

Proyecto: Desarrollo de una Aplicación con Arquitectura en N Capas

Brayan Patiño

Andrés Jácome

Departamento de Ciencias de la Computación, Universidad de las Fuerzas Armadas –

ESPE

Ing. Darío Morales

23 de junio de 2024

Tabla de contenidos

Introducción	3
Objetivo general.....	3
Objetivos específicos	3
Alcance del proyecto	4
Fases del proyecto.....	4
Fase de Análisis	4
Requisitos funcionales	4
Requisitos no funcionales	5
Diagrama de casos de uso	6
Análisis de riesgos	7
Fase de Diseño	7
Arquitectura en N Capas.....	8
Diagrama de bases de datos	8
Diseño de interfaces.....	9
Fase de Desarrollo	11
Base de datos	11
Solución del proyecto	11
Capa de Data.....	12
Capa de negocio.....	16
Capa de Servicio	18
Capa de Presentación	22
Fase de pruebas.....	24
Prueba de Funcionalidad del Repositorio	24
Prueba de Funcionalidad del Repositorio	25
Prueba de Integración Básica.....	25
Conclusiones.....	26
Recomendaciones	27
Bibliografía	28

Introducción

La gestión eficiente de una librería implica el manejo adecuado de una amplia variedad de datos, que incluyen libros, autores y los préstamos realizados. Con el avance de la tecnología, se ha vuelto crucial contar con herramientas que permitan una administración precisa y rápida de estos recursos. Este proyecto tiene como objetivo desarrollar una aplicación de librería que optimice estos procesos, mejorando la eficiencia y reduciendo los errores humanos debido a que adoptará una arquitectura en N capas, teniendo como capa de presentación una aplicación de escritorio como una aplicación web.

Objetivo general

Desarrollar una aplicación integral de gestión de librería que permita a los usuarios administrar libros, autores y préstamos de manera eficiente, utilizando una arquitectura en N capas para garantizar la escalabilidad, mantenibilidad y robustez del sistema.

Objetivos específicos

Llevar a cabo un análisis exhaustivo de los requisitos del sistema para asegurar una comprensión profunda de las necesidades de los usuarios y establecer una base sólida para el diseño y desarrollo.

Crear una arquitectura en N capas que separe las responsabilidades de la lógica de negocios, la capa de datos y la interfaz de usuario, promoviendo una mayor modularidad y facilidad de mantenimiento.

Implementar la aplicación utilizando tecnologías modernas y siguiendo las mejores prácticas de desarrollo de software, asegurando un código limpio, eficiente y escalable.

Alcance del proyecto

Alcance del Proyecto	Descripción
Capa de Presentación	Web: Desarrollo de una interfaz web accesible desde cualquier navegador. Escritorio: Creación de una aplicación de escritorio para usuarios que prefieren o requieren una solución nativa.
Capa de Negocios	Lógica de Negocios: Implementación de los procesos y reglas de negocio necesarios para la gestión de libros, autores y préstamos, asegurando que todas las operaciones cumplan con los requisitos funcionales definidos.
Capa de Datos	Base de Datos: Diseño y gestión de un sistema de almacenamiento de datos que permita un acceso rápido y seguro a la información de libros, autores y préstamos. Acceso a Datos: Desarrollo de interfaces y servicios para interactuar con la base de datos, garantizando una comunicación eficiente y segura entre la capa de datos y las demás.
Servicios	Servicios Web: Implementación de API y servicios necesarios para la integración y comunicación entre las diferentes capas de la aplicación, permitiendo la interoperabilidad.

Fases del proyecto

Fase de Análisis

La fase de análisis es fundamental para el éxito del proyecto, ya que establece la base sobre la cual se construirá el sistema. A continuación, se describen las actividades clave realizadas durante esta fase:

Requisitos funcionales

1. Gestión de Libros:

- Añadir, modificar, eliminar y consultar libros.
- Asignación de categorías y etiquetas a los libros.
- Registro de detalles como título, autor, editorial, año de publicación y número de copias disponibles.

2. Gestión de Autores:

- Añadir, modificar, eliminar y consultar autores.
- Vinculación de libros con sus respectivos autores.

3. Gestión de Préstamos:

- Registrar préstamos de libros a usuarios.
- Registrar devoluciones y renovaciones de préstamos.
- Gestión de fechas de vencimiento y recordatorios para devoluciones.

4. Interfaz de Usuario:

- Interfaces intuitivas tanto para la web como para la aplicación de escritorio.
- Funcionalidades de búsqueda avanzada y filtrado de resultados.

Requisitos no funcionales

1. Rendimiento:

- El sistema debe ser capaz de manejar al menos 1000 transacciones simultáneas sin degradación significativa del rendimiento.

2. Seguridad:

- Autenticación y autorización de usuarios para acceder a las diferentes funcionalidades del sistema.
- Encriptación de datos sensibles tanto en tránsito como en reposo.

3. Usabilidad:

- La interfaz debe ser intuitiva y fácil de usar, minimizando la necesidad de capacitación extensa.

- Diseño responsivo para garantizar una experiencia de usuario consistente en diferentes dispositivos.

4. Mantenibilidad:

- El sistema debe estar diseñado de manera modular para facilitar la actualización y mantenimiento.
- Documentación completa del código y del sistema.

Diagrama de casos de uso

Una vez recopilados los requisitos, se procedió a modelar los casos de uso para visualizar las interacciones entre los usuarios y el sistema. El diagrama de casos de uso presenta las siguientes características principales:



Análisis de riesgos

Riesgo	Descripción	Impacto	Probabilidad	Mitigación
Riesgo Técnico	Posibles dificultades técnicas en la integración de la aplicación web y de escritorio.	Alto	Medio	Planificación de pruebas de integración tempranas y frecuentes.
Riesgo de Seguridad	Vulnerabilidades de seguridad que podrían comprometer los datos del sistema.	Muy Alto	Medio	Implementación de medidas de seguridad robustas y realización de auditorías de seguridad periódicas.
Riesgo de Cronograma	Retrasos en el cronograma de desarrollo debido a cambios en los requisitos o problemas técnicos imprevistos.	Alto	Alto	Uso de metodologías ágiles para permitir una mayor flexibilidad y adaptación a cambios.
Riesgo de Rendimiento	Posible degradación del rendimiento del sistema bajo alta carga.	Medio	Bajo	Pruebas de carga y rendimiento continuas, y optimización del código.
Riesgo de Usabilidad	La interfaz de usuario podría no ser intuitiva para todos los usuarios.	Medio	Medio	Pruebas de usabilidad y feedback continuo de los usuarios durante el desarrollo.
Riesgo de Mantenimiento	Dificultades en la actualización y mantenimiento del sistema.	Medio	Bajo	Diseño modular del sistema y documentación completa del código y del sistema.

Fase de Diseño

La fase de diseño es crucial para definir la estructura del sistema y sus componentes, garantizando que el desarrollo y la implementación se realicen de manera eficiente y

efectiva. Esta fase se enfoca en la creación de una arquitectura robusta y escalable que cumpla con los requisitos establecidos durante la fase de análisis.

Arquitectura en N Capas

La arquitectura en N capas separa las responsabilidades del sistema en distintas capas, cada una con sus propias funciones específicas. Las principales capas de la arquitectura, sus objetivos y sus principales componentes para este proyecto se presentan a continuación en la siguiente figura:

ARQUITECTURA DE LA APLICACIÓN

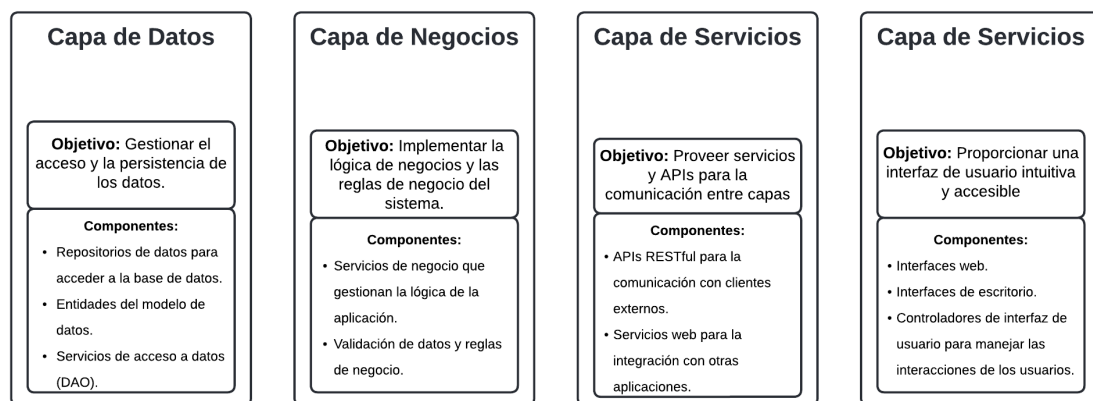
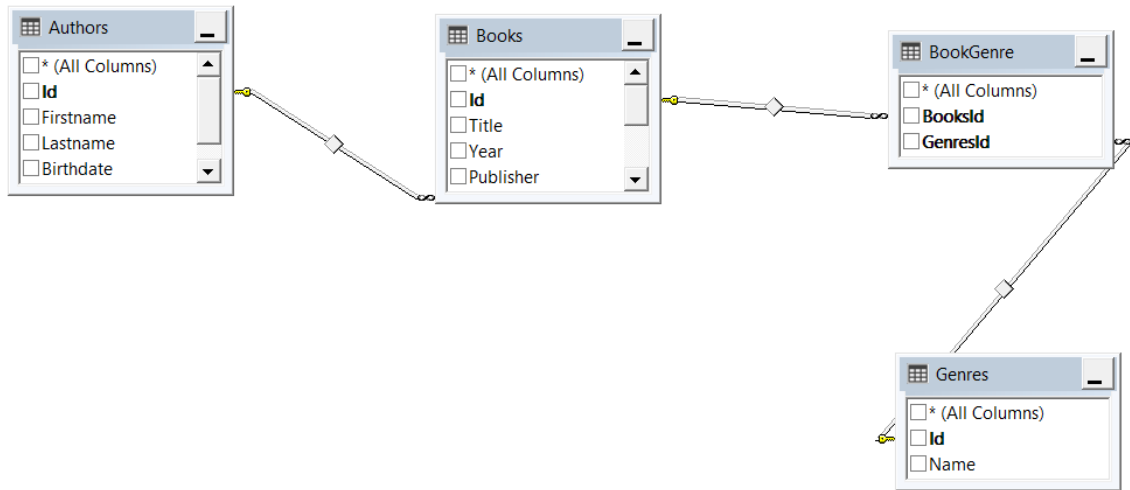


Diagrama de bases de datos

El diseño de la base de datos se centra en la normalización y en asegurar el acceso eficiente a los datos. Las tablas principales en la base de datos, junto a sus principales tributos, son los que se presentan a continuación en la figura:



Diseño de interfaces

El diseño de las interfaces de usuario se basa en principios de usabilidad y accesibilidad, asegurando una experiencia intuitiva y consistente. Las principales interfaces diseñadas son las que se muestran a continuación:

Primero se tiene la interfaz de inicio, en donde habrá las opciones de lo que se quiere hacer dentro de la aplicación:



Luego, se tiene la interfaz se tiene el formulario para agregar un nuevo libro:

Título *

ISBN

Año *

 ^ v

Editorial

Precio *

 ^ v

Autor

 Q

Géneros

Name

Name

Name

Name

Name

Imagen

También se tiene cómo se va a editar cualquier libro:

Título *

ISBN

Año *

 ^ v

Editorial

Precio *

 ^ v

Autor

 Q

Géneros

Name

Name

Name

Name

Name

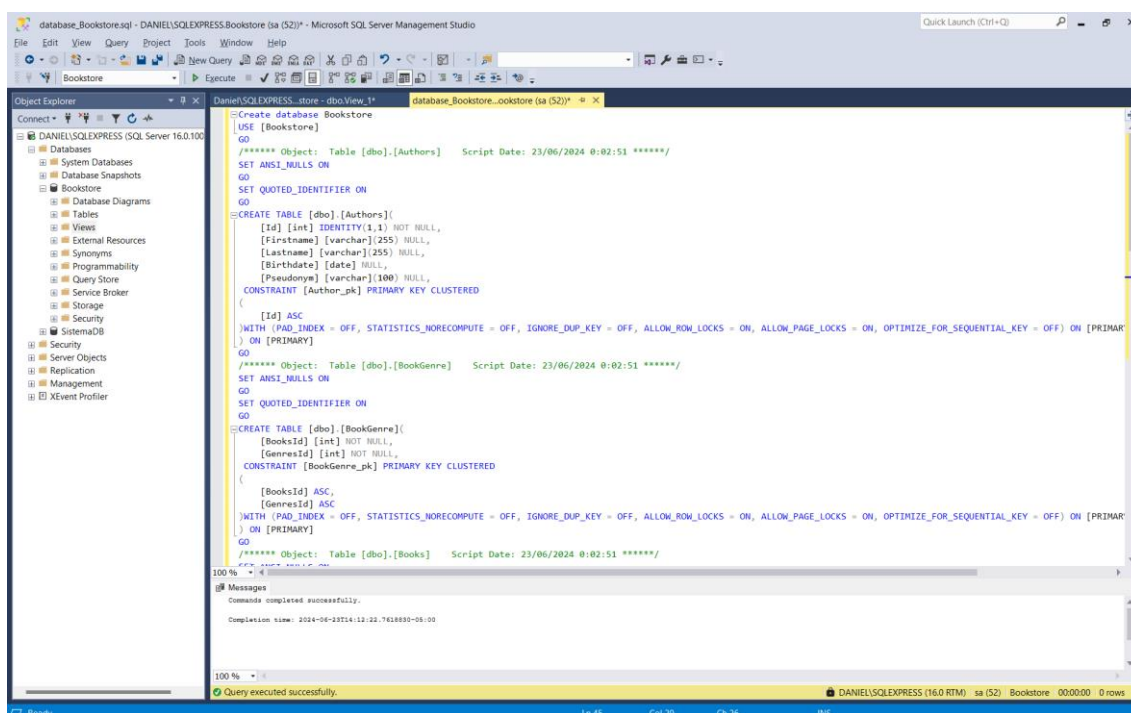
Imagen

Fase de Desarrollo

La fase de desarrollo es el núcleo del proyecto, donde se traduce el diseño en código funcional. Durante esta fase, se implementan todos los componentes del sistema siguiendo las especificaciones detalladas en la fase de diseño. Se adopta una metodología ágil para facilitar iteraciones rápidas y la integración de feedback continuo, asegurando la calidad y relevancia del producto final.

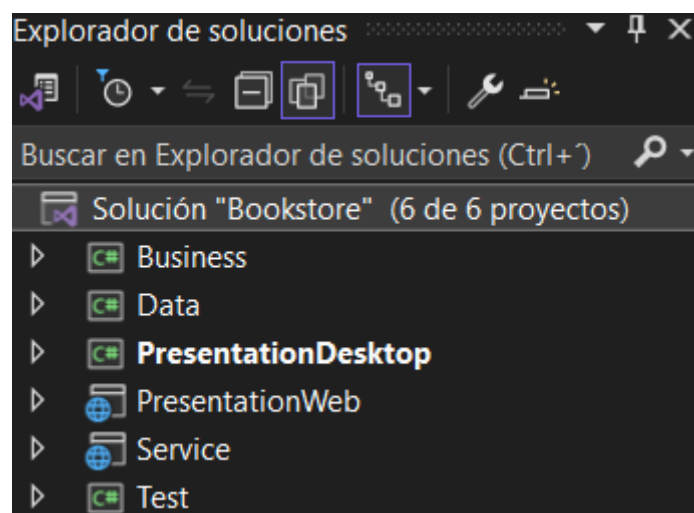
Base de datos

En un inicio, se creó la base de datos que se utilizará para este proyecto, la cual denominamos Bookstore, en donde contiene las tablas y atributos correspondientes como se mencionó anteriormente.



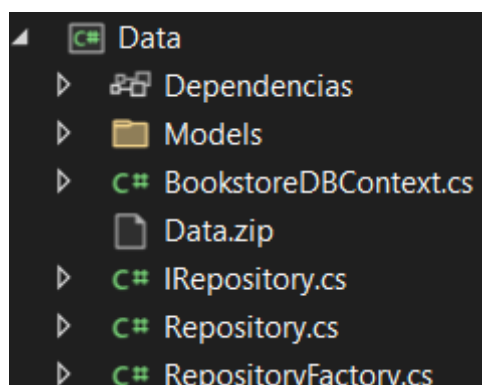
Solución del proyecto

Luego, se creó la solución en blanco del proyecto utilizando la IDE de Visual Studio 2022, en donde se agregó cada capa que se requería para organizar este programa, tan como se ve a continuación en la figura.

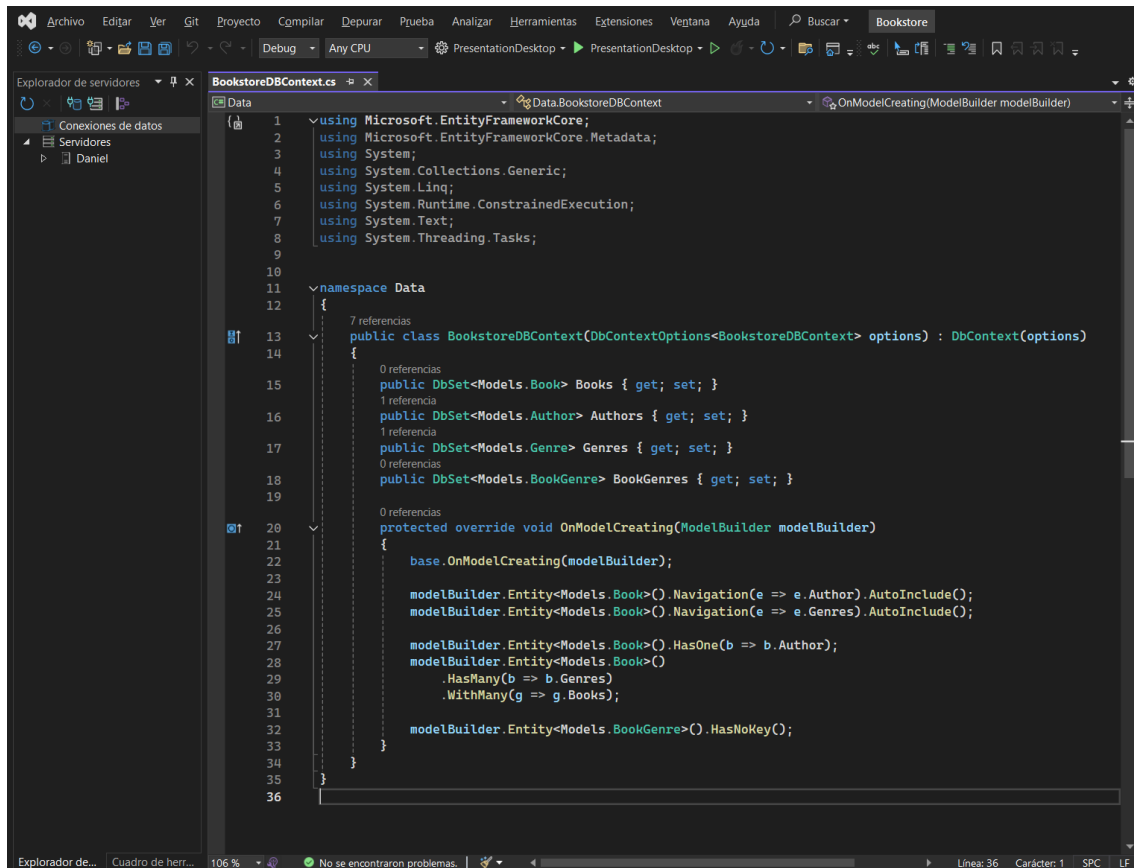


Capa de Data

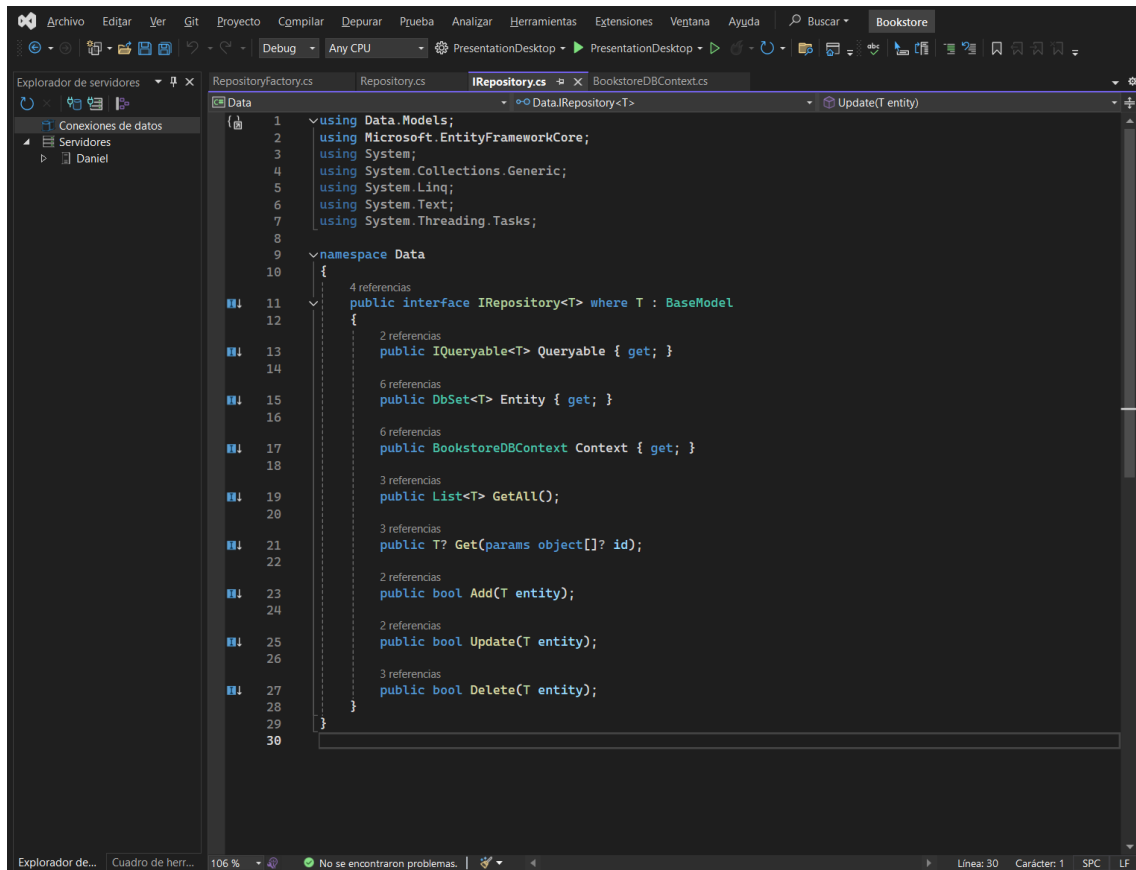
Ahora bien, dentro de la capa de Data, tal como se menciona anteriormente, se encarga de la conexión con la base de datos, en donde se tiene las siguientes clases:



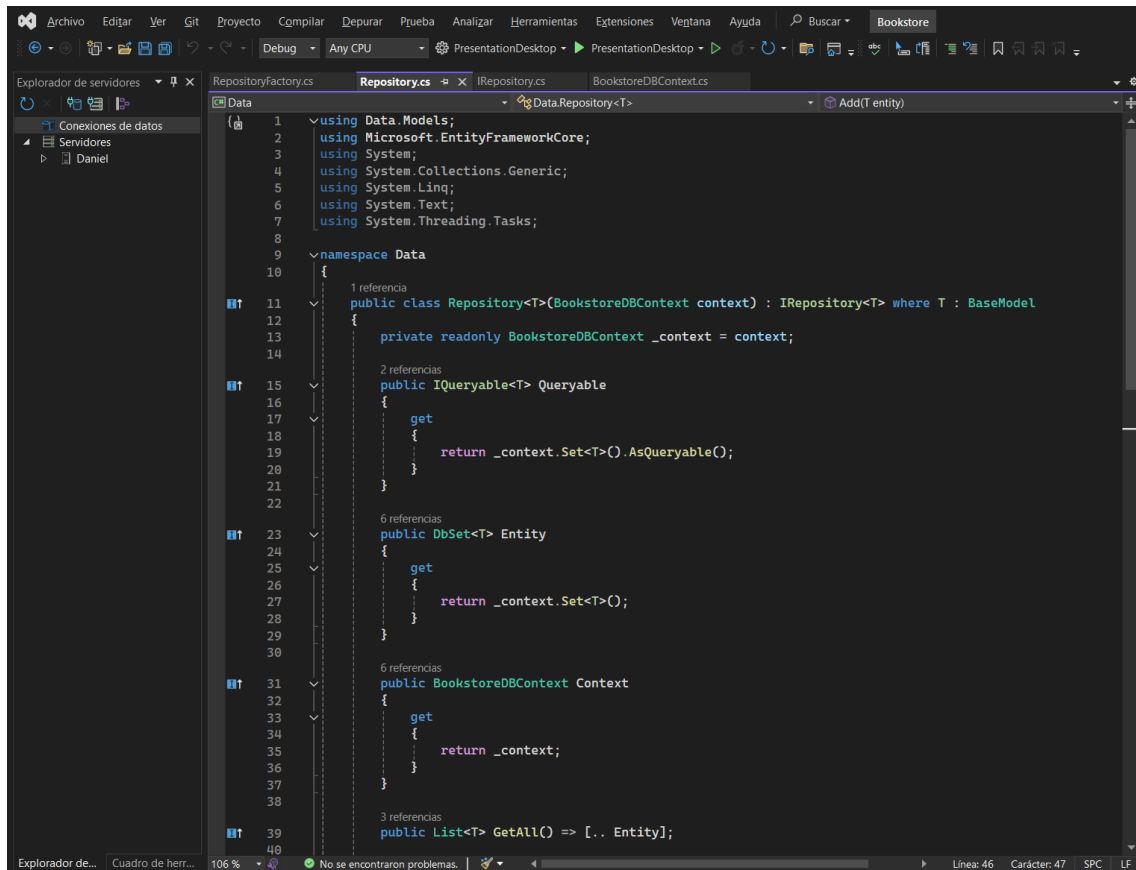
La clase `BookstoreDBContext` extiende `DbContext` y configura el contexto de la base de datos para una librería, definiendo conjuntos de entidades (`DbSet`) para libros, autores, géneros y la relación muchos a muchos entre libros y géneros. Además, sobrecarga el método `OnModelCreating` para establecer relaciones y configuraciones específicas, como la inclusión automática de las entidades relacionadas y la definición de una entidad sin clave para la tabla de relación `BookGenre`.



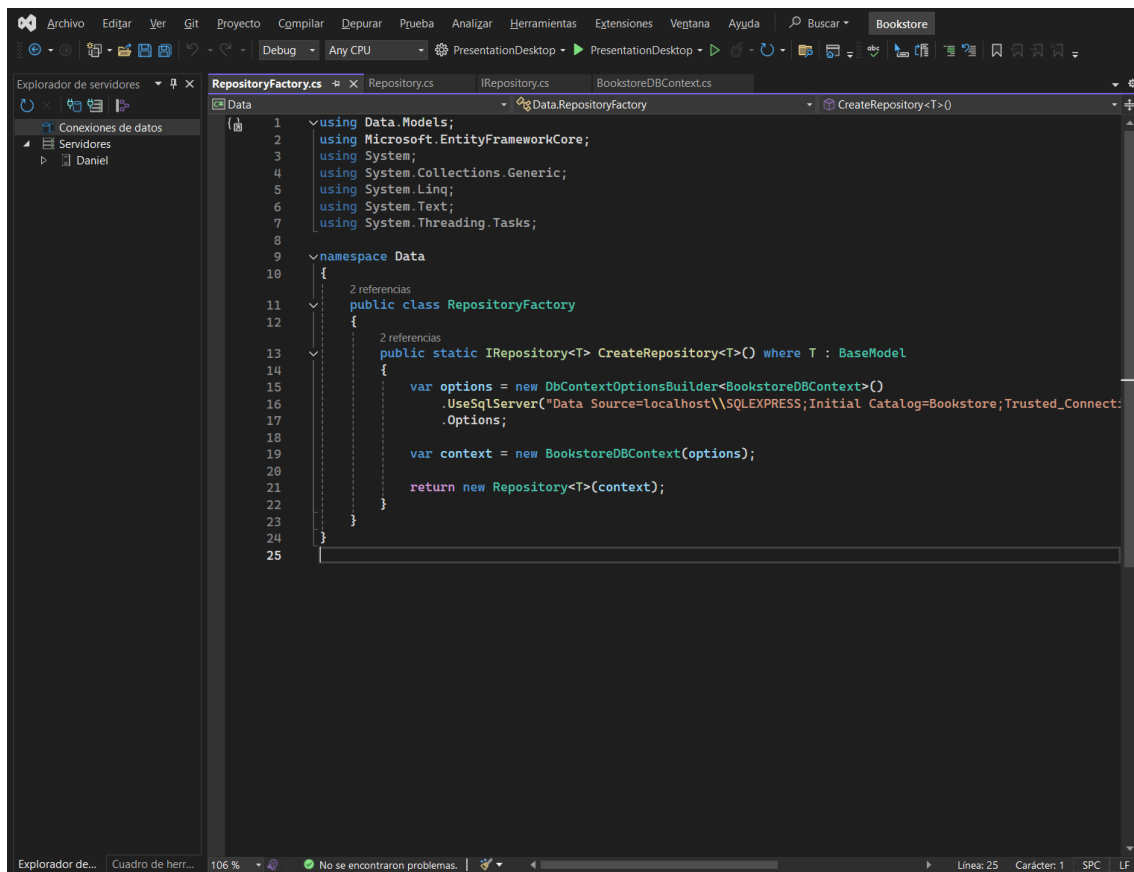
También se tiene la clase `IRepository<T>` define un repositorio genérico para manejar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en una base de datos, utilizando Entity Framework. Proporciona propiedades para acceder a una consulta de tipo `IQueryable<T>`, un conjunto de entidades `DbSet<T>`, y el contexto de la base de datos `BookstoreDBContext`. Incluye métodos para obtener todos los registros, obtener un registro por su ID, agregar, actualizar y eliminar entidades.



La clase Repository<T> implementa la interfaz IRepository<T> para proporcionar operaciones CRUD genéricas sobre entidades que heredan de BaseModel utilizando Entity Framework. Utiliza un contexto de base de datos BookstoreDBContext para manejar las operaciones.

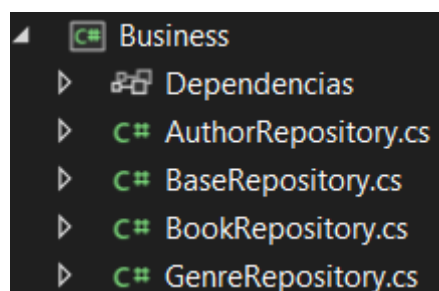


La clase RepositoryFactory proporciona un método estático CreateRepository<T>() que crea y devuelve un repositorio específico para una entidad genérica T, que debe heredar de BaseModel. Utiliza Entity Framework Core para establecer una conexión a una base de datos SQL Server local (localhost\\SQLEXPRESS) con una base de datos llamada Bookstore. El método construye un contexto de base de datos BookstoreDBContext con las opciones de configuración proporcionadas y luego instancia un objeto Repository<T> utilizando este contexto, que implementa la interfaz IRepository<T> para manejar operaciones CRUD en la entidad T.



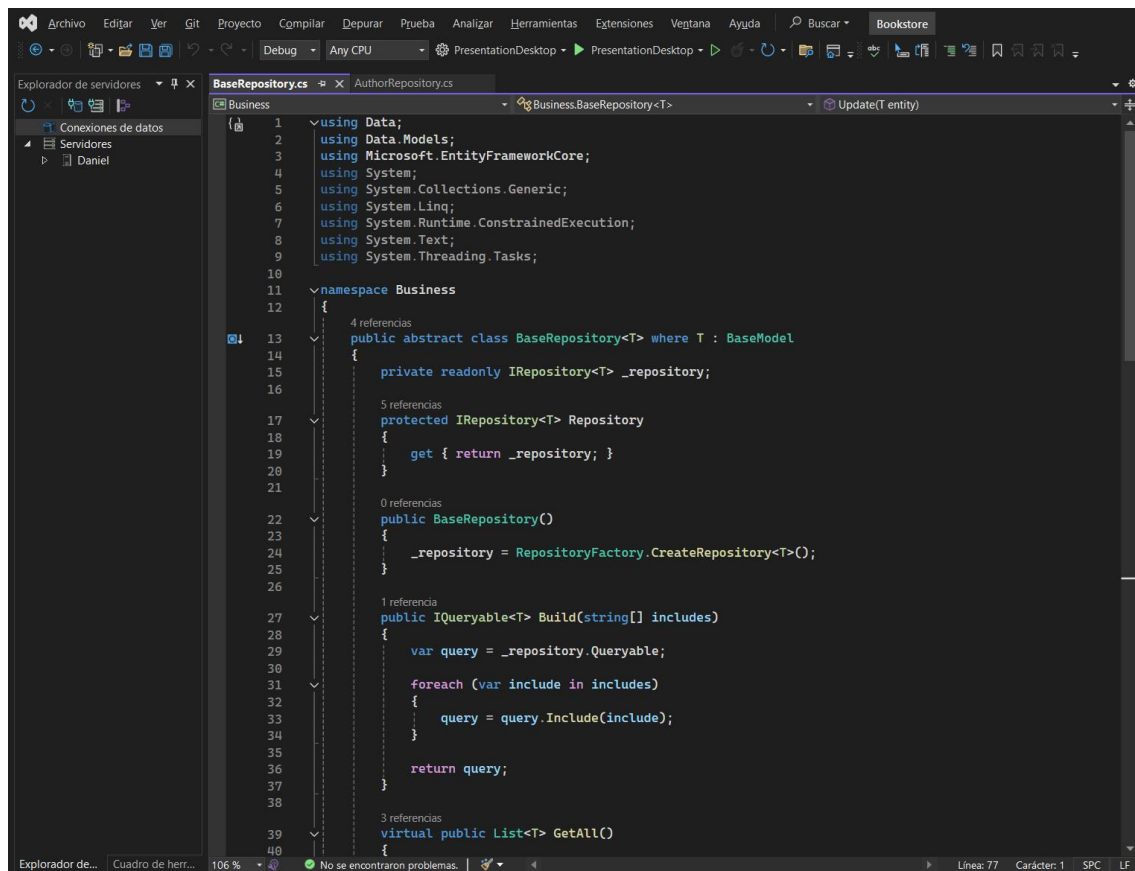
Capa de negocio

Consecuentemente, dentro de la capa de negocio, Business, se tiene las siguientes clases:



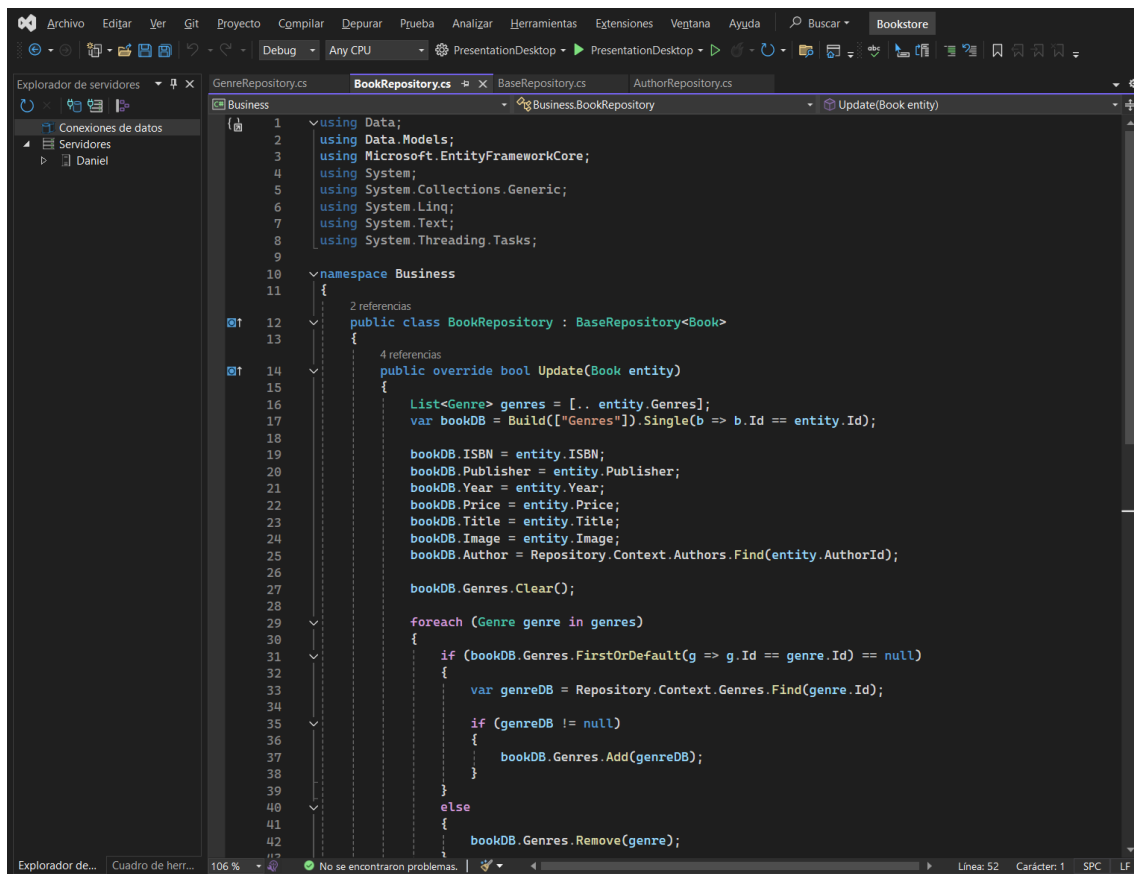
Dentro de la clase BaseRepository es una abstracción genérica que facilita la interacción con entidades del tipo T mediante métodos CRUD básicos. Utiliza un repositorio genérico `IRepository<T>` obtenido a través de `RepositoryFactory.CreateRepository<T>()`, permitiendo operaciones como obtener todos los registros (`GetAll()`), buscar por ID (`FindById()`), crear (`Create()`), actualizar

(Update()), y eliminar (Delete()), proporcionando flexibilidad y reusabilidad en el manejo de entidades a través de Entity Framework Core.



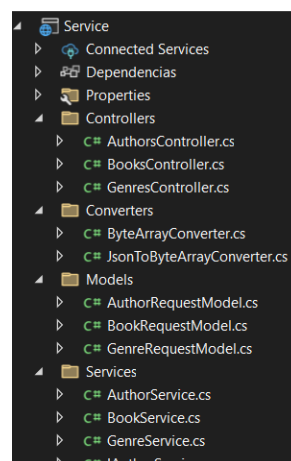
Dentro de la clase BookRepository extiende BaseRepository<Book> y sobrescribe el método Update(Book entity) para actualizar un libro en particular en la base de datos. Primero, obtiene los géneros asociados al libro y luego busca y actualiza el libro específico utilizando el método Build(["Genres"]).Single(b => b.Id == entity.Id) para incluir los géneros relacionados. Luego, actualiza las propiedades del libro con los valores proporcionados por entity, incluyendo ISBN, editor, año, precio, título e imagen. Asocia al libro el autor correspondiente utilizando Repository.Context.Authors.Find(entity.AuthorId) y limpia la colección de géneros del libro con bookDB.Genres.Clear(). Después, itera sobre los géneros proporcionados para actualizar la colección de géneros del libro, añadiendo géneros nuevos y eliminando los que ya no están asociados. Finalmente, adjunta el libro actualizado al contexto, marca su

estado como modificado y guarda los cambios en la base de datos, devolviendo true si la operación de actualización fue exitosa.

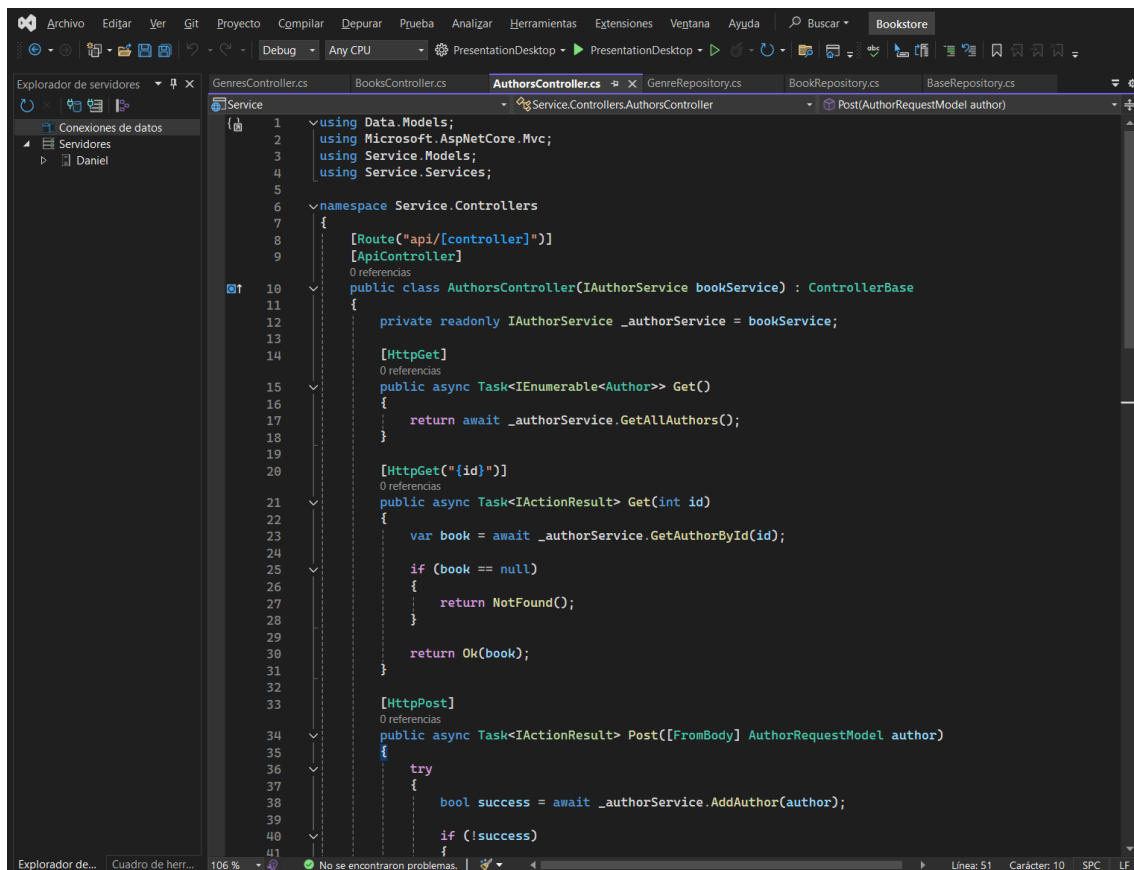


Capa de Servicio

Dentro de la capa de servicio, se estructuró en controladores, convertidores, modelos y servicios, todo ello para que esté más organizado y poder utilizarlo de buena manera a lo largo del proyecto.



Por ejemplo, el controlador `AuthorsController` implementa operaciones CRUD básicas para la gestión de autores a través de una API RESTful. Utiliza inyección de dependencias para obtener un servicio de autores (`IAuthorService`), permitiendo así la recuperación de todos los autores, la obtención de uno específico por ID, la creación de nuevos autores, la actualización de datos de autores existentes y la eliminación de autores. Cada método HTTP (GET, POST, PUT, DELETE) se encarga de manejar las solicitudes correspondientes y responder con los códigos de estado apropiados junto con los datos o mensajes relevantes, asegurando una gestión eficiente y estructurada de la información sobre autores dentro del sistema.



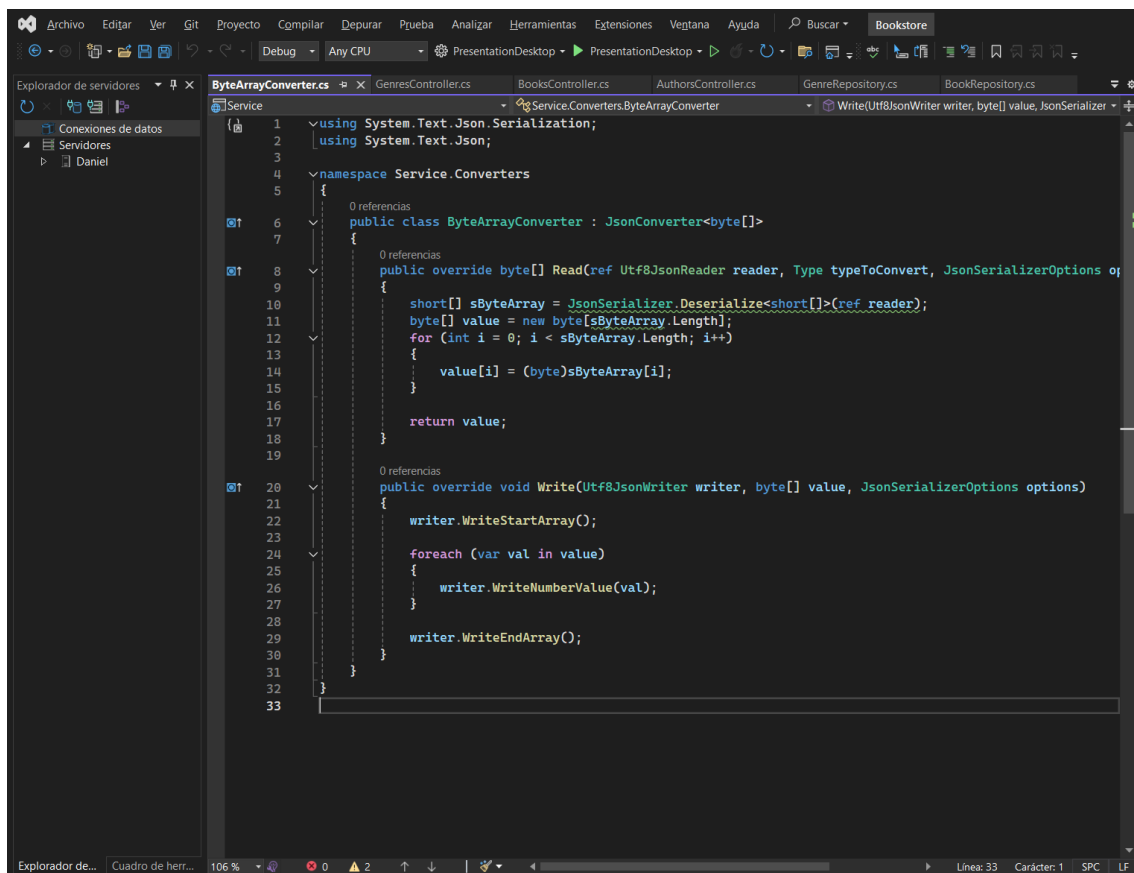
```

1  using Data.Models;
2  using Microsoft.AspNetCore.Mvc;
3  using Service.Models;
4  using Service.Services;
5
6  namespace Service.Controllers
7  {
8      [Route("api/[controller]")]
9      [ApiController]
10     public class AuthorsController(IAuthorService bookService) : ControllerBase
11     {
12         private readonly IAuthorService _authorService = bookService;
13
14         [HttpGet]
15         public async Task<IEnumerable<Author>> Get()
16         {
17             return await _authorService.GetAllAuthors();
18         }
19
20         [HttpGet("{id}")]
21         public async Task<ActionResult> Get(int id)
22         {
23             var book = await _authorService.GetAuthorById(id);
24
25             if (book == null)
26             {
27                 return NotFound();
28             }
29
30             return Ok(book);
31         }
32
33         [HttpPost]
34         public async Task<ActionResult> Post([FromBody] AuthorRequestModel author)
35         {
36             try
37             {
38                 bool success = await _authorService.AddAuthor(author);
39
40                 if (!success)
41                 {
42                     return BadRequest();
43                 }
44             }
45             catch { }
46         }
47     }

```

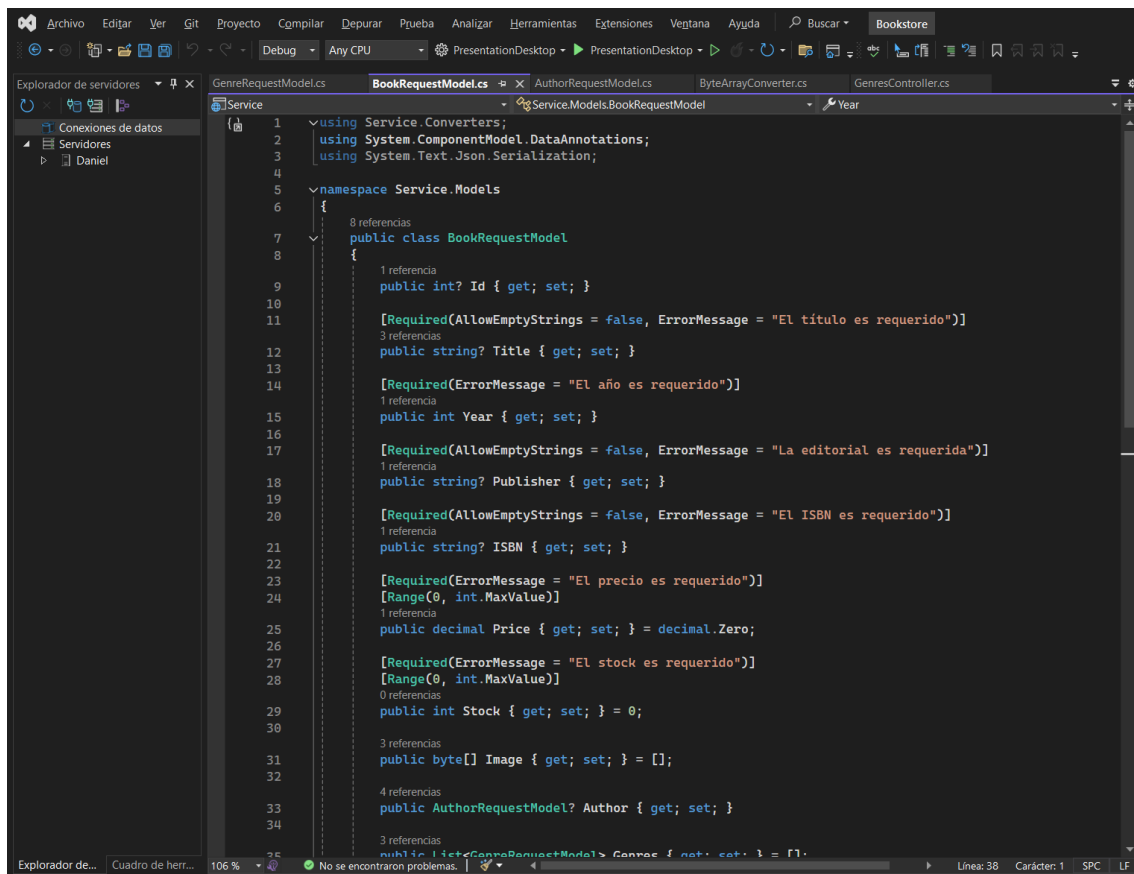
Dentro de la carpeta `Converters` se tiene la clase `ByteArrayConverter` implementa `JsonConverter<byte[]>` para facilitar la serialización y deserialización personalizada de arreglos de bytes utilizando `System.Text.Json`. El método `Read` convierte un arreglo de enteros cortos (`short[]`) deserializado desde JSON a un arreglo de bytes (`byte[]`). Por

otro lado, el método `Write` serializa un arreglo de bytes a JSON, escribiendo cada byte como un valor numérico en un arreglo JSON. Esta clase es útil cuando se necesita manejar datos binarios, como imágenes o archivos, en aplicaciones que utilizan `System.Text.Json` para el intercambio de datos estructurados en formato JSON, proporcionando una forma personalizada de convertir entre bytes y JSON de manera eficiente y controlada.



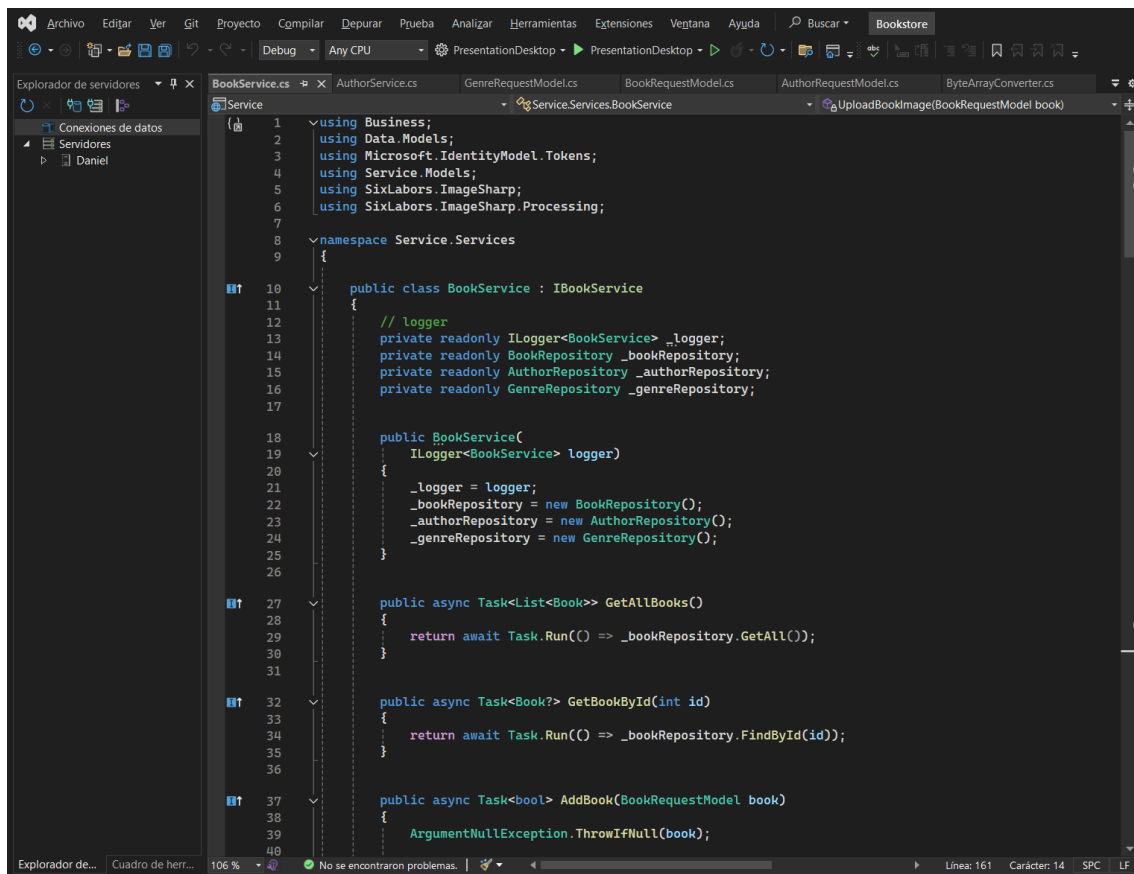
Asimismo, en `Models`, se tiene la clase `BookRequestModel` en el espacio de nombres `Service.Models` define un modelo de solicitud que encapsula los datos esenciales para la creación o actualización de libros dentro de una aplicación. Incluye propiedades como `Id`, `Title`, `Year`, `Publisher`, `ISBN`, `Price`, `Stock`, `Image`, `Author`, y `Genres`, cada una con validaciones específicas utilizando atributos como `[Required]` para asegurar que los campos obligatorios no estén vacíos y `[Range]` para garantizar valores dentro de rangos específicos. Esto permite una representación estructurada de los datos del libro,

facilitando su manejo y validación en operaciones CRUD, asegurando consistencia y precisión en la interacción con los datos de libros en la aplicación.



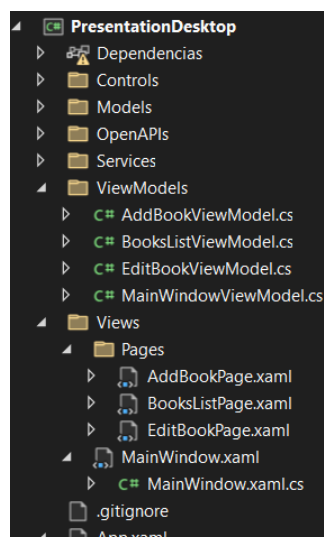
Finalmente, dentro de Servicios se tiene diferentes clases que tienen la misma funcionalidad, por ejemplo, la clase BookService en el espacio de nombres Service.Services implementa la lógica de negocio para la gestión de libros dentro de una aplicación. Utiliza repositorios (BookRepository, AuthorRepository, GenreRepository) para interactuar con la capa de datos y realizar operaciones como obtener todos los libros, buscar un libro por ID, añadir, actualizar y eliminar libros. El servicio valida la integridad de los datos de entrada, como asegurarse de que el título no esté vacío y que al menos una categoría esté asociada al libro antes de procesar las operaciones. Además, gestiona la actualización de imágenes de libros, redimensionándolas y almacenándolas localmente en el sistema de archivos. Este servicio encapsula la lógica compleja

relacionada con los libros, asegurando un manejo eficiente y consistente de la información dentro de la aplicación.



Capa de Presentación

En la capa de presentación, se tiene primero la aplicación de escritorio, la cual está estructurada de la siguiente manera:



En esta estructura, se pretende facilitar la interacción entre la interfaz de usuario y la lógica de negocio en una aplicación de escritorio WPF utilizando el patrón MVVM.

Por ejemplo, en la clase AddBookViewModel se centraliza la lógica para agregar libros, gestionando la carga de datos de autores y géneros desde una API

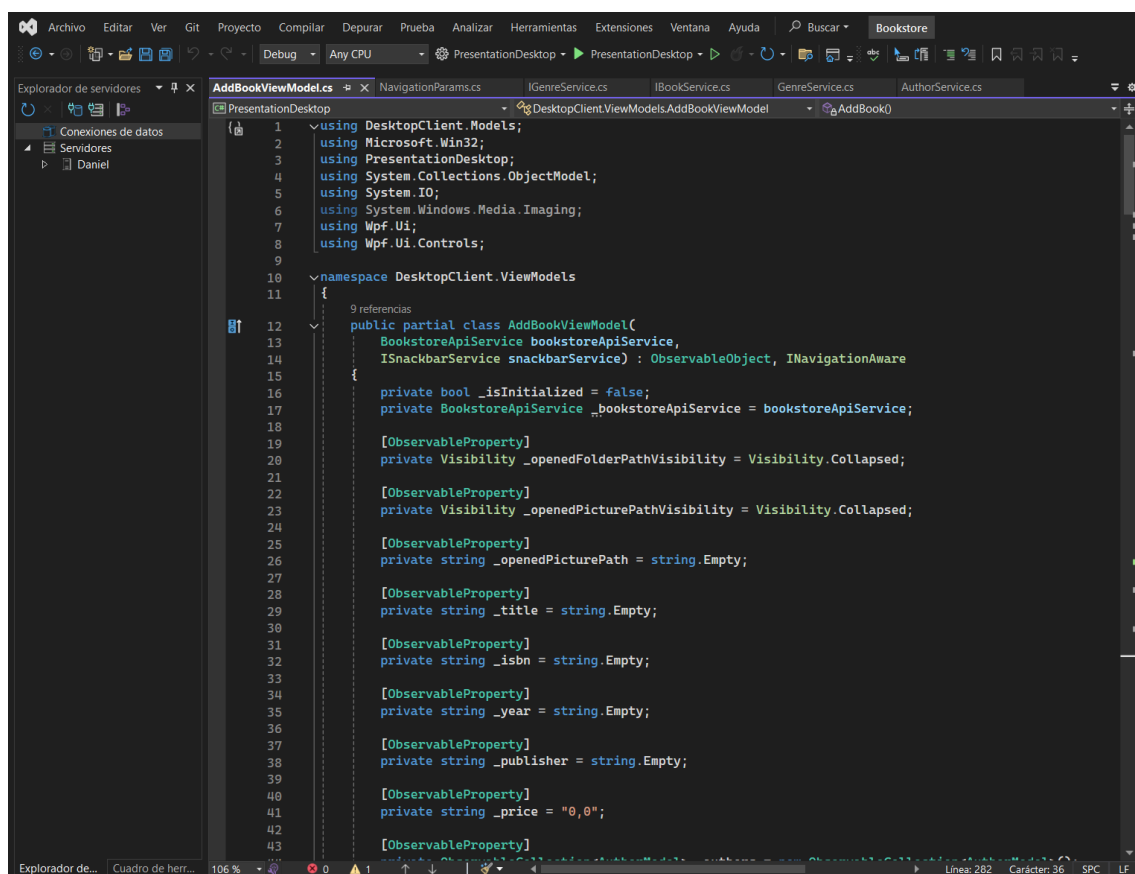
(BookstoreApiService), así como la validación de campos obligatorios como título, año, autor y género. Utiliza propiedades observables para enlazar datos con la vista y

comandos (RelayCommand) para manejar acciones como la selección de imágenes,

envío de datos al servidor y gestión de errores mediante mensajes (snackbarService). En

conjunto, AddBookViewModel facilita una estructura organizada y mantenible que

mejora la separación de responsabilidades y la reactividad de la interfaz de usuario en la aplicación WPF.

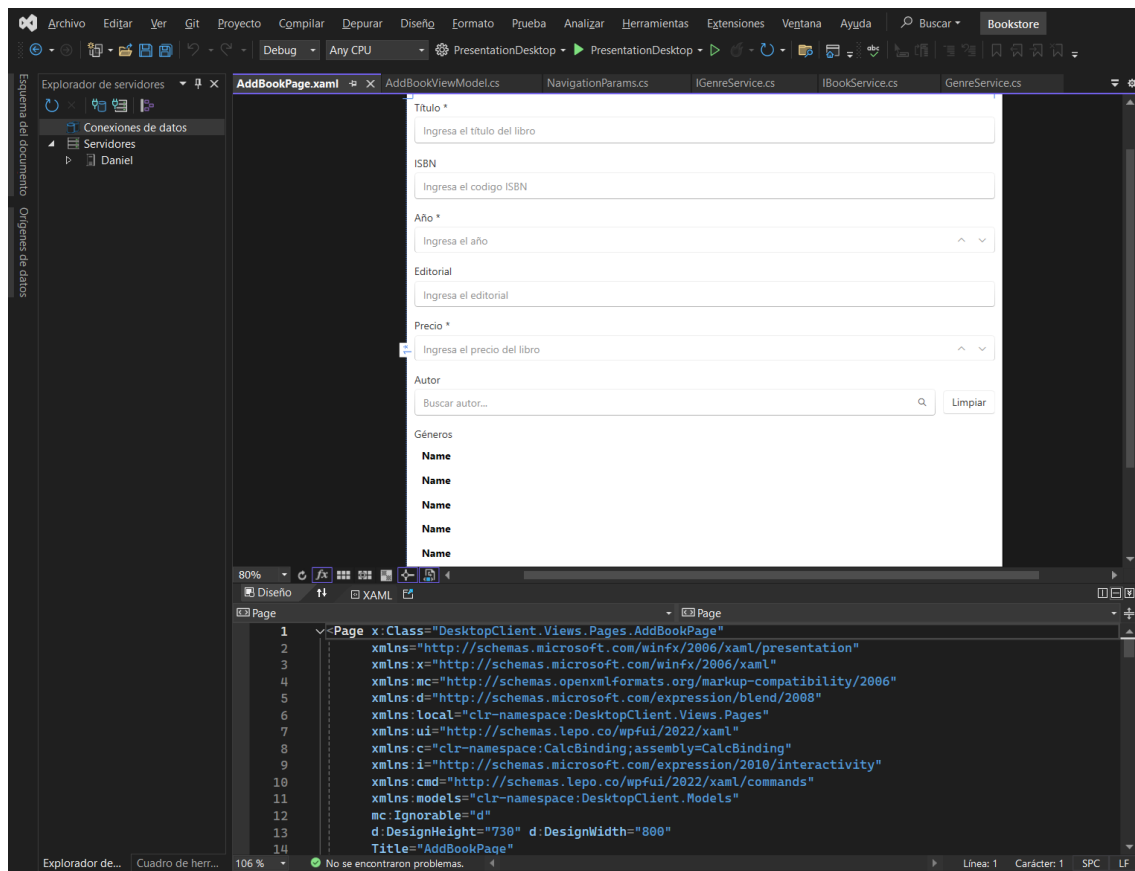


```

1  using DesktopClient.Models;
2  using Microsoft.Win32;
3  using PresentationDesktop;
4  using System.Collections.ObjectModel;
5  using System.IO;
6  using System.Windows.Media.Imaging;
7  using Wpf.Ui;
8  using Wpf.Ui.Controls;
9
10 namespace DesktopClient.ViewModels
11 {
12     9 referencias
13     public partial class AddBookViewModel(
14         BookstoreApiService bookstoreApiService,
15         ISnackbarService snackbarService) : ObservableObject, INavigationAware
16     {
17         private bool _isInitialized = false;
18         private BookstoreApiService _bookstoreApiService = bookstoreApiService;
19
20         [ObservableProperty]
21         private Visibility _openedFolderPathVisibility = Visibility.Collapsed;
22
23         [ObservableProperty]
24         private Visibility _openedPicturePathVisibility = Visibility.Collapsed;
25
26         [ObservableProperty]
27         private string _openedPicturePath = string.Empty;
28
29         [ObservableProperty]
30         private string _title = string.Empty;
31
32         [ObservableProperty]
33         private string _isbn = string.Empty;
34
35         [ObservableProperty]
36         private string _year = string.Empty;
37
38         [ObservableProperty]
39         private string _publisher = string.Empty;
40
41         [ObservableProperty]
42         private string _price = "0,0";
43         [ObservableProperty]

```

Y también se tienen las diferentes interfaces del programa, por ejemplo, el formulario para agregar un nuevo libro.



Fase de pruebas

En la fase de pruebas del proyecto de software como el descrito, se implementarían varios tipos de pruebas para garantizar la calidad y el funcionamiento adecuado del sistema.

Prueba de Funcionalidad del Repositorio

- **Propósito:** Verificar que el repositorio pueda recuperar correctamente todos los autores almacenados en la base de datos.
- **Descripción:** Se utiliza el método `repo.GetAll<Author>()` para obtener todos los autores del repositorio. Luego, se itera sobre cada autor para mostrar su nombre completo en la consola.

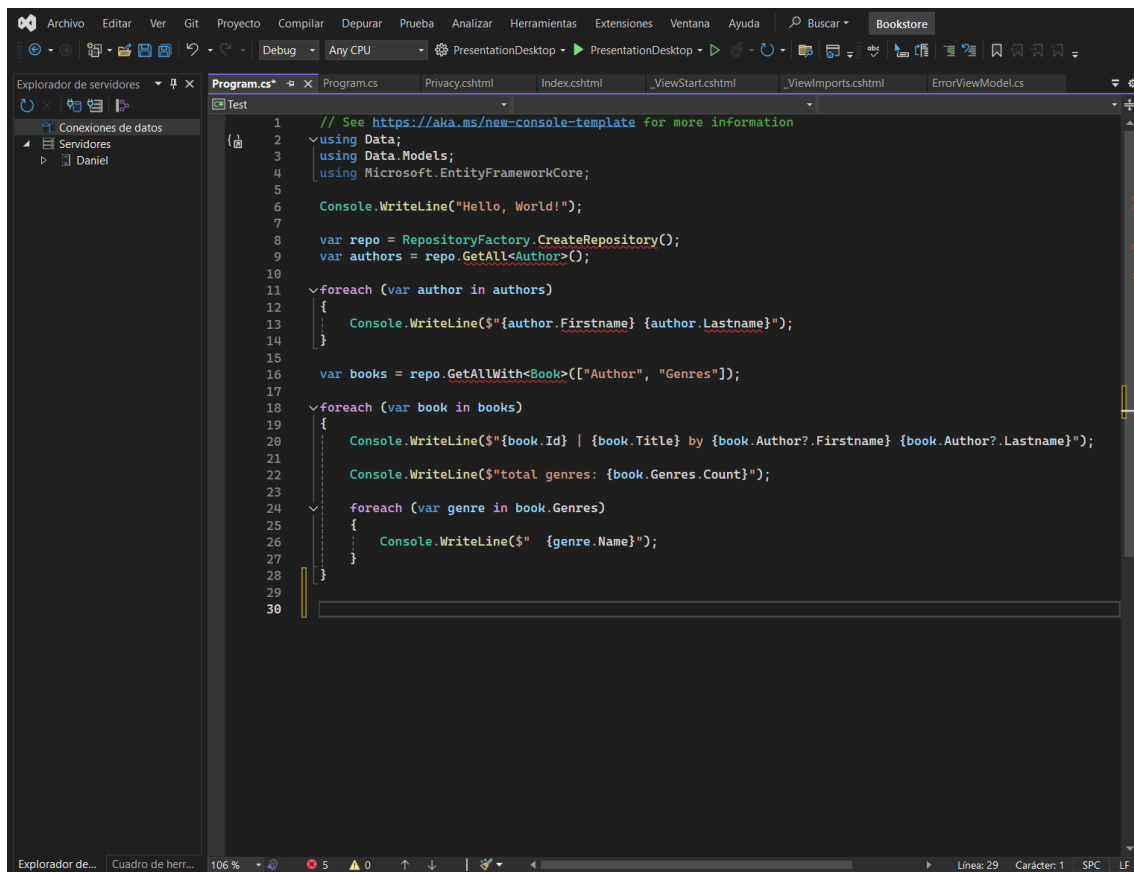
- **Resultado Esperado:** Se espera que todos los autores almacenados se recuperen y se muestren correctamente en la consola.

Prueba de Funcionalidad del Repositorio

- **Propósito:** Verificar que el repositorio pueda recuperar correctamente todos los libros junto con sus autores y géneros relacionados desde la base de datos.
- **Descripción:** Se utiliza el método `repo.GetAllWith<Book>(["Author", "Genres"])` para obtener todos los libros del repositorio junto con las entidades relacionadas de autores y géneros. Luego, se itera sobre cada libro para mostrar su título, el nombre completo del autor y la lista de géneros asociados en la consola.
- **Resultado Esperado:** Se espera que todos los libros, junto con sus relaciones de autor y géneros, se recuperen y se muestren correctamente en la consola.

Prueba de Integración Básica

- **Propósito:** Verificar la interacción correcta entre el código de la aplicación y la base de datos.
- **Descripción:** Se ejecutan consultas directas al repositorio para obtener datos reales de la base de datos. Esto incluye la verificación de que las relaciones entre entidades (como autor-libro y libro-género) se carguen correctamente.
- **Resultado Esperado:** Se espera que las consultas a la base de datos se realicen con éxito y que los datos recuperados reflejen correctamente las relaciones definidas en el modelo de datos.



```
1 // See https://aka.ms/new-console-template for more information
2 using Data;
3 using Data.Models;
4 using Microsoft.EntityFrameworkCore;
5
6 Console.WriteLine("Hello, World!");
7
8 var repo = RepositoryFactory.CreateRepository();
9 var authors = repo.GetAll<Author>();
10
11 foreach (var author in authors)
12 {
13     Console.WriteLine($"{author.Firstname} {author.Lastname}");
14 }
15
16 var books = repo.GetAllWith<Book>(["Author", "Genres"]);
17
18 foreach (var book in books)
19 {
20     Console.WriteLine($"{book.Id} | {book.Title} by {book.Author?.Firstname} {book.Author?.Lastname}");
21     Console.WriteLine($"total genres: {book.Genres.Count}");
22
23     foreach (var genre in book.Genres)
24     {
25         Console.WriteLine($" {genre.Name}");
26     }
27 }
28
29
30
```

Conclusiones

La arquitectura está claramente estructurada utilizando principios de separación de capas: datos, negocio y servicio. Esto promueve la modularidad y escalabilidad del código, facilitando futuras expansiones y mantenimiento del sistema.

El uso de Entity Framework Core para la persistencia de datos muestra una decisión acertada, aprovechando las capacidades de ORM para simplificar las operaciones de base de datos y mejorar la portabilidad entre diferentes proveedores de bases de datos relacionales. Además, la implementación de repositorios genéricos y específicos demuestra una metodología coherente para el acceso a datos, promoviendo la reutilización del código y la coherencia en la aplicación.

La integración de servicios y la lógica de negocio en los controladores y modelos de vista del lado del servidor, junto con la interfaz gráfica del cliente, demuestra un enfoque completo hacia la usabilidad y la experiencia del usuario final. La implementación de pruebas en varios niveles, incluyendo pruebas unitarias y funcionales, es fundamental para asegurar la calidad del software, garantizando que todas las funcionalidades se comporten como se espera antes de ser implementadas en producción.

Finalmente, la inclusión de gestión de errores, manejo de excepciones y logging muestra un compromiso con la robustez y la confiabilidad del sistema, asegurando que cualquier problema pueda ser identificado y resuelto eficientemente.

Recomendaciones

Asegúrate de documentar exhaustivamente el código, especialmente las decisiones arquitectónicas y el funcionamiento de los módulos clave. Esto facilitará el mantenimiento futuro y la integración de nuevos desarrolladores al equipo.

Implementa medidas adicionales de seguridad, como la validación de entradas tanto en el lado del cliente como del servidor. Esto ayudará a prevenir vulnerabilidades comunes como inyecciones SQL y ataques XSS.

Considera centralizar el manejo de errores utilizando técnicas como middleware de excepciones o interceptores. Esto mejorará la coherencia en el manejo de errores y simplificará la resolución de problemas.

Implementa un sistema de logging robusto que registre adecuadamente las acciones del usuario y los eventos del sistema. Esto facilitará la depuración y el análisis de problemas en producción.

Bibliografía

Smith, J. (2023). *Agile Software Development: Principles, Patterns, and Practices*. Pearson Education. Recuperado de <https://www.pearson.com/us/higher-education/program/Martin-Agile-Software-Development-Principles-Patterns-and-Practices/PGM114010.html>

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Recuperado de <https://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612>

Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall. Recuperado de <https://www.pearson.com/us/higher-education/program/Martin-Clean-Code-A-Handbook-of-Agile-Software-Craftsmanship/PGM103703.html>

Bass, L., Clements, P., & Kazman, R. (2012). *Software Architecture in Practice* (3rd ed.). Addison-Wesley. Recuperado de <https://www.informit.com/store/software-architecture-in-practice-9780321815736>

Hunt, A., & Thomas, D. (1999). *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley. Recuperado de <https://www.amazon.com/Pragmatic-Programmer-Journeyman-Master/dp/020161622X>

Freeman, E., Robson, E., Bates, B., Sierra, K., & Kathy, S. (2004). *Head First Design Patterns*. O'Reilly Media. Recuperado de <https://www.oreilly.com/library/view/head-first-design/0596007124>

Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley. Recuperado de <https://www.amazon.com/Patterns-Enterprise-Application-Architecture-Martin/dp/0321127420>

McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction* (2nd ed.). Microsoft Press. Recuperado de <https://www.microsoftpressstore.com/store/code-complete-9780735619678>

Pressman, R. S. (2014). *Software Engineering: A Practitioner's Approach* (8th ed.). McGraw-Hill Education. Recuperado de <https://www.mheducation.com/highered/product/software-engineering-practitioner-s-approach-pressman/9780078022128.html>

Sommerville, I. (2016). *Software Engineering* (10th ed.). Pearson Education. Recuperado de <https://www.pearson.com/store/p/software-engineering/P100000813927>