Clean Architecture
(Arquitectura Limpia)

Principios de arquitectura para el desarrollo de aplicaciones poco acopladas

Autor: Luz Elena Galindo Gamboa

Principios de Arquitectura

Autor: Luz Elena Galindo Gamboa

Separation of concerns - Separación de intereses

Este principio afirma que **el software se debe separar en función de los tipos de trabajo que realiza.** Un ejemplo rápido sería una aplicación con lógica para identificar elementos a mostrar y otra que le da formato a dichos elementos. Ambos comportamientos deben mantenerse separados.

Desde el punto de vista de la arquitectura, las aplicaciones se pueden crear de forma lógica para seguir este principio mediante la **separación del comportamiento de negocios principal de la lógica de la interfaz de usuario y la infraestructura.**

Autor: Luz Elena Galindo Gamboa

Encapsulation - Encapsulación

Las diferentes partes de una aplicación deben usar la encapsulación para aislarse de otras partes de la aplicación. Las capas y los componentes de la aplicación deben poder ajustar su implementación interna sin interrumpir a sus colaboradores mientras no se infrinjan los externos.

El uso correcto de la encapsulación contribuye a lograr el acoplamiento flexible y la modularidad en los diseños de aplicaciones, ya que **los objetos y paquetes se pueden reemplazar con implementaciones alternativas, siempre y cuando se mantenga la misma interfaz.**

Autor: Luz Elena Galindo Gamboa

Dependency inversion - Inversión de dependencias

La dirección de dependencia dentro de la aplicación debe estar en la dirección de la abstracción, no de los detalles de implementación.

La Inversión de dependencias es una parte fundamental de la creación de aplicaciones de acoplamiento flexible, ya que **se pueden escribir detalles de implementación de los que depender e implementar abstracciones de nivel superior**, en lugar de hacerlo al revés.

Autor: Luz Elena Galindo Gamboa

Explicit dependencies - Dependencias explícitas

Los métodos y las clases deben requerir explícitamente todos los objetos de colaboración que necesiten para funcionar correctamente.

Los **constructores de clases** proporcionan una oportunidad para que las clases **identifiquen lo que necesitan para** poder tener un estado válido y **funcionar correctamente.**

Seguir el principio hace que el código sea más autoexplicativo y los contratos de codificación más fáciles de usar.

Autor: Luz Elena Galindo Gamboa

Single responsibility - Responsabilidad única

Indica que **los objetos solo deben tener una responsabilidad y solo una razón para cambiar.** En concreto, la única situación en la que **el objeto debe cambiar** es **si hay que actualizar la manera en la que lleva a cabo su única responsabilidad**.

Agregar clases nuevas siempre es más seguro que cambiar las existentes, puesto que todavía no hay código que dependa de las clases nuevas. También se puede aplicar el principio de responsabilidad única a las capas de la aplicación.

Cuando este principio se aplica a la arquitectura de la aplicación y se lleva a su punto de conexión lógico, se obtienen microservicios.

Autor: Luz Elena Galindo Gamboa

Don't repeat yourself (DRY) - No te repitas

La aplicación debe evitar especificar el comportamiento relacionado con un determinado concepto en varios lugares, ya que esta práctica es una fuente de errores frecuente.

En lugar de duplicar la lógica, se puede encapsular en una construcción de programación. Convierta esta construcción en la única autoridad sobre este comportamiento y haga que cualquier otro elemento de la aplicación que requiera este comportamiento use la nueva construcción.

Autor: Luz Elena Galindo Gamboa

Persistence ignorance - Omisión de persistencia

La Omisión de persistencia (PI) hace referencia a los tipos que se deben conservar, pero cuyo código **no se ve afectado por la elección de la tecnología de persistencia**.

En una aplicación, **las clases que modelan el dominio del negocio no deben verse afectadas por la forma en que persisten los datos.**

Autor: Luz Elena Galindo Gamboa

Bounded contexts - Contextos delimitados

Los contextos delimitados son un patrón esencial en el Domain-Driven Design. **Proporcionan una manera de abordar la complejidad en organizaciones o aplicaciones de gran tamaño dividiéndola en módulos conceptuales independientes**.

Cada módulo conceptual representa un contexto que está separado de otros contextos (por tanto, delimitado) y que puede evolucionar independientemente. Cada contexto delimitado debería poder **elegir sus propios nombres** para los conceptos que contiene y **tener acceso exclusivo a su propio almacén de persistencia.**

La comunicación entre los contextos delimitados se realiza a través de interfaces de programación, en lugar de una base de datos compartida.

Autor: Luz Elena Galindo Gamboa

Capas en la Arquitectura de software

Autor: Luz Elena Galindo Gamboa

¿Qué son las capas?

Cuando aumenta la complejidad de las aplicaciones, una manera de administrarla consiste en dividir la aplicación según sus responsabilidades o intereses.

Al organizar el código en capas, la funcionalidad común de bajo nivel se puede reutilizar en toda la aplicación.

Con una arquitectura en capas, las aplicaciones pueden aplicar **restricciones sobre qué capas se pueden comunicar con otras capas.** Esta arquitectura permite lograr la encapsulación. Cuando se cambia o reemplaza una capa, solo deberían verse afectadas aquellas capas que funcionan con ella.

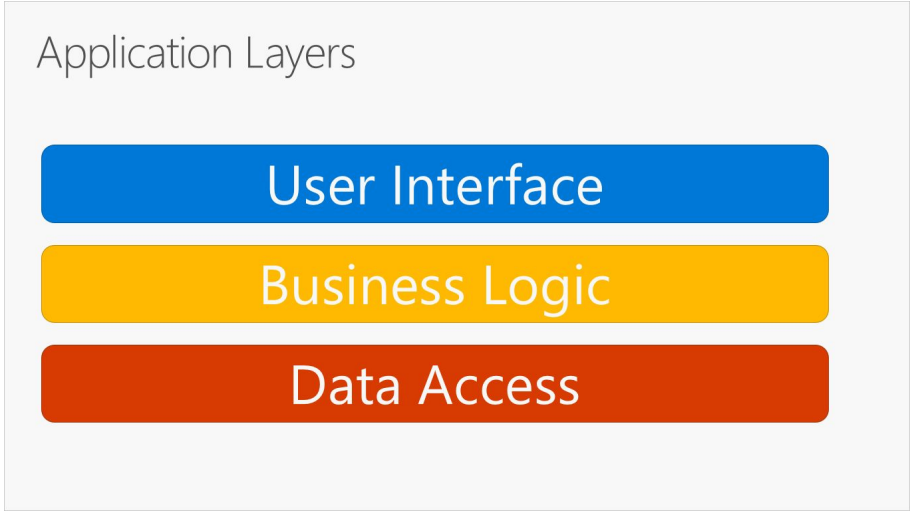
Autor: Luz Elena Galindo Gamboa

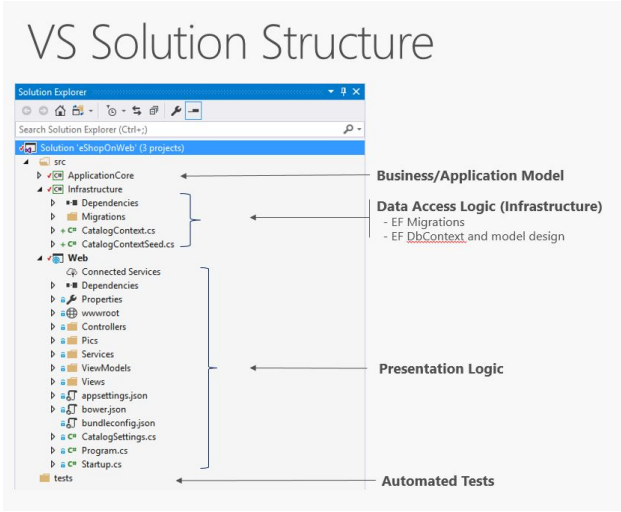
¿Qué son las capas?

Además de la posibilidad de intercambiar las implementaciones en respuesta a cambios futuros en los requisitos, las capas de aplicación también facilitan el intercambio de implementaciones con fines de prueba.

Aplicaciones tradicionales de arquitectura de "N capas"

Estas capas se suelen abreviar como UI (interfaz de usuario), BLL (capa de lógica de negocios) y DAL (capa de acceso a datos).





Autor: Luz Elena Galindo Gamboa

Aplicaciones tradicionales de arquitectura de "N capas"

Una desventaja de este enfoque de distribución en capas tradicional es que las dependencias de tiempo de compilación se ejecutan desde la parte superior a la inferior. Es decir, la capa de interfaz de usuario depende de BLL, que depende de DAL.

Clean Architecture - Arquitectura Limpia

Autor: Luz Elena Galindo Gamboa

Clean Architecture - Arquitectura Limpia

Las aplicaciones que siguen los principios de **Dependency Inversion**, así como de **Domain-Driven Design** tienden a tener arquitecturas bastante similares.

Estas arquitecturas han recibido varios nombres:

- Hexagonal Architecture - Arquitectura Hexagonal
- Ports-and-Adapters - Puertos y Adaptadores
- Onion Architecture - Arquitectura de Cebolla
- Clean Architecture - Arquitectura Limpia

Autor: Luz Elena Galindo Gamboa

Clean Architecture - Arquitectura Limpia

La arquitectura limpia coloca el modelo de lógica de negocios y aplicación en el centro de la aplicación.

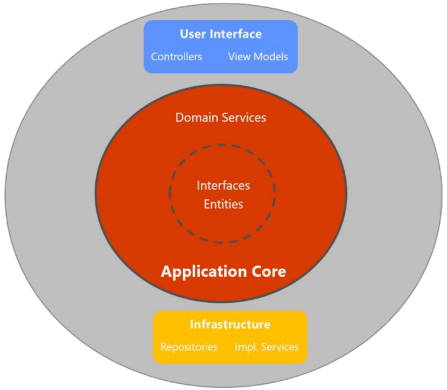
En lugar de tener lógica de negocios que depende del acceso a datos o de otros aspectos de infraestructura, esta dependencia se invierte: **los detalles de la infraestructura y la implementación dependen del núcleo de la aplicación**.

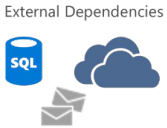
Esta función se logra mediante la definición de abstracciones o interfaces, en el núcleo de la aplicación, que después se implementan mediante tipos definidos en el nivel de infraestructura.

Autor: Luz Elena Galindo Gamboa

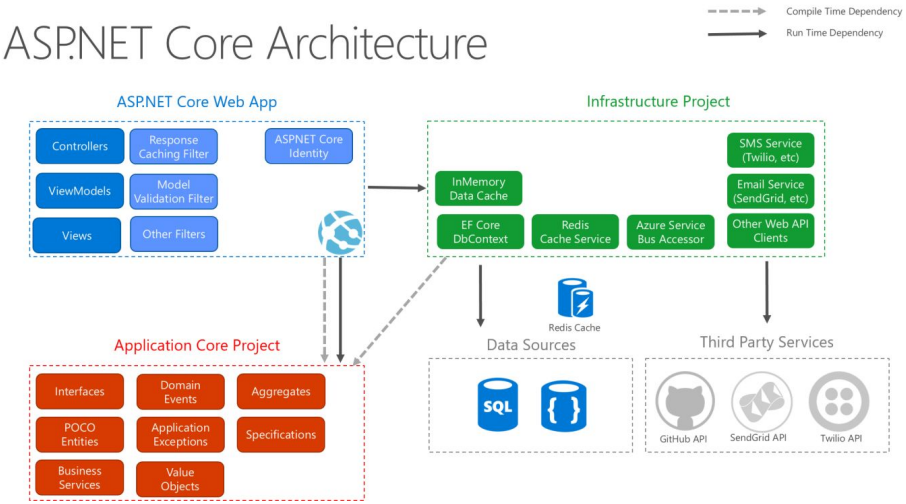
Clean Architecture - Arquitectura Limpia

Clean Architecture Layers (Onion view)



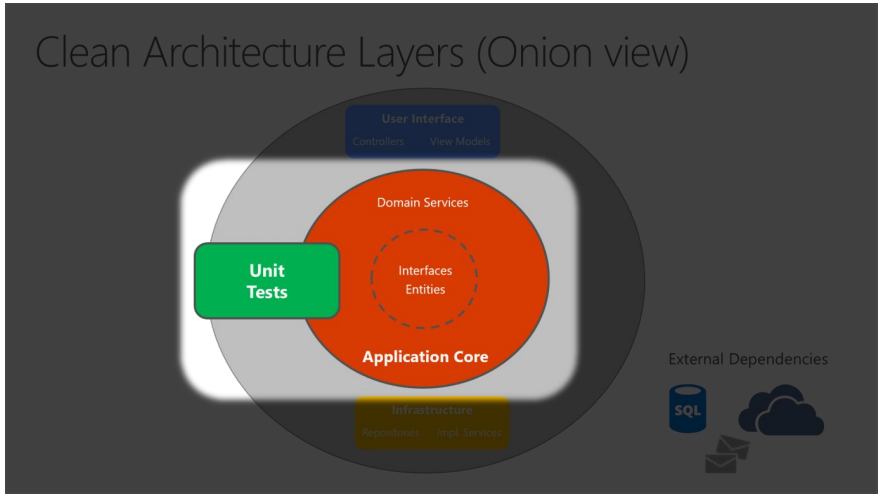


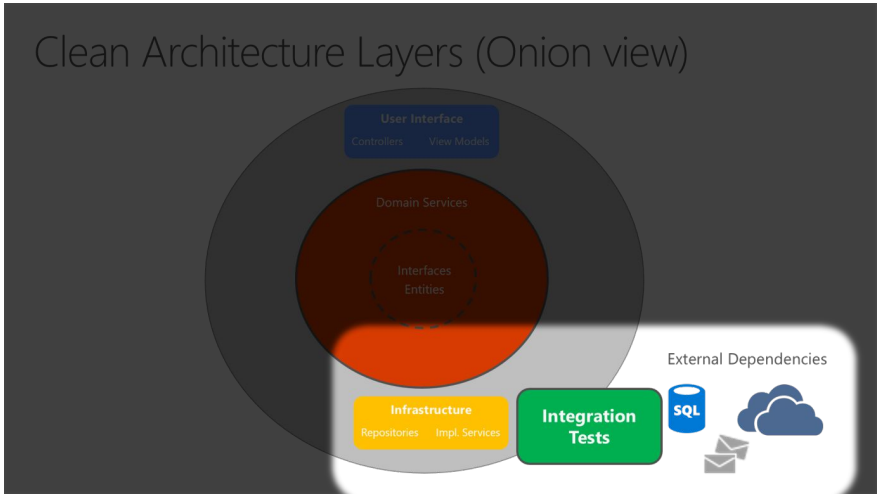
ASP.NET Core Architecture



Autor: Luz Elena Galindo Gamboa

Clean Architecture - Arquitectura Limpia





Autor: Luz Elena Galindo Gamboa

Organización del código

Autor: Luz Elena Galindo Gamboa

Application Core - Núcleo de la aplicación

El núcleo de la aplicación contiene el modelo de negocio, que incluye entidades, servicios e interfaces.

Estas interfaces incluyen abstracciones para las operaciones que se llevarán a cabo mediante la infraestructura, como el acceso a datos, el acceso al sistema de archivos, las llamadas de red, etc.

En ocasiones los servicios o interfaces definidos en este nivel tendrán que trabajar con tipos sin entidad que no tienen dependencias en la interfaz de usuario o la infraestructura. Estos se pueden definir como Objetos de transferencia de datos (DTO) simples.

Autor: Luz Elena Galindo Gamboa

Application Core Types - Tipos de núcleo de la aplicación

- Entities - Entidades (las clases de modelo de negocio que se conservan)
- Interfaces - Interfaces
- Services - Servicios
- DTOs - Data Transfer Objects

Autor: Luz Elena Galindo Gamboa

Infrastructure - Infraestructura

El proyecto de infraestructura incluye normalmente las implementaciones de acceso a datos.

Además de las implementaciones de acceso a datos, **el proyecto de infraestructura debe contener las implementaciones de los servicios que tienen que interactuar con los intereses de infraestructura.** Estos servicios deben implementar interfaces definidas en el núcleo de la aplicación, por lo que la infraestructura deberá tener una referencia al proyecto del núcleo de la aplicación.

Infrastructure Types - Tipos de Infraestructura

- EF Core types (DbContext, Migration)
- Repositories - Repositorios (Tipos de implementación de acceso a datos)
- Servicios específicos de la infraestructura (ej. FileLogger or SmtpNotifier)

Autor: Luz Elena Galindo Gamboa

UI Layer - Interfaz de Usuario

La capa de interfaz de usuario en una aplicación ASP.NET Core MVC es el punto de entrada para la aplicación. Este proyecto debe hacer referencia al proyecto Application Core y sus tipos deben interactuar con la infraestructura estrictamente a través de las interfaces definidas en Application Core.

En la capa de interfaz de usuario no se debe permitir la creación de instancias directas o llamadas estáticas a los tipos de la capa de infraestructura.

La clase Startup es responsable de configurar la aplicación y de conectar los tipos de implementación a las interfaces, lo que permite que la inserción de dependencias funcione correctamente en tiempo de ejecución.

Autor: Luz Elena Galindo Gamboa

UI Layer Types - Tipos de capa de interfaz de usuario

- Controllers - Controladores
- Filters - Filtros
- Views - Vistas
- ViewModels
- Startup

Autor: Luz Elena Galindo Gamboa

Fuentes

Autor: Luz Elena Galindo Gamboa

Fuentes

https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/

https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/architectural-principles

https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures

https://deviq.com/principles/persistence-ignorance