Unidad 1:
El Ecosistema .NET

.Net es un framework con herramientas para el desarrollo de software creado por Microsoft y lanzado en 2002.

Runtimes
NET Framework

El código es abierto para ver pero no se puede contribuir.

Todo el framework funciona solo en SOs de Windows, todo el framework es centrado en windows.
Ofrece una gran gama de tipos de aplicaciones que se pueden crear. Algunas de estos tipos (Workloads) son los siguientes:

- Console Applications: aplicaciones de consola.
- Windows Communications Foundations (WCF): para comunicación de servidores web
- Windows Workflow Foundation (WF): automatizar procesos de negocios
- Windows Presentation Foundation (WPF): aplicaciones de escritorio con complejas UI
- Windows Form: aplicaciones de escritorio pero sencillas
- ASP .NET: aplicaciones web con Web Forms, Web API o MVC
- Azure (WebJobs, Cloud Services)

Net Core

Completamente Open Source y el código se encuentra disponible en github. Es un framework multiplataforma, más liviano que net framework. La filosofía de Net Core es tener solo las librerías necesarias. de hacer falta se instalan paquetes NuGet.

Net core puede correr SOs de Windows Red Hat(Linux), Ubuntu, Red Hat, Unix y Mac.

Los tipos de aplicaciones son los siguientes:

- Console applications
- ASP.Net Core: MVC y API (una evulucion de net framework)
- UWP: para aplicaciones de escritorio

Mono for Xamarin

Completamente Open Source y el código se encuentra disponible en github. Es un runtime para crear aplicaciones móviles. Es la implementación Open Source y Multiplataforma del CLR original de Net Framework, es de 2001 y solo acepta C#.

Comparación entre Runtimes

	.NET Framework	.NET Core	Mono for Xamarin
Workloads	WPF, Windows Forms, ASP.NET	ASP.NET Core, UWP	IOS, Mac OS X, Android
Cross-platform		X	X
Side-by-side	Only major versions	X	X
Self-contained		X	X
Main purpose	Windows desktop apps	Cross-platform web and desktop apps	Cross-platform mobile apps

Net

Standard

El propósito principal de Net Standard es compartir código entre runtimes, cada nueva versión incluye todas las APIs de la versión anterior más las nuevas propias de la versión actual. Sin embargo por esto mismo las versiones anteriores de Net Standard son compatibles por más versiones de runtimes. Se pueden crear Librerías de Clases que usen una determinada versión de .Net Standard como objetivo y de esta forma se puede reutilizar estas clases en todos los runtimes que implementen la versión de Net Standard objetivo de nuestra Librería de Clases creada.
Ejemplo: Realizó una Librería de Clases que implementa una clase MonedaArgentina y está implementa métodos de conversión a otras divisas esta clase. Esta librería de clases tiene como objetivo Net Standard 1.1, de esta forma para todos mis proyectos de Net Framework 4.5 van a tener disponible esta Librería de Clases

Common Infrastructure

El proceso de compilación para Net Core y para Net Framework y en términos generales también para Mono, es el siguiente:

1. La solución que nosotros generamos produce tres archivos que se usan para compilar. El archivo sln que es el ejecutable de la solución (que se usa para abrir la solución en visual studio), el archivo .csproj junto con .vbproj
2. Cuando le damos a ejecutar se invoca al compilador de la plataforma .Net para c# (también conocido como Roslyn) o bien el F# Compiler, en el caso de que se haya programado en ese lenguaje. Estos compiladores generan un archivo ejecutable (.Exe, .DLL) el cual esta escrito en IL (Intermediate Language)
3. El common language runtime o CLR (en el caso de Net Core es el Core CLR) interpreta estos ejecutables en IL y ejecuta el JIT Compiler (Just in Time) apropiado en tiempo de ejecución (el JIT es específico para el hardware en donde se ejecuta la aplicación). El JIT pasa de IL a lenguaje de máquina, es decir lenguaje nativo para el hardware y lo hace en tiempo de ejecución, es decir que si hay algún error que no fue detectado por el compilador de C# o F# que que paso el codigo a IL, entonces es detectado por el JIT cuando se lo invoca para que compile la parte de código errónea que se requiere ejecutar

Introducción a la programación web

Cliente y Servidor

El modelo cliente-servidor, también conocido como “principio cliente-servidor”, es un modelo de comunicación que permite la distribución de tareas dentro de una red de ordenadores.

Un servidor es un hardware que proporciona los recursos necesarios para otros ordenadores o programas, pero un servidor también puede ser un programa informático que se comunica con los clientes. Un servidor acepta las peticiones del cliente, las procesa y proporciona la respuesta solicitada. También existen diferentes tipos de clientes. Un ordenador o un programa informático se comunica con el servidor, envía solicitudes y recibe respuestas del servidor. En cuanto al modelo cliente-servidor, representa la interacción entre el servidor y el cliente.

Hay una clara distribución de tareas entre los clientes y los servidores. El servidor es el responsable de proporcionar los servicios. Se encarga de ejecutar los servicios solicitados y entrega la respuesta esperada. El cliente, en cambio, utiliza y solicita los servicios proporcionados. Finalmente, recibe la respuesta del servidor (Por ejemplo, renderiza los datos en la ventana del navegador).

En el modelo cliente-servidor, un servidor sirve a varios clientes y, por ende, procesa múltiples peticiones de diferentes clientes. Para ello, presta su servicio de forma permanente y pasiva. Por su parte, el cliente solicita activamente los servicios del servidor e inicia las tareas del servidor.

Ventajas

Administración central

La administración central es una de las principales ventajas. El servidor está en el centro de la red. Todos los usuarios o clientes lo utilizan. Los recursos importantes, como bases de datos, se encuentran en el servidor y son accesibles de forma centralizada. Esto simplifica la administración y el mantenimiento de los recursos importantes que requieren protección. La ubicación central del servidor hace que la realización de actualizaciones sea cómoda y de bajo riesgo.

Derechos de acceso controlados globalmente

El almacenamiento central de recursos importantes permite una gestión segura y global de los derechos de acceso. Cuando se trata de datos sensibles, es importante saber quién puede ver los datos y quién puede manipularlos.

Un servidor para muchos clientes

Los clientes comparten los recursos del servidor. También es posible que el servidor esté situado en un lugar distinto al de los clientes. Lo más importante es que el servidor y los clientes estén conectados a través de una red. Y por ende, no es necesario que los recursos estén en el mismo sitio.

Inconvenientes

Caída del servidor

Debido a la disposición centralizada y a la dependencia en un modelo cliente-servidor, la caída del servidor conlleva la caída de todo el sistema.

Recursos de un servidor

El servidor realiza las tareas que requieren muchos recursos. La demanda de recursos de los clientes es mucho menor. Si el servidor tiene muy pocos recursos, afecta a todos los clientes.

Inversión de tiempo

Otro factor que no hay que subestimar es el tiempo necesario para hacer funcionar tu servidor. Además de los conocimientos técnicos correspondientes, por ejemplo, para proteger y configurar servidores, su uso requiere una considerable inversión de tiempo.

Frontend y Backend

La división Frontend y Backend apareció en el año 2008 con la salida de HTML5 y la aparición de las nuevas APIs. Frontend es la parte del sitio web que interactúa con los usuarios, por eso decimos que está del lado del cliente. Backend es la parte que se conecta con la base de datos y el servidor que utiliza dicho sitio web, por eso decimos que el backend corre del lado del servidor.

¿Qué es una API?

Una API (Application Programming Interface) es una interfaz para que programas de software se comuniquen entre ellos y compartan datos bajo diferentes estándares. Pueden ser locales o remotas estas últimas utilizan un servicio web y sus arquitecturas pueden ser SOAP o REST , el tipo más común hoy en día es REST (Representational State Transfer), generalmente los archivos de información enviados y recibidos son JSON (objetos de javaScript). Las Api pueden ser públicas o privadas, las privadas requieren un autenticación que se realiza mediante un TOKEN.

Request y Responses

En informática, request–response o request–reply es uno de los métodos básicos que utilizan las computadoras para comunicarse entre sí en una red, en el que la primera computadora envía una solicitud de algunos datos y la segunda responde a la solicitud. Más específicamente, es un patrón de intercambio de mensajes en el que un solicitante envía un mensaje de solicitud a un sistema de respuesta, que recibe y procesa la solicitud y, en última instancia, devuelve un mensaje como respuesta. Es similar a una llamada telefónica, en la que la persona que llama debe esperar a que el destinatario atienda antes de que se pueda hablar de algo.
Para simplificar, este patrón generalmente se implementa de forma puramente sincrónica, como en las llamadas de servicio web a través de HTTP, que mantiene una conexión abierta y espera hasta que se entrega la respuesta o expira el período de tiempo de espera. Sin embargo, la solicitud-respuesta también se puede implementar de forma asincrónica, con una respuesta que se devuelve en un momento posterior desconocido.
En programación web las request y las responses tiene una forma específica con secciones y atributos que deben estar presente y con determinados valores. Esto los especifica el protocolo HTTP o HTTPS con el cual se realizan casi todas las comunicaciones web

Arquitecturas multi-servidor

También puede haber varios servidores para ofrecer un servicio, estos pueden tener diferentes “direcciones” para accederlos y ciertas configuraciones para que si la disponibilidad de algunos de ellos se ve comprometida entonces el servicio lo ofrece alguno de los otros que se encuentra en correcto estado, esta tecnica se llama “**mirroring**” es bastante común en sitios web de descarga o streaming piratas para así evadir los controles ubicando los servidores “mirror” en diferentes países.

Otra forma de implementar la arquitectura es mediante la adquisición de capacidad de procesamiento de la nube mediante el uso de servicios de AWS, GCP, o Azure (también hay otros pero estos claramente son los más completos y por lejos acaparan la inmensa mayoría de la oferta). Estas nubes permiten disponibilizar el servicio que queremos sin que nos tengamos que hacer cargo de la infraestructura (el o los servidores físicos) ellos se encargan de asegurar un elevado ratio de disponibilidad a cambio del pago correspondiente. Esto se llama **procesamiento distribuido**

Los inconvenientes planteados anteriormente se ven enormemente mitigados por los servicios proveídos por las nubes mencionadas: las caídas se vuelven algo casi imposible de que suceda si contrata el plan adecuado ya que los recursos necesarios para brindar el servicio se encuentran siempre disponible y escalan cuando la demanda escala de esta manera los recursos del servidor ya no es algo de lo que nos tengamos que preocupar. La inversión de tiempo también se reduce considerablemente debido a las facilidades que las nubes nos dan así como también la disponibilidad de múltiples formas de capacitación y documentación que las mismas nos brindan.

Navegadores, Motores de búsqueda y Postman

Además de estas aplicaciones están los navegadores o browsers que nos permiten navegar por la internet de manera mucho menos ilimitada que una aplicación específica, los navegadores son la pieza clave del cliente en las aplicaciones web. En esencia un navegador no es más que un “intérprete” de HTML CSS y Javascript el cual interpreta el código fuente de los sitios web con los que interactúa y realiza las solicitudes al servidor especificadas en el código y procesa las respuestas que recibe del servidor. Un navegador es la herramienta única y necesaria para acceder a cualquier sitio web de la internet.

Los motores de búsqueda (Google, Bing, The Pirate Bay) son servicios web que nos proveen de las direcciones de sitios web mediante la búsqueda en enorme base de datos de sitios webs con el uso de algoritmos que intentan brindarnos las mejores aproximaciones de lo que buscamos mediante coincidencias en el contenido de los sitios y nuestros parametros de busqueda. Los navegadores (Chrome, Mozilla, Firefox, Safari, Thor) son aplicaciones que pueden usar estos buscadores para acceder a sitios webs, esencialmente lo único que necesitan los navegadores para acceder a un sitio web es su dirección (dominio o ip) y conexión a internet

Postman es una aplicación que nos permite realizar solicitudes “custom” a servidores, es decir no tiene que interpretar código fuente para realizar una solicitud como lo hace un navegador con los sitios web, sino que podemos armar la solicitud como queramos. El uso de esta herramienta está limitado a probar el funcionamiento de un servidor, esto es muy útil en la programación o en el testing.

----------UNIDAD 2----------

Conceptos de Análisis y Diseño. Programación Orientada a Objetos

Introducción a UML

¿Qué es UML?

Unified Model Language o lenguaje de modelos unificado es una herramienta que nos provee una serie de convenciones, estructuras, estándares, reglas para el análisis y diseño orientado a objetos. Es una herramienta fundamental para los programadores orientados a objetos, que nos permite pensar en los objetos de una forma ordenada y metodológica.

¿Qué es UP?

Unified Process o proceso unificado 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 (es cualquier documento diagrama dibujo estandarizado por UML y que se usa en el proceso de desarrollo del software) y entregables que este proceso crea y utiliza. En pocas palabras UP es una metodología de desarrollo de software que nos propone una forma de hacer software y utiliza UML para elaborar sus diagramas y documentos
El UP fomenta muchas buenas prácticas, pero una destaca sobre las demás: el desarrollo iterativo. En este enfoque el desarrollo se organiza en una serie de mini proyectos cortos de duración fija (por ejemplo, de cuatro semanas) llamados iteraciones, el resultado de cada uno es un sistema (software o no) que puede ser probado, integrado y ejecutado. Cada iteración incluye sus propias actividades de análisis de requisitos, diseño, implementación y pruebas.

Fases y Disciplinas

El UP hace uso de una serie de conceptos para explicar su metodología, dos de ellos son las Fases y las Disciplinas. Para UP las Fases son aquellas que están asociadas a una transición temporal, en cambio las disciplinas están asociadas al tipo de trabajo que en ellas se realiza. Si bien algunas disciplinas están asociadas a algunas fases, ya que es lógico que ciertos tipos de trabajo se hagan en cierta fase de la elaboración del software, es importante entender que no son lo mismo ni tampoco su diferencia es redundante ya que la relación entre fases y disciplinas es bastante dinámica.

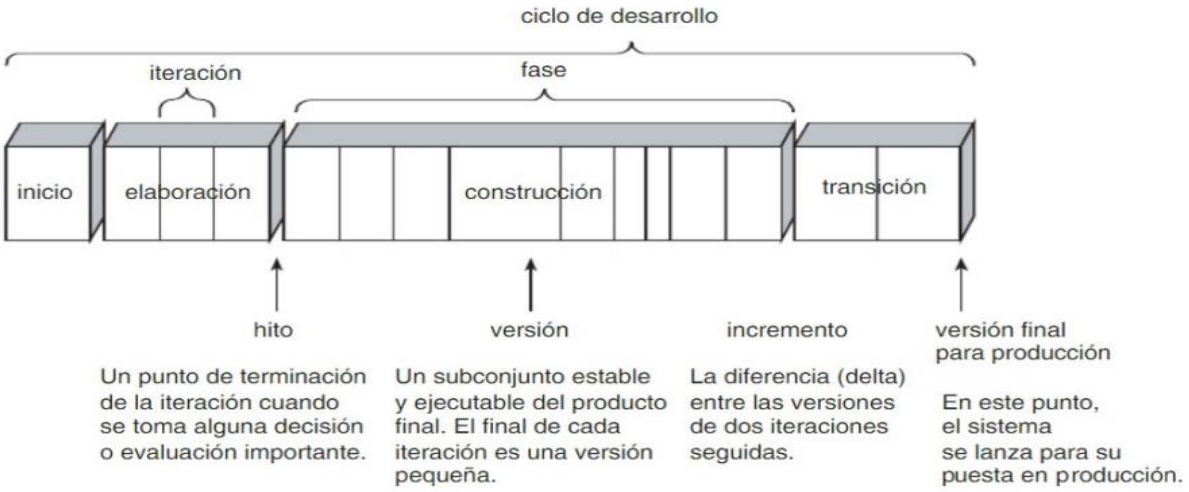
Fases

- Inicio: es una visión aproximada de él/los problemas a resolver en ella se hace análisis del negocio, se determina el alcance a abarcar, se realizan estimaciones indefectiblemente imprecisas. Es una fase de viabilidad

- Elaboración: visión refinada, implementación iterativa del núcleo central de la arquitectura, resolución de riesgos altos, identificación de más requerimientos o requisitos. Se logran estimaciones más realistas. Es una fase de implementación

- Construcción: se resuelven de forma iterativa los demás riesgos, se prepara para el despliegue

- Transición: Se hacen pruebas beta (pruebas en un entorno relativamente real, con algunos usuarios finales o bien casos de usos factibles en el negocio). Se realiza el despliegue del producto elaborado



Disciplinas

A continuación, se presentan las principales disciplinas de UP. Las disciplinas marcadas en azul están presentes en todas las iteraciones.

- Planificación

  En esta fase se incluyen tareas como la determinación del ámbito del proyecto, un estudio de viabilidad, análisis de riesgos, costes estimados, asignación de recursos en las distintas etapas, etc. Son tareas que influyen en el éxito del proyecto, por eso es necesaria una planificación inicial.

- Análisis

  Proceso en el que se trata de descubrir lo que se necesita y cómo llegar a las características que el sistema debe poseer.

- Diseño

  Se estudian las posibles implementaciones que hay que construir y la estructura general del software. Es una etapa complicada, y si la solución inicial no es la más adecuada, habrá que redefinirla.

- Implementación

  Se trata de elegir las herramientas adecuadas, un entorno de desarrollo que haga más sencillo el trabajo y el lenguaje de programación óptimo.
  Esta decisión va a depender del diseño y el entorno elegido.
  Es importante tener en cuenta la adquisición de productos necesarios para que el software funcione.

- Pruebas

  Conseguiremos detectar los fallos que se hayan cometido en etapas anteriores, para que no repercuta en el usuario final.
  Esta fase del ciclo de vida del software hay que repetirla tantas veces como sea necesaria, ya que la calidad y estabilidad final del software dependerá de esta fase.

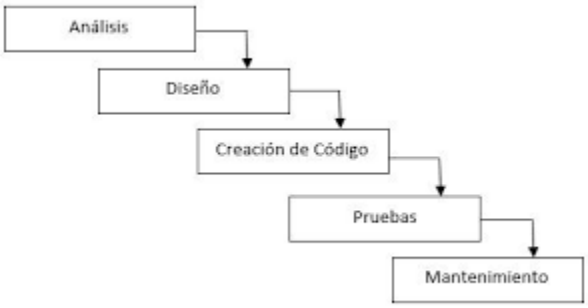
- Despliegue

  En esta fase pondremos el software en funcionamiento.

Ciclos de vida del proceso de desarrollo

Modelo en cascada

En este modelo del ciclo de vida de un software, se espera a finalizar una etapa para comenzar con la siguiente.
Es un proceso secuencial en el que el desarrollo va fluyendo de arriba hacia abajo.
Aunque en ocasiones ha sido criticado debido a su rigidez, sigue siendo el más seguido a día de hoy.



(En la imagen creación de código se refiere a la etapa de implementación)

Modelo de prototipos

Comienza con la recolección de requisitos y definición de objetivos globales, llevando a un diseño rápido y a un prototipo.
El prototipo es evaluado por el cliente, y nos permite refinar los requisitos hasta llegar a lo que el cliente espera.

Modelo de desarrollo iterativo e incremental

Se basa en la ampliación y refinamiento sucesivos del sistema mediante múltiples iteraciones, con retroalimentación cíclica y adaptación como elementos principales. El sistema crece incrementalmente a lo largo del tiempo, iteración tras iteración, y por ello su nombre

Programación Orientada a objetos y principios fundamentales

Encapsulación

El concepto de encapsulación es el más evidente de todos. Pero, precisamente por su sencillez, a veces pasa inadvertido.

La encapsulación es la característica de un lenguaje POO que permite que todo lo referente a un objeto quede aislado dentro de éste. Es decir, que todos los datos referentes a un objeto queden "encerrados" dentro de éste y sólo se puede acceder a ellos a través de los miembros que la clase proporciona (propiedades y métodos).

Abstracción

Este concepto está muy relacionado con el anterior.

Como la propia palabra indica, el principio de abstracción lo que implica es que la clase debe representar las características de la entidad hacia el mundo exterior, pero ocultando la complejidad que llevan aparejada. O sea, nos abstrae de la complejidad que haya dentro dándonos una serie de atributos y comportamientos (propiedades y funciones) que podemos usar sin preocuparnos de qué pasa por dentro cuando lo hagamos.

Así, una clase (y por lo tanto todos los objetos que se crean a partir de ella) debe exponer para su uso solo lo que sea necesario. Cómo se haga "por dentro" es irrelevante para los programas que hagan uso de los objetos de esa clase.

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

Dado que una clase es un patrón que define cómo es y cómo se comporta una cierta entidad, 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 le llama clase base o padre, y a la clase que hereda de ésta se le llama clase derivada o hija.

Polimorfismo

La palabra polimorfismo viene del griego "polys" (muchos) y "morfo" (forma), y quiere decir "cualidad de tener muchas formas".

En POO, el concepto de polimorfismo se refiere al hecho de que varios objetos de diferentes clases, pero con una base común, se pueden usar de manera indistinta, sin tener que saber de qué clase exacta son para poder hacerlo.

El polimorfismo permite que una clase hija puede redefinir un método de la clase padre. En el caso de que la clase hija tuviese un método definido con el mismo nombre pero con el atributo “override”, la que es válida para objeto instancia de la clase hija es el método redefinido en la clase hija.

Artefactos UML fundamentales

Minuta de relevamiento

Este artefacto no está definido por el estándar UML pero es por el cual se parte para realizar todos los demás. Normalmente hecho por el analista del equipo, una minuta de relevamiento es una narración descriptiva de él/los procesos de negocio de la empresa que son relevantes para el sistema que se precisa desarrollar.

Este relevamiento no es para nada sencillo de realizar y de hacer, ya que requiere captar de manera objetiva, sencilla y metódica todos los pasos del proceso de negocio así como de sus alternativas. El analista tiene que escribir esta minuta pensando en aquellos que después la leerán para obtenerlos demás artefactos.

Se suele escribir con oraciones cortas en presente simple, siendo consistente con los términos utilizados (Ej al cliente no se le dice de distintas formas como consumidor o comprador)

Lista de requerimientos

Los requerimientos son capacidades y condiciones con las cuales debe ser conforme el sistema y más ampliamente el proyecto que se desea construir. Por lo tanto, el primer desafío con el que nos encontramos a la hora de desarrollar un sistema es con encontrar comunicar y registrar los requerimientos.
Normalmente son definidas por el cliente, sin embargo, esto es todo un proceso porque usualmente el cliente no sabe lo que quiere, e incluso aunque lo sepa sus necesidades pueden cambiar a lo largo del desarrollo del sistema. El negocio, la presión del mercado, innovaciones, cambio de necesidades en sus clientes, diferentes formas de comercialización,etc son algunos de los motivos por lo que debemos considerar a los requerimientos como un conjunto dinámico que debe ser revisado periódicamente.
Para UML estos requerimientos se explicitan en un artefacto denominado “Lista de requerimientos”

Tipos de requerimientos

UML define una clasificación que es muy útil debido a que los tipos que se definen permiten dividir el trabajo para resolución de requisitos similares. La clasificación de UML se denomina modelo FURPS+:

F Funcional: Características, capacidades, seguridad. Deben ser cumplidas para que el sistema se pueda usar.

U Facilidad de uso (Usability): factores humanos, ayuda, documentación.
R Fiabilidad (Reliability): frecuencia de fallos, capacidad de recuperación de un fallo, grado de previsión.
P Rendimiento (Performance): tiempo de respuesta, productividad, precisión, disponibilidad, uso de recursos
S Soporte (Supportability): adaptabilidad, facilidad de mantenimiento, internacionalización, configurabilidad.
+ Implementación, Interfaz, Legales, Operaciones, Empaquetamiento, etc.

Casos de uso

Un caso de uso esta compuesto por:

- Actor primario: es el que usa el sistema
- Actor/es secundarios: de haberlos son aquellos que solicitan o proveen alguna información necesaria para que el caso de uso se pueda realizar
- Acción disparadora del caso de uso: un evento que hace que el caso de uso inicie
- Estado de éxito: es el escenario que resulta de finalizar el caso de uso con éxito
- Estado de fracaso:es el escenario que resulta de finalizar el caso de uso con fracaso
- Un camino básico en donde se describe el paso a paso de la utilización del sistema por un actor.
- Caminos alternativos: son las variantes que se pueden dar si el escenario no transcurre por el camino básico.

  Es importante destacar que para cada requerimiento de tipo funcional se debe corresponder con un caso de uso que narra cómo se lo satisface.

Tipo de caso de uso

Difieren en la perspectiva y el nivel de detalle que cada uno contempla.

- Casos de uso resumen de negocio (CURN).: son los casos de uso desde el punto de vista del proceso de negocio antes de tener el sistema. Es uno por requerimiento funcional.
- Caso de uso resumen de sistema (CURS): es uno por cada caso de uso resumen de negocio pero desde el punto de vista del proceso con el sistema hecho
- Caso de uso usuario (CUU): estos casos de uso son producto de partir los CURS en pequeñas partes enfocándose en uso especifico de sistema y corto en el tiempo. Cada CUU se corresponde solo a un CURS.

Diagrama de Clases

Relaciones

Una relación identifica una dependencia. Esta dependencia puede ser entre dos o más clases (más común) o una clase hacía sí misma (menos común, pero existen), este último tipo de dependencia se denomina dependencia reflexiva. Las relaciones se representan con una línea que une las clases, esta línea variará dependiendo del tipo de relación

Las relaciones en el diagrama de clases tienen varias propiedades, que dependiendo la profundidad que se quiera dar al diagrama se representarán o no. Estas propiedades son las siguientes:

- Multiplicidad. Es decir, el número de elementos de una clase que participan en una relación. Se puede indicar un número, un rango… Se utiliza n o * para identificar un número cualquiera.
- Nombre de la asociación. En ocasiones se escribe una indicación de la asociación que ayuda a entender la relación que tienen dos clases. Suelen utilizarse verbos como por ejemplo: “Una empresa contrata a n empleados”

Tipos principales de relaciones

- Asociación

Este tipo de relación es el más común y se utiliza para representar dependencia semántica. Se representa con una simple línea continua que une las clases que están incluidas en la asociación.

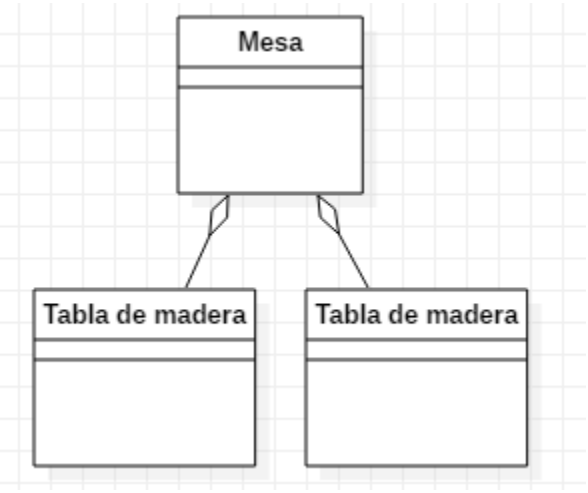
Un ejemplo de asociación podría ser: “Una mascota pertenece a una persona”.

- Agregación o composición

Es una representación jerárquica que indica a un objeto y las partes que componen ese objeto. Es decir, representa relaciones en las que un objeto es parte de otro, pero aun así debe tener existencia en sí mismo.

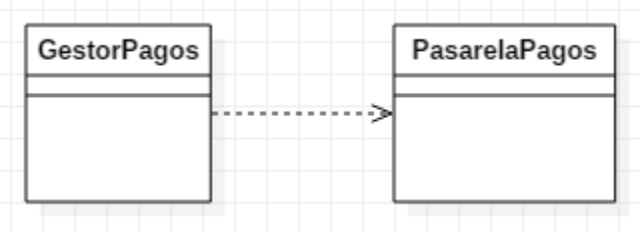
Se representa con una línea que tiene un rombo en la parte de la clase que es una agregación de la otra clase (es decir, en la clase que contiene las otras).

Un ejemplo de esta relación podría ser: “Las mesas están formadas por tablas de madera y tornillos o, dicho de otra manera, los tornillos y las tablas forman parte de una mesa”. Cómo ves, el tornillo podría formar parte de más objetos, por lo que interesa especialmente su abstracción en otra clase.



- Dependencia

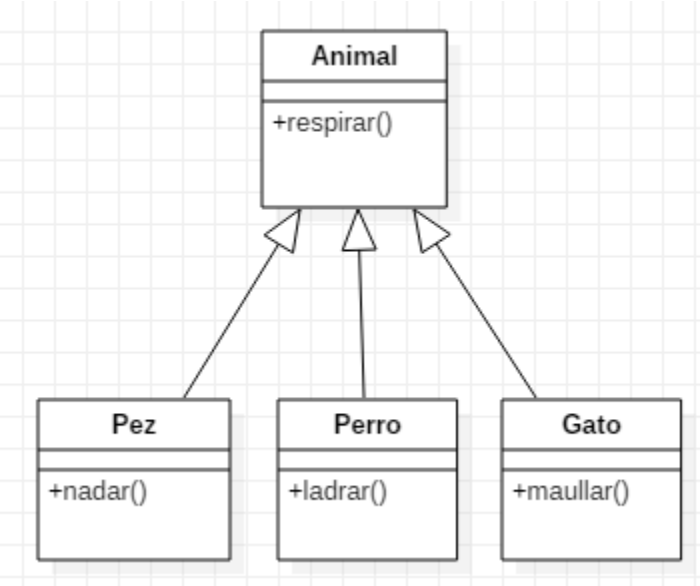
Se utiliza este tipo de relación para representar que una clase requiere de otra para ofrecer sus funcionalidades. Es muy sencilla y se representa con una flecha discontinua que va desde la clase que necesita la utilidad de la otra flecha hasta esta misma.



- Herencia

Otra relación muy común en el diagrama de clases es la herencia. Este tipo de relaciones permiten que una clase (clase hija o subclase) reciba los atributos y métodos de otra clase (clase padre o superclase). Estos atributos y métodos recibidos se suman a los que la clase tiene por sí misma. Se utiliza en relaciones “es un”.

Un ejemplo de esta relación podría ser la siguiente: Un pez, un perro y un gato son animales.



Cómo dibujar un diagrama de clases

Los diagramas de clase van de la mano con el diseño orientado a objetos. Por lo tanto, saber lo básico de este tipo de diseño es una parte clave para poder dibujar diagramas de clase eficaces.

Este tipo de diagramas son solicitados cuando se está describiendo la vista estática del sistema o sus funcionalidades. Unos pequeños pasos que puedes utilizar de guía para construir estos diagramas son los siguientes:

- Identifica los nombres de la clase

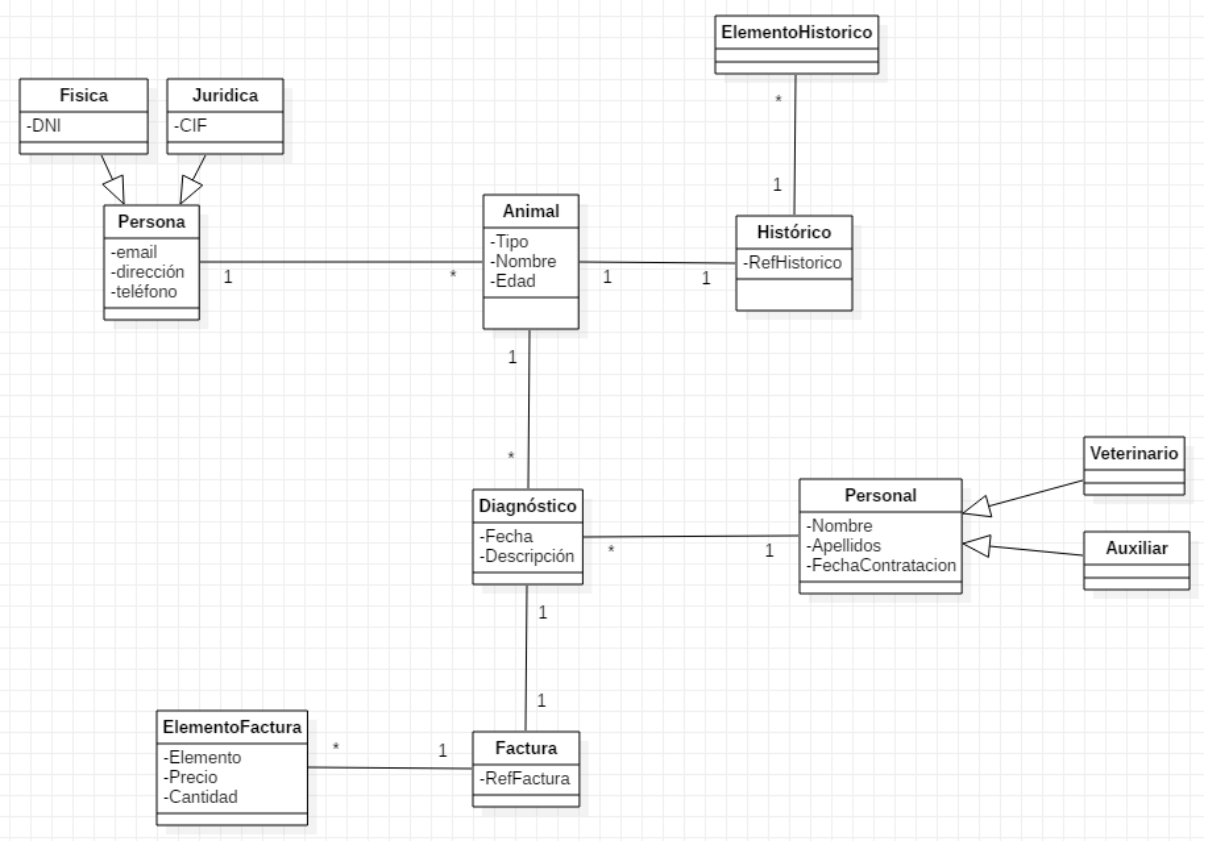
El primer paso es identificar los objetos primarios del sistema. Las clases suelen corresponder a sustantivos dentro del dominio del problema.

- Distingue las relaciones

El siguiente paso es determinar cómo cada una de las clases u objetos están relacionados entre sí. Busque los puntos en común y las abstracciones entre ellos; esto le ayudará a agruparlos al dibujar el diagrama de clase.

- Crea la estructura

Primero, agregue los nombres de clase y vincule con los conectores apropiados, prestando especial atención a la cardinalidad o las herencias. Deje los atributos y funciones para más tarde, una vez que esté la estructura del diagrama resuelta.



----------UNIDAD 3----------

Arquitecturas por capas

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

Además, la arquitectura establece reglas de cómo se deben comunicar las capas.

Primera regla

Cada capa debe tener una responsabilidad única. Es decir que las capas deben estar perfectamente delimitadas de que se ocupa cada una de ellas.

Segunda regla

Las capas deben respetar una estructura jerárquica estricta. Quiere decir que cada capa puede comunicarse sólo con la que está debajo suyo, pero NO al revé Y cuando nos referimos a la próxima más baja significa que no se puede saltar capas. Veamos un ejemplo gráfico.

Ventajas

Entre sus ventajas se encuentran las siguientes:

- Es fácil testear cada capa por separado debido a la separación clara de responsabilidades que existe entre ellas.
- Al momento de hacer un cambio, si se implementó bien la separación de responsabilidad, este cambio solo debe impactar a la capa responsable y no a todas. Esto se conoce como desacople.

Desventajas

- Si se implementaron demasiadas capas el rendimiento de la aplicación puede verse afectado
- Ciertas operaciones al ser modificadas pueden afectar a todas las capas, haciendo visible que no existe un 100% de desacople entre estas.

Arquitectura de tres capas

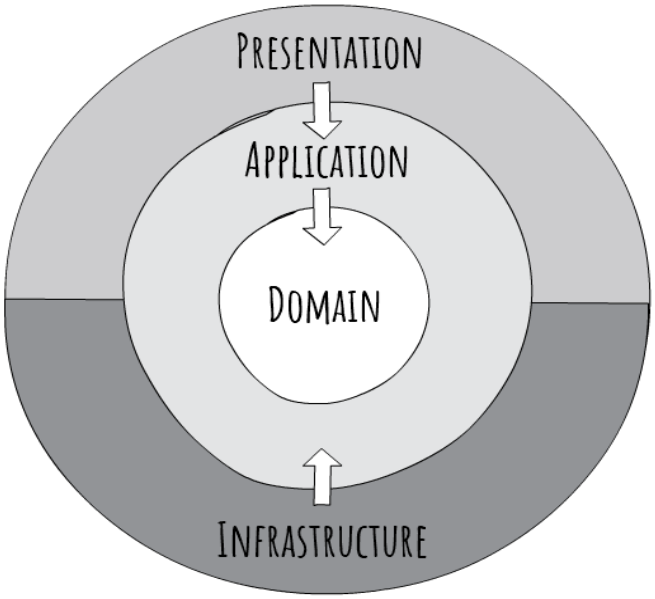
Esta arquitectura permite desarrollar n número de capas siempre y cuando se respeten las dos reglas expuestas.
La arquitectura de tres capas en particular es muy utilizada ya que propone dividir las capas en 3 específicas que son frecuentes en aplicaciones empresariales web. Como veníamos mencionando en los ejemplos las capas que propone son las siguientes.

Clean architecture

Descripción del patrón

En Clean architecture, la capa de dominio y aplicación son el centro del diseño y se conocen como el core del sistema.
La capa de dominio contiene lógica empresarial y la capa de aplicación contiene lógica de negocio. La diferencia está en que la lógica empresarial se puede compartir entre múltiples sistemas y la lógica de negocio corresponde a un sistema en particular.
El core no debe depender de el acceso a la base de datos ni a otras cuestiones de infraestructura, por lo tanto las dependencias son invertidas.
Esto se obtiene agregando interfaces o abstracciones en el core, que son implementadas en las capas exteriores. Por ejemplo, si se quiere implementar el patrón Repositorio, se debe definir la interfaz dentro del core y la implementación se hace en la capa de infraestructura.
Todas las dependencias apuntan hacia el core, y el core no tiene dependencia de ninguna otra capa. Las capas de infraestructura y presentación dependen del core, pero no la una de la otra.

Representación gráfica



Estructura básica de carpetas en cada proyecto

Esta es una estructura sugerida para ordenar el código. La misma puede variar según criterio del arquitecto de software y de los patrones que se deseen implementar en la solución.

- 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

El término API es una abreviatura de Application Programming Interfaces, que en español significa interfaz de programación de aplicaciones. Se trata de un conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones, permitiendo la comunicación entre dos aplicaciones de software a través de un conjunto de reglas.

Así pues, podemos hablar de una API como una especificación formal que establece cómo un módulo de un software se comunica o interactúa con otro para cumplir una o muchas funciones. Todo dependiendo de las aplicaciones que las vayan a utilizar, y de los permisos que les dé el propietario de la API a los desarrolladores de terceros.

Una de las principales funciones de las API es poder facilitar el trabajo a los desarrolladores y ahorrarles tiempo y dinero. Por ejemplo, si estás creando una aplicación que es una tienda online, no necesitarás crear desde cero un sistema de pagos u otro para verificar si hay stock disponible de un producto. Podrás utilizar la API de un servicio de pago ya existente, por ejemplo PayPal, y pedirle a tu distribuidor una API que te permita saber el stock que ellos tienen.

Con ello, no será necesario tener que reinventar la rueda con cada servicio que se crea, ya que podrás utilizar piezas o funciones que otros ya han creado. Imagínate que cada tienda online tuviera que tener su propio sistema de pago, para los usuarios normales es mucho más cómodo poder hacerlo con los principales servicios que casi todos utilizan.

Request y Response

HTTP REQUEST y RESPONSE son dos conceptos básicos en el desarrollo web, pero no siempre le quedan claros a los programadores.
HTTP significa HyperText Transfer Protocol. Esta es la forma de comunicación de datos básica en Internet.
La comunicación de datos empieza con un request enviado del cliente, y termina con la respuesta del servidor web.
Por ejemplo, si fuera un ejemplo clásico con un ser humano visitando una página Web:

4. Un sitio web que empieza con la URL http:// es entrado en un navegador web de la computadora del cliente. El navegador puede ser Chrome, Firefox, o Internet explorer, no importa.
5. El navegador envía un request al servidor web que está hospedado en el website.
6. El servidor web regresa una respuesta como un página de HTML, o algún otro formato de documento al navegador (puede ser un mp4, mp3, pdf, doc, entre otros soportados por el navegador)
7. El navegador despliega el response del servidor al usuario. Por supuesto esto dependerá de los formatos que soporte el navegador.

No todas las requests se hacen desde los navegadores de usuarios finales, de hecho muchas peticiones se hacen desde aplicaciones(Discord, Steam, Whatsapp, Instagram) o incluso de dispositivos IoT(Google Home, Alarmas).
De esta forma mucha de la comunicación vía web, ya no se hace directamente entre humano-máquina, cada vez más son aplicaciones automatizadas de ambos lados, las que envían datos. Es decir por ejemplo yo tengo un programa en PHP, APEX, o algún otro lenguaje que utilizo para enviar request y recibir response, de manera que la comunicación es de máquina a máquina.
Las peticiones o requests se hacen siempre a un endpoint. Es decir un punto de acceso de una API que procesa dicha solicitud y responde de acuerdo a su implementación interna. Aquí se puede ver que las APIs toman claramente la encapsulación de la POO.
Ahora bien, ¿Qué contienen el request y el response?

HTTP Request Structure from Client
Un HTTP request se compone de:

- Método: GET, POST, PUT, etc. Indica que tipo de request es.
- Path: la URL que se solicita, donde se encuentra el resource.
- Protocolo: contiene HTTP y su versión, actualmente 1.1.
- Headers. Son esquemas de key: value que contienen información sobre el HTTP request y el navegador. Aquí también se encuentran los datos de las cookies. La mayoría de los headers son opcionales.
- Body. Si se envía información al servidor a través de POST o PUT, ésta va en el body.
-

HTTP Response Structure from Web Server

Una vez que el navegador envía el HTTP request, el servidor responde con un HTTP response, compuesto por:

- Protocolo. Contiene HTTP y su versión, actualmente 1.1.
- Status code. El código de respuesta, por ejemplo: 200 OK, que significa que el GET request ha sido satisfactorio y el servidor devolverá los contenidos del documento solicitado. Otro ejemplo es 404 Not Found, el servidor no ha encontrado el resource solicitado.
- Headers. Contienen información sobre el software del servidor, cuando se modificó por última vez el resource solicitado, el mime type, etc. De nuevo la mayoría son opcionales.
- Body. Si el servidor devuelve información que no sean headers ésta va en el body.

Verbos HTTP

También conocidos como métodos HTTP. Vimos como una request contiene un Request Method, pues bien eso es un verbo HTTP. Indican que tipo de petición queremos hacer al servidor.
Hay distintos tipos de peticiones que podemos hacer hacia el servidor mismo.

Estos verbos indican qué acción queremos realizar sobre el servidor y son GET, POST, PUT, PATCH, DELETE, HEAD, CONNECT, OPTIONS y TRACE. Cada uno indica una acción diferente a la que el servidor debe responder.
A continuación haremos una pequeña descripción de los más importantes.

GET

El método GET solicita una representación de un recurso específico. Las peticiones que usan el método GET sólo deben recuperar datos.

POST

El método POST se utiliza para enviar una entidad a un recurso en específico. Se uso más frecuente es para crear un nuevo recurso.

PUT

El modo PUT reemplaza todas las representaciones actuales del recurso de destino con la carga útil de la petición. Es para hacer una modificación total de un recurso.

DELETE

El método DELETE borra un recurso en específico.

PATCH

El método PATCH es utilizado para aplicar modificaciones parciales a un recurso.

Model View Controller (MVC)

En líneas generales, MVC es una propuesta de arquitectura del software utilizada para separar el código por sus distintas responsabilidades, manteniendo distintas capas que se encargan de hacer una tarea muy concreta, lo que ofrece beneficios diversos.

MVC se usa inicialmente en sistemas donde se requiere el uso de interfaces de usuario, aunque en la práctica el mismo patrón de arquitectura se puede utilizar para distintos tipos de aplicaciones. Surge de la necesidad de crear software más robusto con un ciclo de vida más adecuado, donde se potencie la facilidad de mantenimiento, reutilización del código y la separación de conceptos.

Su fundamento es la separación del código en tres capas diferentes, acotadas por su responsabilidad, en lo que se llaman Modelos, Vistas y Controladores.

Por qué MVC

La rama de la ingeniería del software se preocupa por crear procesos que aseguren calidad en los programas que se realizan y esa calidad atiende a diversos parámetros que son deseables para todo desarrollo, como la estructuración de los programas o reutilización del código, lo que debe influir positivamente en la facilidad de desarrollo y el mantenimiento.
Los ingenieros del software se dedican a estudiar de qué manera se pueden mejorar los procesos de creación de software y una de las soluciones a las que han llegado es la arquitectura basada en capas que separan el código en función de sus responsabilidades o conceptos. Por tanto, cuando estudiamos MVC lo primero que tenemos que saber es que está ahí para ayudarnos a crear aplicaciones con mayor calidad.

Capas

Modelos

Es la capa donde se trabaja con los datos, por tanto contendrá mecanismos para acceder a la información y también para actualizar su estado. Los datos los tendremos habitualmente en una base de datos, por lo que en los modelos tendremos todas las funciones que accederán a las tablas y harán los correspondientes selects, updates, inserts, etc.

No obstante, cabe mencionar que cuando se trabaja con MVC lo habitual también es utilizar otras librerías o algún ORM como Entity Framework, que nos permiten trabajar con abstracción de bases de datos y persistencia en objetos. Por ello, en vez de usar directamente sentencias SQL, que suelen depender del motor de base de datos con el que se esté trabajando, se utiliza un dialecto de acceso a datos basado en clases y objetos.

Vistas

Las vistas, como su nombre nos hace entender, contienen el código de nuestra aplicación que va a producir la visualización de las interfaces de usuario, o sea, el código que nos permitirá renderizar los estados de nuestra

aplicación en HTML. En las vistas nada más tenemos los códigos HTML y PHP que nos permite mostrar la salida.

En la vista generalmente trabajamos con los datos, sin embargo, no se realiza un acceso directo a éstos. Las vistas requerirán los datos a los modelos y ellas se generará la salida, tal como nuestra aplicación requiera.

Controladores

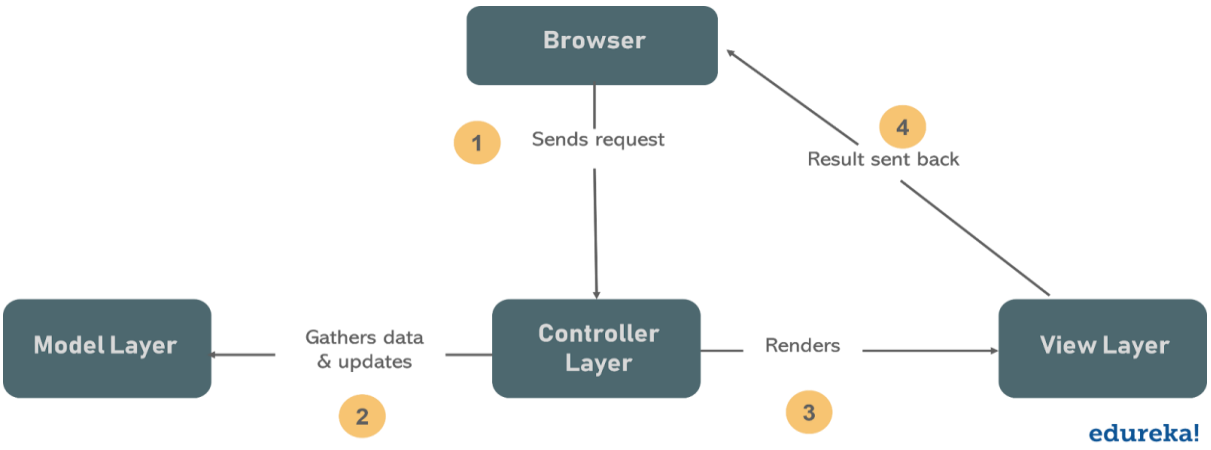
Contiene el código necesario para responder a las acciones que se solicitan en la aplicación, como visualizar un elemento, realizar una compra, una búsqueda de información, etc.

En realidad es una capa que sirve de enlace entre las vistas y los modelos, respondiendo a los mecanismos que puedan requerirse para implementar las necesidades de nuestra aplicación. Sin embargo, su responsabilidad no es manipular directamente datos, ni mostrar ningún tipo de salida, sino servir de enlace entre los modelos y las vistas para implementar las diversas necesidades del desarrollo.

Arquitectura de aplicaciones MVC

A continuación encontrarás un diagrama que te servirá para entender un poco mejor cómo colaboran las distintas capas que componen la arquitectura de desarrollo de software en el patrón MVC.

En esta imagen hemos representado con flechas los modos de colaboración entre los distintos elementos que formarían una aplicación MVC, junto con el usuario. Como se puede ver, los controladores, con su lógica de negocio, hacen de puente entre los modelos y las vistas. Pero además en algunos casos los modelos pueden enviar datos a las vistas. Veamos paso a paso cómo sería el flujo de trabajo característico en un esquema MVC.



El usuario(a través del browser) realiza una solicitud a nuestro sitio web. Generalmente estará desencadenada por acceder a una página de nuestro sitio. Esa solicitud le llega al controlador.
El controlador se comunica tanto con modelos como con vistas. A los modelos les solicita datos o les manda realizar actualizaciones de los datos. A las vistas les solicita la salida correspondiente, una vez se hayan realizado las operaciones pertinentes según la lógica del negocio.

Para producir la salida, en ocasiones las vistas pueden solicitar más información a los modelos. En ocasiones, el controlador será el responsable de solicitar todos los datos a los modelos y de enviarlos a las vistas, haciendo de puente entre unos y otros. Sería corriente tanto una cosa como la otra, todo depende de nuestra implementación; por eso esa flecha la hemos coloreado de otro color.

En resumen, en MVC la lógica se divide en dos capas distintas, cierta lógica (lógica de negocios) está presente en la capa de Modelos o en la capa que hayamos definido para ella y cierta lógica en la capa de controladores(lógica de aplicación). Es importante destacar que en la capa de vistas nunca hay ningún tipo de lógica.

----------UNIDAD 4----------

Bases de datos relacionales

Utilidades y ventajas

El modelo relacional es el mejor para mantener la consistencia de los datos en todas las aplicaciones y copias de la base de datos (denominadas instancias). Por ejemplo, cuando un cliente deposita dinero en un cajero automático y, luego, mira el saldo de la cuenta en un teléfono móvil, el cliente espera ver que ese depósito se refleja inmediatamente en un saldo de cuenta actualizado. Las bases de datos relacionales se destacan en este tipo de consistencia de datos, lo que **garantiza que múltiples instancias de una base de datos tengan los mismos datos todo el tiempo**.

Es difícil para otros tipos de bases de datos mantener este nivel de coherencia oportuna con grandes cantidades de datos. Algunas bases de datos recientes, como NoSQL, sólo pueden proporcionar “coherencia eventual”. Según este principio, cuando se escala la base de datos o cuando varios usuarios acceden a los mismos datos al mismo tiempo, los datos necesitan algo de tiempo para “ponerse al día”. La coherencia eventual es aceptable para algunos usos, como mantener listados en un catálogo de productos, pero para operaciones comerciales críticas, como transacciones de carrito de compras, la base de datos relacional sigue siendo lo ideal.

Estructura

En una base de datos relacional los datos se almacenan en tablas. Las tablas tienen columnas en cada una de ellas se especifica el nombre del atributo a almacenar y el tipo de dato.

Cada tabla debe tener una **clave primaria**, es decir un dato o conjunto de datos que permiten identificar cada registro de la misma. No puede haber dos registros con una clave idéntica. Las claves primarias pueden ser simples, compuestas por un único atributo, o compuestas, es decir compuestas por más de un atributo.

Tambien estan las **claves foráneas,** estas permiten relacionar un registro de una tabla con otro registro de otra tabla. pueden ser simples o compuestas.

Object Relational Mapper - ORM

¿Qué es?

Un ORM es un modelo de programación para mapear estructuras de una base de datos relacional, como por ejemplo SQL Server, MySQL u Oracle, entre otras. Su objetivo es simplificar y acelerar el desarrollo de las aplicaciones.

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 (crear, leer, actualizar o borrar) se puedan realizar de manera indirecta a través del ORM.

No solo permiten mapear, sino también liberarnos de picar código SQL que habitualmente hace falta para las queries o consultas y gestionar la persistencia de datos Por lo que, los objetos se pueden manipular mediante lenguajes según el tipo de ORM utilizado. Un ejemplo es LINQ sobre Entity Framework de Microsoft.

Asimismo, los ORMs más completos ofrecen distintos servicios y sin escribir código de SQL. Es una ventaja, dado que permite atacar las entidades de la Base de datos virtual sin generar el código. Por lo que, se acelera el desarrollo de las aplicaciones.

¿Para qué sirve un mapeador de objetos relacionales?

Su principal funcionalidad pasa porque el proceso de programación de la BD sea más rápido, dado que se consiguen reducir los códigos y que el mapeo sea más automático. De tal forma que el programador o desarrollador no tenga que adaptar los códigos a las tablas en base a las necesidades específicas de cada aplicación.

Es muy complicado convertir la información que recibes de la base de datos (que suele ser en tablas) a los objetos del programa en sí y viceversa. Con la utilización de un ORM, conseguirás que este proceso de mapeo sea automático e independiente de la BD, pudiendo realizar cambios, añadir más campos, etc sin tener que tocar todo el código.

También presenta importantes ventajas en cuanto a seguridad, debido a la capa de acceso que actúa como barrera frente a los ataques. A su vez la forma con la que los ORMs realizan sus consultas a las bases de datos suelen ser muy óptimas y seguras ya que están hechas para que se hagan de la mejor manera. Por lo que es otra importante característica a tener en cuenta.

Es más fácil de mantener, aunque para ello se recomienda tener conocimientos avanzados sobre su funcionamiento. Es una tarea que ante la duda se puede dejar en manos de un tercero para que sea el programador profesional el que se encargue de todo el back-end de tu página o aplicación.

En definitiva, funciona como una solución intermedia que facilita la tarea al programador y permite que el acceso a los datos sea automático. Por lo que es una herramienta interesante para quienes manejan diferentes bases de datos de empresas, organizaciones, entidades, etc

Cada lenguaje tiene uno o varios ORMs, en el caso de todo el ambiente de .NET el ORM por excelencia es Entity Framework en sus diferentes versiones, sin embargo existen otras opciones como NHibernate

Consumo de servicios e Inyección de dependencia

Cuando un piloto va a aterrizar en una pista, necesita tener el espacio para poder descender y que el tránsito actual del aeropuerto no interfiera con su aterrizaje. Pero el no va a comunicarse directamente con el controlador de Tierra o el controlador de Ruta o Área. En lugar de ello, se comunicara con la torre de control quien se encarga de gestionar todo el proceso y darle toda la información que necesita para el aterrizaje.

El aeropuerto hace la función de Contenedor de dependencia, pues se encarga de gestionar todo el proceso para que los vuelos puedan tener lugar. Los controladores aéreos, hacen la función de Framework, pues se encargan de inyectar las dependencias a los módulos dependientes y gestionar los recursos para que los módulos puedan funcionar correctamente, las interfaces son los lineamientos que se necesitan para que el avión pueda aterrizar. Todos estos elementos en conjunto conforman la inyección de dependencia y la inversión de control.

Ventajas de Utilizar Inyección de Dependencias

Flexible

- No hay necesidad de tener un código de búsqueda en la lógica de negocio.
- Elimina el acoplamiento entre módulos

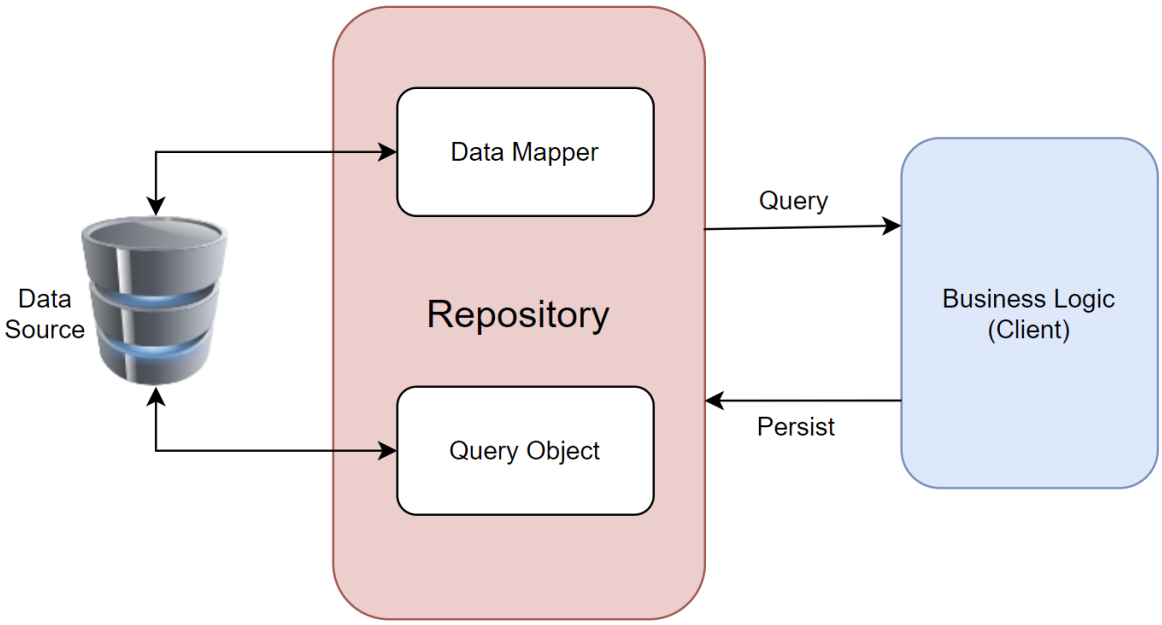
Testeable

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

Mantenible

- Permite la reutilización en diferentes entornos de aplicaciones modificando los archivos de configuración en lugar del código.
- Promueve un enfoque coherente en todos aplicaciones y equipos

Patrón repository

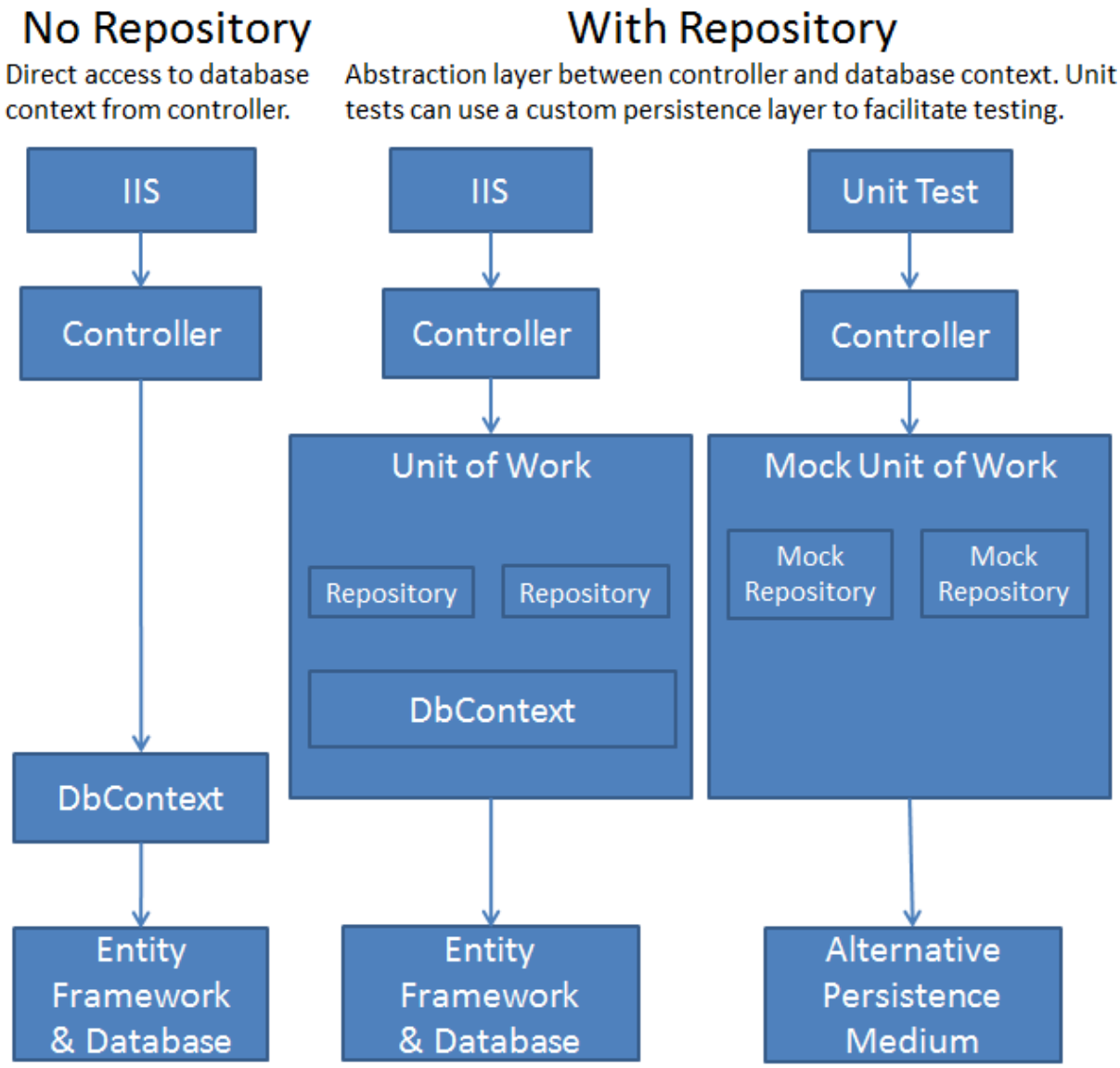


Está diseñado para crear una capa de abstracción entre la capa de acceso a datos y la capa de lógica de negocios de una aplicación. Implementar estos patrones puede ayudar a aislar la aplicación de cambios en el almacén de datos y puede facilitar la realización de pruebas unitarias automatizadas o el desarrollo controlado por pruebas (TDD).

Este patrón permite crear una clase (o varias) en donde estén definidos todos los métodos que interactúen con la base de datos, esta clase consume el contexto de la base de datos.

Otro de los aspectos fundamentales es que esta clase/s que se encarga de los métodos de datos implementando una interfaz en donde se definen la firma de los métodos(tipo nombre y parámetros del método) que componen a la clase que hace de repository. Si tengo dividido el repository en varias clases entonces debo definir una interfaz por clase y hacer que dicha clase implemente la interfaz.

La interfaz tiene la utilidad de que si en un futuro quiero implementar otro repository ya sea que porque tengo otro contexto debido a que algunos datos se manejan en otra base datos para inteligencia de negocio o por motivos de disponibilidad, entonces solo hace falta implementar la misma interfaz en la nueva clase repository análoga y de esta manera nos aseguramos que la clase contenga la totalidad de los métodos definidos de la capa de datos y definidos de la misma manera así nos evitamos inconvenientes en las capas lógicas que utilizan estos métodos



----------UNIDAD 5----------

Autenticación con JWT en .Net

Modificación de strings

Codificación

Es una técnica en la que los datos se transforman de un formato a otro, para que puedan ser entendidos y utilizados por diferentes sistemas. No se utiliza ninguna “clave” en la codificación.

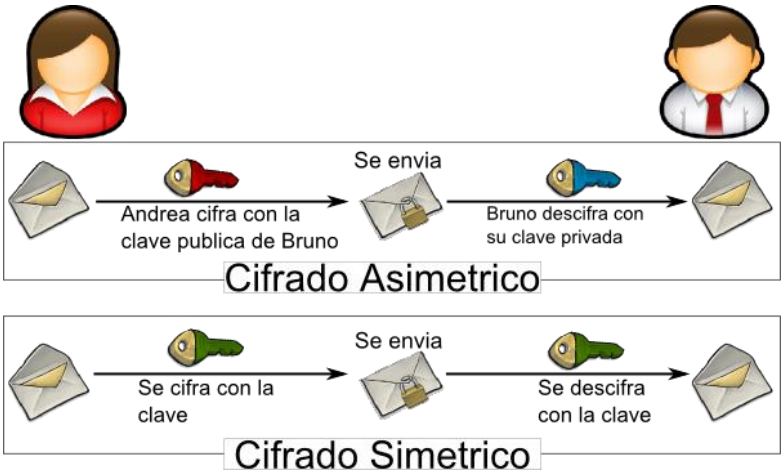
El mismo algoritmo se utiliza para decodificar los datos que se utilizaron para codificarlos en primer lugar. Por esta razón, es muy fácil para un atacante decodificar los datos si tiene la información codificada. El ejemplo de tales algoritmos son ASCII, Unicode, Base64, etc.

Encriptacion

Los datos en Internet se mantienen confidenciales y seguros mediante cifrado. El cifrado hace que los datos sean ilegibles y difíciles de decodificar para un atacante y evita que sean robados.

El cifrado utiliza “claves criptográficas”. Con la clave, los datos se cifran en el extremo del remitente. Y, con la misma clave o una clave diferente, los datos se descifran en el extremo del receptor. Según el tipo de clave que se utiliza para cifrar/descifrar la información, el cifrado se clasifica en dos categorías:
– cifrado de clave simétrica
– cifrado de clave asimétrica

Analicemos estos dos tipos de técnicas de cifrado:

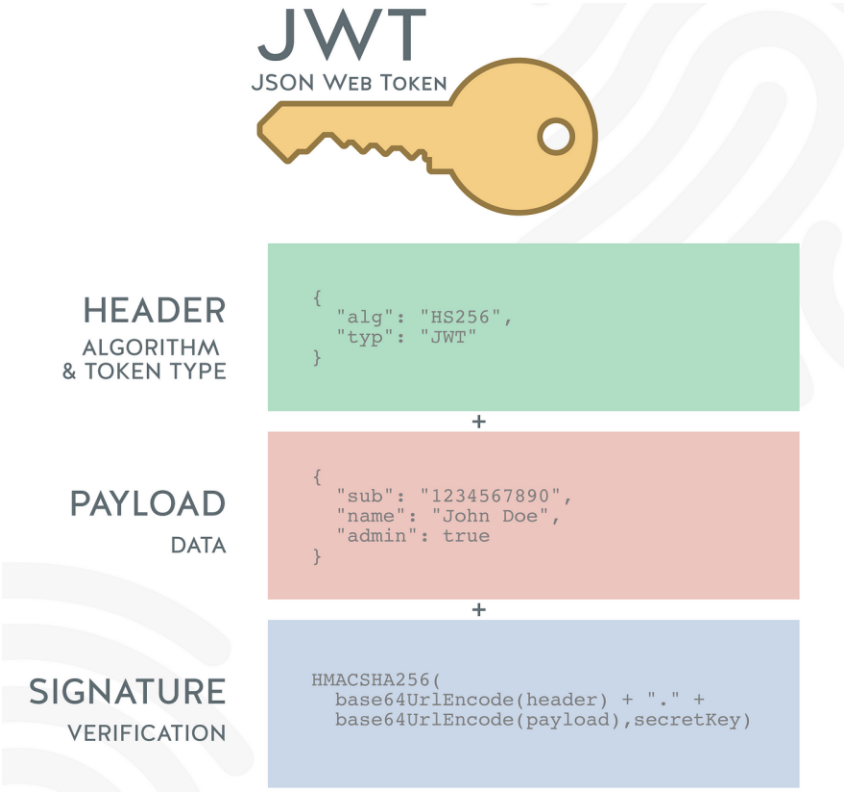


Hash

Un Hash es básicamente una cadena que se genera a partir de la cadena de entrada pasándola a través de un algoritmo Hash. Esta cadena hash es siempre de una longitud fija sin importar el tamaño de la cadena de entrada. El hash también se puede considerar como “cifrado unidireccional”, es decir, los datos una vez procesados no se pueden revertir a su forma original.

Con eso, significa que se asegura de que incluso si se cambia una sola cosa, tu puedes saber que se ha cambiado. El hash protege tus datos contra posibles alteraciones para que tus datos no se modifiquen ni un poco.

Estructura del JWT



Un JSON web token es un estándar de código abierto que se utiliza para transmitir información de manera segura entre dos partes en formato JSON. Está diseñado para ser un mecanismo de autenticación y autorización en aplicaciones web y servicios API. Los JWT son especialmente populares en el contexto de aplicaciones basadas en RESTful API y aplicaciones de una sola página (SPA).

Un JWT consta de tres partes: la cabecera (header), el payload y la firma. La cabecera contiene información sobre el algoritmo de cifrado utilizado y el tipo de token. El payload es el contenido propiamente dicho del token y contiene la información que se quiere transmitir.

La cabecera del JWT

La cabecera dentro de la estructura de un JSON web token es la que indica qué formato criptográfico está utilizando el token.

El payload del JWT

En el payload, como parte de la estructura de un JSON web token, es donde se encuentra la información propiamente dicha. Esta parte está codificada en Base64, lo que permite que cualquiera pueda ver su contenido.
En término de estructura el payload es un diccionario en donde cada par clave valor lo denominamos una “claim”

La firma del token

La firma (signature) es la parte de la estructura de un JSON web token que garantiza la autenticidad del JWT. Aquí es donde entra en juego la clave privada. La firma se genera usando el contenido de la cabecera y el

payload, además de una clave secreta que solo conoce el servidor. De esta manera, al recibir el JWT, el servidor puede verificar si ha sido alterado en el camino.

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

Codificación para transferencia en la web

La estructura de un JSON web token es bastante sencilla. Se compone de tres partes que van separadas por un punto. Cada una de estas partes está codificada en Base64. Por tanto, un JWT tendría esta apariencia:

//Estructura de un JSON web token
[Base64(header)].[Base64(payload)].[firma]
Lo importante de esto es que CUALQUIERA puede ver el contenido de cada parte utilizando alguna herramienta de decodificación Base64.

----------Preguntas parcial----------
Escriba una sentencia con LINQ que filtre los productos con un precio entre 20 y 30, ordenarlos por precio descendente. Elegir la opción con la sentencia de LINQ que falta

```
public class Producto
{
    public string Nombre { get; set; }
    public decimal Precio { get; set; }
}

public class ProductoTest
{
    List<Producto> productos = new List<Producto>
    {
        new Producto { Nombre = "Producto A", Precio = 25.99m },
        new Producto { Nombre = "Producto B", Precio = 15.50m },
        new Producto { Nombre = "Producto C", Precio = 30.75m },
        new Producto { Nombre = "Producto D", Precio = 20.00m },
        new Producto { Nombre = "Producto E", Precio = 22.40m }
    };

    public List<Producto> FiltrarProductosyOrdenarPorPrecio()
    {
        // La sentencia de LINQ va acá
    }
}
```

productos.Where(p => p.Precio >= 20 && p.Precio <= 30).OrderByDescending(p => p.Precio).ToList();

Dado el código representado en la imagen, de una API ejecutándose localmente en el puerto 7209.

Si hago una request con el verbo "GET" a la url " https://localhost:7209/WeatherForecast/Get" , sin indicar el header " Authorization".
¿Que respuesta obtengo?

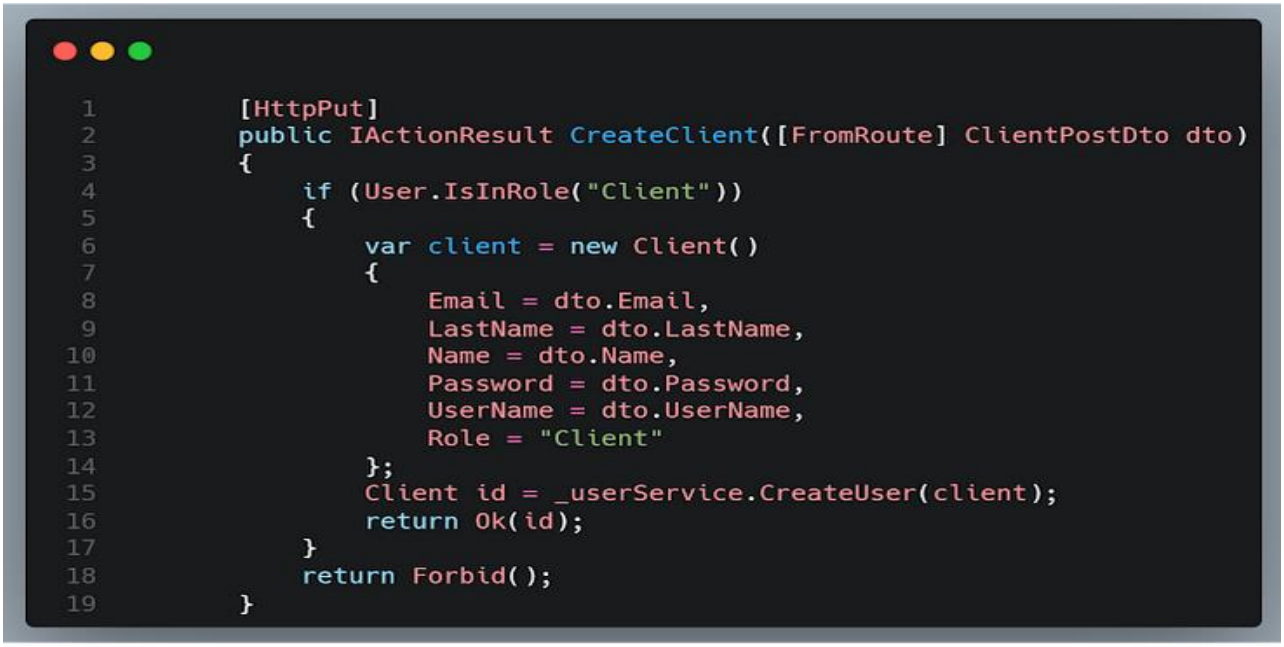
Status code: 404 (Not Found)
¿Cuales de las siguientes opciones son mecanismos para definir el modelo de datos en Entity Framework?
-Convenciones del framework.
-Data annotations. (También conocidas como mapping attributes)
-Fluent API sobreescribiendo el método OnModelCreating de la clase derivada de Context.

Indique cuales de las siguientes opciones son correctas respecto al patrón de diseño Data Transfer Objects.

-Las clases para transferencia de datos sirven para tomar datos que vienen informados en la request que se hacen a la API

- Los clases para transferencia de datos sirven para definir que datos queremos retornar cuando se hace una request a la API.

Usted se encuentra trabajando en una empresa de desarrollo de software y se le asigna la tarea de hacer una review al PR de un compañero.
Encuentre aquellas cosas que se encuentran mal en el código y que cambios sugeriría. Para cada caso indique la línea de código y el cambio sugerido.



```
1         [HttpPut]
2         public IActionResult CreateClient([FromRoute] ClientPostDto dto)
3         {
4             if (User.IsInRole("Client"))
5             {
6                 var client = new Client()
7                 {
8                     Email = dto.Email,
9                     LastName = dto.LastName,
10                    Name = dto.Name,
11                    Password = dto.Password,
12                    UserName = dto.UserName,
13                    Role = "Client"
14                };
15                Client id = _userService.CreateUser(client);
16                return Ok(id);
17            }
18            return Forbid();
19        }
```

Linea 1. (HttpPost) (ACLARACION: no me salen los corchetes)
Linea 2. public IActionResult CreateClient((FromBody) ClientPostDto dto)
Linea 15. client.id = _userService.CreateUser(client);
Linea 16. return Ok(client)

Comentarios

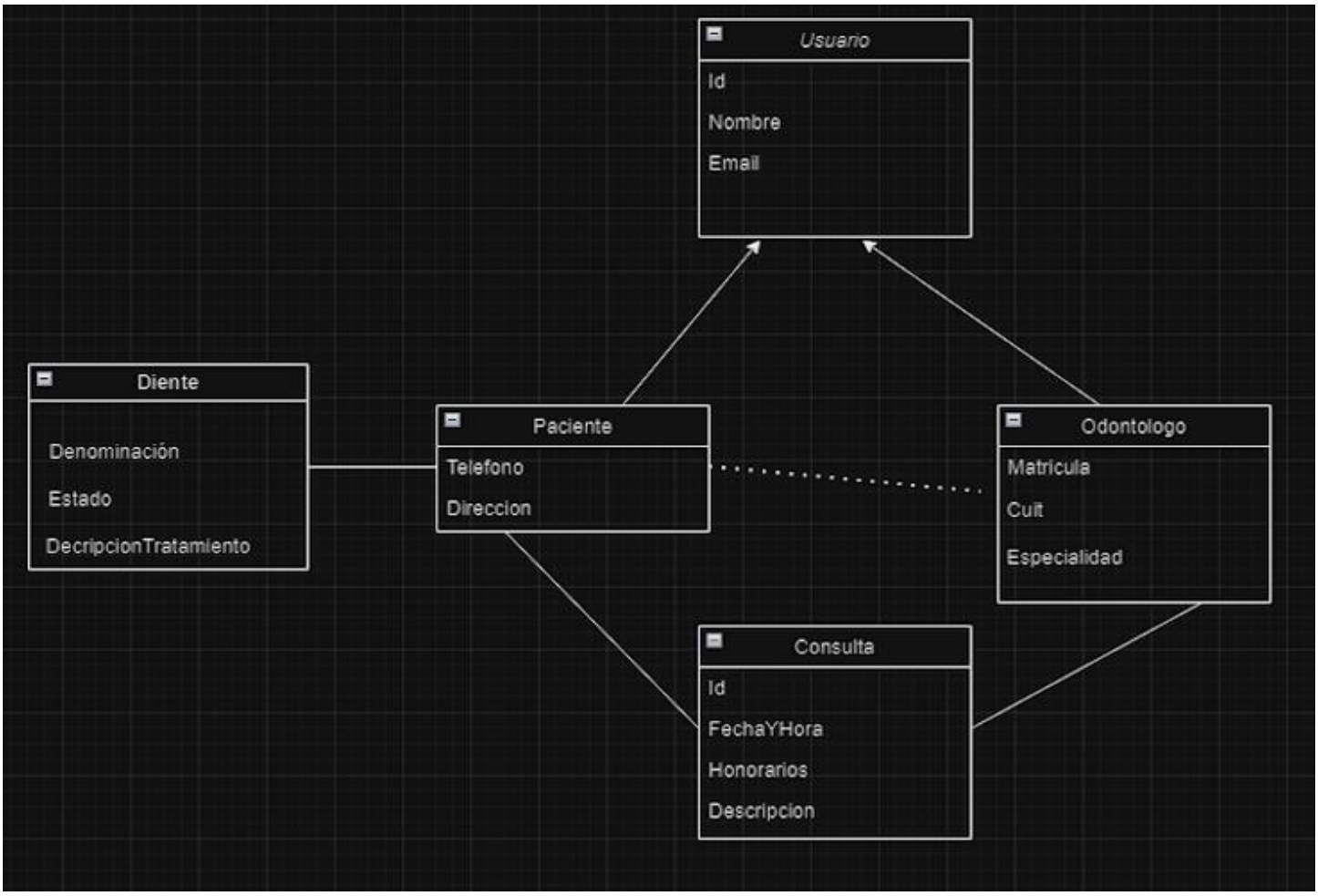
1-El httpverb correspondiente es un post y no un put
2-El dto deberia venir por el body con un [FromBody]
3-El role cliente no debería poder crear otro cliente, tendría que chequearse un rol admin o superior.
4-No debería retornar el objeto entero sino solo el id del elemento creado o nada debido a que hay información sensible en dicho objeto

¿Cuales de las siguientes opciones son correctas respecto al patrón de diseño de arquitectura en capas "Clean architecture"?
- 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.
- 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.

¿Cuales de las siguientes opciones son verdaderas respecto a los JWT (JSON Web Token)?
- Sirven para realizar el proceso de autentificación y autorización.
- Al estar firmados digitalmente, me aseguran que el contenido del mismo no fue alterado.

Observe los siguientes diagrama de clase y marque las opciones que crea correcta
En una clínica odontológica los pacientes son registrados en el sistema así como la/las consultas que ha tenido con un odontólogo. Cada consulta tiene sus honorarios y se registra en un día y hora determinado sin importar a la hora efectiva que se hace ya sea por retraso del odontólogo o el paciente.

Se guarda registro de los dientes de cada paciente asi como de los tratamientos que cada diente ha tenido, si fuera el caso, y el estado de dicho diente para este paciente.
Tanto el odontólogo como el paciente tienen un usuario en el sistema para poder ver, y cargar en caso de los dentistas, los datos de las consultas que les conciernan a cada uno



-La relación entre paciente y consulta es 1 a n
- La relación entre usuario, paciente y odontólogo esta mal representada
Usted se encuentra iniciando un proyecto de 0. El analista de su equipo le hace entrega de un bosquejo del diagrama de clases del sistema a desarrollar.

Se le pide en su tarea asignada la creación de un proyecto en .net y su base de datos relacional que refleje el diagrama de clase enunciado siguiendo el paradigma code first.
¿Cual de los siguientes caminos le parece mejor solución?
Justifique en Otros
-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.
Justificacion: Lo haria de esa manera ya que las tablas y migraciones en algunos casos cuando se realizan cambios en el codigo, tenemos que crearlas de vuelta, de esta manera la hacemos al final ahorrandonos el tema de tener que repetir ese proceso.

¿Cuales de las siguientes opciones son atributos que se utilizan para definir que action method (método perteneciente a un Controller) se va a invocar a partir de la request recibida por la API.?
-[HttpGet]
-[Route]

Sos un desarrollador junior trabajando en una aplicación de gestión de ventas. Tenés una lista de ventas donde cada una tiene información sobre el Id, Producto, Cantidad y Precio. Necesitas encontrar todos los productos que se han vendido por más de $1000 en total. La lista de ventas se define como:

```
public class Venta
{
    public int Id { get; set; }
    public string Producto { get; set; }
    public int Cantidad { get; set; }
    public decimal Precio { get; set; }
}

List<Venta> ventas = new List<Venta>
{
    new Venta { Id = 1, Producto = "Laptop", Cantidad = 5, Precio = 300 },
    new Venta { Id = 2, Producto = "Monitor", Cantidad = 2, Precio = 150 },
    new Venta { Id = 3, Producto = "Teclado", Cantidad = 10, Precio = 30 },
    new Venta { Id = 4, Producto = "Mouse", Cantidad = 15, Precio = 20 },
    new Venta { Id = 5, Producto = "Laptop", Cantidad = 3, Precio = 300 },
    new Venta { Id = 6, Producto = "Monitor", Cantidad = 3, Precio = 150 }
};
```

- ○ var productos = ventas.Where(v => v.Precio > 1000).Select(v => v.Producto).Distinct();
- ◉ var productos = ventas.GroupBy(v => v.Producto).Where(g => g.Sum(v => v.Cantidad * v.Precio) > 1000).Select(g => g.Key);
- ○ var productos = ventas.Select(v => v.Producto).Where(p => ventas.Sum(v => v.Cantidad * v.Precio) > 1000);
- ○ var productos = ventas.GroupBy(v => v.Producto).Where(g => g.Average(v => v.Cantidad * v.Precio) > 1000).Select(g => g.Key);

¿Cuales de las siguientes opciones son atributos que se utilizan para definir que action method (método perteneciente a un Controller) se va a invocar a partir de la request recibida por la API.?

[HttpPut, HttpPost]

En donde se encuentra la vulnerabilidad del mecanismo de Autenticación con JWT?

- [x] El payload y el header es decodificable y por lo tanto las credenciales que se pueden sustraer del mismo
- [ ] Dependiendo el algoritmo de hashing puede ser roto y extraerse el salt alli oculto
- [ ] Si el JWT se pierde no se puede acceder mas a la cuenta, solo reseteando las credenciales
- [x] Si el JWT es robado cualquiera puede hacer acciones en nombre del legitimo propietario
- [ ] Ninguna, el JWT garantiza que nadie robe la identidad del propietario

Respuesta correcta

- [x] Si el JWT es robado cualquiera puede hacer acciones en nombre del legitimo propietario

Dado el código representado en la imagen, de una API ejecutándose localmente en el puerto 7209, configurada para autentificar por JWT token.

Si hago una request con el verbo "GET" a la url " https://localhost:7209/api/WeatherForecast/Get" , sin indicar el header " Authorization".

¿Que respuesta obtengo?

- ○ Status code: 200, Response body: Hola.
- ○ Status code: 401(Unauthorized)
- ◉ Status code: 404 (Not Found)
- ○ Status code: 500 (Internal error)

Respuesta correcta

- ◉ Status code: 401(Unauthorized)

El siguiente código hace referencia a un contexto en una aplicación de Ventas, donde se manejan las entidades Pedidos, Clientes, Productos (asumir que las entidades ya están creadas) con el enfoque codefirst. Quisiera representar la relación entre pedidos y productos. ¿Qué acción / acciones debería agregar, teniendo en cuenta la acción y el código propuesto para cada acción?

```
public class VentasContext : DbContext
{
    public DbSet<Cliente> Clientes { get; set; }
    public DbSet<Pedido> Pedidos { get; set; }
    public DbSet<Producto> Productos { get; set; }

    public VentasContext(DbContextOptions<VentasContext> options)
            : base(options)
            {

            }
    protected override void OnModelCreating(ModelBuilder
mode{Builder)
        modelBuilder.Entity<Cliente>()
            .HasMany(c => c.Pedidos)
            .WithOne(p => p.Cliente)
            .HasForeignKey(p => p.ClienteId);

        /* ¿Qué acción(es) debería agregar aquí para representar
            la relación muchos a muchos entre Pedidos y Productos? */
    }
}
```

- [x] Definir una relación muchos a muchos entre Pedidos y Productos

- [ ] Definir una nueva entidad llamada PedidoProducto para representar la relación muchos a muchos:

- [ ] Definir relación muchos a muchos entre Pedidos y Productos con una tabla de unión personalizada llamada PedidoProducto

- [ ] Agregar una columna adicional a la entidad Pedido que mantenga una lista de ProductoIds

- [ ] Agregar una tabla de unión a la base de datos manualmente en SQL y configurar la relación en el modelo.

Usted es desarrollador backend en un equipo de desarrollo. Su tech lead le pide que lo ayude con la review de un pr de un colega suyo. La aplicación desarrollada por su equipo sigue las practicas dadas por la catedra de p3 *

Enuncie los comentarios que haría a dicho a pr con los cambios que sugeriría. Por cado comentario indique línea o líneas de código y sugerencia de cambio.

Tenga presente que la metodología de corrección es idéntica a los multiplechoice. Comentario acerado suma, indistinto o irrelevante no suma ni resta y equivocado resta sobre el puntaje obtenido en la pregunta

```
1 using Data.Repositories;
2 using Microsoft.AspNetCore.Mvc;
3
4 [ApiController]
5 [Route("api/Article")]
6 public class ArticleController : ControllerBase
7 {
8     private  readonly ArticleRepository _repo;
9     public ArticleController(ArticleRepository repo)
10    {
11        _repo = repo;
12    }
13
14    [HttpGet("{idArticle}")]
15    public ActionResult Get([FromBody]int idArticle)
16    {
17        var articleToReturn = _repo.Get(idArticle);
18        return Ok(articleToReturn);
19    }
20
21    [HttpGet]
22    public ActionResult AddArticle([FromBody] ArticleToAdd articleDto)
23    {
24        _repo.AddArticle(new Article() {});
25        return NoContent();
26    }
27
28 }
```

1: Línea 11: Según clean arquitecture los controladores no deberían tener interacción con la capa de repositorios, debería haber una inyección de un servicio hecho en la capa de aplicación
2: Línea 15: un get no debería recibir un body, recibirlo por ruta con un [FromRoute]
3: Línea 2: el http ver debería ser un POST
4: Línea 24: no se esta añadiendo un articulo con el contenido del articulo recibido por body sino un articulo vacío. Enviar el dto al servicio