RESUMEN P3

Linq:

● Nos proporciona comprobaciones de tipo consultas durante la compilación

● Sirve para consultar y guardar datos de diferentes orígenes de datos

● Ejemplo: var people = From p in Personas

Where p.Edad >= 20 && p.Edad <= 30

Select p

Return people

POO:

● Herencia: Heredar los atributos de otra clase. Vamos a poder agregar o modificar su comportamiento

● Encapsulamiento: Modificadores: **TEORÍA ABAJO.**

1. Public (acceso total)

2. Internal (acceso del mismo proyecto)

3. Private (acceso solo la clase)

4. Protected (acceso a la clase y las que heredan)

● Polimorfismo: Sobreescritura de métodos: **TEORÍA ABAJO.**

● Virtual (le decimos que va a ser sobreescrito)

● Override (le decimos que lo sobreescriba)

● Abstracción: Proceso de simplificar un sistema complejo

Interfaz: ● Contrato de la firma de los métodos, se compromete a usar todos los métodos. ● Puede implementar múltiples interfaces ● Cualquier clase que tenga implementada esta interfaz es válida como tipo de retorno.

Mecanismos para definir el modelo de datos en Entity Framework

1- Convenciones del Framework

2- Data Annotations (mapping atributes)

3- Fluent API sobreescribiendo el método OnModelCreating de la clase derivada de context.

¿Qué es una API?

Una API es una interfaz para que programas de software se comuniquen entre ellos y compartan datos bajo diferentes estándares. Pueden ser locales o remotas y pueden tener arquitecturas SOAP o REST (Representational State Transfer). Las API pueden ser públicas o privadas, las privadas requieren una autenticación que se realiza mediante un token.

¿Qué es UML?

Unified Model Language es una herramienta que nos provee una serie de convenciones, estructuras, estándares, reglas para el análisis y diseño orientado a objetos.

¿Qué es UP?

Unified Process es una forma de construir un sistema que introduce una serie de disciplinas y fases en ese proceso y que usa a UML como standard en la elaboración de los artefactos y entregables que este proceso crea y utiliza.

POO

Encapsulación

La encapsulación es la característica de un lenguaje de POO que permite que todo lo referente a un objeto quede aislado dentro de este. Solo se puede acceder a ellos a través de los miembros que la clase proporciona (propiedades y métodos). **MODIFICADORES ARRIBA.**

Abstracción

La abstracción implica que la clase debe representar las características de la entidad hacia el mundo exterior, pero ocultando la complejidad que llevan aparejada. La abstracción está muy relacionada con la encapsulación, pero va un paso más allá pues no sólo controla el

acceso a la información, sino también oculta la complejidad de los procesos que estemos implementando.

Herencia

Una clase que hereda de otra obtiene todos los rasgos de la primera y añade otros nuevos y además también puede modificar algunos de los que ha heredado. A la clase de la que se hereda se la llama clase padre o base, y la clase que hereda se llama clase derivada o hija. La herencia fomenta la reutilización de código.

Polimorfismo

El polimorfismo permite que una clase hija pueda redefinir un método de la clase padre.

ARQUITECTURA POR CAPAS

¿Qué es una capa?

Una capa es un conjunto de “cosas” (ejemplo: conjunto de clases) que tienen cierta responsabilidad. Cuando nos referimos a capas es una abstracción de responsabilidades.

La arquitectura define reglas de cómo deben comunicarse las capas.

Primera regla

Cada capa debe tener una responsabilidad única. Las capas deben estar delimitadas de que se ocupa cada una de ellas.

Segunda regla

Las capas deben respetar una estructura jerárquica estricta. Esto quiere decir que cada capa puede comunicarse sólo con la capa que está debajo suyo, pero **NO** al revés.

Ventajas

- Es fácil testear cada capa por separado, gracias a la separación de responsabilidades.
- Al momento de hacer algún cambio, si se implementó bien la separación de responsabilidades, este cambio solo debe impactar en la capa responsable. Esto se conoce como **DESACOPLE.**

Desventajas

- Si se implementaron varias capas, el rendimiento de la aplicación puede verse afectado.
- Algunas operaciones pueden afectar a todas las capas, demostrando que no existe un desacople completo de estas.

Arquitectura de tres capas.

Esta arquitectura permite desarrollar n número de capas siempre y cuando se respeten las dos reglas. Esta arquitectura propone dividir las capas en 3 específicas. Estas son:

Presentación

Atiende los eventos del cliente y representa los datos para el mismo. Esta sera la capa que vea el cliente.

Lógica de Negocio

En esta capa se encuentra todo lo referido a las reglas a las reglas que se encuentren en el negocio, en pocas palabras, los requerimientos funcionales de nuestro sistema.

Acceso a Datos

Mediante esta capa, podremos obtener o guardar los datos que utilizará nuestra aplicación.

Afirmaciones de las arquitecturas por capas

- Separa de forma clara los roles y responsabilidades de los componentes del sistema.
- Un cambio en una capa superior debería afectar poco o nada a una capa de nivel inferior.

Clean Architecture

En clean architecture, la capa dominio y aplicación son el centro del diseño y se conocen como core del sistema. **En el patrón de diseño Clean architecture, las capas del core (Capa de dominio y aplicación) dependen de abstracciones y no de clases concretas de la capa de Infraestructura.** La capa de dominio contiene lógica empresarial y la capa aplicación posee lógica de negocio. La lógica empresarial se puede compartir entre múltiples sistemas, la lógica de negocio corresponde a un sistema en particular.
El core no depende de cuestiones de infraestructura, al contrario, la infraestructura depende del core.
Todas las dependencias apuntan al core, y el core no tiene dependencia de ninguna capa.

El patrón de arquitectura "Clean architecture" me permite modelar la lógica de negocio de mi aplicación, independizándola de las implementaciones de mecanismos específicos como ser el acceso a base de datos, tecnologías de notificaciones, etc.

Estructura de carpetas

- Domain
    - Entities: Entidades del modelo de dominio.

    - Enums: Enumeraciones.

    - Exceptions: Custom Exceptions.

    - Interfaces (Según criterio del arquitecto): Interfaces de repositorios u otras clases que se podrían compartir con otros sistemas de la organización.

- Application
    - Services: Servicios que orquestan las clases necesarias para cumplir con las request (casos de uso). Dependen de abstracciones y nunca de clases de librerías externas.

    - Models: Dtos tanto para requests como para responses.

    - Interfaces: Interfaces de los servicios de la capa de Application e Infrastructure. También se puede incluir las interfaces de los repositorios.

- Infrastructure
    - Data: Clases para implementar el patrón repositorio. Clases de acceso a datos, como el Context.
        - Migrations: Clases correspondientes a las migraciones y snapshot de la base de datos. (Para entity framework)

    - Services: Clases correspondiente a servicios que cumplen una función específica y generalmente utilizan librerías externas.

- Presentation (Web o API)
    - Controllers: Objetos controllers, que se encargan de recibir la request, hacer el data binding, data validation y mapear el action method correspondiente, para finalmente generar una response.

Web API

Una api se trata de un conjunto de definiciones y protocolos que se utilizan para desarrollar e integrar software de las aplicaciones, permitiendo la comunicación entre dos aplicaciones de software mediante un conjunto de reglas.

Request y Response

Las requests siempre se hacen a un Endpoint, punto de acceso de una API, que procesa dicha solicitud y responde de acuerdo a su implementación interna.

¿Qué contienen el Request y Response?

Estructura HTTP Request

Un Request HTTP se compone de:

- Método: **GET**, **POST**, **PUT**, **DELETE**, **PATCH**, **ETC**.
- Path: La URL que se solicita.
- Protocolo: Contiene HTTP y su versión.
- Headers: Son esquemas de key value que contienen información acerca del Request HTTP y el navegador. La mayoría de los headers son opcionales.
- Body: Si se envía información al servidor a través de los métodos **POST** o **PUT**.

Estructura HTTP Response

Un Response HTTP se compone de:

- Protocolo: Contiene HTTP y su versión.
- Status code: El código de respuesta. Ejemplos: 200 OK, 401 Unauthorized, 404 Not Found, 500 Internal Error, etc.
- Headers: Contienen información acerca del software del servidor.
- Body: Si el servidor devuelve información que no sean headers esta va en el body.

Verbos HTTP

Los verbos HTTP indican que acción deseamos realizar sobre el servidor, estos son **GET**, **POST**, **PUT**, **PATCH**, **DELETE**, **HEAD**, **CONNECT**, **OPTIONS** y **TRACE.**

Los más importantes son:

- **GET:** El método **GET** solicita una representación de un recurso específico. Las peticiones que utilizan este método deben recuperar datos.
- **POST:** El método **POST** se utiliza para enviar una entidad de un recurso en específico. Se utiliza para crear un nuevo recurso.
- **PUT:** El método **PUT** se usa para realizar la modificación total de un recurso.
- **DELETE:** Este método borra un recurso en específico.
- **PATCH:** El método **PATCH** se utiliza para aplicar modificaciones parciales a un recurso.

Los atributos que se utilizan para definir que action method se va a invocar a partir de la request recibida por la API son el método [Route] y el método [HttpGet].

Algunas diferencias entre el método HttpGet y el método HttpPost son las siguientes:

En una petición GET, los parámetros son enviados en la URL de la solicitud, generalmente como parte de la cadena de consulta (query string), visible en la barra de direcciones del navegador. En cambio, en una petición POST, los parámetros son enviados en el cuerpo (body) de la solicitud, ocultos al usuario comúnmente.

Las peticiones GET son idempotentes, mientras que las peticiones POST no lo son.

Los pasos necesarios para manejar una petición HttpPost son los siguientes:

Definir un controlador que herede de la clase ControllerBase y un método decorado con la atributo [HttpPost]. Dentro de este método, se realiza la lógica necesaria y se retorna un objeto ActionResult<T> con el resultado en formato JSON y la ruta de acceso al recurso dentro de un return Created().

Model View Controller (MVC)

MVC es una propuesta de arquitectura de software utilizada para separar código por sus responsabilidades. MVC es útil para cualquier desarrollo en el que intervengan interfaces de usuario.

Capas

Modelos

Capa donde se trabaja con los datos, esta contendrá mecanismos para acceder a la información y también actualizar su estado.

Vistas

Las vistas contienen el código de nuestra aplicación que producirá la visualización de las interfaces de usuario. Generalmente se trabaja con los datos, pero, no se realiza un acceso directo a estos.

Controladores

Los controladores contienen el código necesario para responder a las acciones que se soliciten en la aplicación.

Capa de Datos

Base de Datos Relacional ¿Qué es?

Una base de datos relacional es un sistema de almacenamiento de datos que organiza la información en tablas con filas y columnas. Estas se relacionan entre sí con claves

primarias y foráneas. Algunos ejemplos de bases de datos relacionales son SQL, MySQL, etc.

Este tipo de bases de datos garantizan que múltiples instancias de una base de datos tengan los mismos datos todo el tiempo.

Estructura

La estructura de una base de datos es mediante tablas, estas consisten en columnas (campos) y filas (registros). En estas tablas se pueden gestionar y consultar los datos guardados. Uno de los aspectos más importantes de las bases de datos es que se puede identificar univocamente cada registro que haya en esta mediante las **claves**.
Cada tabla debe tener una **Clave Primaria** (dato que permita identificar un registro en la misma), estas claves pueden ser **simples**, compuesta por un solo atributo, o compuestas, compuestas por más de un atributo. También existen las **Claves Foráneas**, estas permiten que se pueda relacionar un registro de una tabla con otro registro de otra tabla. Estas tambien pueden ser simples o compuestas.

ORM (Object Relational Mapper)

Un ORM es un modelo de programación que se utiliza para mapear estructuras de una base de datos relacional. El objetivo de estas es simplificar y acelerar el desarrollo.

Las estructuras de la Base de datos relacional se vinculan con las entidades lógicas o con la BD virtual que define el ORM, para que las distintas acciones de CRUD se puedan realizar de manera indirecta a través de ORM.

Ejemplo de ORM => Entity FrameWork

Utilidad de un mapeador de objetos relacionales

Su principal funcionalidad es la de agilizar el proceso de programación de la BD, mediante la reducción de código y realizando un mapeo automático.

Pasos para la creación de una BD Relacional

Crear las entidades y plasmarlas en el contexto, configurar Entitiy Framework para MySql y realizar la migración y el update de la base de datos.

Inyeccion de Dependencias

La inyeccion de dependencias es un patron de diseño orientado a objetos en el que se suministran objetos a una clase en lugar de ser la propia clase la que los crea.

Ventajas

Es **Flexible:**

- No hay necesidad de tener un código de búsqueda en la lógica de negocio.

- Elimina el acoplamiento entre módulos.

Es **Testeable:**

- No se necesita un espacio específico de testeo.
- Testeo automático como parte de las construcciones.

Es Mantenible:

- Permite la reutilización de código en diferentes entornos de la aplicación.
- Promueve enfoque coherente en toda la aplicación y equipo.

¿Cuáles son los pasos para realizar la inyección de dependencias?

1- Crear una propiedad privada readonly del tipo de servicio que queremos inyectar.
2- Crear un constructor que reciba un parámetro de la clase que queremos inyectar y asignar dicho parámetro a la propiedad privada
3- Asignarle un ciclo de vida a el servicio a inyectar usando el objeto builder.Services

¿Cual es el propósito principal de la inyección de dependencias?

- Evitar la dependencia entre clases y simplificar la creación y gestión de objetos en una aplicación.

Patron Repository

Diseñado para crear una capa de abstracción entre la capa de acceso a datos y la capa lógica de negocios de una aplicación. La implementación de estos patrones pueden ayudar a aislar la aplicación de cambios en el almacén de datos asi facilitando la realización de pruebas unitarias.

Permite la creación de clases en donde esten definidos los metodos que interactuan con la BD.
Código más limpio y fácil de mantener.

Resuelve la dispersión de la interacción con la BD.

Afirmaciones sobre el patron repository

- El Repository actúa como una capa de acceso a datos que oculta los detalles de la fuente de datos subyacente.
- Facilita la reutilización de código al proporcionar una abstracción entre la lógica de negocio y la capa de acceso a datos.

Autenticación

Codificación

La codificación es una técnica en la que los datos se transforman de un formato a otro, para que estos puedan ser entendidos y utilizados por diferentes sistemas.

Para un atacante es muy fácil decodificar los datos si tienen la información codificada ya que el mismo algoritmo se utiliza para decodificar los datos que se codificaron en primer lugar.

Encriptación

El cifrado hace que los datos sean ilegibles y difíciles de decodificar para un atacante, así evitando que estos sean robados.

Para el cifrado se utilizan las **claves criptográficas.** Con esta clave, los datos se cifran en el extremo del remitente. Con esta clave, o con una distinta, se descifran los datos en el extremo del receptor. El cifrado se clasifica en dos categorías:

- cifrado de **clave simétrica**
- cifrado de **clave asimétrica**

Clave simétrica

La clave simétrica es cuando el remitente y el receptor utilizan la misma clave privada tanto para cifrar como para descifrar los datos enviados/recibidos. El principal problema de este tipo de claves es que, para que el receptor conozca la clave, el remitente debe transmitirle la clave de acceso al receptor, si esto se hace mediante una red no segura existe una alta posibilidad de que esta sea interceptada por un atacante.

Clave asimétrica

La clave asimétrica abarca dos claves, una para el cifrado de los datos, esta clave es de conocimiento público y la otra clave se utiliza para el descifrado de los mismos, esta se conoce como **clave privada** y solo la conocen el remitente y el receptor.

Hash

Un **Hash** es básicamente una cadena que se genera a partir de la cadena de entrada que es pasada a través de un algoritmo hash. El **Hash** es un “cifrado unidireccional”, los datos una vez procesados ya no se pueden revertir a su forma original. Con esto se asegura de que si se modifica alguna cosa, vos estes al tanto de esto. El hasheo se encarga de proteger tus datos contra posibles alteraciones para que tus datos no se modifiquen.

Autenticación JWT

1. El cliente solicita mediante una request a un endpoint de autenticación, el token con el cual va a poder acceder a los servicios de la api. Para esto manda en dicha request las credenciales de autenticación que, en muchos casos, es el usuario y la contraseña.

2. La API verifica que el usuario exista en la base de datos y además verifica si realmente es quien dice ser a través de la confirmación de que la contraseña es la correspondiente para el usuario.

3. La API retorna, como respuesta de la request del paso 1, el token de autenticación que el cliente va a usar a futuro si corresponde. En caso de no reconocer el usuario o de que la validación de las credenciales haya fallado entonces retorna una response con status code 401.

4. Desde ese momento, cada vez que el cliente quiera usar un endpoint de la api va a mandar en la request en un header el token que obtuvo en el paso 3 de la API. Generalmente anteponiendo “Bearer ” Ej: Bearer tokenquerecibiodelaapienelpaso3

5. La api verifica que el token lo haya generado ella y que nadie lo haya alterado. Este proceso de verificación lo veremos mejor en el último título de esta sección.

6. La api procede a procesar la request del cliente normalmente y a responderle lo que corresponda

Estructura de JWT

Un token JWT se compone de 3 partes: el header, el payload y el signature.

Header
El header de un JWT es la que indica que formato criptográfico está utilizando el token. Acá se especifica el algoritmo que se está utilizando en la firma. Teniendo así una mayor seguridad y confidencialidad.

Payload
El payload de un JWT es el que contiene la información del token, codificada en base64, esto permite que el contenido pueda ser visto por cualquiera, pero no es modificable. En el payload se definen las **claims**, estas se transmitiran y existen dos tipos de claims, las obligatorias (ejemplos: **exp** (fecha de expiración), **iat** (fecha de creación).) y las que el servidor quiera agregar a la hora de generar el **jwt**.

Signature
El signature es la parte que garantiza la autenticidad del JWT. En esta parte entra en juego la clave privada. La firma se genera usando el contenido del header y el payload, ademas de la clave secreta conocida solo por el servidor. Al recibir el JWT, el servidor puede verificar si este fue alterado en el camino o no.

Signature = Hash(Base64(Header)+Base64(Payload)+Secret)

Como el secreto es algo que el servidor solo conoce y a partir de la signature nadie puede deducir el Secret porque los hash son irreversibles entonces no hay forma de que alguien puede generar una Signature válida si se crean o modifican un payload y/o un header porque les falta un parte para poder generarla, el Secret.

Afirmaciones de JWT

- Al estar firmados digitalmente, me aseguran que el contenido del mismo no fue alterado.
- Sirven para realizar el proceso de autentificación y autorización.
- Un JWT es un token que hace referencia a una sesion y permite autenticarse
- Las secciones de un JWT son Header Payload y Signature
- El mecanismo de seguridad de un jwt reside en que la signature se arma a partir de un salt que solo conoce el servidor
- Las claims son agregadas en el servidor cuando se genera el token

● Issuer: es quien creó el token, el identificado de la api que generó el JWT. Se coloca como claim en el payload.
● Audience: a quién va dirigido dicho token. Se coloca como claim en el payload
● Claims personalizadas: una lista de claims que se van a agregar al payload del JWT.
● Fecha de creación: en UTC que es GMT0 para poder hablar un lenguaje común
● Fecha de expiración: un periodo a partir de la fecha de creación en donde el JWT va a ser válido. Este valor es una claim del payload
● Signature: es la firma del JWT que pre generamos con anterioridad que ahora él constructora la va usar para armar la firma definitiva del JWT usando el payload entero del JWT con todos los valores que le pasamos.

Verificación de la autenticidad del JWT
Para que el chequeo de autenticación se haga efectivo tenemos que explicitar en los lugares que así lo deseamos para que se aplique con el decorador [Authorize], este decorador puede ser aplicado a todo el controlador poniéndolo justo de la definición de la clase controller o bien a cada endpoint en particular.