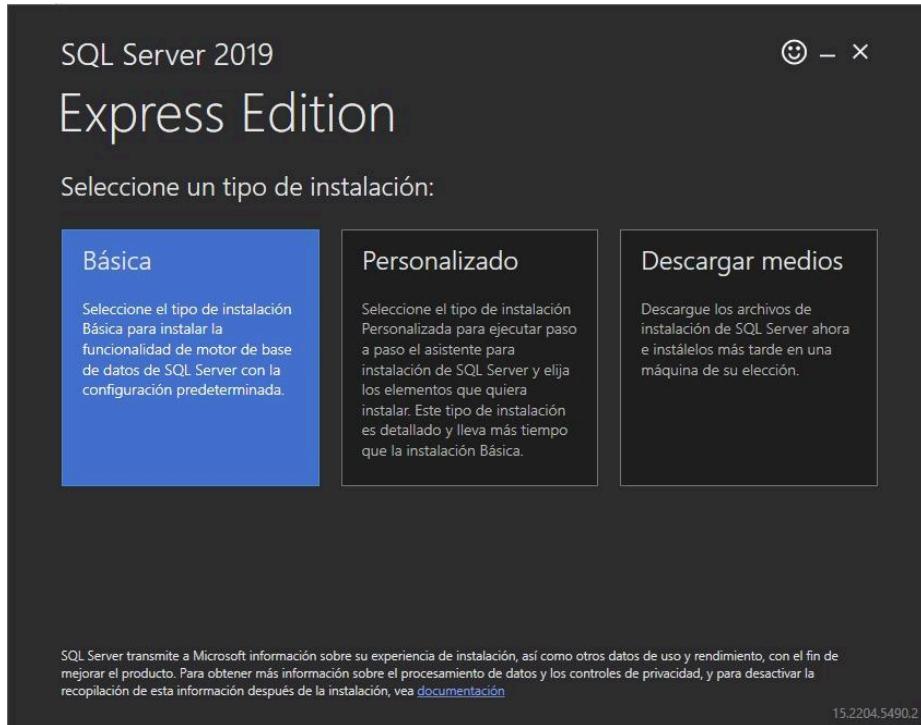


Ilustración 2. Instalación de SQL Server 2019 modo básico.

Instalación SQL Server Management Studio 18

De igual forma, se realizó la instalación de SQL Server Management Studio. Que es el entorno gráfico para la administración de la base de datos.

Microsoft SQL Server Management Studio

v18.0 Preview 4

© 2018 Microsoft.
All rights reserved.

Ilustración 3. Instalación de SQL Server Management Studio.

Creación la base de datos

Se realizó la creación de la base de datos con el nombre de persona_db y se le dio propiedad al usuario sa.

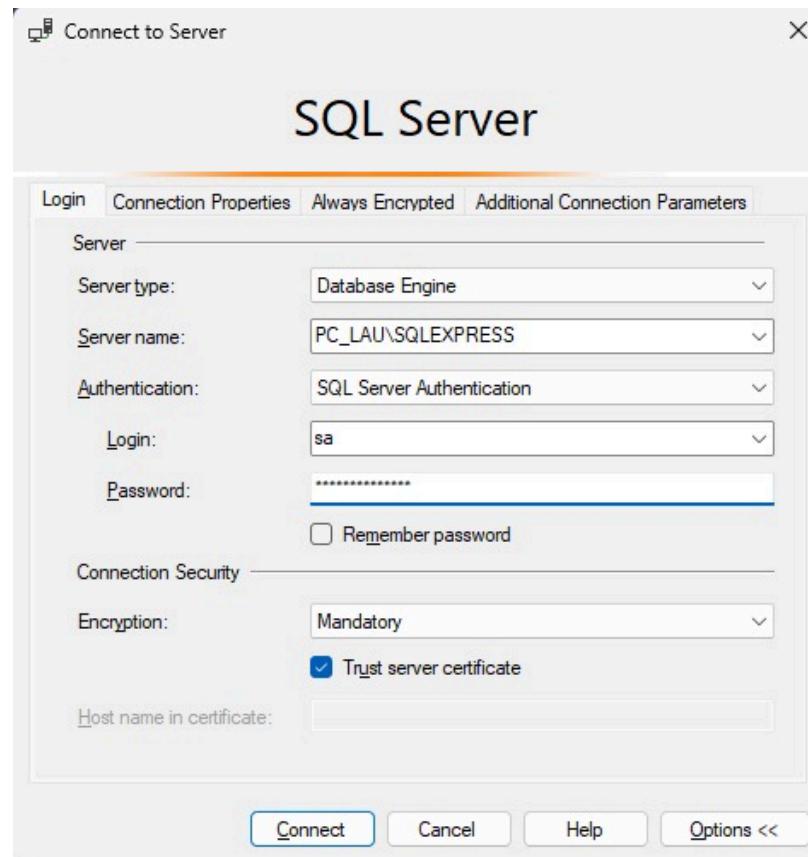


Ilustración 4. Creación base de datos y usuario sa.

Creación las tablas según el modelo

Se realizó la creación de las tablas de acuerdo al modelo propuesto inicialmente en la fase de diseño. Se crearon correctamente los atributos y sus llaves primarias y foráneas.

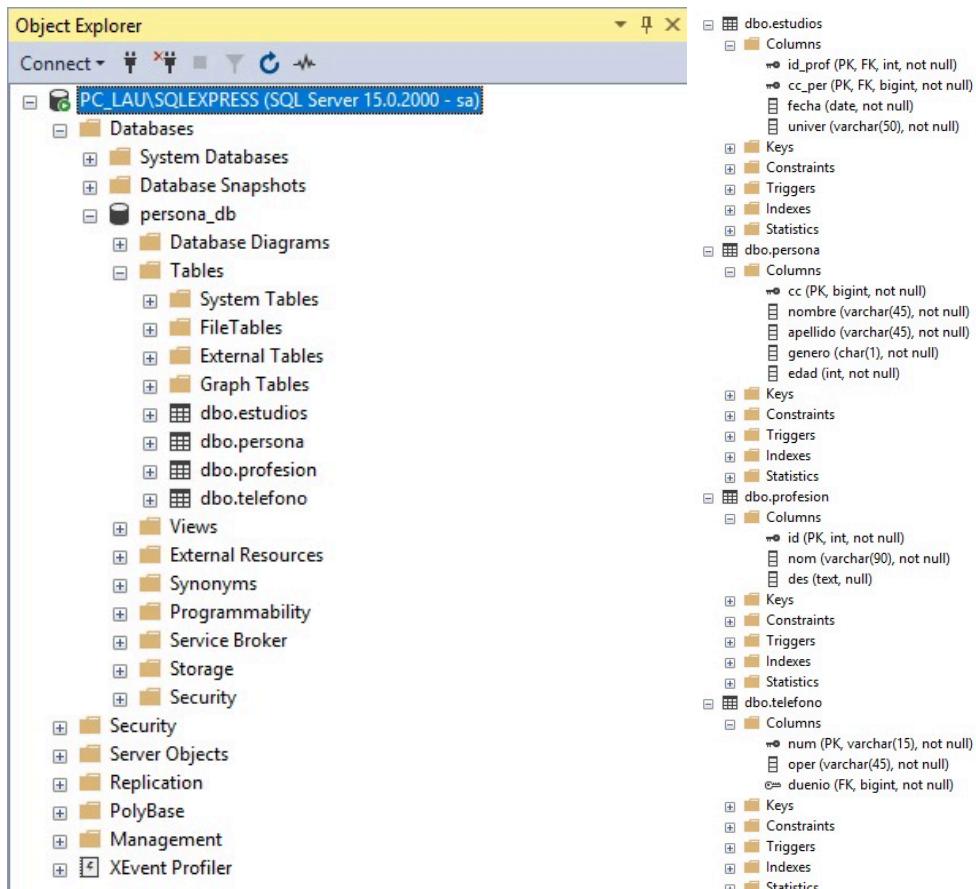


Ilustración 5. Creación de tablas en la base de datos

Instalación Visual Studio Community 2022 con los complementos

Se realizó la instalación de Visual Studio Community 2022 junto con sus complementos. Este es un entorno de desarrollo integrado (IDE) necesario para compilar y administrar proyectos .NET. Durante la instalación de Visual Studio, se seleccionaron los complementos necesarios para el desarrollo del sistema:

- Desarrollo ASP.NET y web
- Almacenamiento y procesamiento de datos
- Plantillas de proyecto y elementos de .NET Framework
- Características avanzadas de ASP.NET

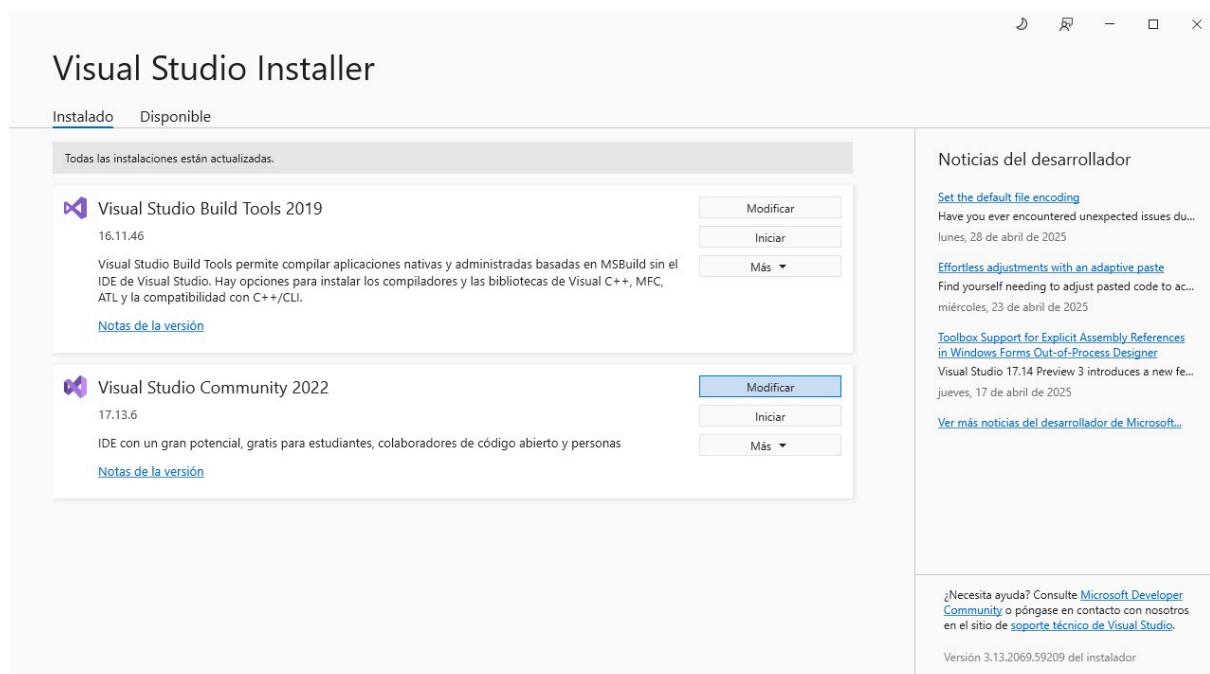


Ilustración 6. Instalación de Visual Studio Community 2022.

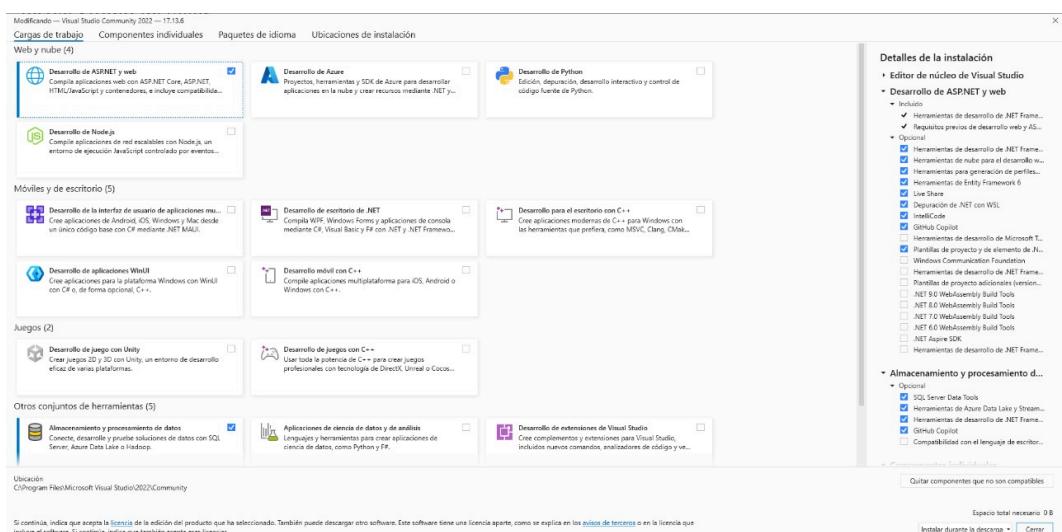


Ilustración 7. Instalación de complementos

Clonar el repositorio local git a partir del remoto creado previamente

Posteriormente, mediante Git, se clonó el repositorio remoto previamente creado en GitHub a un repositorio local.

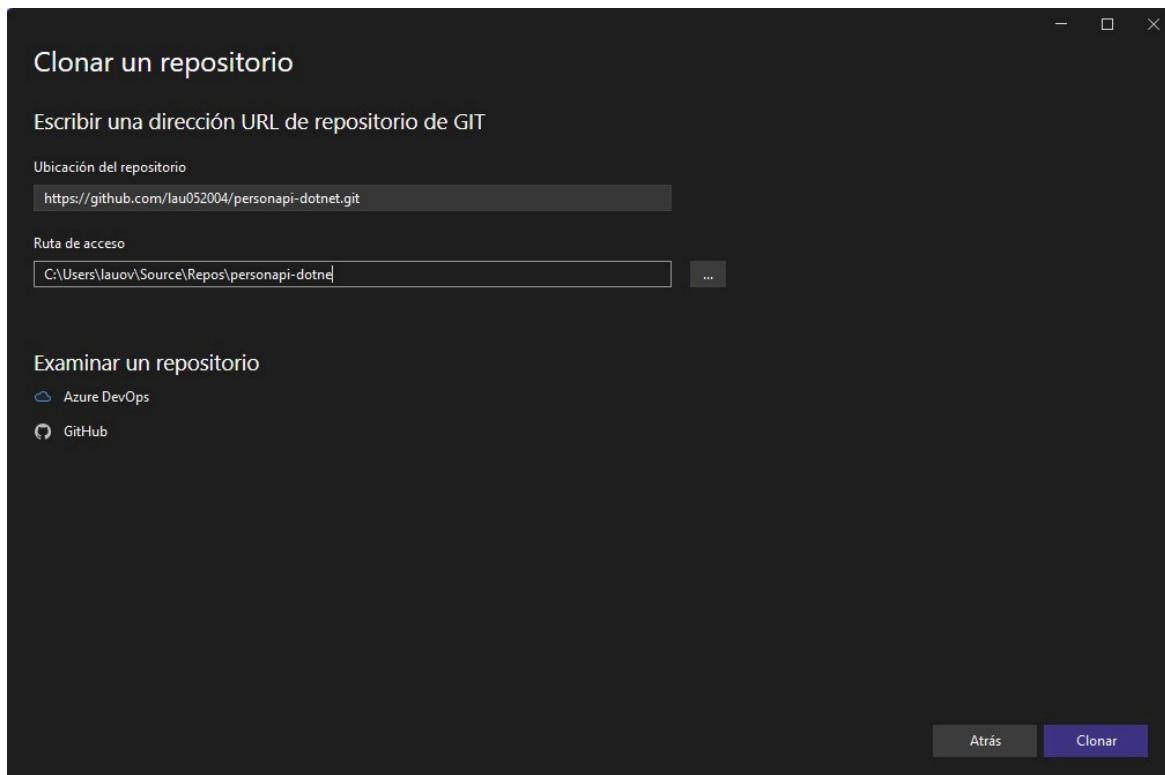


Ilustración 8. Clonar repositorio de Github

En Visual Studio Community 2022

A continuación, en la herramienta de VS Community 2022, se realizó el siguiente procedimiento para el desarrollo principal del laboratorio.

Se realizó la creación del proyecto a través de la plantilla de Aplicación web de ASP.NET Core (Modelo-Vista-Controlador).

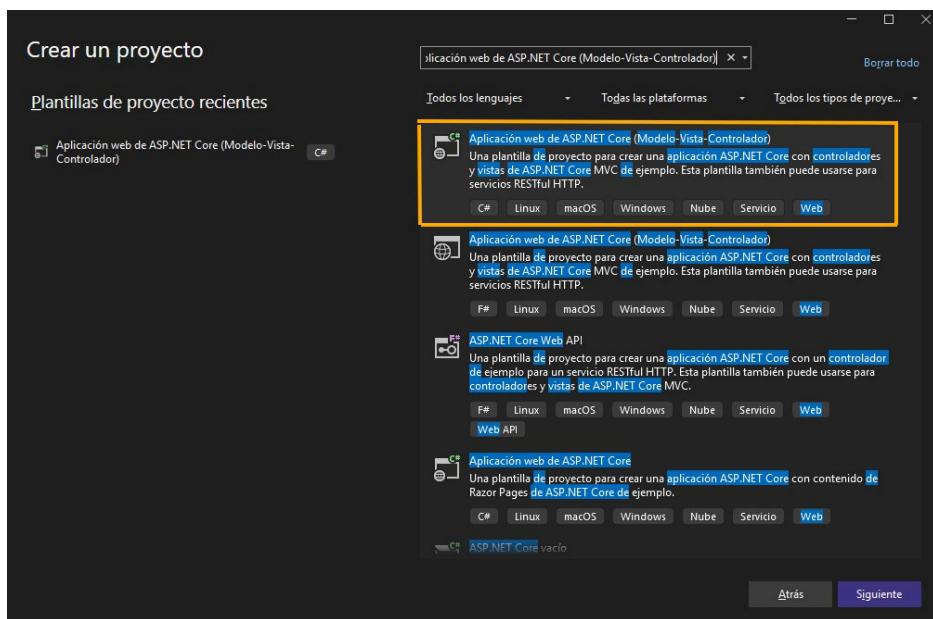


Ilustración 9. Creación del proyecto.

Luego, se realizó la configuración del mismo, en donde el nombre del proyecto corresponde a personal-dotnet.

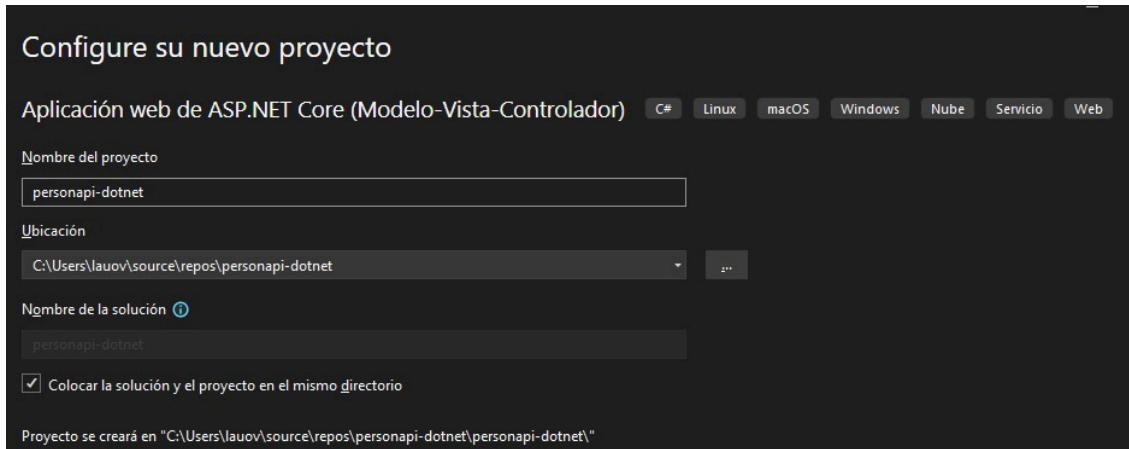


Ilustración 10. Configuración del proyecto.

Se utilizó el framework .NET 6.0, sin autenticación y sin configuración HTTPS, de acuerdo con los lineamientos establecidos.

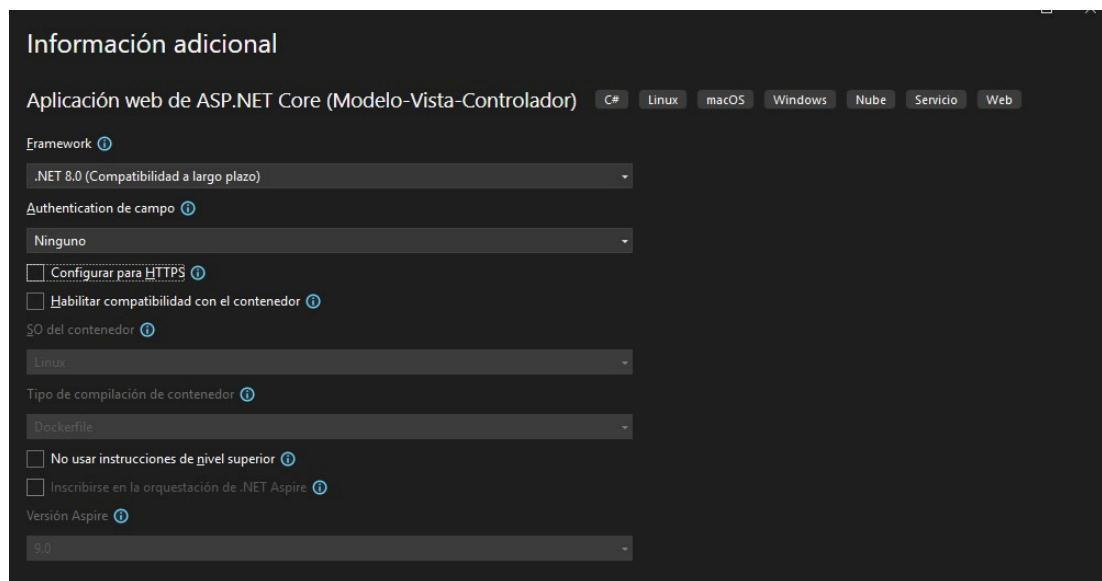


Ilustración 11. Configuración del proyecto.

En el menú Ver, se activó el Explorador de objetos de SQL Server para verificar y agregar una conexión con la instancia local de SQL Server Express (localhost\SQLEXPRESS).

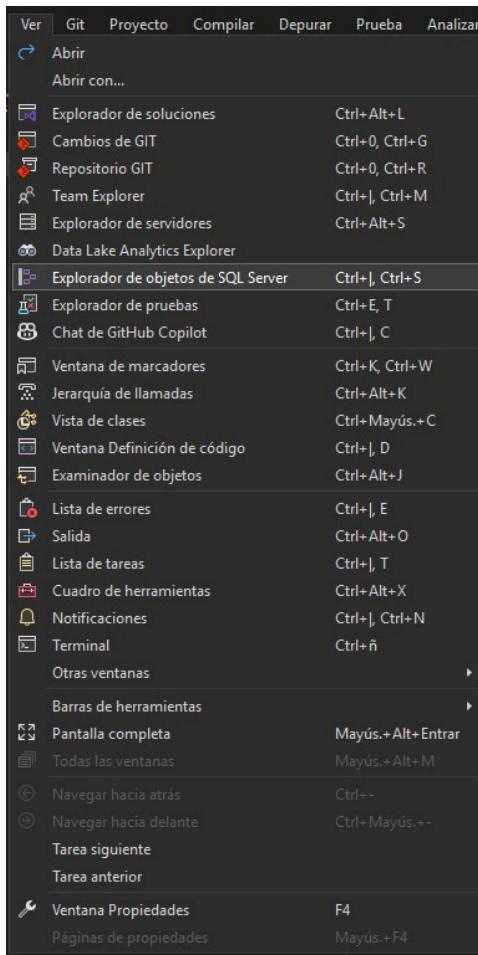


Ilustración 12. Explorador de objetos de SQL Server.

Fue necesario agregar la conexión y realizar la prueba.

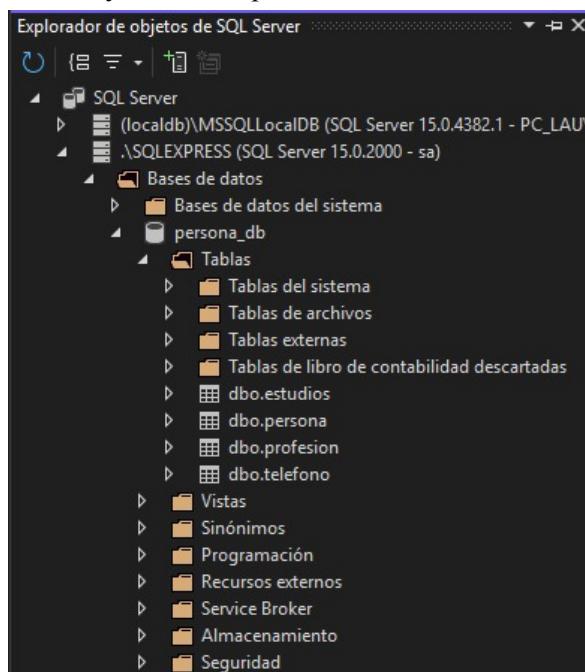


Ilustración 13. Explorador de objetos de SQL Server.

Desde el Administrador de paquetes NuGet, se instalaron por la terminal los siguientes paquetes esenciales para la comunicación con SQL Server y la generación del modelo:

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

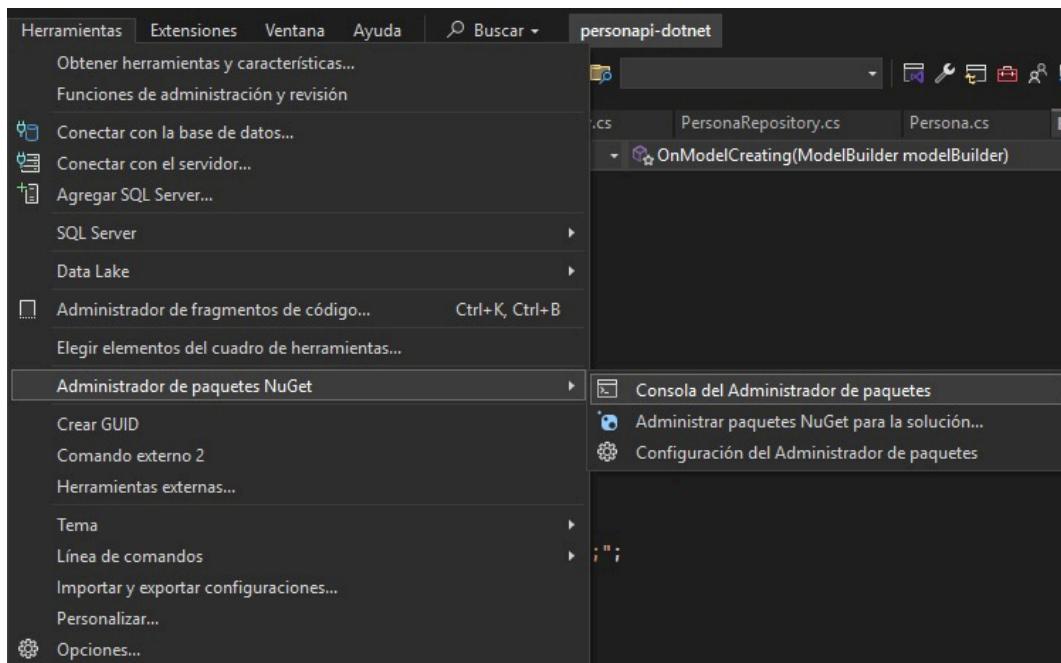


Ilustración 14. Instalación de paquetes.

Se procedió con la creación de entidades:

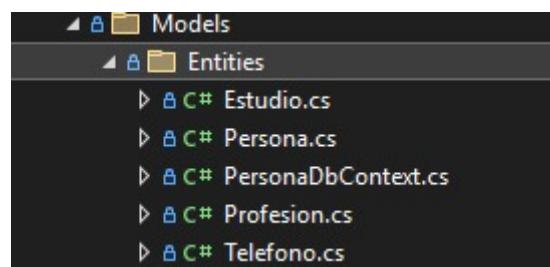


Ilustración 15. Creación de entidades.

A continuación, se realizó la creación de toda la lógica del negocio, con el desarrollo de los siguientes componentes:

- **Interfaces** para abstraer la lógica de acceso a datos.

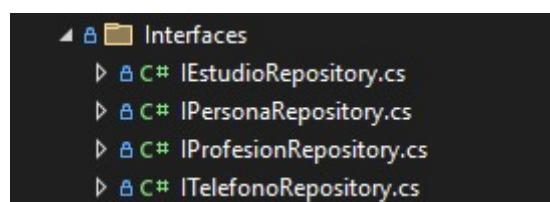


Ilustración 16. Creación de interfaces.

- **Repositorios** para implementar las operaciones CRUD mediante Entity Framework.

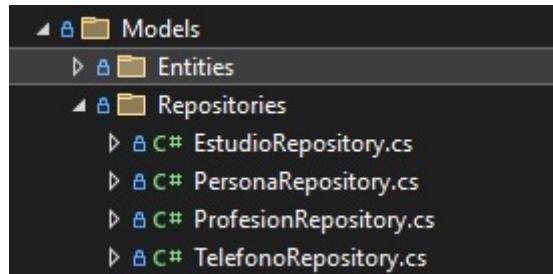


Ilustración 17. Creación de entidades.

- **Controladores** para gestionar las peticiones entrantes y mapearlas a los servicios correspondientes.

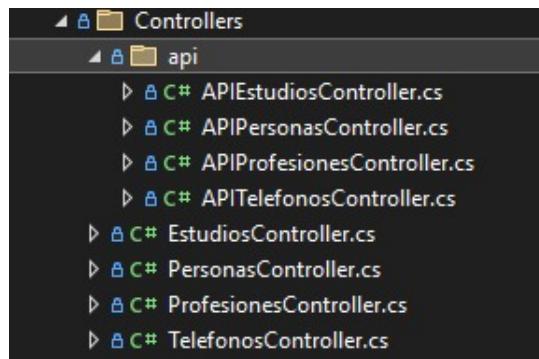


Ilustración 18. Creación de Controladores.

Finalmente, se ejecutó el proceso de despliegue local para validar el funcionamiento del sistema. Fueron creados unos archivos adicionales para el despliegue.

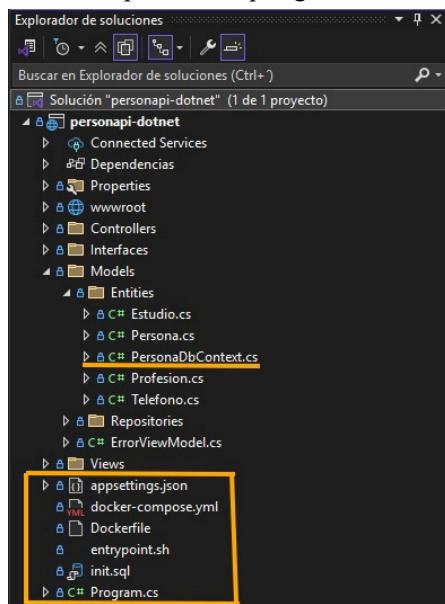


Ilustración 19. Creación de archivos adicionales para el despliegue.

Realizar push al repositorio

Se realizó push al repositorio y se probó el despliegue.

```
PS C:\Users\lauov\source\repos\personapi-dotnet> docker compose up --build -d
time="2025-05-03T08:51:58-05:00" level=warning msg="C:\\\\Users\\\\lauov\\\\source\\\\repos\\\\personapi-dotnet\\\\docker-compose.yml: the attribute `version` is obsolete, it will be ignored, please remove onfusiom"
[+] Building 14.2s (16/16) FINISHED
--> [webapi internal] load build definition from Dockerfile
--> [webapi internal] transfering dockerfile: 610B
--> [webapi internal] load metadata for mcr.microsoft.com/dotnet/sdk:8.0
--> [webapi internal] load metadata for mcr.microsoft.com/dotnet/sdk:8.0
--> [webapi internal] load Dockerfileignore
--> [webapi internal] transfering context: 2B
--> [webapi build-env 1/6] FROM mcr.microsoft.com/dotnet/sdk:8.0@sha256:bab06772f296ed5f5a1350334f15d9e2ce84ad4b3c70c90f2e43db2752c30f6
--> [webapi internal] resolve mcr.microsoft.com/dotnet/sdk:8.0@sha256:bab06772f296ed5f5a1350334f15d9e2ce84ad4b3c70c90f2e43db2752c30f6
--> [webapi stage-1 1/3] FROM mcr.microsoft.com/dotnet/aspnet:8.0@sha256:6159cf66274cf52730d7a0c7bb05cf0af94b79370176886ac58286ab6ccb7faf
--> [webapi internal] load build context
--> [webapi internal] transfering context: 1.32MB
--> [CACHED] [webapi build-env 2/6] WORKDIR /src
--> [CACHED] [webapi build-env 3/6] COPY *.csproj .
--> [CACHED] [webapi build-env 4/6] RUN dotnet restore
--> [webapi build-env 5/6] COPY . .
--> [webapi build-env 6/6] RUN dotnet publish -c Release -o /app
--> [webapi internal] [stage-1 2/3] WORKDIR /app
--> [CACHED] [webapi stage-1 3/3] COPY --from=build-env /app /
--> [webapi] exporting to image
--> [webapi] exporting layers
--> [webapi] exporting manifest sha256:79aa09ec7cc0e65667b6714c4fdb180c407567ffcb284e4fs32a5d9baa6a8a
--> [webapi] exporting config sha256:7bccf5e4074dc7b4459c3d6ad8b840747cb00dd47045050a811d570276f10
--> [webapi] exporting attestation manifest sha256:809dd46296c06e7425031c1c795c6c479ba759b18a4baa031821bf6fa828199
--> [webapi] exporting manifest list sha256:ed45331580fsafe446d578cc86aa893fc4ef6514d18d0845bbfd247d433918
--> [webapi] naming to docker.io/library/personapi-dotnet-webapi:latest
--> [webapi] resolving provenance for metadata file
[+] Running 5/5
- webapi          Built
- Network personapi-dotnet_webapp-network Created
- Volume "personapi-dotnet_sql_data" Created
- Container persona_db Started
Container personapi_dotnet_webapi Started
```

Ilustración 20. Despliegue funcionando.

Realizar pruebas con Swagger.

A continuación se presentan algunas de las pruebas con los diferentes métodos REST hacia los diferentes Endpoints con sus respuestas:

The screenshot shows the Swagger UI interface for a REST API. The URL is /api/estudios/{ccPer}/{idProf}. The method is GET. There are two parameters: ccPer (required, integer, value: 1001288968) and idProf (required, integer, value: 2). Below the parameters, there are sections for Responses, Request URL, Server response, and Details. The Response section shows a JSON object with fields like id, nombre, telefono, email, etc. The Request URL is http://localhost:5002/api/estudios/1001288968/2. The Server response shows a status code of 200 and a JSON body. The JSON body contains a large object with nested properties like telefono, email, estudios, and otrosEstudios. At the bottom, there are sections for Response headers, Responses, and Links.

Ilustración 21. Método GET al Endpoint de estudios.

POST /api/personas

Parameters

Name	Description
CC	integer(\$int32) (query)
nombre	string (query)
apellido	string (query)
genero	string (query)
edad	integer(\$int32) (query)

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:5062/api/personas?cc=1001288968&nombre=carlos&apellido=perez&genero=M&edad=30' \
  -H 'accept: */*' \
  -d ''
```

Request URL

<http://localhost:5062/api/personas?cc=1001288968&nombre=carlos&apellido=perez&genero=M&edad=30>

Server response

Code	Details	Links
201	Response body	No links

```
{
  "id": 1,
  "cc": 1001288968,
  "nombre": "carlos",
  "apellido": "perez",
  "genero": "M",
  "edad": 30,
  "estudios": [],
  "telefono": [],
  "email": []
}
```

Responses

Code	Description	Links
200	OK	No links

Ilustración 21. Método POST al Endpoint de personas.

DELETE /api/profesiones/{id}

Parameters

Name	Description
id * required	integer(\$int32) (path)

Responses

Curl

```
curl -X 'DELETE' \
  'http://localhost:5062/api/profesiones/2' \
  -H 'accept: */*'
```

Request URL

<http://localhost:5062/api/profesiones/2>

Server response

Code	Details	Links
204	Response headers	No links

```
date: Sat, 03 May 2025 21:03:04 GMT
server: Kestrel
```

Responses

Code	Description	Links
200	OK	No links

Ilustración 22. Método DELETE al Endpoint de profesiones.

The screenshot shows the Swagger UI interface for a `PUT` request to the endpoint `/api/telefonos/{numero}`. The `Parameters` section displays three fields:

- `numero` (required, string, path) value: `3208558763`
- `operador` (string, query) value: `WOM`
- `dueñoId` (integer(\$int32), query) value: `1001288968`

Below the parameters are two buttons: `Execute` (highlighted in blue) and `Clear`.

The `Responses` section includes:

- `Curl` command: `curl -X 'PUT' \ 'http://localhost:5062/api/telefonos/3208558763?operador=WOM&dueñoId=1001288968' \ -H 'accept: */*'`
- `Request URL`: `http://localhost:5062/api/telefonos/3208558763?operador=WOM&dueñoId=1001288968`
- `Server response` (Code: 204, Response headers: `Date: Sat, 03 May 2025 20:56:39 GMT`, `Server: Kestrel`)
- `Responses` table: Code 200 OK, Description: OK, Links: No links

Ilustración 23. Método PUT al Endpoint de teléfonos.

CONCLUSIONES

El desarrollo del laboratorio permitió consolidar los fundamentos teóricos y prácticos relacionados con la arquitectura monolítica y los patrones de diseño MVC y DAO, aplicados en la implementación del sistema persona-dotnet. La integración del stack tecnológico, compuesto por .NET 8, MS SQL Server 2022 y Swagger 3, demostró ser una combinación efectiva para construir una aplicación web robusta, con una clara separación de responsabilidades y una interfaz RESTful bien documentada.

La implementación exitosa de las operaciones CRUD sobre las entidades del sistema (Personas, Estudios, Teléfonos y Profesiones) validó la viabilidad del diseño propuesto, destacando la importancia de una base de datos relacional bien estructurada y la utilidad de herramientas como Entity Framework Core para la persistencia de datos. Asimismo, el despliegue mediante contenedores Docker, con docker-compose, demostró ser una solución eficiente para garantizar la portabilidad y reproducibilidad del sistema en diferentes entornos. Las pruebas realizadas con Swagger confirmaron la funcionalidad de los endpoints y la facilidad de interacción con la API, lo que refuerza la adecuación de esta herramienta para el desarrollo y validación de servicios web.

En términos generales, este laboratorio evidenció que, a pesar de las limitaciones inherentes a la arquitectura monolítica (como la dificultad de escalabilidad en sistemas grandes), su simplicidad y rapidez la convierten en una opción adecuada para proyectos iniciales o de alcance limitado, siempre que se complemente con buenas prácticas de diseño y herramientas modernas.

LECCIONES APRENDIDAS

A lo largo del desarrollo del proyecto personapi-dotnet, se adquirieron valiosas lecciones que fortalecieron las competencias técnicas del equipo. Se destacó la importancia de una planificación arquitectónica clara desde el inicio, como lo permite el modelo C4, y la necesidad de familiarizarse previamente con las herramientas del stack tecnológico, especialmente Entity Framework Core y Swagger. También se aprendió la relevancia de validar de forma continua la configuración de la base de datos y del entorno de desarrollo, así como de realizar pruebas iterativas para detectar errores tempranos. Asimismo, se evidenció que la retroalimentación constante, especialmente durante las pruebas de los endpoints, mejora significativamente la calidad del sistema. Cabe resaltar que se presentaron varias dificultades en las configuraciones iniciales debido a que era la primera vez que el equipo realizaba este tipo de implementación, lo que hizo del proceso un reto técnico y formativo.

REFERENCIAS

- ❖ <https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith>
- ❖ <https://www.geeksforgeeks.org/mvc-architecture-system-design/>
- ❖ <https://www.oscarblancarteblog.com/2018/12/10/data-access-object-dao-pattern/>
- ❖ <https://learn.microsoft.com/en-us/dotnet/core/whats-new/dotnet-8/overview>
- ❖ <https://learn.microsoft.com/en-us/sql/sql-server/what-s-new-in-sql-server-2022?view=sql-server-ver16>
- ❖ <https://dev.to/veronicaguamann/api-con-aspnet-mvc-6-y-sql-server-mediante-entity-framework-core-6-code-first-parte-2-4lbg>
- ❖ <https://c4model.com/diagrams>