

- Manejar la propagación de resultados y excepciones.

Beneficios y Usos:

- **Interfaces de Usuario Responsivas:** Evita que la aplicación se "congele" al realizar operaciones que consumen mucho tiempo, permitiendo que el usuario siga interactuando.
- **Escalabilidad:** Mejora la capacidad de una aplicación (especialmente las aplicaciones web) para manejar múltiples peticiones concurrentemente, al liberar hilos que de otro modo estarían bloqueados esperando operaciones externas.
- **Eficiencia de Hilos:** Utiliza los hilos de manera más eficiente al no bloquearlos innecesariamente mientras se espera una respuesta de una operación de I/O (por ejemplo, una descarga de internet o una consulta a una base de datos). Para operaciones ligadas a la CPU (cálculos intensivos), Task.Run se combina con async/await para delegar el trabajo a un grupo de hilos.
- **Código más Limpio:** Permite escribir código asincrónico de una manera que se parece mucho al código sincrónico, evitando las complejidades de las devoluciones de llamada anidadas (callbacks).

UNIDAD 3

PRIMERA PARTE

Definición

Vistas

Proceso

Arquitectura



Marco
Teórico

Requisitos
Funcionales
y No
Funcionales

Estilos

Estándares

MARCO TEORICO

CONCEPTOS

INTERFACES

Es una colección de operaciones que sirven para especificar un servicio de una clase o un componente. Una interfaz bien estructurada nos da una clara separación entre las vistas externas e internas de una abstracción, haciendo posible comprender y abordar una abstracción sin tener que sumergirse en los detalles de su implementación.

Ejemplo: Un cable USB-C a USB-A, es una interfaz. Nos permite conectar dos componentes sin importar cuales sean.

COMPONENTES

Un componente es una parte lógica y reemplazable de un sistema que conforma y proporciona la realización de interfaces.

Los buenos componentes definen abstracciones precisas, con interfaces bien definidas.

PAQUETES

Se utilizan para organizar los elementos de modelado en partes mayores que se pueden manipular como un grupo. La visibilidad puede controlarse.

Un paquete bien diseñado agrupa elementos cercanos semánticamente que suelen cambiar juntos

Representación graficologica de una arquitectura

ARTEFACTOS Y NODOS

Se utilizan para modelar los elementos físicos que pueden hallarse en un nodo. Por ejemplo, ejecutables, bibliotecas, tablas, archivos y documentos.

Un nodo es un elemento físico que existe en runtime y representa un recurso computacional. Un nodo representa un procesador o un dispositivo sobre el que se pueden desplegar los artefactos.

Los buenos nodos representan con claridad el vocabulario de hardware en el dominio de la solución

PRINCIPIOS

Son guías generales o reglas fundamentales que ayudan a diseñar, escribir y mantener código de buena calidad. Buscan mejorar la legibilidad, mantenibilidad, escalabilidad y robustez de las aplicaciones, facilitando el trabajo en equipo y la evolución del software a lo largo del tiempo.

A la hora de diseñar un sistema, ayudan a crear una arquitectura que se ajuste a las prácticas demostradas, que minimicen los costes de mantenimiento y maximicen la usabilidad y extensibilidad.

SEPARACION DE INTERESES (SEPARATION OF CONCERNS)

A un nivel bajo, se relaciona con el Principio de Responsabilidad Única, evitando que diferentes funcionalidades se mezclen en el código o diseño.

La separación de preocupaciones suele ser una consecuencia natural de seguir el principio "**No te repitas**" (**DRY**), ya que obliga a crear abstracciones que encapsulan conceptos repetidos. Si estas abstracciones están lógicamente agrupadas, se logrará la separación de preocupaciones.

Finalmente, la separación de preocupaciones es un principio rector de la **Arquitectura Limpia**, que enfatiza una fuerte división entre la lógica de negocio y las decisiones técnicas.

SOLID

Single Responsibility Principle (Responsabilidad Única):

- Una clase debe tener una y solo una razón para cambiar.

Open Close Princple (Abierto/Cerrado)

- Las entidades deben estar abiertas para extensión pero cerradas para modificación

Liskov Substitution Principile (Substitución de Liskov)

- Si un programa utiliza un tipo base, debe poder usar cualquier subtipo de ese tipo base sin que el programa se rompa. Los objetos deben ser reemplazables por instancias de sus subtipos sin alterar la corrección de ese programa.

Interface Segregation Principle (Segregación de Interfaces o ISP)

- Es mejor tener muchas interfaces pequeñas y específicas que una interfaz grande y monolítica

Dependency Inversion Principile (Inversión de Dependencias)

- Los módulos de alto nivel no deben depender de módulos de bajo nivel.

PATRONES

Son soluciones habituales a problemas comunes en el diseño de software. Cada patrón es como un plano que se puede resolver un problema de diseño particular del código.

ADAPTER O ADAPTADOR

Es un patrón que permite la colaboración entre objetos con interfaces incompatibles

FACADE

Proporciona una interfaz simplificada a una biblioteca, un frameworks o cualquier otro grupo complejo de clases

INYECCIÓN DE DEPENDENCIAS

Un patrón donde un objeto no crea directamente sus dependencias, sino que estas le son proporcionadas externamente.

MVC O MODELO VISTA CONTROLADOR

Busca separar las responsabilidades en:

- Modelo: Representa la lógica de negocio y los datos de la aplicación

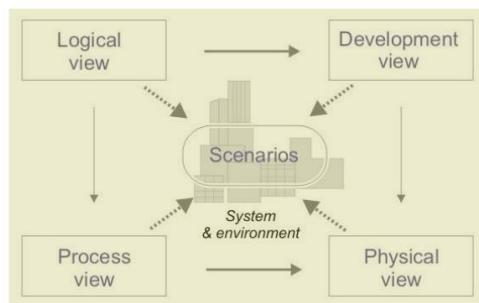
- Interactua con la base de datos
 - Contiene reglas de validación y lógica para manipular la información
 - Es independiente de la interfaz de usuario
- Vista
 - Se encarga de la presentación de datos al usuario
 - Define como se muestra la información(interfaz)
 - No tiene lógica de negocio.
- Controlador
 - Actua como intermediario entre el modelo y la vista
 - Recibe solicitudes del usuario, como un URL o un evento.
 - Procesa esas solicitudes
 - Decide que vista debe mostrarse y le pasa los datos relevantes del modelo
 - No tiene lógica de negocio.

ARQUITECTURA

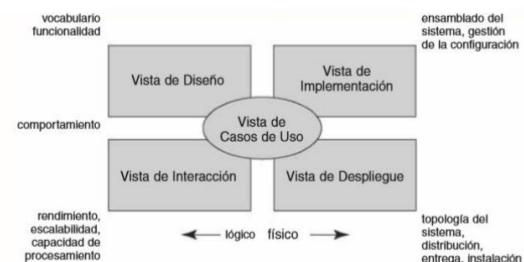
La Arquitectura de Software es el conjunto de las decisiones de diseño fundamentales y significativas sobre la organización de un sistema de software. Estas decisiones abarcan:

- Estructuras del sistema
- Estilo y patrones
- La forma en que el sistema se divide y como sus partes colaboran con requisitos funcionales y requisitos de calidad
- Abstracciones de alto nivel.

Es imposible representar toda la información relevante sobre la arquitectura en un solo modelo arquitectónico. Para el desarrollo, por lo general, se necesita presentar múltiples vistas de la arquitectura de software

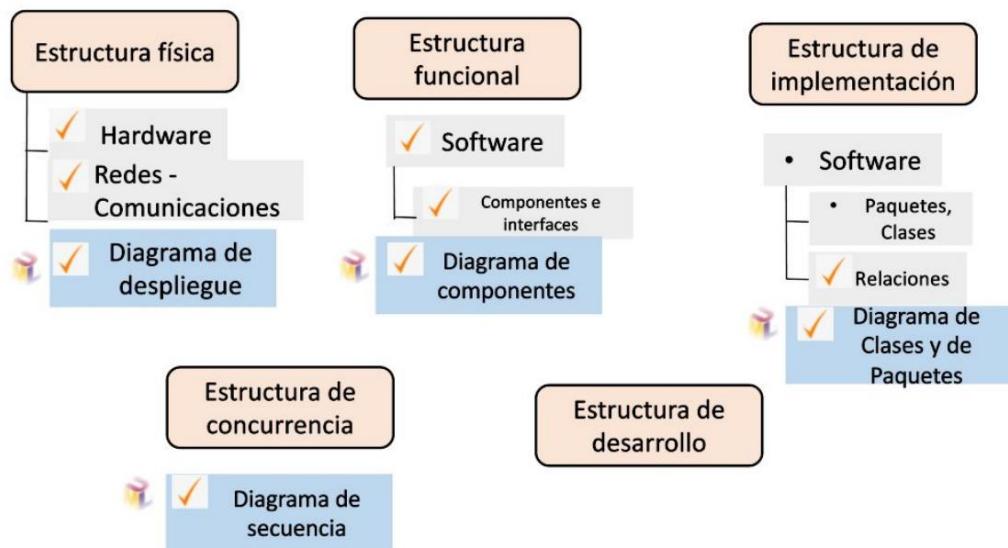


Vistas 4 + 1 (Kruchten)



5 vistas del UP

Diagramas para modelar la Arquitectura



El Proceso de Diseño de la Arquitectura



CLEAN ARCHITECTURE

Es un conjunto de principios de diseño de software que busca crear sistemas independientes, testables, mantenibles y flexibles a lo largo del tiempo. Su objetivo principal es proteger la lógica de negocio central de los detalles de implementación externos, como frameworks, bases de datos o interfaces de usuario.

Se basa en el principio de **separación de preocupaciones** y, fundamentalmente, en la **Regla de Dependencia**: las dependencias siempre deben apuntar hacia adentro, de las capas externas a las capas internas. Las capas internas no deben saber nada sobre las capas externas.

CAPAS INTERNAS Y EXTERNAS

Las capas internas (Entidades y Casos de Uso) contienen la lógica de negocio central y son independientes de todo

Las capas eternas (Adaptadores de interfaz, Frameworks y Drivers) manejan los detalles de la implementación y dependen de las capas enteras.

REST Y RESTFUL

REST es un estilo arquitectónico para WWW. El objetivo es lograr un sistema distribuido escalable, robusto y eficiente.

Se basa en la idea de que los recursos pueden ser manipulados utilizando un conjunto limitado de operaciones predefinidas(métodos HTTP estándar) sobre representaciones de esos recursos.

Las **restricciones arquitectónicas** clave de REST son:

1. **Client-Server (Cliente-Servidor):** Clara separación de responsabilidades entre la interfaz de usuario (cliente) y el almacenamiento de datos (servidor).
2. **Stateless (Sin Estado):** Cada solicitud del cliente al servidor debe contener toda la información necesaria para que el servidor la entienda y la procese. El servidor no debe almacenar ningún estado de sesión del cliente entre solicitudes.
3. **Cacheable (Cachable):** Las respuestas del servidor deben indicar si los datos son cachables o no, para que los clientes puedan almacenar en caché las respuestas y mejorar el rendimiento.
4. **Uniform Interface (Interfaz Uniforme):** La interacción entre cliente y servidor debe ser consistente y genérica, facilitando la escalabilidad y simplificando el sistema. Esto se logra mediante:
 - **Identificación de recursos:** Los recursos se identifican mediante URIs (Uniform Resource Identifiers).
 - **Manipulación de recursos a través de representaciones:** El cliente interactúa con el recurso a través de su representación (ej. JSON, XML).
 - **Mensajes autodocumentados:** Cada mensaje contiene suficiente información para describir cómo procesarlo.
 - **HATEOAS (Hypermedia as the Engine of Application State):** El servidor guía al cliente a través de la aplicación proporcionando enlaces dentro de las representaciones de los recursos.
5. **Layered System (Sistema en Capas):** Permite que un sistema esté compuesto por múltiples capas (proxies, gateways, balanceadores de carga) sin que el cliente se dé cuenta.
6. **Code on Demand (Código bajo demanda - opcional):** Permite al servidor extender la funcionalidad del cliente enviando código ejecutable (ej. JavaScript).

Una API es considerada **RESTful** si cumple con las restricciones arquitectónicas de REST. Es decir que, si REST es el estilo, RESTful es la implementación práctica de ese estilo.

Todas las APIs RESTful son, por definición, implementaciones de REST. Pero no todas las APIs que usan HTTP son realmente "RESTful" si no cumplen con todas las restricciones de REST (especialmente la de "interfaz uniforme" y "sin estado"). Una API "RESTful" es una API que sigue el estilo REST de manera apropiada.

Buenas Prácticas (Nombres, Verbos HTTP, Estado)

Para construir APIs RESTful de calidad, se siguen ciertas convenciones y buenas prácticas:

- **Nombres de Recursos (URIs):**
 - **Usar sustantivos (en plural) para representar colecciones:** Evitar verbos en las URIs. Los verbos son la función de los métodos HTTP.
 - /users
 - ✗ /getAllUsers
 - **Identificar recursos individuales con IDs:**
 - /users/123
 - ✗ /getUserById/123
 - **Reflejar relaciones entre recursos:**
 - /users/123/orders (órdenes del usuario 123)
 - /orders/456/user (usuario de la orden 456, si aplica)
 - **Nombres en plural y consistentes:** Si decides usar plurales, úsalos siempre.
 - /products, /orders
 - ✗ /product, /order (aunque algunos prefieren singular, la pluralidad es más común para colecciones)
- **Verbos HTTP (Métodos HTTP):** Utilizar los verbos HTTP de manera semántica para indicar la acción deseada sobre el recurso.
- **Códigos de Estado HTTP:** Utilizar los códigos de estado HTTP para indicar el resultado de la operación.

HTTP

HyperText Transfer Protocol, es un protocolo para la comunicación de datos en la WWW. Define como los clientes y los servidores se comunican para intercambiar información

VERBOS HTTP

Son métodos que indican la acción deseada. Comandos que un cliente envía a un servidor

GET:

- Sirve para recuperar una representación de un recurso o una colección de recursos

POST:

- Sirve para enviar datos al servidor para crear un nuevo recurso subordinado a la URI dada

PUT:

- Sirve para reemplazar completamente la representación de un recurso existente con la nueva representación proporcionada.

DELETE:

- Eliminar el recurso identificado por la URI

PATH:

- Aplica modificaciones parciales a un recurso

CÓDIGOS DE ESTADO

- 1xx : Informacional, la solicitud fue recibida y se continua el proceso
- 2xx: Éxito, la acción fue recibida, entendida y aceptada con éxito
 - 200: OK
 - 201: CREATED
 - 204: NO CONTENT
- 3xx: Redirección
 - 301 MOVED PERMANENTLY
 - 302 FOUND
- 4xx: Errores del cliente
 - 400 BAD REQUEST
 - 401 UNAUTHORIZED
 - 404 FORBIDDEN
 - 404 NOT FOUND
 - 409 CONFLICT
 - 422 UNPROCESSABLE ENTITY
- 5xx: Errores de Servidor
 - 500 INTERNAL SERVER ERROR
 - 502 BAD GATEWAY
 - 503 SERVICE UNAVAILABLE

HEADERS, BODY, MÉTODOS SEGUROS VS. IDEMPOTENTES

Los headers son campos adicionales en el inicio de la solicitud o respuesta HTTP que proporcionan metadatos sobre la comunicación

El body es la parte principal de una solicitud o respuesta HTTP que contiene los datos reales que se están transfiriendo

Los métodos seguros, son solo así si, al ser invocados, no causan ningún cambio de estado en el servidor.

Los métodos son idempotente, si al invocarlo múltiples veces con los mismos parámetros, el efecto final en el servidor es el mismo que si se hubiera invocado una sola vez

B. ASP.NET CORE

Es un Framework de código abierto y multiplataforma para construir aplicaciones modernas basadas en la nube, como aplicaciones web, APIs REST y microservicios.

Middleware: Es un componente de ASP.NET CORE que se encarga de procesar las solicitudes HTTP y las respuestas. Puede realizar una acción específica y luego pasar la solicitud al siguiente middleware en la cadena o terminar la solicitud.

Pipeline de ejecución: Cuando la solicitud HTTP llega a una aplicación, atraviesa una serie de middlewares configurados secuencialmente.

TIPOS DE API

REST API vs. gRPC:

- **REST API (Representational State Transfer API):**

- **Protocolo:** Basada en HTTP/1.1 (predominantemente).
- **Formato de Datos:** Generalmente JSON, a veces XML.
- **Comunicación:** Request/Response síncrono.
- **Filosofía:** Centrada en recursos (URIs) y métodos HTTP (GET, POST, PUT, DELETE).
- **Ventajas:** Ubicua, fácil de entender, compatible con navegadores, muchos clientes disponibles, legible por humanos.
- **Desventajas:** Más verbosa (JSON), menos eficiente para comunicaciones de alto rendimiento o streaming, no tiene tipado fuerte de los contratos de servicio por defecto (depende de documentación como OpenAPI/Swagger).
- **Casos de Uso:** APIs públicas, integración con navegadores, aplicaciones móviles, sistemas donde la legibilidad y la simplicidad son prioritarias.

- **gRPC (gRPC Remote Procedure Call):**

- **Protocolo:** Basado en HTTP/2.
- **Formato de Datos:** Protocol Buffers (protobuf) serializado binario, que es mucho más compacto y eficiente que JSON/XML.
- **Comunicación:** Soporta Request/Response, Server Streaming, Client Streaming y Bidirectional Streaming.
- **Filosofía:** Centrada en servicios y funciones/métodos (RPC - Remote Procedure Call).
- **Ventajas:** Mucho más eficiente y rápido (HTTP/2, protobuf), soporta streaming, tipado fuerte de los contratos de servicio (.proto files), generación de código para múltiples lenguajes (Polyglot).
- **Desventajas:** Menos amigable para el desarrollo web directo (no se puede llamar desde el navegador sin un proxy), curva de aprendizaje ligeramente mayor, herramientas y ecosistema menos maduros que REST en algunos aspectos.
- **Casos de Uso:** Microservicios inter-servicio de alto rendimiento, comunicación interna en sistemas distribuidos, IoT, servicios en tiempo real, entornos polyglot.

Minimal API vs. API con Controlador:

- **API con Controlador (Controller-based API):**

- **Estructura:** Utiliza clases que heredan de ControllerBase (o Controller para APIs con vistas) y organizan las acciones de la API en métodos dentro de esos controladores.
- **Convención:** Sigue un patrón MVC para APIs. Cada controlador maneja un conjunto de recursos relacionados.
- **Características:**
 - Soporte completo para atributos de enrutamiento, validación, filtros de acción, filtros de autorización, etc.
 - Mayor estructura y separación de responsabilidades.
 - Ideal para APIs grandes, complejas o cuando se necesita una estructura más organizada.
- **Configuración:** Se habilita con builder.Services.AddControllers(); y app.MapControllers();.

- **Minimal API:**

- **Estructura:** Permite definir endpoints HTTP directamente en el archivo Program.cs (o en archivos separados con WebApplication.Map* métodos) utilizando expresiones lambda. No requiere clases de controlador.
- **Filosofía:** Diseñada para ser simple y concisa, ideal para microservicios pequeños, APIs con pocos endpoints o prototipos rápidos.
- **Características:**
 - Menos boilerplate (código repetitivo).
 - Enrutamiento directo y manejo de solicitudes en línea.
 - Aún soporta inyección de dependencias, filtros y validación, pero de una manera más concisa.
- **Configuración:** Habilitada por defecto en proyectos .NET 6+ WebApplication.CreateBuilder(). Los endpoints se mapean con app.MapGet(), app.MapPost(), etc.

¿Cuándo usar cuál?

Minimal APIs: Pequeñas APIs, microservicios simples, prototipos, aprender ASP.NET Core de forma rápida.

API con Controlador: APIs complejas, grandes, cuando se requiere mucha organización y aplicación de patrones de diseño, equipos grandes, cuando se mezclan APIs con vistas (MVC).

Los Health Checks en ASP.NET Core permiten que un servicio o herramienta externa (ej., un orquestador de contenedores como Kubernetes, o un balanceador de carga) determine el **estado de salud de tu aplicación**.

Se implementan creando clases que implementan la interfaz `IHealthCheck`. Cada clase contiene la lógica para verificar el estado de un componente específico (ej., conectividad a la base de datos, disponibilidad de un servicio externo, espacio en disco).

El método `CheckHealthAsync` de la interfaz `IHealthCheck` debe devolver un `HealthCheckResult` que indica `Healthy`, `Degraded` o `Unhealthy`, junto con un mensaje opcional y datos de diagnóstico.

Configuración: Se agregan al servicio de Health Checks en `Program.cs`:

```
builder.Services.AddHealthChecks().AddCheck<MyDatabaseHealthCheck>("MyDatabase");
```

/health o /healthz:

Una vez configurados los Health Checks, se exponen como endpoints HTTP.

Se mapean en el pipeline de middleware: `app.MapHealthChecks("/healthz")` o `app.MapHealthChecks("/health")`.

Cuando una solicitud llega a `/healthz`, ASP.NET Core ejecuta todos los Health Checks registrados y devuelve una respuesta HTTP basada en su estado combinado.

Respuestas comunes:

- Si todos los checks son `Healthy`.
- `503 Service Unavailable`: Si al menos un check es `Unhealthy`.

Variantes de endpoints:

- `MapHealthChecks("/healthz")`: Endpoint simple que devuelve solo el código de estado (útil para balanceadores de carga).
- `MapHealthChecks("/health")`: Puede configurarse para devolver un JSON más detallado con el estado de cada componente, útil para monitoreo. También se puede usar `MapHealthChecks("/health/ready")` y `MapHealthChecks("/health/live")` para distinguir entre "listo para recibir tráfico" y "vivo pero quizás no listo".

ENTITY FRAMEWORK (EF CORE)

Es un ORM, un mapeador objeto-relacional que permite interactuar con bases de datos relacionales usando clases C# y LINQ, abstracto directamente el SQL. Rastrea cambios y genera SQL al guardar.

MIGRACIONES

Es una característica para evolucionar el esquema de la base de datos de forma versionada. `Add-Migration` genera archivos de cambios; `Update-Database` los aplica.

DBCONTEXT

Es la clase central en EFCORE, actuando como puerta de entrada principal a la base de datos. La aplicación tendrá una clase que hereda de `DbContext` y dentro de esta clase definirías propiedades `DbSet<T>`.

Una propiedad **DbSet< TEntity >** representa una colección de entidades de un tipo específico en tu base de datos, mapeándose a una tabla.

FLUENT API

Es una API basada en código que se utiliza en el método OnModelCreating del DbContext. Ofrece una forma potente y flexible de configurar el modelo.

Se usa en configuraciones complejas que los atributos no pueden manejar (ej., claves compuestas, índices únicos, relaciones many-to-many sin entidad de unión explícita).

También es preferible cuando quieras **separar completamente tu modelo de dominio (tus clases de entidad) de la lógica de persistencia**, lo cual es crucial en arquitecturas limpias donde las entidades no deberían depender de librerías de infraestructura como EF Core.

DATA ANNOTATIONS

Son atributos de C# que se aplican directamente a las propiedades de las clases de entidad. Son buenas para configuraciones sencillas y comunes, ya que son fáciles de leer y están junto a la propiedad que modifican.

SWAGGER / OPENAPI

Swagger (o más precisamente, la **especificación OpenAPI** y sus herramientas como Swagger UI) es un conjunto de herramientas esenciales para el desarrollo de APIs RESTful.

Documentación Automática: Herramientas como Swashbuckle.AspNetCore en .NET Core inspeccionan tu código (controladores, acciones, modelos) y **generan automáticamente un archivo JSON** que describe tu API según la especificación OpenAPI. Esta documentación detalla tus endpoints, los métodos HTTP que soportan, los parámetros de entrada, las estructuras de datos esperadas y los posibles códigos de respuesta HTTP. Lo mejor es que se mantiene sincronizada con tu código, reduciendo la carga de documentación manual.

Herramienta para Probar Endpoints: La **Swagger UI** es una interfaz de usuario web interactiva generada a partir de esa documentación. Se expone como una página en tu aplicación (comúnmente en /swagger o /swagger/index.html). Permite:

- Visualizar todos los endpoints de tu API.
- Ver sus detalles (parámetros, tipos de retorno, códigos de estado).
- Lo más importante: **probar los endpoints directamente desde el navegador**. Puedes introducir los datos de la solicitud y ver la respuesta en tiempo real. Esto es invaluable para desarrolladores (tanto de backend como frontend) y para equipos de QA, acelerando el desarrollo y la depuración.