Dia 1

**Funciones del sistema final**

* Sistema de ventas con ASP.NET Core 6, patrón MVC y SQL Server.
* Funciones incluyen gestión de usuarios, roles, ventas, historial, reportes con exportación Excel y PDF.
* Manejo de roles que determinan acceso a menús y acciones.
* Ejemplo de flujo de venta, detalle y reporte de ventas.
* Opciones para recuperación y cambio de contraseña.

**Contenido general del Proyecto.**

* Programación orientada a objetos en .NET Core.
* Operaciones CRUD con Entity Framework.
* Envío de correos usando Gmail.
* Guardado de archivos en la nube (Firebase Storage).
* Autenticación y acceso con roles.
* Menús dinámicos por rol.
* Generación de reportes PDF y gráficos.

**Creación de base de datos y tablas**

* Creación de base de datos llamada "dbventas".
* Tablas principales:
  + Usuario, Rol, Menu, RolMenu, Categoria, Producto, Venta, DetalleVenta.
  + Configuración para recursos externos (correo, nube).
  + Número correlativo para ventas.
* Población inicial con datos de ejemplo.
* Creación de procedimientos almacenados para productos y categorías.
* Diagrama de base de datos con relaciones.

**Estructura del proyecto**

* Capas: Aplicación Web, Negocio, Acceso a Datos, Entidad, Inversión de Control.
* Uso de Entity Framework Core para facilitar acceso a base de datos.

Dia 2

1. Crear la solución en Visual Studio 2022 y agregar las capas necesarias:
   * Capa de entidades (Entity).
   * Capa de acceso a datos (DAL).
   * Capa de lógica de negocio (BLL).
   * Capa de inyección de dependencias (IOC).
   * Capa web MVC para la interfaz.
2. Instalar y configurar Entity Framework Core en el proyecto:
   * Instalar paquetes EF Core para SQL Server y herramientas EF Core.
   * Crear el contexto de datos (DbContext).
   * Mapear las tablas de la base de datos a entidades con Scaffold-DbContext.
3. Organizar los proyectos y las referencias entre capas para mantener la arquitectura en capas.
4. Configurar la cadena de conexión a la base de datos en appsettings.json y pasarla al DbContext vía inyección de dependencias.
5. Crear y configurar la interfaz de usuario:
   * Crear menús principales: ventas, productos, usuarios.
   * Formularios para iniciar sesión, agregar/editar productos, registros de ventas y gestión de usuarios.
6. Implementar lógica para:
   * Registrar ventas seleccionando productos y documentos asociados.
   * Mantener detalle de ventas con productos, cantidades y precios.
   * Crear, leer, actualizar y eliminar productos y usuarios.
7. Crear la base de datos SQL Server con las tablas necesarias:
   * Categorías, productos, ventas, detalle de ventas, usuarios.
   * Insertar datos iniciales y crear procedimientos almacenados para CRUD.
8. Se probo el sistema ejecutando la aplicación desde Visual Studio y verificar que se puedan realizar operaciones sin errores.

Dia 3

1. Descargar y descomprimir una carpeta con plantillas HTML y recursos (estilos CSS, imágenes, scripts JS).
2. Limpiar la carpeta wwwroot del proyecto y copiar las carpetas de estilos, imágenes y scripts de la plantilla descargada a wwwroot, para que estén disponibles en el proyecto.
3. Abrir la vista \_Layout.cshtml (la master page de Razor que comparte estructura y estilos para todas las vistas).
4. Reemplazar el contenido del layout con el contenido de la plantilla HTML que se quiere integrar. Se tiene cuidado de mantener las instrucciones de Razor importantes como @RenderBody() para que las vistas se rendericen correctamente dentro del layout.
5. Se ajustan las rutas de los recursos (CSS, JS, imágenes) en el layout usando el símbolo "~/" para apuntar a la raíz wwwroot y que las referencias funcionen bien en el proyecto.
6. Se crean controladores en ASP.NET Core MVC para diferentes secciones del sistema (Dashboard, Usuarios, Negocio, Categoría, Producto, Venta, Reportes).
7. Para cada controlador, se crean vistas Razor (vistas del revisor) y se integran las secciones HTML copiadas de las plantillas (contenido de container-fluid, scripts, estilos) en las respectivas vistas.
8. Se enlazan los menús del layout a los controladores y vistas adecuados para que la navegación funcione.
9. Se prueba el proyecto ejecutándolo para verificar que los estilos, scripts y menús funcionan y se visualizan correctamente sin errores.

Dia 4

1. Se crea la vista "Perfil" para el usuario, integrando su plantilla HTML con estilos y scripts dentro del layout del sistema.
2. Implementación de una interfaz genérica IRepository<T> que define operaciones comunes para todas las entidades (obtener, crear, editar, eliminar, consultar) usando expresiones lambda para filtros.
3. Se crea la clase GenericRepository<T> que implementa IGenericRepository<T> con lógica asíncrona para interactuar con el DbContext de Entity Framework Core, utilizando métodos como AddAsync, Update, Remove y consultas usando LINQ.
4. Se desarrolla un repositorio específico para la gestión de ventas (IventaRepository), que hereda del genérico pero añade métodos específicos para registrar ventas con transacciones y obtener reportes.
5. En la implementación de VentaRepository se maneja la lógica de transacciones para asegurar que si alguna operación falla durante el registro de ventas, se hace rollback para mantener la integridad de datos.
6. El método de registro de ventas también actualiza el stock de productos, genera número correlativo para las ventas y almacena datos con EF Core.
7. Se crea un método para generar reportes que incluye detalles como productos, usuarios y tipo de documentos, utilizando consultas con Include para cargar relaciones.
8. Finalmente, se integra esta lógica en la inyección de dependencias para ser utilizada en la capa de negocio y la capa web.

Dia 5

**Configuración del Servicio de Correo**

* Crear una cuenta de Gmail para usarla como servicio de envío de correos desde la aplicación.
* Activar la verificación en dos pasos en Google para esta cuenta y generar una contraseña específica para la aplicación.
* Esta contraseña especial será usada para autenticarse en el servicio SMTP de Gmail y poder enviar correos desde la aplicación sin utilizar la contraseña principal.

**Inserción de Configuración en la Base de Datos**

* Se agregan a la tabla de configuración en la base de datos las credenciales necesarias para el servicio de correo:
  + correo electrónico,
  + contraseña generada para aplicación,
  + alias,
  + host SMTP de Gmail,
  + puerto SMTP,
* Estos datos son esenciales para que la aplicación pueda enviar correos correctamente.

**Creación de la Interfaz del Servicio de Correo**

* Se crea la interfaz ICorreoService en la capa de negocio con un método EnviarCorreo que recibe:
  + correo destino,
  + asunto,
  + cuerpo del mensaje.

**Implementación del Servicio de Correo**

* La clase CorreoService implementa la interfaz y gestiona la lógica para enviar correos con SMTP usando System.Net.Mail.
* Se extraen las configuraciones desde la base de datos y se almacenan en un diccionario para fácil acceso.
* Se construye el correo electrónico configurando remitente, destinatario, asunto y cuerpo en formato HTML.
* Se configura el cliente SMTP con host, puerto, credenciales y protocolo de seguridad SSL.
* Finalmente, se envía el correo de manera asíncrona y se maneja cualquier excepción retornando false en caso de error.

**Integración y Pruebas**

* Se añade la inyección del servicio en la clase de dependencias para que esté disponible para el resto de la aplicación.
* Se ejecuta y compila el proyecto para asegurar que todo funciona correctamente.

Dia 6

**Configuración e Implementación del Servicio de Firebase Storage**

* Crear un proyecto en Firebase para usar su servicio de almacenamiento en la nube, necesario para subir archivos multimedia.
* Se configura el proyecto en Firebase Console, incluyendo la selección de región y creación de un bucket para almacenamiento.
* Se ajustan las reglas de seguridad para permitir lectura y escritura solo a usuarios autenticados.
* Se habilita la autenticación por correo electrónico y se crea un usuario desde Firebase para pruebas del servicio.

**Inserción de Configuración en la Base de Datos**

* Se prepara la tabla de configuración de la base de datos para almacenar las credenciales necesarias:
  + correo,
  + clave,
  + URL del bucket,
  + API key de Firebase.
* Este paso es fundamental para que el sistema pueda conectarse al servicio Firebase Storage con las credenciales correctas.

**Instalación de Paquetes NuGet para Firebase Storage**

* Se instalan los paquetes necesarios:
  + Google.Apis.Auth
  + Firebase.Storage.net
* Estos paquetes permiten la integración de la aplicación ASP.NET Core 6 con Firebase para operaciones de almacenamiento.

**Implementación del Servicio para Firebase Storage**

* Se crea una interfaz IFirebaseStorageService con métodos para:
  + Subir un archivo a Firebase Storage,
  + Eliminar un archivo del almacenamiento.
* La clase que implementa esta interfaz (FirebaseStorageService) incluye:
  + Inyección de repositorio genérico para configuración,
  + Lógica para autenticarse con Firebase mediante el JSON del proyecto y credenciales,
  + Uso de la API de Firebase para subir archivos y retornar la URL donde se puede acceder a dicho archivo,
  + Método para eliminar archivos de Firebase Storage.

**Manejo de Excepciones y Retorno**

* Los métodos se colocan dentro de bloques try-catch para manejar posibles errores,
* Se retorna la URL generada para el archivo subido o un valor booleano indicando éxito o falla en eliminación.

**Integración y Pruebas**

* Se inyecta el servicio Firebase Storage en la capa de dependencias para que pueda ser usado donde se necesite.
* Finalmente se ejecuta y compila el proyecto para verificar que no hay errores y que la integración funciona correctamente.

Dia 7

**Implementación del Servicio de Utilidades**

* Se crea una interfaz llamada IUtilidadesService en la carpeta de interfaces con dos métodos:
  + GenerarClave: genera una clave aleatoria para el usuario.
  + ConvertirSha256: recibe un texto y devuelve su encriptación SHA-256.
* Se implementa esta interfaz en la clase UtilidadService donde:
  + GenerarClave genera una clave de 6 caracteres alfanuméricos aleatorios.
  + ConvertirSha256 utiliza la clase SHA256 para encriptar el texto, convirtiéndolo a un formato hexadecimal.

**Implementación del Servicio de Roles**

* Se crea la interfaz IRolService con un método Listar que retorna una lista de roles asincrónicamente.
* La clase RolService implementa esta interfaz y utiliza un repositorio genérico para obtener la lista de roles desde la base de datos.
* Este servicio se inyecta mediante dependencia para ser utilizado en diferentes partes del sistema.

**Plantillas HTML para Correos Electrónicos**

* Se crean vistas Razor para enviar correos con dos plantillas:
  + Envío de contraseña: para notificar al usuario su clave al crearse su cuenta.
  + Restablecimiento de contraseña: para cuando el usuario solicite cambiar su clave.
* Se utilizan ViewData para pasar información dinámica (correo, clave, url) a las vistas.
* Las plantillas incluyen formateo HTML para el cuerpo del correo, con enlaces y mensajes amigables.

**Respuesta Genérica para Solicitudes HTTP**

* Se crea una carpeta de utilidades y dentro una clase GenericResponse<T>.
* La clase define propiedades como:
  + Estado (bool) para indicar éxito o fracaso,
  + Mensaje para enviar información adicional,
  + Objeto para devolver un objeto genérico,
  + Lista que puede contener una lista genérica de objetos.
* Esta clase se prepara para estandarizar la respuesta a las solicitudes HTTP en toda la aplicación.

Día 8

**1. Introducción al servicio de usuario**

* Se investiga la necesidad de encapsular la lógica relacionada con los usuarios en un servicio específico.
* Se prepara el entorno para trabajar con la capa de servicios, siguiendo el patrón de arquitectura limpia.

**2. Creación de la interfaz IUsuarioService**

* Se define la interfaz que contendrá los métodos necesarios para gestionar usuarios.
* Se incluyen métodos como Lista(), que devolverá una lista de usuarios desde la base de datos.

**3. Implementación de la clase UsuarioService**

* Se crea la clase que implementa la interfaz IUsuarioService.
* Se inyecta el contexto de la base de datos (DbContext) para acceder a la tabla de usuarios.
* Se utiliza LINQ para consultar y transformar los datos en objetos DTO (Data Transfer Object).

**4. Configuración en la capa OIC en dependencia**

* Se registra el servicio en el contenedor de dependencias para que pueda ser inyectado en los controladores.
* Se usa AddScoped<IUsuarioService, UsuarioService>() para asegurar una instancia por solicitud.

**5. Pruebas del servicio**

* Se realiza una prueba rápida desde el controlador para verificar que el servicio devuelve correctamente la lista de usuarios.
* Se muestra cómo retornar los datos en formato JSON desde una API RESTful.

**6. Buenas prácticas y estructura**

* Se refuerza el uso de interfaces para desacoplar la lógica del negocio.
* Se promueve la separación de responsabilidades entre capas: controlador, servicio y acceso a datos.

Dia 9

**Creación de ViewModels (BioModels)**

* Se investiga la importancia de utilizar ViewModels para interactuar con las vistas en lugar de usar directamente los modelos de entidad.
* Se crea una carpeta llamada ViewModel donde se agregan clases que representan las propiedades necesarias para cada vista.
* Se hacen ViewModels para entidades como:
  + Rol: solo con ID y descripción necesarias para la vista.
  + Usuario: incluye ID, nombre, correo, teléfono, ID rol, nombre del rol, URL de la foto, y estado como entero (1 o 0).
  + Negocio, Categoría, Producto, Documento de Venta, Venta, Detalle de Venta, Menú, Reporte de Venta, Usuario Login, Cambio de Clave y otros específicos como Ventas por Semana y Dashboard.
* Cada ViewModel contiene solo las propiedades necesarias para la interacción con la vista, muchas veces omitiendo constructores o propiedades irrelevantes.

**Instalación y Configuración de AutoMapper**

* Se instala el paquete NuGet AutoMapper y su extensión AutoMapper.Extensions.Microsoft.DependencyInjection para ASP.NET Core.
* Se crea una carpeta AutoMapper con una clase AutoMapperProfile que hereda de Profile de AutoMapper.
* Dentro del constructor de esta clase se define la configuración para mapear entre modelos y ViewModels.
* Ejemplo para Rol: mapeo de entidad Rol a VmRol y viceversa con .ReverseMap().
* Para Usuario, se configura un mapeo más específico:
  + Se convierte el booleano EsActivo a entero (1 o 0) para el ViewModel, y viceversa.
  + Se mapea también el NombreRol accediendo a la propiedad de navegación IDRolNavigation.Descripcion en la entidad.
  + Se ignoran ciertas propiedades para evitar mapeos innecesarios o problemáticos.
* Se replican configuraciones similares para negociación, categoría, producto, tipo de documento, venta, detalle de venta, menú y reportes, haciendo conversiones específicas como de decimal a string con cultura, y formatos de fecha.

**Integración en el Proyecto**

* Se importa el perfil de AutoMapper en el archivo Program.cs.
* Se registra AutoMapper en los servicios con builder.Services.AddAutoMapper(typeof(AutoMapperProfile));
* Se verifica que el proyecto compila y ejecuta sin errores, confirmando que la configuración funciona correctamente.

Día 10