EDUCACIÓN

SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO

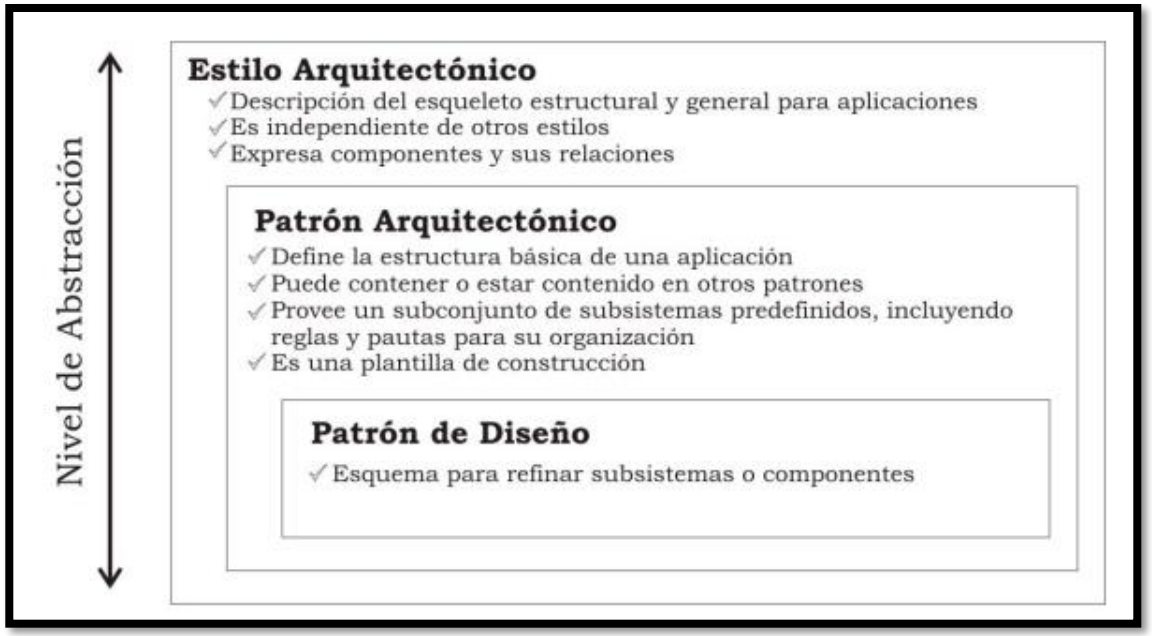
Especialidad Análisis, diseño y desarrollo de software

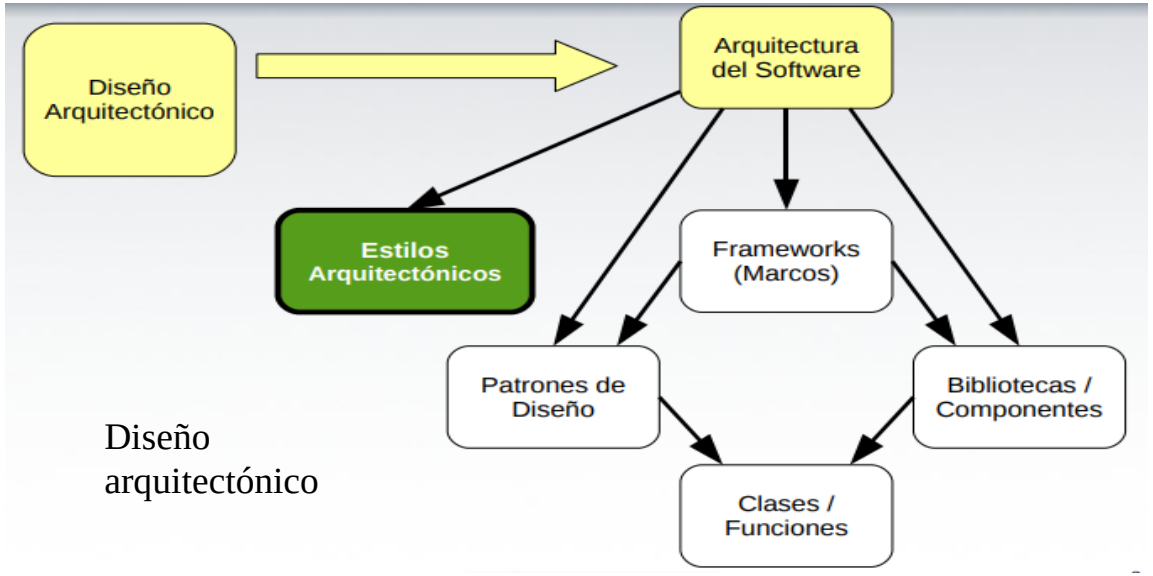
D3.2 Estilos, patrones y diseño arquitectónico de software

Asesor: M.T.I.C. Leonardo Enriquez

Ingeniero electrónico, sistemas digitales

Diseño arquitectónico





Diseño arquitectónico

Estilo	Descripción	Atributos asociados	Atributos en conflicto
Datos Centralizados	Sistemas en los cuales cierto número de clientes accede y actualiza datos compartidos de un repositorio de manera frecuente.	Integrabilidad Escalabilidad Modificabilidad	Desempeño
Flujo de Datos	El sistema es visto como una serie de transformaciones sobre piezas sucesivas de datos de entrada. El dato ingresa en el sistema, y fluye entre los componentes, de uno en uno, hasta que se le asigne un destino final (salida o repositorio).	Reusabilidad Modificabilidad Mantenibilidad	Desempeño
Máquinas Virtuales	Simulan alguna funcionalidad que no es nativa al hardware o software sobre el que está implementado.	Portabilidad	Desempeño
Llamada y Retorno	El sistema se constituye de un programa principal que tiene el control del sistema y varios subprogramas que se comunican con éste mediante el uso de llamadas.	Modificabilidad Escalabilidad Desempeño	Mantenibilidad Desempeño
Componentes Independientes	Consiste en un número de procesos u objetos independientes que se comunican a través de mensajes.	Modificabilidad Escalabilidad	Desempeño Integrabilidad

Patrón Arquitectónico	Descripción	Atributos asociados	Atributos en conflicto
Layers	Consiste en estructurar aplicaciones que pueden ser descompuestas en grupos de subtareas, las cuales se clasifican de acuerdo a un nivel particular de abstracción.	Reusabilidad Portabilidad Facilidad de Prueba	Desempeño Mantenibilidad
Pipes and Filters	Provee una estructura para los sistemas que procesan un flujo de datos. Cada paso de procesamiento está encapsulado en un componente filtro (*filter*). El dato pasa a través de conexiones (*pipes*), entre filtros adyacentes.	Reusabilidad Mantenibilidad	Desempeño
Blackboard	Aplica para problemas cuya solución utiliza estrategias no determinísticas. Varios subsistemas ensamblan su conocimiento para construir una posible solución parcial ó aproximada.	Modificabilidad Mantenibilidad Reusabilidad Integridad	Desempeño Facilidad de Prueba
Broker	Puede ser usado para estructurar sistemas de software distribuido con componentes desacoplados que interactúan por invocaciones a servicios remotos. Un componente *broker* es responsable de coordinar la comunicación, como el reenvío de solicitudes, así como también la transmisión de resultados y excepciones.	Modificabilidad Portabilidad Reusabilidad Escalabilidad Interoperabilidad	Desempeño
Model-View-Controler	Divide una aplicación interactiva en tres componentes. El modelo *(model)* contiene la información central y los datos. Las vistas *(view)* despliegan información al usuario. Los controladores *(controlers)* capturan la entrada del usuario. Las vistas y los controladores constituyen la interfaz del usuario.	Funcionalidad Mantenibilidad	Desempeño Portabilidad

Patrón de Arquitectura

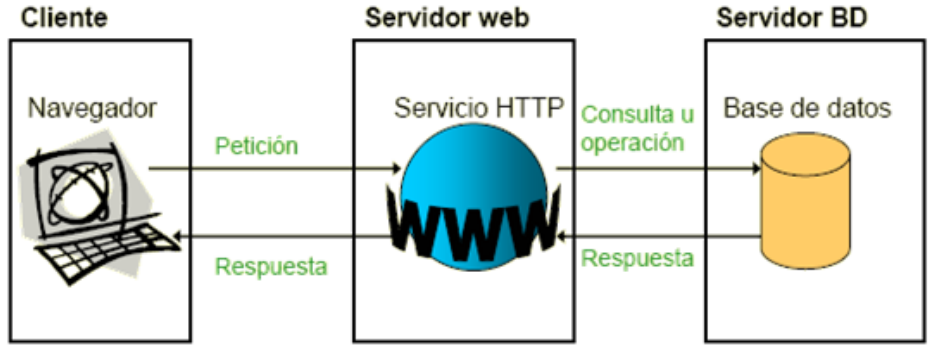
Cliente-Servidor

Patrones de Arquitectura Cliente-Servidor

Componentes que conforman la arquitectura

Los principales componentes de este modelo son:

1. Conjunto de **servidores** que ofrecen servicios a otros componentes (servidor de archivos, servidor compilador, servidor de impresión)
2. Conjunto de **clientes** que solicitan los servicios que ofrecen los servidores.
3. Una **red** que permite a los clientes acceder a dichos servicios.

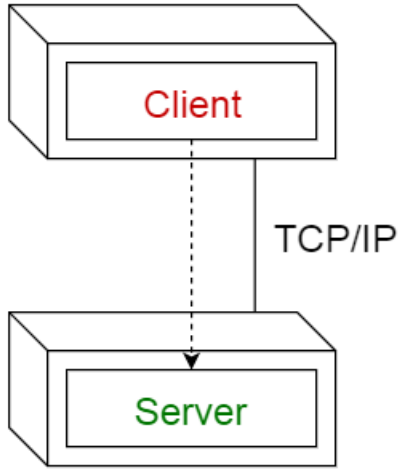


- **Red**: Una red es un conjunto de clientes, servidores y base de datos unidos de una manera física o no física en el que existen protocolos de transmisión de información establecidos.
- **Cliente**: El concepto de cliente hace referencia a un demandante de servicios, este cliente puede ser un ordenador como también una aplicación de informática, la cual requiere información proveniente de la red para funcionar.
- **Servidor**: Un servidor hace referencia a un proveedor de servicios, este servidor a su vez puede ser un ordenador o una aplicación informática la cual envía información a los demás agentes de la red.
- **Protocolo**: Un protocolo es un conjunto de normas o reglas y pasos establecidos de manera clara y concreta sobre el flujo de información en una red estructurada.
- **Servicios**: Un servicio es un conjunto de información que busca responder las necesidades de un cliente, donde esta información pueden ser mail, música, mensajes simples entre software, videos, etc.
- **Base de datos**: Son bancos de información ordenada, categorizada y clasificada que forman parte de la red, que son sitios de almacenaje para la utilización de los servidores y también directamente de los clientes.

Arquitectura Cliente-Servidor

Características de la Arquitectura

- Este patrón consiste en dos partes; un **servidor** y múltiples **clientes** . El componente del servidor proporcionará servicios a múltiples componentes del cliente. Los clientes solicitan servicios del servidor y el servidor proporciona servicios relevantes a esos clientes. Además, el servidor sigue escuchando las solicitudes de los clientes.



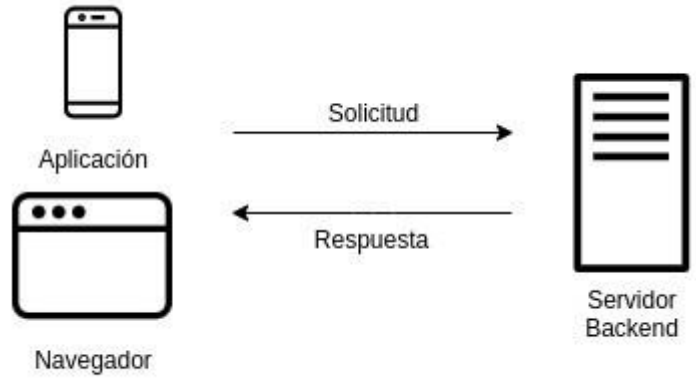
Patrón cliente-servidor

- En una arquitectura cliente-servidor, la funcionalidad del sistema se organiza en servicios y cada servicio lo entrega un servidor independiente. Los clientes son usuarios de dichos servicios y para utilizarlos ingresan a los servidores.
- Esta arquitectura consiste en un cliente que realiza peticiones a otro programa (el servidor) que le da respuesta.
- En esta arquitectura la computadora de cada uno de los usuarios, llamada cliente, produce una demanda de información a cualquiera de las computadoras que proporcionan información, conocidas como servidores estos últimos responden a la demanda del cliente que la produjo.
- Los clientes y los servidores pueden estar conectados a una red local o una red amplia, como la que se puede implementar en una empresa o a una red mundial como lo es la Internet.
- Bajo este modelo cada usuario tiene la libertad de obtener la información que requiera en un momento dado proveniente de una o varias fuentes locales o distantes y de procesarla como según le convenga. Los distintos servidores también pueden intercambiar información dentro de esta arquitectura.

Patrones de Arquitectura Cliente-Servidor

Características de la Arquitectura

- En una arquitectura cliente-servidor, la funcionalidad del sistema se organiza en servicios y cada servicio lo entrega un servidor independiente. Los clientes son usuarios de dichos servicios y para utilizarlos ingresan a los servidores.
- Esta arquitectura consiste en un cliente que realiza peticiones a otro programa (el servidor) que le da respuesta.
- En esta arquitectura la computadora de cada uno de los usuarios, llamada cliente, produce una demanda de información a cualquiera de las computadoras que proporcionan información, conocidas como servidores estos últimos responden a la demanda del cliente que la produjo.
- Los clientes y los servidores pueden estar conectados a una red local o una red amplia, como la que se puede implementar en una empresa o a una red mundial como lo es la Internet.
- Bajo este modelo cada usuario tiene la libertad de obtener la información que requiera en un momento dado proveniente de una o varias fuentes locales o distantes y de procesarla como según le convenga. Los distintos servidores también pueden intercambiar información dentro de esta arquitectura.



Arquitectura Cliente - Servidor

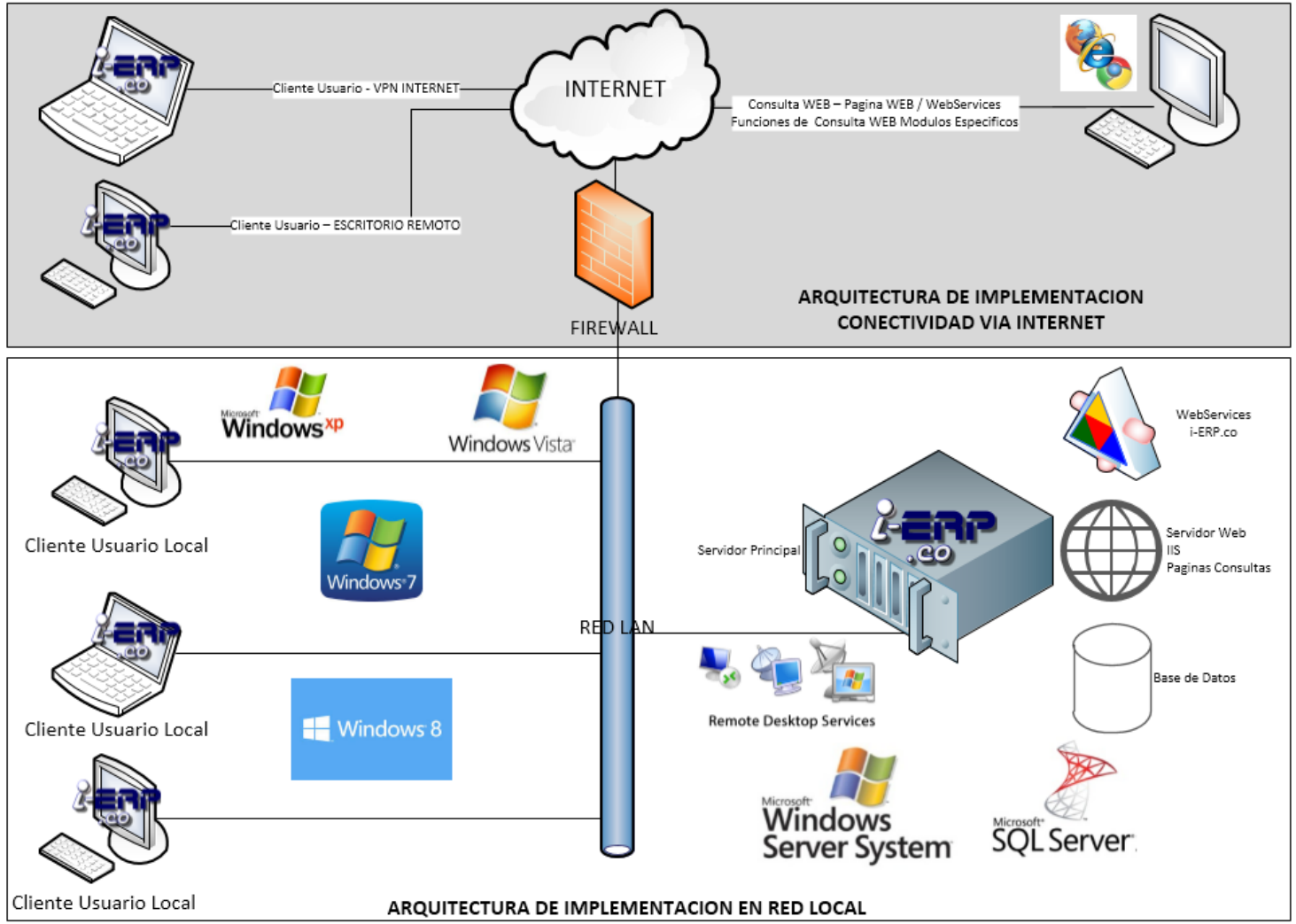
Arquitectura Cliente-Servidor

Casos de uso de la arquitectura (ejemplo 1)

- **Cliente:** Programa ejecutable que participa activamente en el establecimiento de las conexiones. Envía una petición al servidor y se queda esperando por una respuesta. Su tiempo de vida es finito una vez que son servidas sus solicitudes, termina el trabajo.
- **Servidor:** Es un programa que ofrece un servicio que se puede obtener en una red. Acepta la petición desde la red, realiza el servicio y devuelve el resultado al solicitante. Al ser posible implantarlo como aplicaciones de programas, puede ejecutarse en cualquier sistema donde exista TCP/IP y junto con otros programas de aplicación. El servidor comienza su ejecución antes de comenzar la interacción con el cliente. Su tiempo de vida o de interacción es "interminable".

Arquitectura Cliente-Servidor

Casos de uso de la arquitectura (ejemplo 2)



- La aplicación esta desarrollada con la herramienta y estándares de desarrollo de Microsoft Visual Studio .Net, y maneja una **Arquitectura Cliente / Servidor**
- El software a nivel de cliente es una aplicación Windows Forms que se ejecuta en sistemas operativos cliente Microsoft Windows. (Windows 7, Windows 8, Windows 10)
- El software a nivel de servidor esta conformado principalmente por el motor de base de datos Microsoft SQL Server como único repositorio de almacenamiento, y brinda módulos específicos de consulta e interacción por ambiente web, mediante paginas ASP.NET y/o Webservices .NET que se ejecutan en servidores con IIS - Internet Information Server en ambientes Microsoft Windows Server 2008 y 2012.

Arquitectura Cliente-Servidor

Ventajas y desventajas de la arquitectura

- Existencia de plataformas de hardware cada vez más baratas. Esta constituye a su vez una de las más palpables ventajas de este esquema, la posibilidad de utilizar máquinas mucho más baratas que las requeridas por una solución centralizada, basada en sistemas grandes (mainframes). Además, se pueden utilizar componentes, tanto de hardware como de software, de varios fabricantes, lo cual contribuye considerablemente a la reducción de costos y favorece la flexibilidad en la implantación y actualización de soluciones.
- Facilita la integración entre sistemas diferentes y comparte información, permitiendo por ejemplo que las máquinas ya existentes puedan ser utilizadas pero utilizando interfaces más amigables el usuario. De esta manera, se puede integrar PCs con sistemas medianos y grandes, sin necesidad de que todos tengan que utilizar el mismo sistema operativo.
- Al favorecer el uso de interfaces gráficas interactivas, los sistemas construidos bajo este esquema tienen una mayor y más intuitiva con el usuario. En el uso de interfaces gráficas para el usuario, presenta la ventaja, con respecto a uno centralizado, de que no siempre es necesario transmitir información gráfica por la red pues esta puede residir en el cliente, lo cual permite aprovechar mejor el ancho de banda de la red.
- La estructura inherentemente modular facilita además la integración de nuevas tecnologías y el crecimiento de la infraestructura computacional, favoreciendo así la escalabilidad de las soluciones.
- Contribuye además a proporcionar a los diferentes departamentos de una organización, soluciones locales, pero permitiendo la integración de la información.

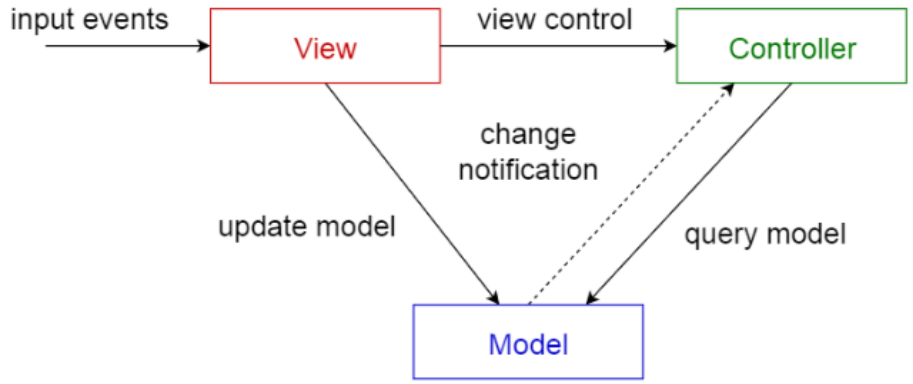
Patrón de Arquitectura

Modelo Vista Controlador

Componentes que conforman la arquitectura

Este patrón, también conocido como patrón MVC, divide una aplicación interactiva en 3 partes, como **modelo** — contiene la funcionalidad y los datos básicos, **vista** : muestra la información al usuario (se puede definir más de una vista), **controlador** : maneja la entrada del usuario

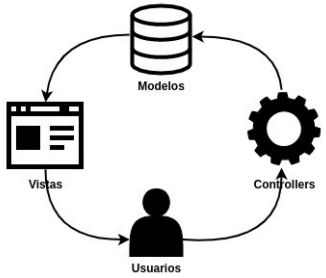
Esto se hace para separar las representaciones internas de información de las formas en que se presenta y acepta la información del usuario. Desacopla los componentes y permite la reutilización eficiente del código.



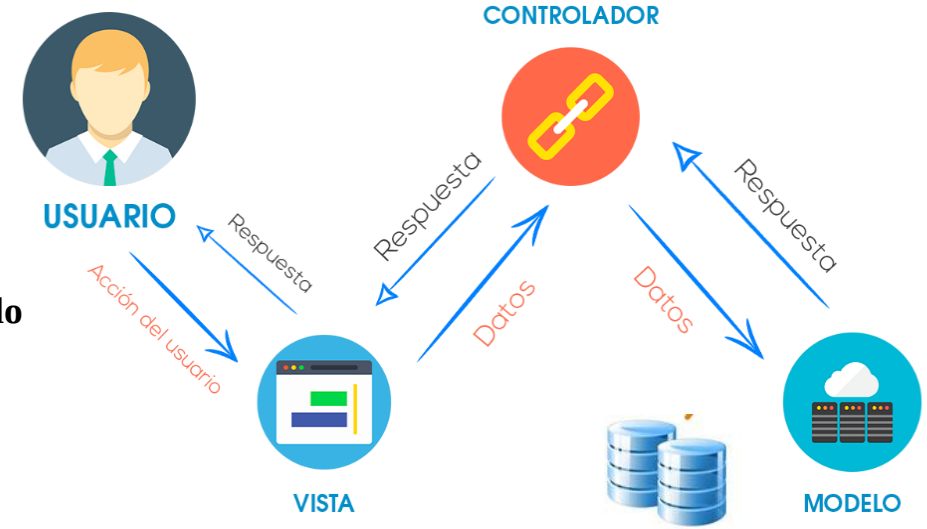
Modelo-vista-controlador

El sistema se estructura en tres **componentes lógicos** que interactúan entre sí.

1. El componente **Modelo** maneja los datos del sistema y las operaciones asociadas a esos datos.
2. El componente **Vista** define y gestiona cómo se presentan los datos al usuario.
3. El componente **Controlador** dirige la interacción del usuario (por ejemplo, teclas oprimidas, clics del mouse, etcétera) y pasa estas interacciones a Vista y Modelo.

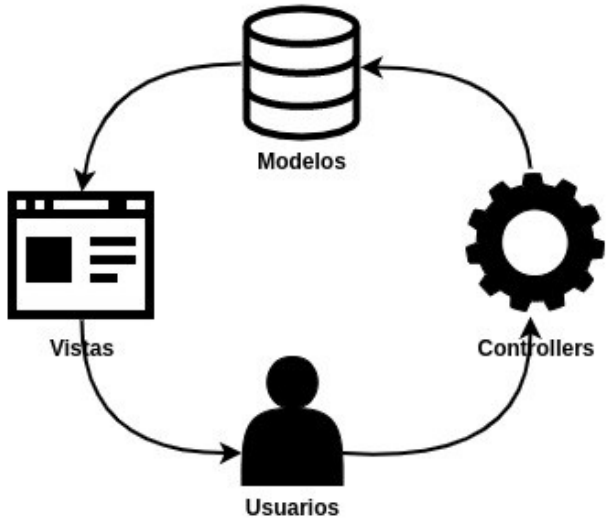


Organización del Modelo Vista Controlador

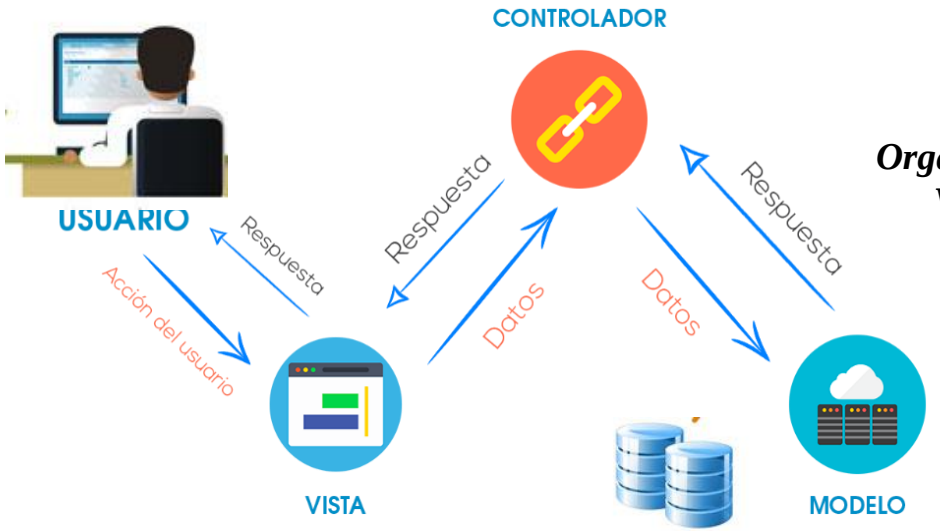


Componentes que conforman la arquitectura

- El MVC es un patrón de arquitectura de software que separa la **lógica de negocios y la capa de representación** entre si, que es tradicionalmente utilizada para interfaces gracias de usuario. Hoy en día, la arquitectura MVC se ha hecho popular para el diseño de aplicaciones Web y aplicaciones Mobile.
- El sistema se estructura en tres **componentes lógicos** que interactúan entre sí.
  1. El componente **Modelo** maneja los datos del sistema y las operaciones asociadas a esos datos.
  2. El componente **Vista** define y gestiona cómo se presentan los datos al usuario.
  3. El componente **Controlador** dirige la interacción del usuario (por ejemplo, teclas oprimidas, clics del mouse, etcétera) y pasa estas interacciones a Vista y Modelo.



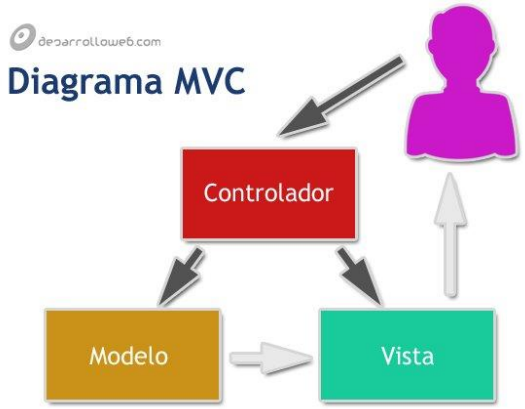
Flujo de datos del patrón Modelo Vista Controlador



Organización del Modelo Vista Controlador

Características de la Arquitectura

Etapa Modelos



- Es la capa donde se trabaja con los datos, por lo 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, update, insert, etc.
- Cuando se trabaja con MVC habitualmente se utilizan otras librerias como PDO o ORM como Doctrine, que permite trabajar con abstracción de base de datos y persistencia en objetos, que suele depender del motor de base de datos utilizando un dialecto de acceso a los datos basado en clases y objetos.

Etapa Vistas

- Las vista, contiene el código de nuestra aplicación que va a producir la visualización de las interfaces de usuario, ósea, el código que nos permitirá renderizar los estado de nuestra aplicación. En las vistas solo tenemos los código que nos permite mostrar la salida.
- En la vista generalmente se trabaja con los datos, sin embargo no se realiza un acceso directo a esto. Las vistas requerirán los datos a los modelos y ellas se generara la salida, tal como la aplicación lo requiera.

Etapa 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.
- El controlador es una capa que sirve de enlace entre las vista y los modelos, respondiente a los mecanismos que puedan requerirse para implementar las necesidades de nuestra aplicación. Su responsabilidad es servir de enlace entre los modelos y las vistas para implementar diversas necesidades del desarrollo.

Casos de uso de la arquitectura (ejemplo 1)

Framework soportados

MVC es un patrón de diseño enfocado a separar las responsabilidades dentro de nuestra aplicación y es muy utilizado en la web por su enfoque y las ventajas que ofrece con respecto a algunas otras formas o patrones de desarrollo de aplicaciones web. Puedes encontrar frameworks prácticamente para cualquier lenguaje web, por ejemplo: ASP.NET MVC (C# , VBasic), Laravel (PHP), django (Python), Ruby on Rails.

Lenguaje	Licencia	Nombre
.NET	Castle Project	MonoRail
.NET	Apache	Spring.NET
.NET	Apache	Maverick.NET
.NET	MS-PL	ASP.NET MVC
.NET	Microsoft Patterns & Practices	User Interface Process (UIP) Application Block

Frameworks MVC .NET

Lenguaje	Licencia	Nombre
JavaScript	GPLv3	Sails.JS
JavaScript	GPLv3	ExtJS 4
JavaScript	MIT	AngularJS
JavaScript	MIT	Nest

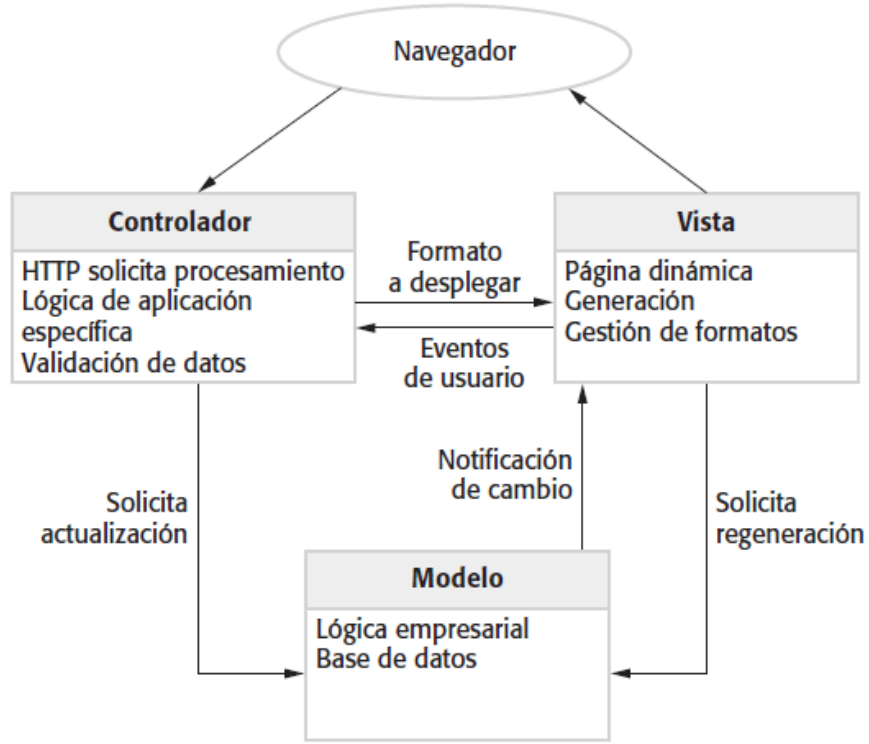
Frameworks MVC JavaScript

Casos de uso de la arquitectura (ejemplo 2)

En la imagen se representa 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 (conjunto de reglas que se siguen en el software para reaccionar ante distintas situaciones), hace de puente entre los modelos y las vistas. En algunos casos los modelos pueden enviar datos a las vistas.

Flujo de trabajo característico en un esquema MVC:

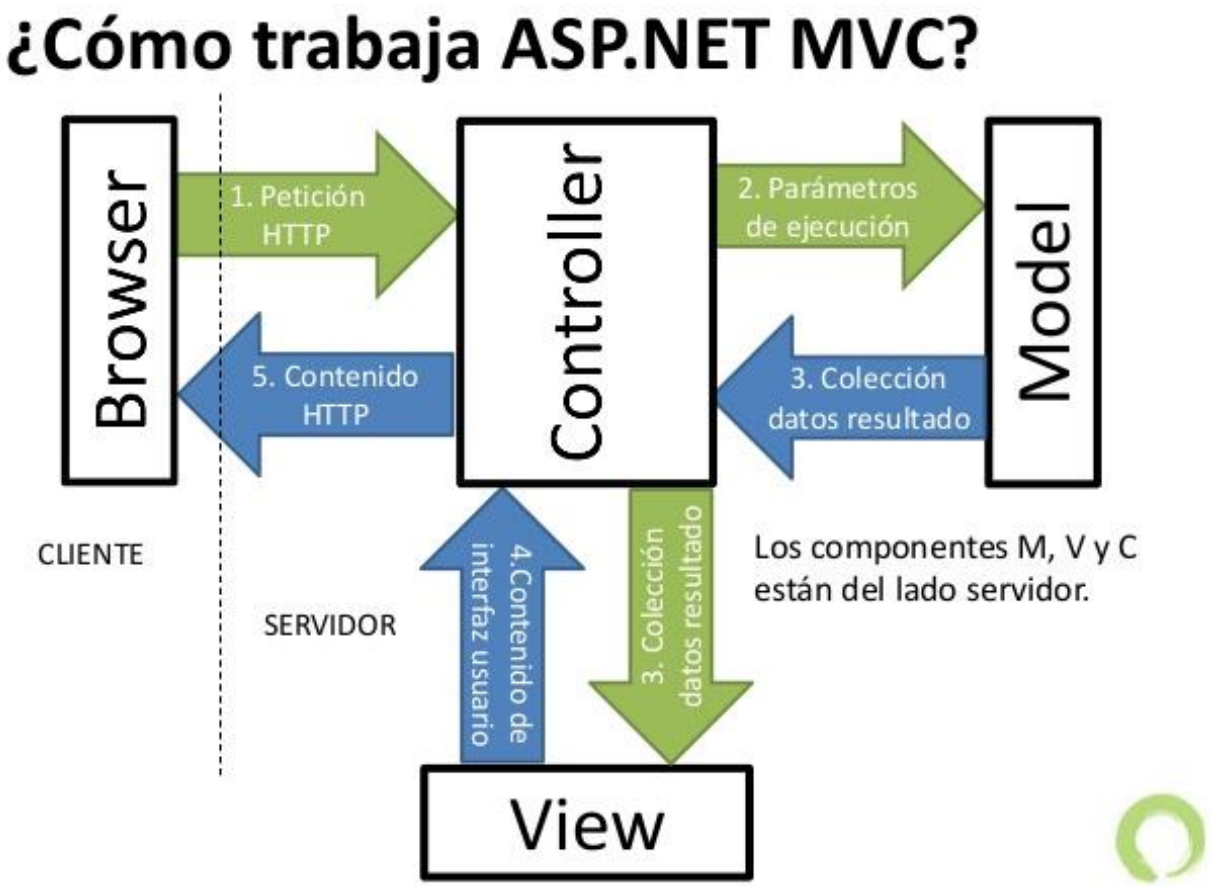
1. El **usuario** realiza una solicitad al sitio web, llegando la solicitud al controlador.
2. El **controlador** comunica tanto con modelos como con vistas. A los modelos les solicita datos o les manda realizar actualizaciones de los datos. A las vistas les solicita la salida correspondiente, una vez que se hayan realizado las operaciones pertinentes según la lógica del negocio.
3. Par producir la salida, en ocasiones las **vistas** pueden solicitar mas 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 puente entre unos y otros.
4. Las **vista** envían al usuario la salida, aunque en ocasiones esa salida puede ir de vuelta al controlador y seria este el que hace el envío al cliente.



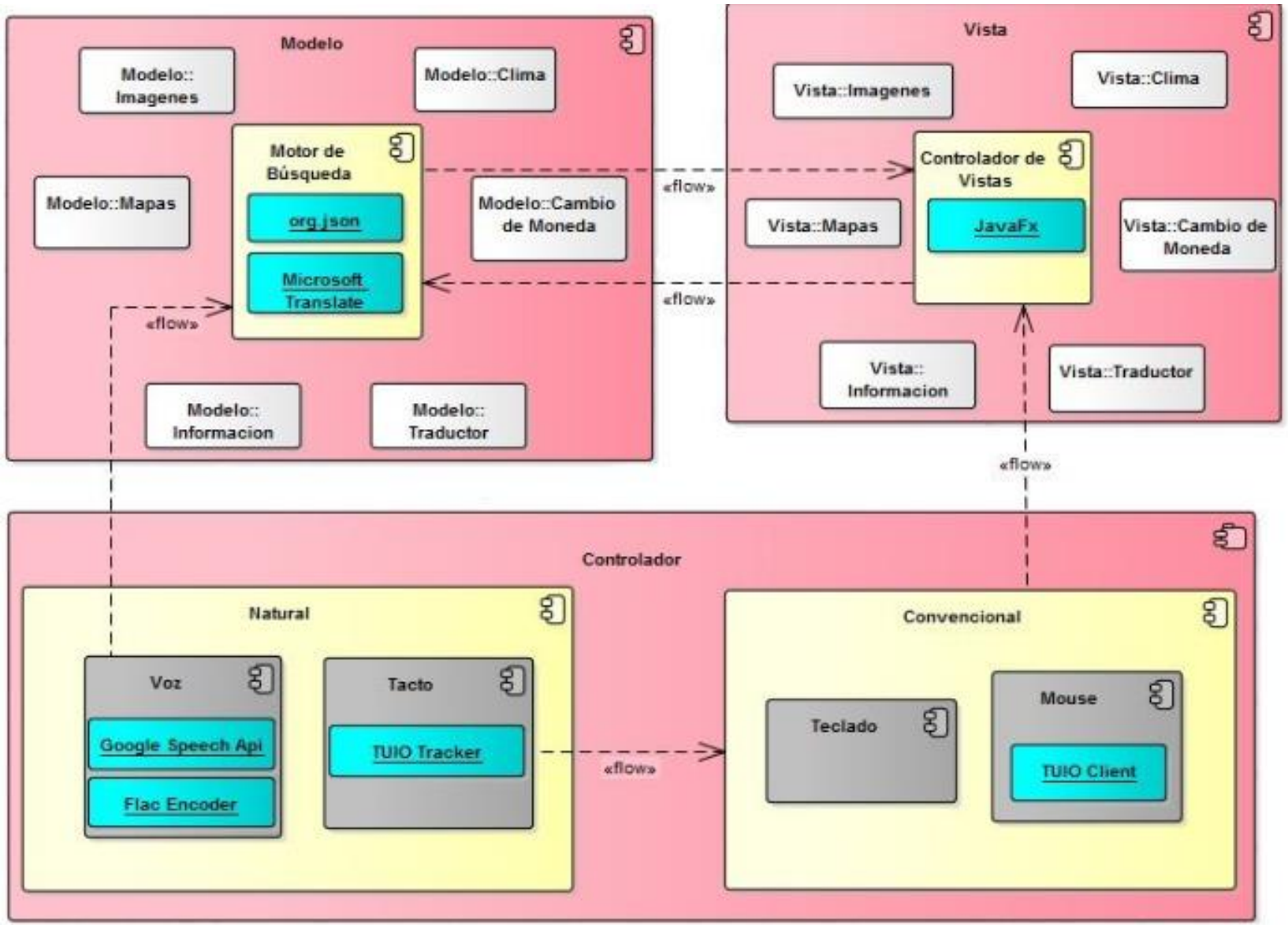
Ejemplo de la arquitectura de un sistema de aplicación basado en la web, que se organiza con el uso del patron MVC.

Casos de uso de la arquitectura (ejemplo 3)

Arquitectura de aplicación Web con el patrón MVC



Casos de uso de la arquitectura (ejemplo 3)



Ventajas y desventajas de la arquitectura

- Los proyectos se podrán dividir la lógica de negocio del diseño, haciéndolo más **escalable**.
- **Facilitará el uso** de URL amigables, importantes para el SEO (Posicionamiento web), la mayoría de frameworks MVC lo controlan.
- Muchos frameworks MVC, incluyen librerías de Javascript como Jquery, facilitarndo validar formularios (Ej. Jquery.Validate) en el cliente y en el servidor.
- Se puede utilizar abstracción de datos, como lo hace Ruby on Rails o con frameworks como **Hibernate para Java o NHibernate para ASP .NET MVC**, facilitando la realización de consultas a la base de datos.
- La mayoría de frameworks controlan el uso de la memoria Caché, hoy en día muy importante para el posicionamiento web, ya que buscadores como google dan prioridad a las webs que tengan menor tiempo de descarga.
- En el caso de proyectos donde hay varios desarrolladores, el seguir métodos comunes de programación, hace que el código sea más entendible entre estos, pudiendo uno continuar el trabajo de otro. En estos casos es conveniente utilizar herramientas de control de versiones como **Subversion**.
- Los frameworks están creados para facilitar el trabajo de los desarrolladores, se encontrarán clases para controlar fechas, URL's, Webservices, etc. lo que tiene una gran ventaja en cuanto a productividad. Inicialmente como es lógico habrá una curva de aprendizaje, pero luego tendrás muchos beneficios.
- Poco a poco el desarrollo web se orienta a lo que se denomina **"Agile Web Development"**, con frameworks como **Ruby on Rails** que ayudan a crear proyectos de calidad y en corto tiempo. Existen varios frameworks en PHP e incluso ASP .NET que en su nueva versión ya contempla el MVC con Visual C#.
- Un Framework MVC te ayuda a controlar los recursos del servidor, evitando **Bugs** que puedan repercutir en el rendimiento, por ejemplo, muchas veces olvidamos cerrar conexiones a la base de datos, sobrecargando el servidor.

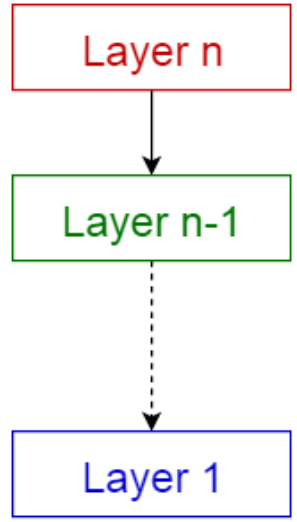
Patrón de Arquitectura en Capas

Componentes que conforman la arquitectura

Este patrón se puede utilizar para estructurar programas que se pueden descomponer en grupos de subtareas, cada una de las cuales se encuentra en un nivel particular de abstracción. Cada capa proporciona servicios a la siguiente capa superior.

Las 4 capas más comúnmente encontradas de un sistema de información general son las siguientes.

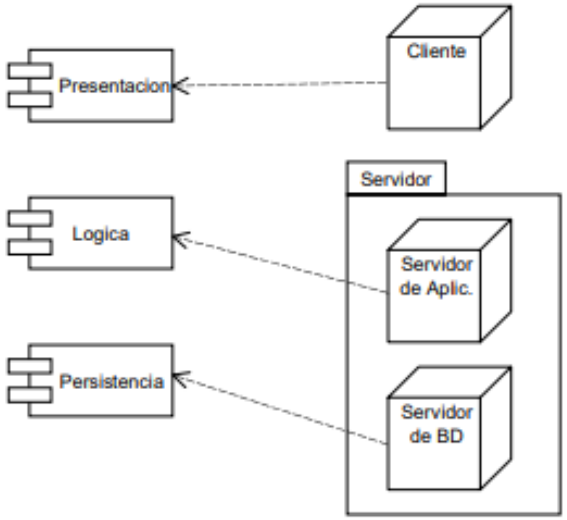
1. **Capa de presentación** (también conocida como **capa UI**)
2. **Capa de aplicación** (también conocida como **capa de servicio**)
3. **Capa de lógica de negocios** (también conocida como **capa de dominio**)
4. **Capa de acceso a datos** (también conocida como **capa de persistencia**)

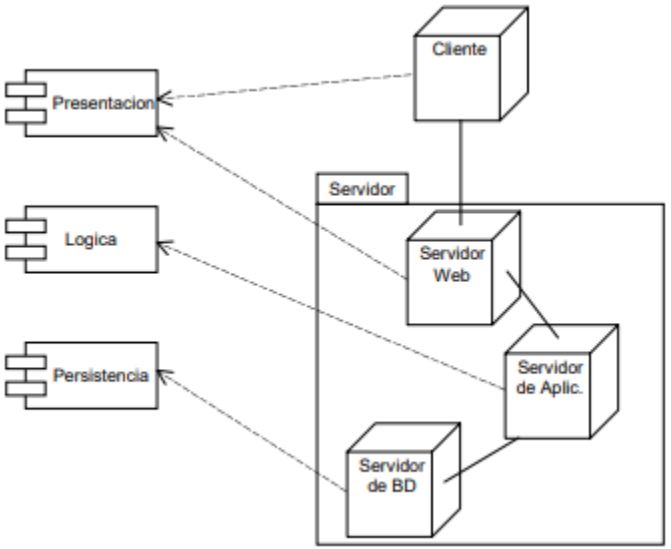


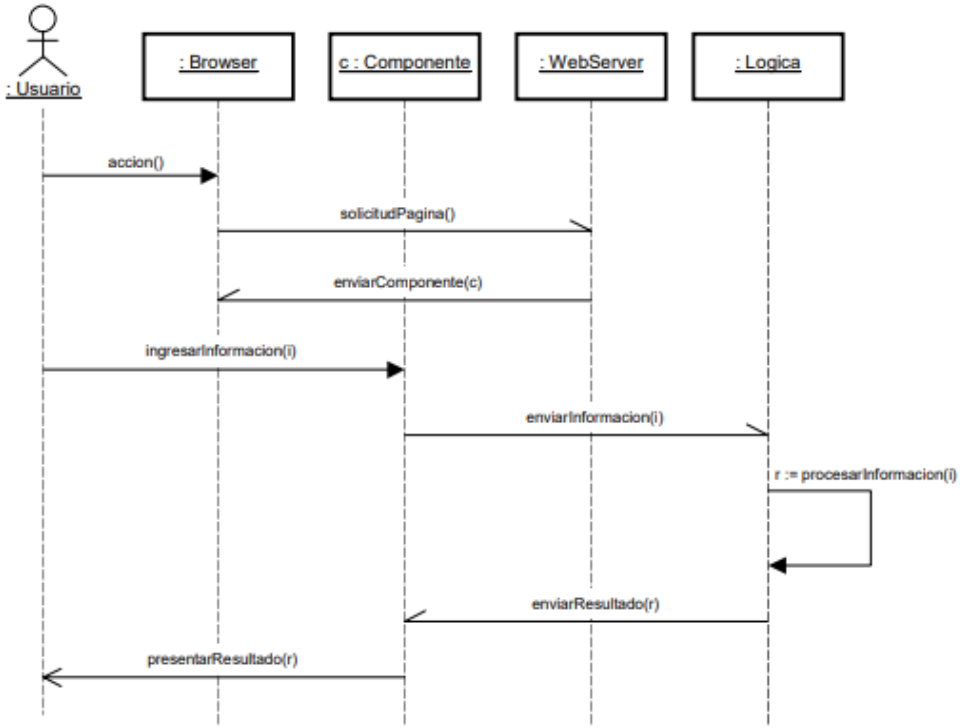
Patrón de capas

- **Capa de presentación** o interfaz de usuario. Esta capa, esta formada por los formularios y los controles. Es la capa con la que interactúa el usuario.
- **Capa de Negocio**. Esta formada por las entidades, que representan objetos que van a ser manejados o utilizados por toda la aplicación. En este caso, están representados por clases y Data Tables que se crean.
- **Capa de acceso a datos**. Contiene clases que interactúan con la base de datos, estas clases altamente especializadas se encuentran en la arquitectura del sistema y permiten, utilizando procedimientos almacenados se realizan las operaciones con la base de datos de forma transparente para la capa de negocio.

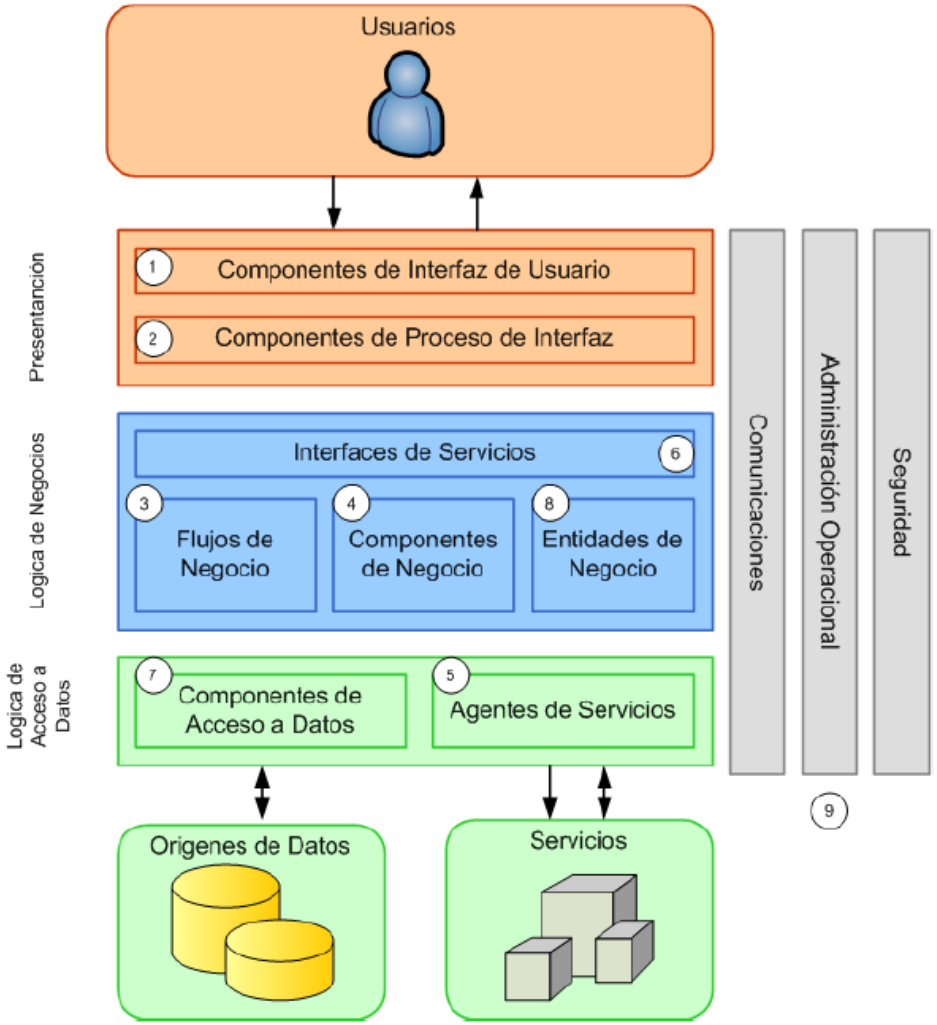
Diagramas que pueden ilustrar el modelado de una arquitectura en capas utilizando UML





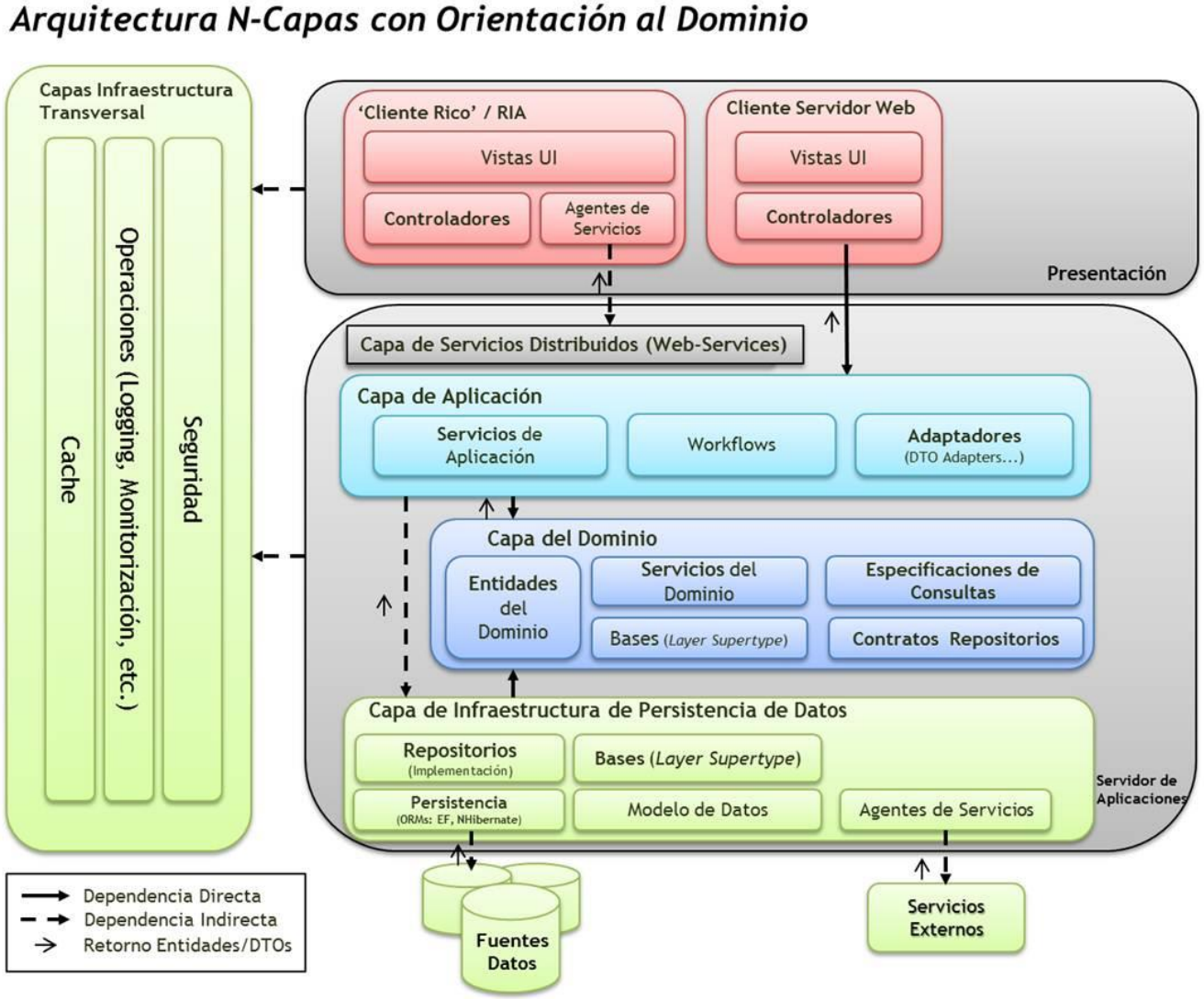


Casos de uso de la arquitectura (ejemplo 1)



Flujo de datos en la arquitectura de 3 capas

Casos de uso de la arquitectura (ejemplo 2)



Casos de uso de la arquitectura (ejemplo 3)

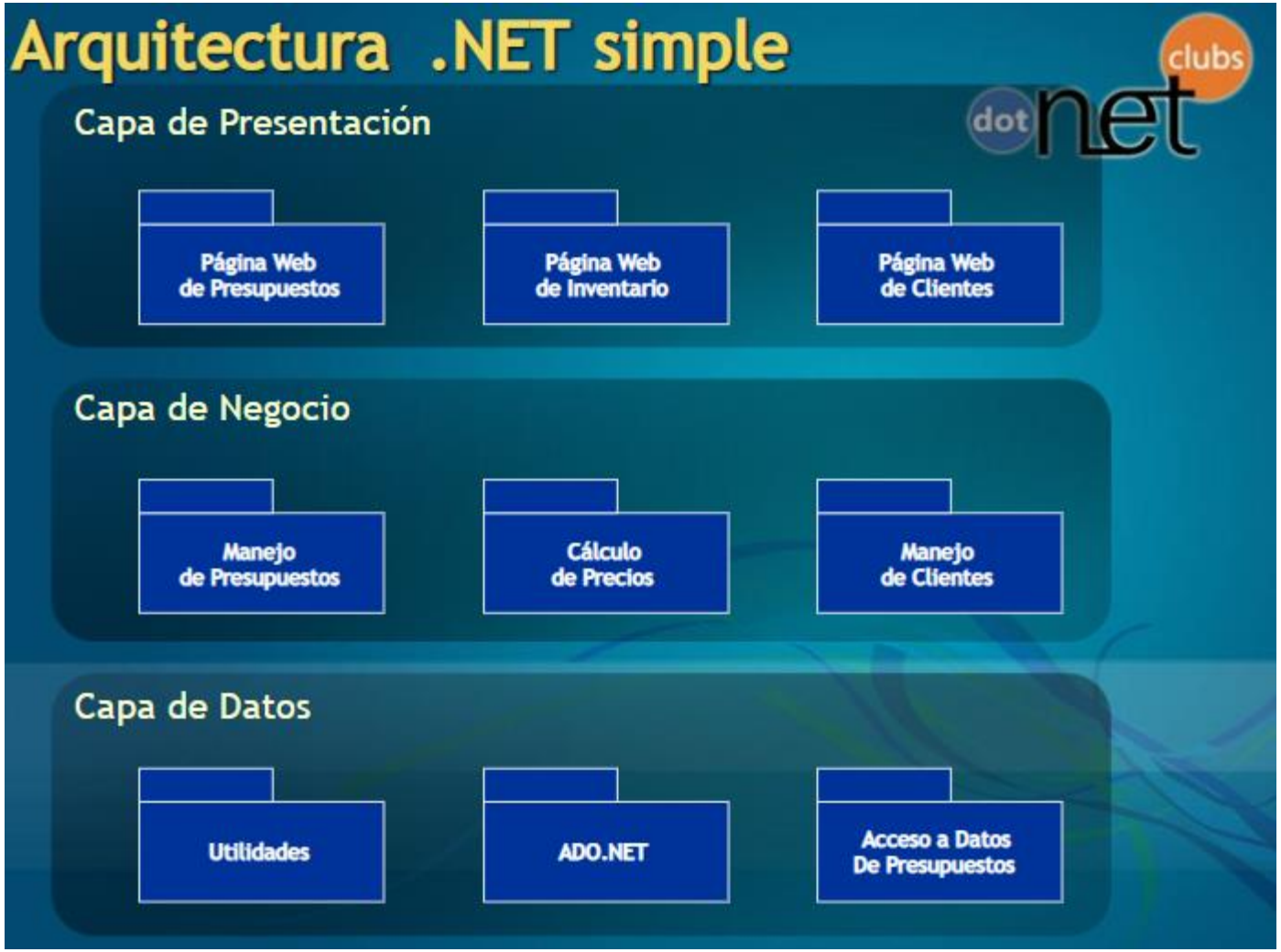
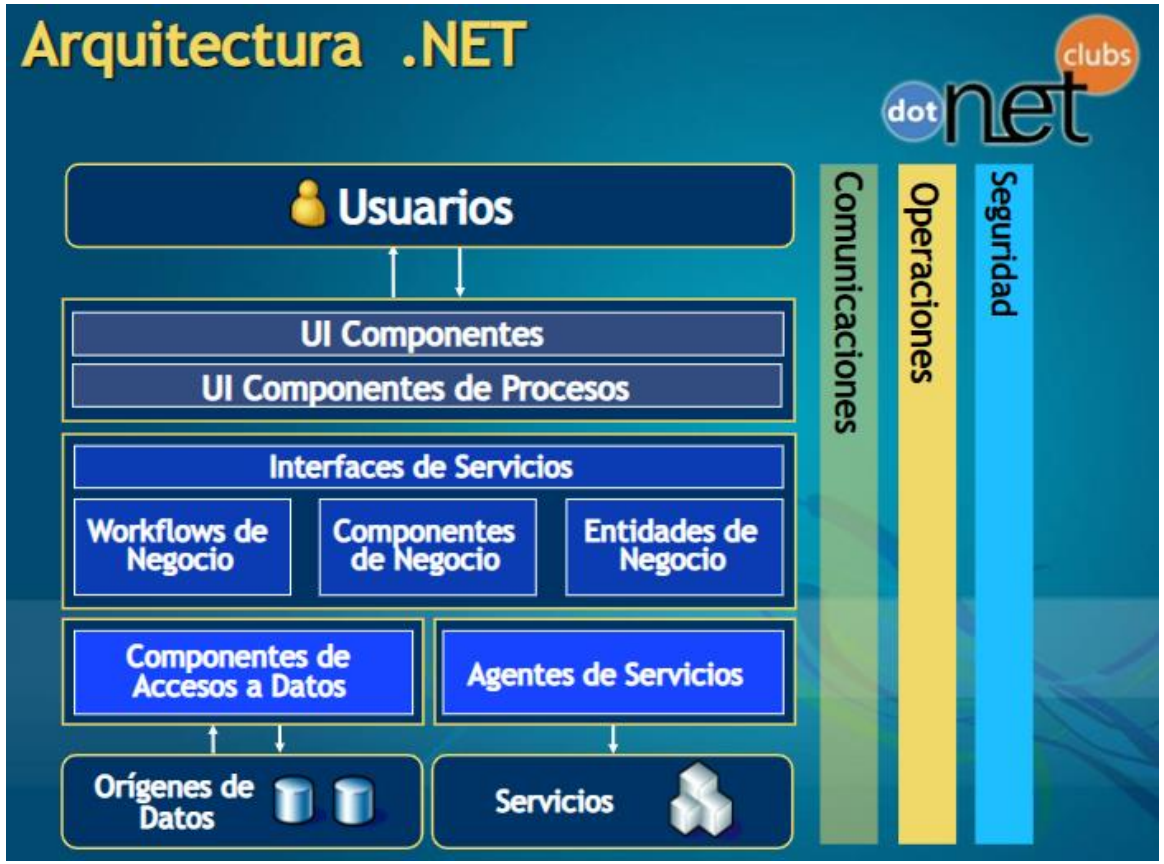


Ilustración que muestra el modelado de una arquitectura en capas

Casos de uso de la arquitectura (ejemplo 3)



Arquitectura en capas con .net

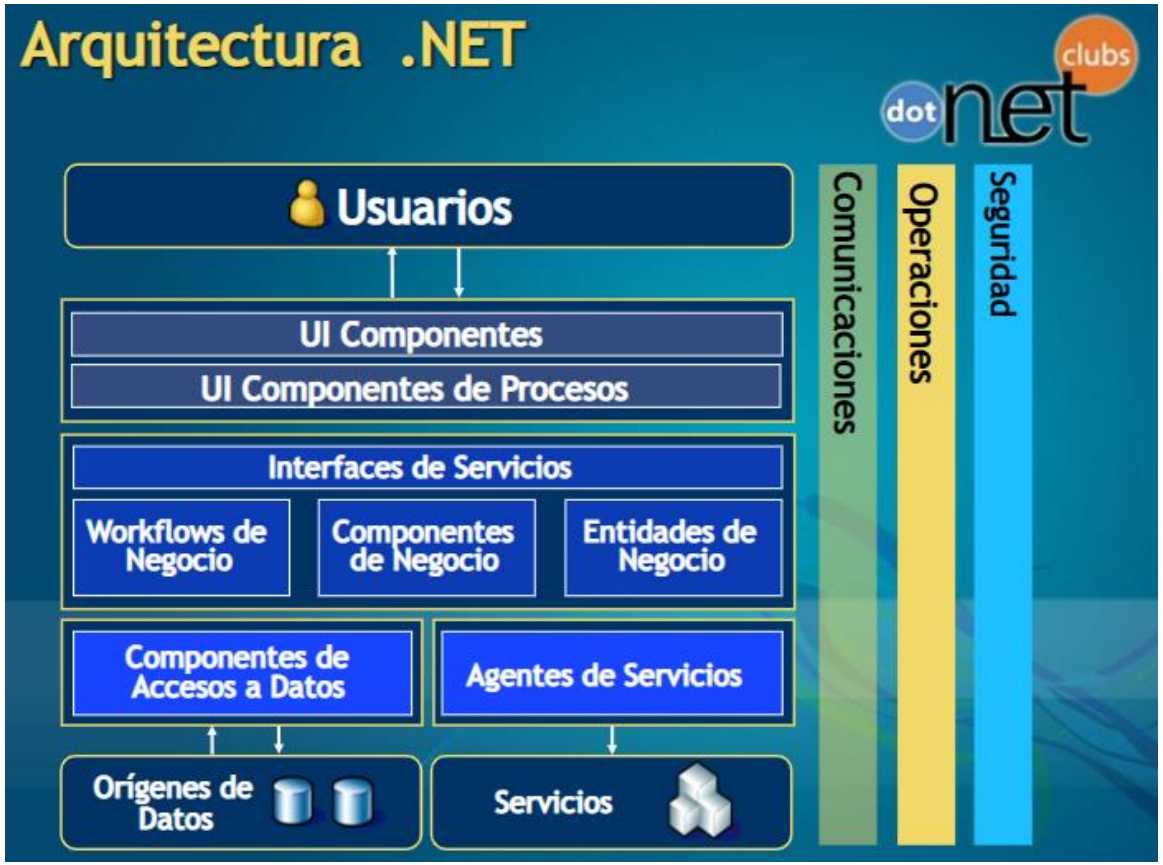
Capa de datos

- Objetos sin comportamiento que solo guardan un Business Entity en la base de datos
- Son clases con métodos estáticos
- Utilizan alguna forma de acceso a datos simplificado como Data Access Application Block
- Deben ser llamados por el Business component de forma que no se tenga en cuenta el origen de los datos
- Realiza todas las conversiones y validaciones necesarias que estén relacionadas con el modelo de base de datos.

Implementación

- Código ADO.NET para cada método de cada objeto: Create, Open, Update, Delete, Find.
- Escribiendo codigo con DAAB Data Access Application Block (Version 2.0 MSDN/Patterns y Practices), Version 3.1 GotDotNet)
- Usando DataAdapter, si se usan DataSets como Business Entities
- Alguna herramienta de Object Relactional Mapping

Casos de uso de la arquitectura (ejemplo 3)



Arquitectura en capas con .net

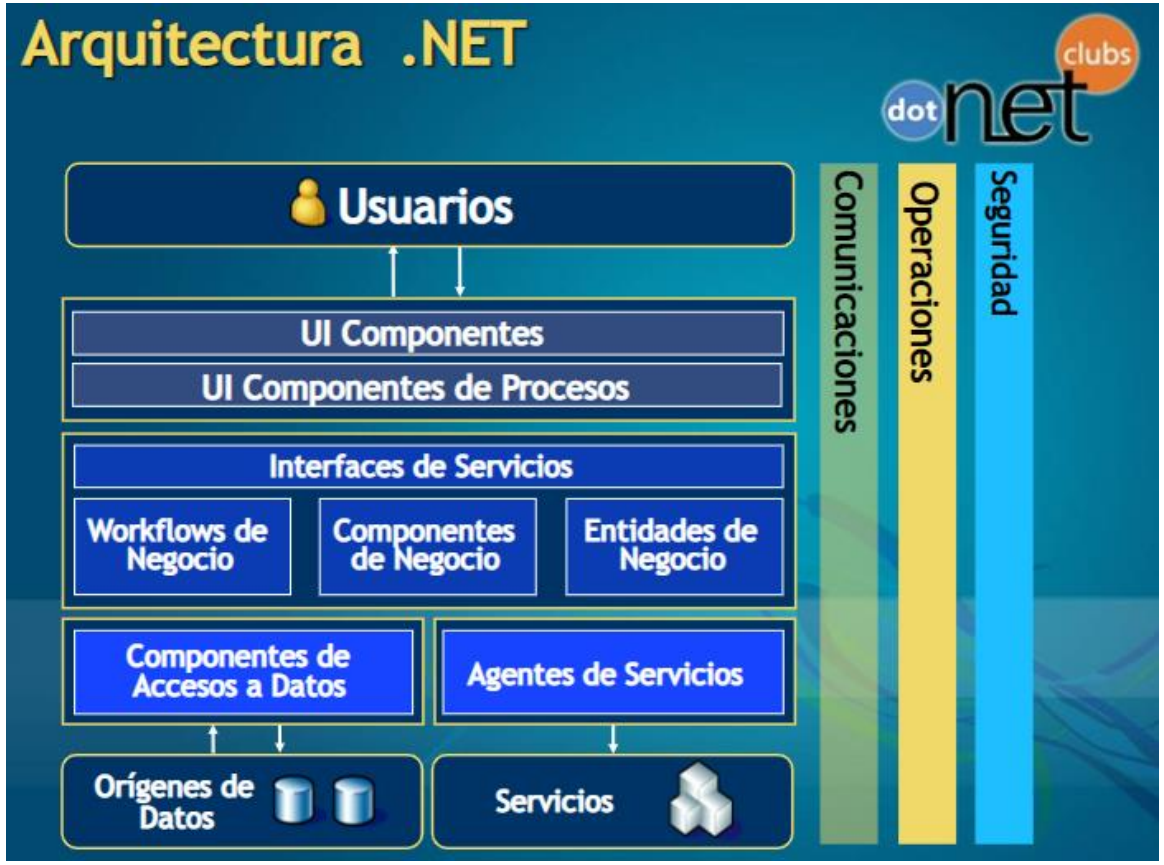
Interfaces de Servicios

- Es el punto de entrada de la capa de negocio
- Expone la funcionalidad que otras aplicaciones pueden consumir
- Se implementa con Servicios Web o Remoting (en caso de aplicaciones distribuidas)
- Se puede implementar con otras tecnologías BizTalk Server, Message Queues..

Entidades de Negocio

- Son contenedores de datos
- Encapsulan y ocultan los detalles de representación de datos
- Pueden encapsular datos que provengan de un Recordset, y luego enviarlos a un XML
- No tienen lógica de negocio
- Es conveniente distinguir entre instancia y colección de instancias
- Se referencian desde la capa de presentación, desde la interfaz de servicio y desde los componentes del negocio
- Alternativas: DataSet, DataSet tipado, Objetos (individuales y colecciones), XML

Casos de uso de la arquitectura (ejemplo 3)



Arquitectura en capas con .net

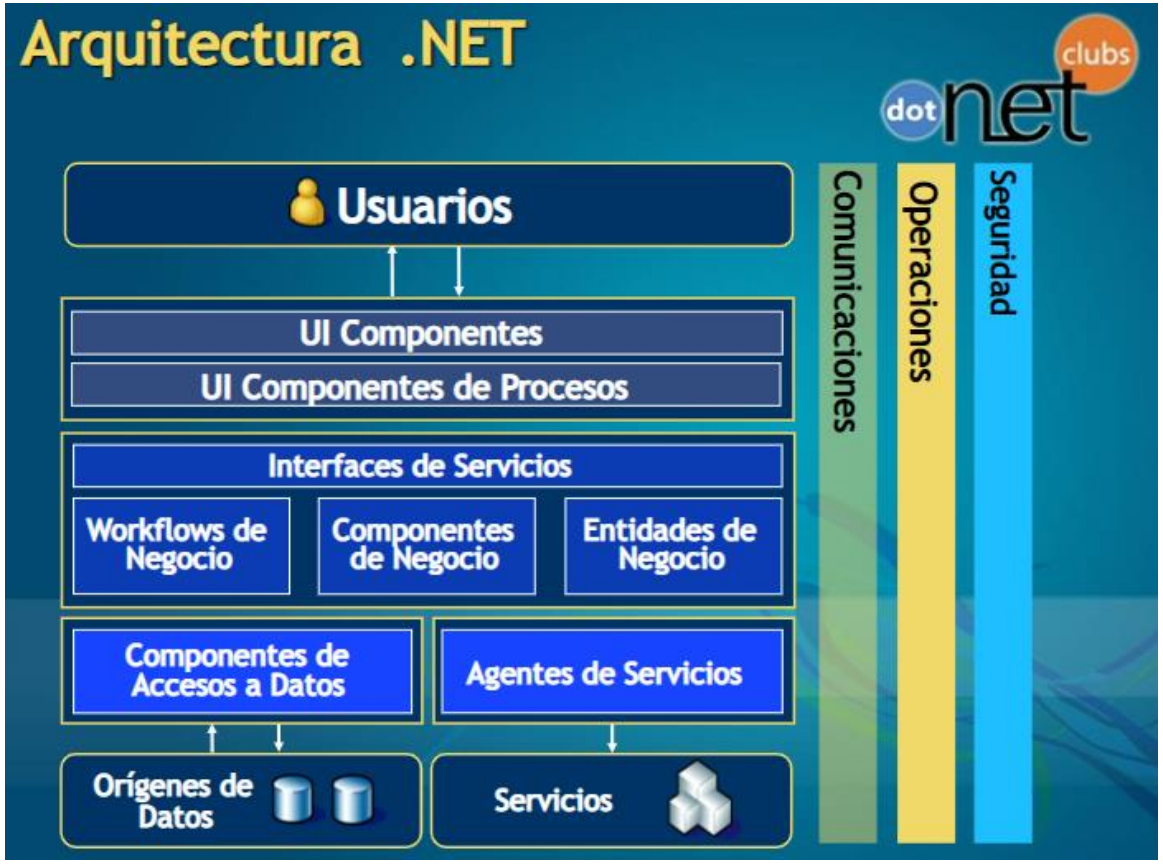
Componentes del negocio

- Implementación en software de conceptos de negocios.
- Encapsulan las reglas de negocio de la aplicación, relacionadas con un Business Entity
- Algunos métodos requieren acceder a la base de datos (Capa de datos)
- Separación de las Business Entities

Business Work

- Implementan las actividades de alto nivel del negocio: proceso de una orden de compra, de una factura.
- Son métodos que no pertenecen a un objeto en particular.
- Se pueden agrupar en objetos o en un objeto por método.
- Cada método de un Service Interface, accede a un Business Workflow o a un componente de Negocio.

Casos de uso de la arquitectura (ejemplo 3)



Arquitectura en capas con .net

Capa de presentación

- Para mucha aplicaciones se usa la metáfora del formulario/informe
- Habrá formularios/paginas web de ingreso y modificación
- Habrá formularios/paginas web de vista de datos
- Son los componentes de Interfaz
- Hay componentes de Proceso de Interfaz

User Interface component

- Muestran datos a los usuarios
- Adquieren y validan la entrada de los usuarios
- Interpretan gestos del usuario para ejecutar una acción.
- No participan, no hacen nada por si solos.

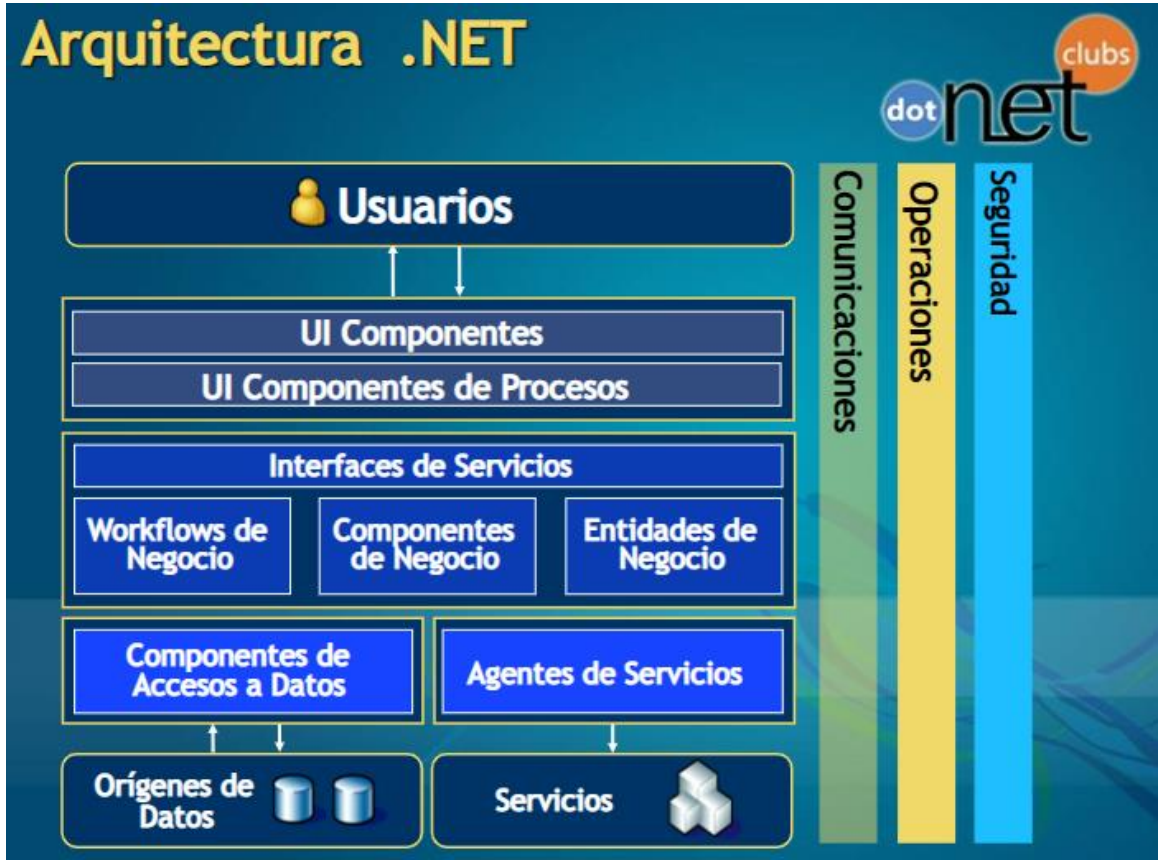
Interfaz Windows

- Para entornos desconectados o clientes ricos (rich clients)
- Opciones de implementación mas simples (Windows forms)

Interfaz Web

- ASP.NET Web Forms
- Basado en componentes en el servidor.
- Usa enlace de datos en los controles.
- Entorno de desarrollo integrado al resto de las soluciones.
- Amplio maneja de estado y caching.
- Nuevo modelo de formularios.

Casos de uso de la arquitectura (ejemplo 3)

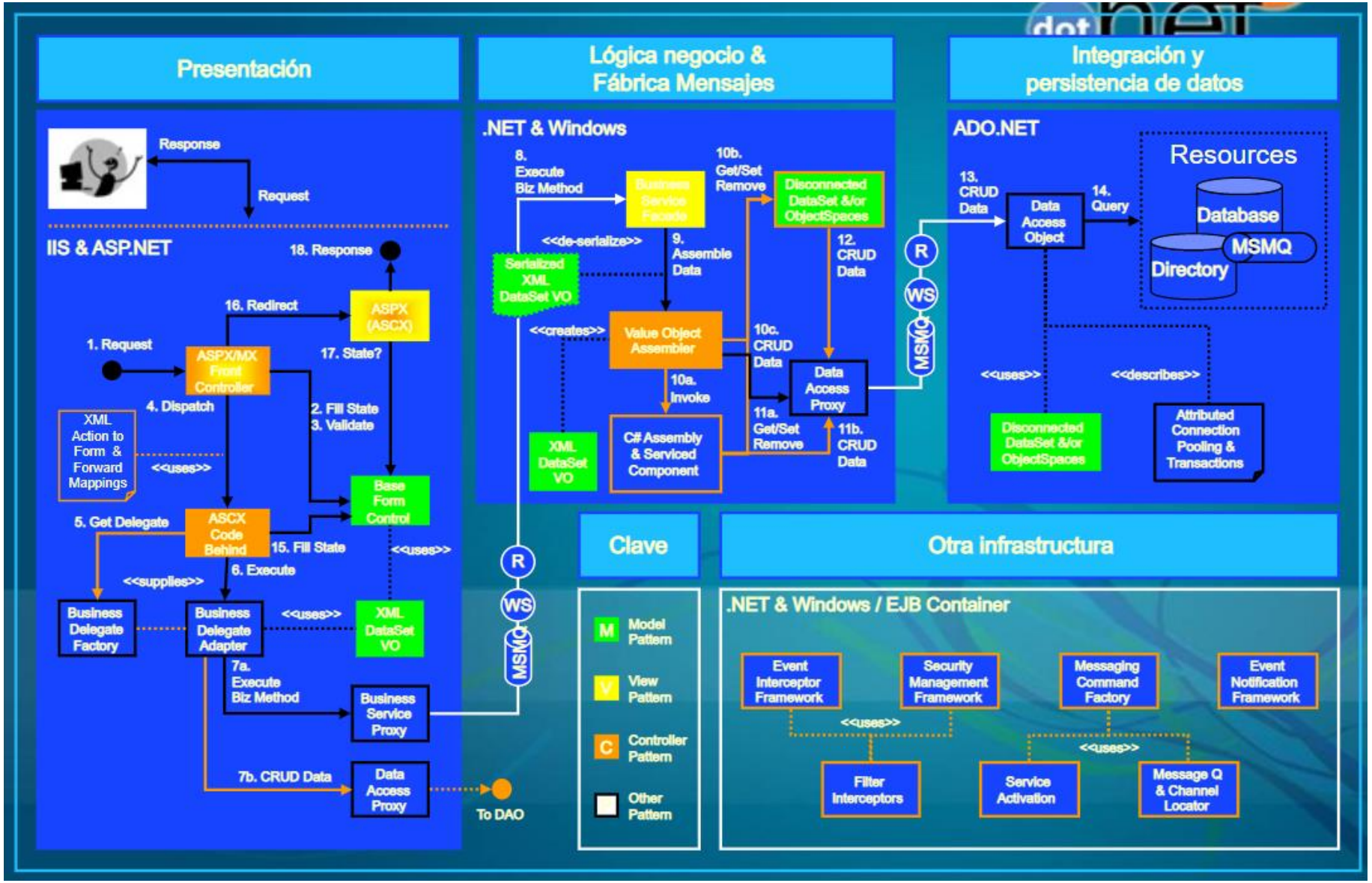


Arquitectura en capas con .net

Capa de infraestructura

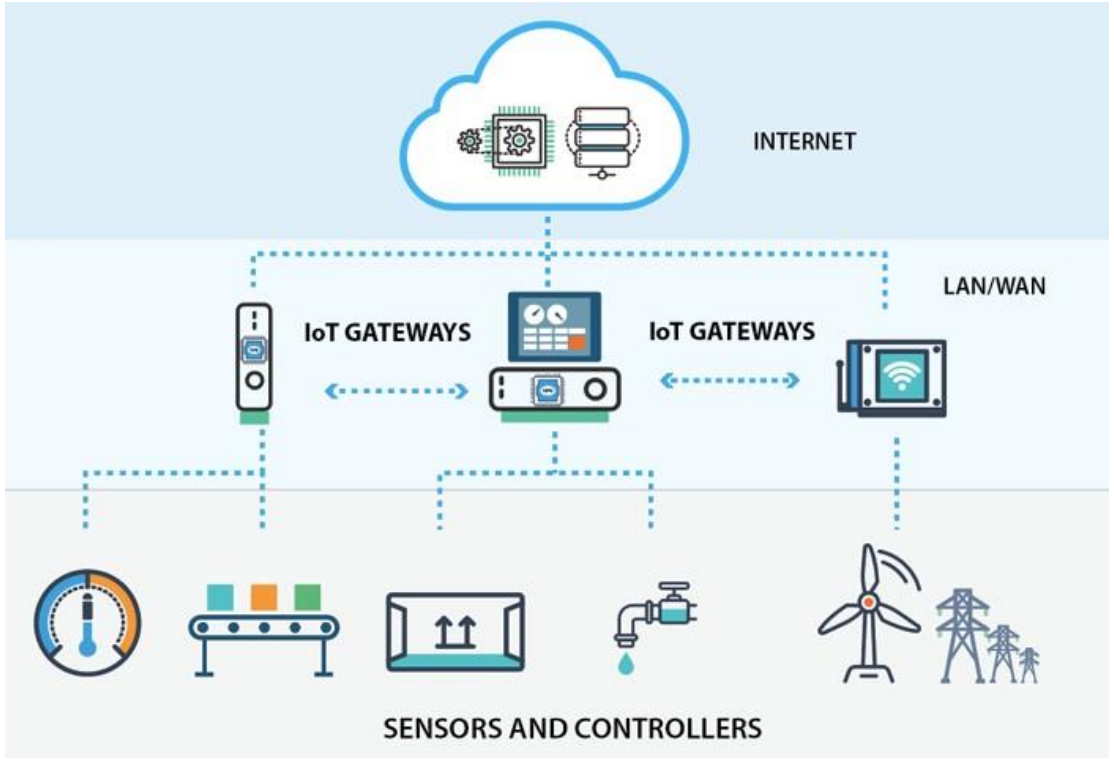
- Son servicios para las aplicaciones
- Dado que no pertenecen a ninguna capa, se definen por fuera aunque en algunos casos se implementen o usen en alguna capa
- Son los siguientes:
  - ✓ Seguridad (AuthZ, AuthN, Comunicación segura, Auditoria, Manejo de perfiles
  - ✓ Operaciones (Manejo de excepciones, monitoreo, execusión asincrónica, metadatos, configuración)
  - ✓ Comunicaciones (formato, protocolo, asincronismo)

Casos de uso de la arquitectura (ejemplo 4)

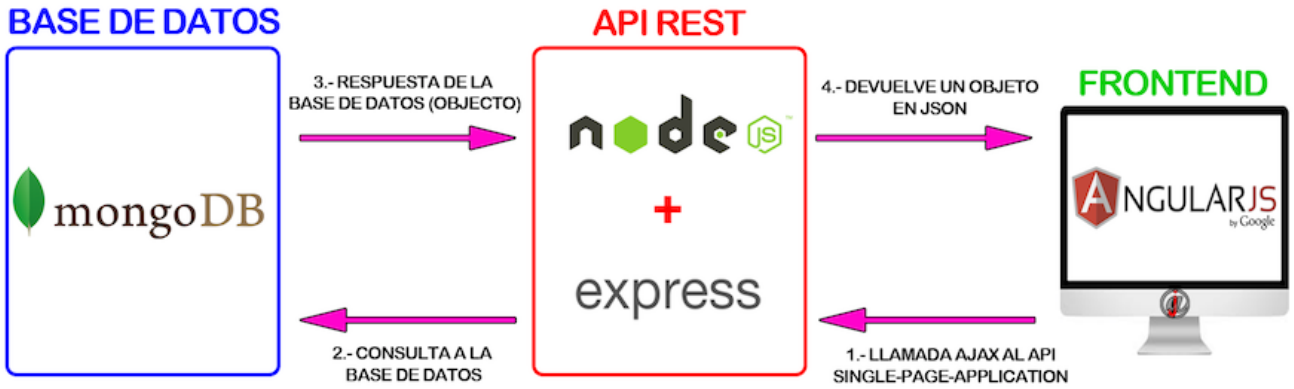


Casos de uso de la arquitectura (ejemplo 5)

Ejemplo de un caso de Arquitectura en capas para IOT



Ejemplo de un caso de Arquitectura en capas para MongoDB, API, Angular JS



Arquitectura en capas

Ventajas / desventajas de la arquitectura

- Las llamadas de la interfaz del usuario en la estación de trabajo, al servidor de capa intermedia, son más flexibles que en el diseño de dos capas, ya que la estación solo necesita transferir parámetros a la capa intermedia.
- Con la arquitectura de tres capas, la interfaz del cliente no es requerida para comprender o comunicarse con el receptor de los datos. Por lo tanto, esa estructura de los datos puede ser modificada sin cambiar la interfaz del usuario en la PC.
- El código de la capa intermedia puede ser reutilizado por múltiples aplicaciones si está diseñado en formato modular.
- La separación de roles en tres capas, hace más fácil reemplazar o modificar una capa sin afectar a los módulos restantes.

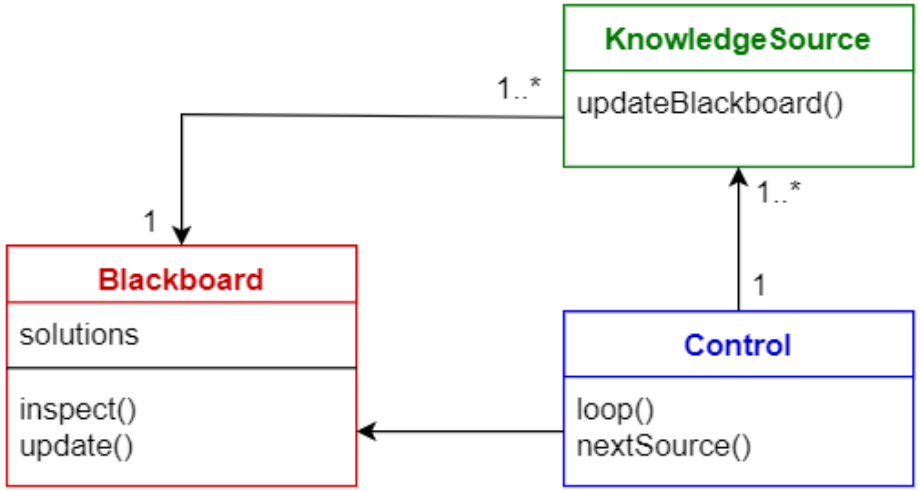
Patrón de Arquitectura de repositorio o Pizarra (Black Board)

Componentes que conforman la arquitectura

Este patrón es útil para problemas para los que no se conocen estrategias de solución deterministas. El patrón de pizarra consta de 3 componentes principales.

- **pizarra** : una memoria global estructurada que contiene objetos del espacio de solución
- **fuente de conocimiento** : módulos especializados con su propia representación
- **componente de control** : selecciona, configura y ejecuta módulos.

Todos los componentes tienen acceso a la pizarra. Los componentes pueden producir nuevos objetos de datos que se agregan a la pizarra. Los componentes buscan tipos particulares de datos en la pizarra, y pueden encontrarlos por coincidencia de patrones con la fuente de conocimiento existente.

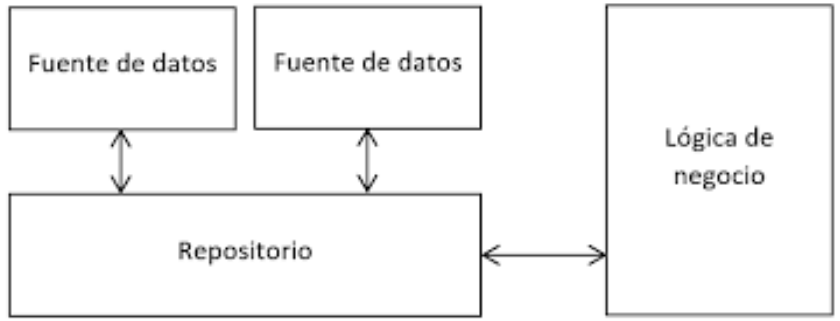


Patrón de pizarra

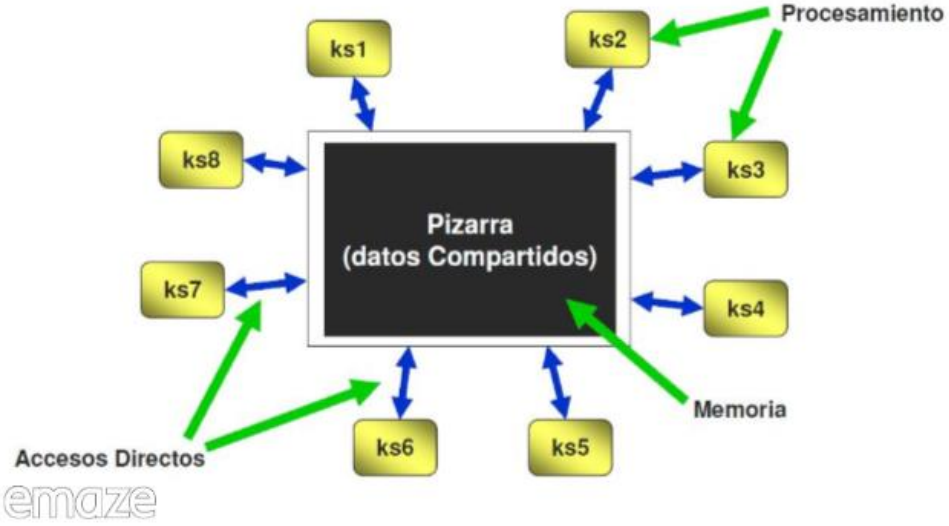
Patrones de Arquitectura de repositorio o pizarra

Características de la arquitectura

- Un repositorio realiza las tareas de intermediario entre las capas de modelo de dominio y nuestros datos, actuando de forma similar a una colección de memoria de objetos del dominio.
- Los repositorios son clases/componentes que encapsulan la lógica requerida para acceder a las fuentes de datos de la aplicación. Centralizan por lo tanto la funcionalidad común de acceso de datos de forma que la aplicación pueda disponer de un mejor mantenimiento y desacoplamiento entre la tecnología y la lógica de las capas.



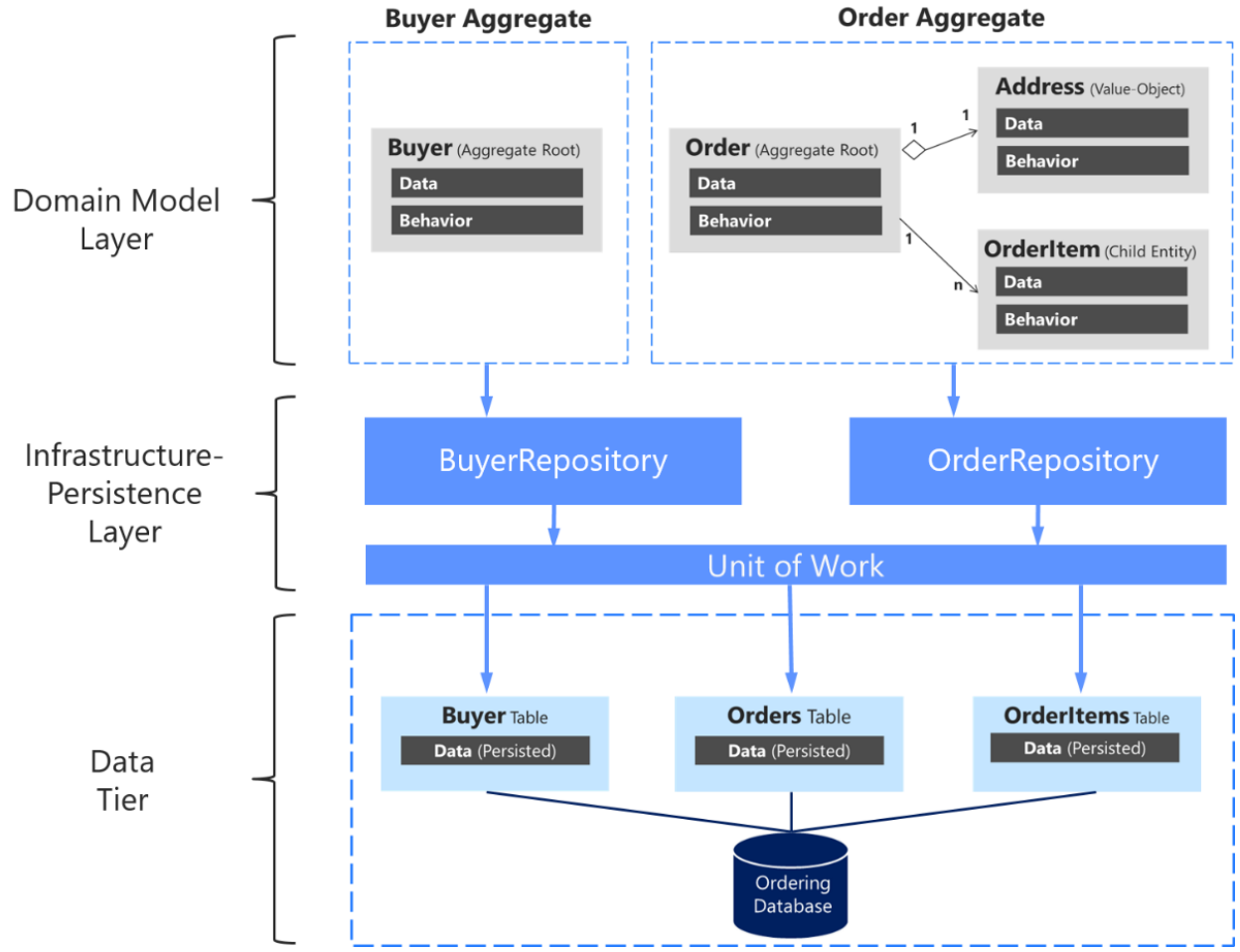
- Esta arquitectura se base en un almacenamiento de datos central (**pizarra**), el cual es accedido en cualquier momento por componentes que actualizan, agregan y borran datos del repositorio.
- Una pizarra tiene dos tareas: coordina a los distintos **agentes**, y facilita su intercomunicación. El estado inicial de la pizarra es una descripción del problema que resolver y el estado final será la solución del problema.
- Los **agentes** suelen estar especializados en una tarea en concreto, cooperando entre ellos para alcanzar una meta en común. El comportamiento básico de cualquier agente consiste en examina la pizarra, realizar su tarea y escribir sus conclusiones en la misma pizarra.



Patrones de Arquitectura de repositorio o pizarra

Casos de uso de la arquitectura (Ejemplo 1)

- Uno por cada modelo de negocio, la forma mas directa es crear un repositorio para cada modelo de negocio clase, cuidando la duplicidad de código o que varios repositorio necesiten interactuar entre sí.
- El uso de la raíz de agregado, una raíz agregada es una clase que puede existir por sí mismo y es responsable de la gestión de las asociaciones con otras clases relacionadas. Por ejemplo, en una aplicación de comercio electrónico, se podría tener un OrderRepository que se ocupe de la creación de una orden y sus elementos de detalle correspondiente.
- El repositorio genérico, en lugar de crear clases de repositorios específicos, un desarrollador puede aprovechar de los genéricos .NET para construir un repositorio común que se puede utilizar a través de múltiples aplicaciones.



El DbContext Entity Framework básicamente se parece a un *Repositorio* (y también a una *Unidad de Trabajo*). No necesariamente tienes que abstraerlo en escenarios simples. La principal ventaja del repositorio es que su dominio puede ser ignorante e independiente del mecanismo de persistencia. En una arquitectura basada en capas, las dependencias apuntan desde la capa UI hacia abajo a través del dominio (o generalmente llamada capa lógica empresarial) a la capa de acceso a datos.

Casos de uso de la arquitectura (Ejemplo 1)

Patrón de repositorio con Entity Framework Core (EF Core) en las librerías, aplicado en el backend del primer módulo de la aplicación.

Puntos Importantes

1. Implementación del patrón de repositorio
2. Facilidades para configurar modelos en Entity Framework Core
3. Aplicación del proceso MDA - Model Driven Architecture
4. Migraciones con Entity Framework Core "Code First"

Herramientas y plataforma

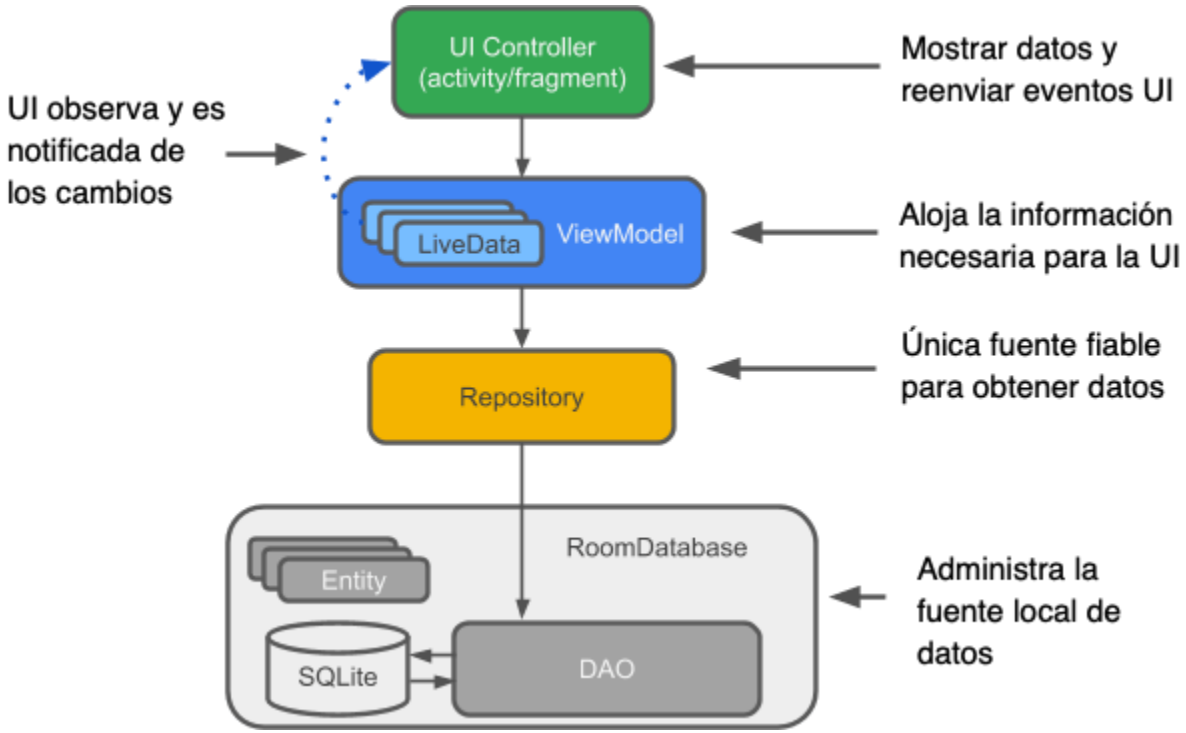
- Visual Studio 2017 Community Edition
  (ver la página de descargas de Visual Studio para otras versiones).
- Productivity Power Tools 2017
- SQL Server Developer Edition
- SQL Server Management Studio
- .NET Core 1.1.2 - x64 Installer
- .NET Core SDK 1.0.4 - x64 Installer
  (ver la página de descargas de .NET Core para otras versiones)

Paquetes NuGet utilizados

- Microsoft.EntityFrameworkCore - 1.1.2
- Microsoft.EntityFrameworkCore.Design - 1.1.2
- Microsoft.EntityFrameworkCore.SqlServer - 1.1.2
- NLog - 5.0.0-beta07
- System.ComponentModel.Annotations - 4.3.0

Contexto

El primer paso para esto es la separación por capas, típicamente Modelo, Datos y Servicios y en segundo lugar la separación en áreas funcionales o módulos



Ventajas del patrón repositorio

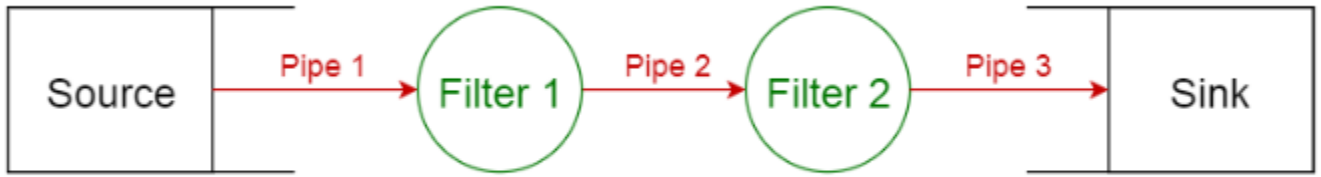
- Se presenta al desarrollador de otras capas del sistema (capas Dominio o Negocios, Aplicación, etc) un modelo mas sencillo e independiente de obtener, objetos/entidades persistidos y gestionar su ciclo de vida.
- Desacopla a las capas superior dominio/negocio y aplicación de la tecnología de persistencia de datos. Por ejemplo, si se tiene un repositorio que use un archivo plano .txt para persistencia luego se quiere uno para guardar en la base de datos, pues se tendría una interfaz que declare el como se ha de hacer, solo se tiene que implementar dicha interfaz.
- Permiten ser sustituido por implementaciones falses de accesos de datos, a ser utilizadas en pruebas unitarias. Imagine que la clase que se quiere testar ejecuta operaciones CRUD en la base de datos. Se vuelve contraproducente si para la prueba de dicha clase has de crear X registros en tu base de datos y luego borrar. Y una solución seria crear un repositorio de provisional para las pruebas sin depender de la base de datos, y no generar basura en la base de datos.

Patrón de Arquitectura de Pipe & Filter

Patrón de tubería y filtro (pipe and filter)

Componentes que conforman la arquitectura

Este patrón se puede usar para estructurar sistemas que producen y procesan una secuencia de datos. Cada paso de procesamiento se incluye dentro de un componente de **filtro** . Los datos que se procesarán se pasan a través de las **tuberías** . Estas tuberías se pueden utilizar para el almacenamiento en búfer o con fines de sincronización.



Patrón de filtro de tubería

Características de la Arquitectura

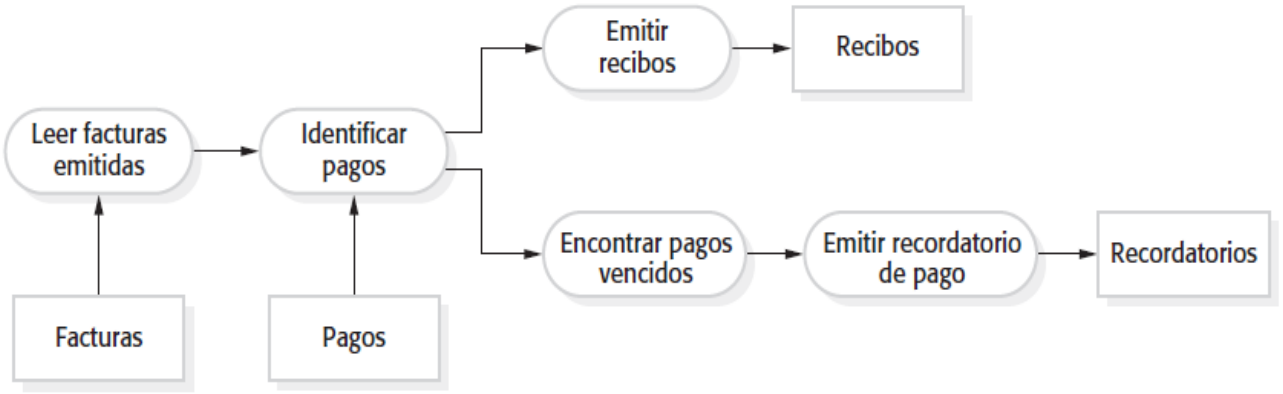
- El procesamiento de datos en un sistema se organiza de forma que cada componente de procesamiento (filtro) sea discreto y realice un tipo de transformación de datos. Los datos fluyen (como en una tubería) de un componente a otro para su procesamiento.
- Una tubería (pipeline) es una popular arquitectura que conecta componentes computaciones (filtros) a través de conectores (pipes), de modo que las computaciones se ejecutan a la manera de un flujo. Los datos se transportan a través de las tuberías entre los filtros, transformando gradualmente las entradas en salidas.

- Una tubería (pipeline) es una popular arquitectura que conecta componentes computaciones (filtros) a través de conectores (pipes), de modo que las computaciones se ejecutan a la manera de un flujo. Los datos se transportan a través de las tuberías entre los filtros, transformando gradualmente las entradas en salidas.
- El sistema tubería-filtros se percibe como una serie de transformaciones sobre sucesivas piezas de los datos de entrada. Los datos entran al sistema y fluyen a través de los componentes. En el estilo secuencial por lotes (batch sequential) los componentes son programas independientes; el supuesto es que cada paso se ejecuta hasta completarse antes que se inicie el paso siguiente.

Patrón de tubería y filtro (pipe and filter)

Casos de uso de la arquitectura (Ejemplo 1)

Este es un modelo de la organización en tiempo de operación de un sistema, donde las transformaciones funcionales procesan sus entradas y producen salidas. Los datos fluyen de uno a otro y se transforman conforme se desplazan a través de la secuencia. Las transformaciones pueden ejecutarse secuencialmente o en forma paralela.



Ejemplo de arquitectura de tubería y filtro

Patrón de tubería y filtro (pipe and filter)

Ventajas y desventajas de la arquitectura

- **Ventajas** Fácil de entender y soporta reutilización de transformación. El estilo del flujo de trabajo coincide con la estructura de muchos procesos empresariales. La evolución al agregar transformaciones es directa. Puede implementarse como un sistema secuencial o como uno concurrente.
- **Desventajas** El formato para la transferencia de datos debe acordarse entre las transformaciones que se comunican. Cada transformación debe analizar sus entradas y sintetizar sus salidas al formato acordado. Esto aumenta la carga del sistema, y puede significar que sea imposible reutilizar transformaciones funcionales que usen estructuras de datos incompatibles.

Patrón de Arquitectura

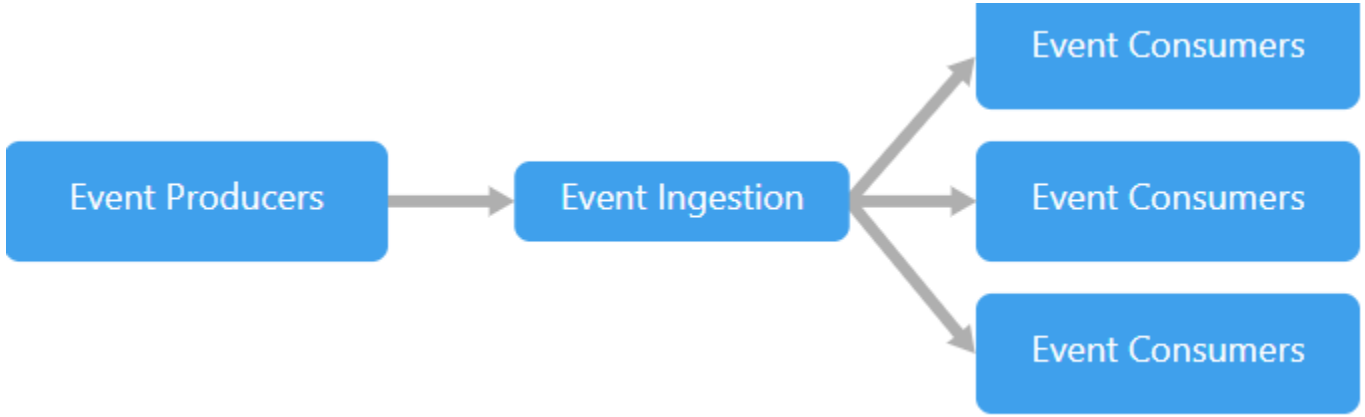
Basada en eventos

Patrón de arquitectura Event-driven

Características de la arquitectura

Una arquitectura basada en eventos consta de **productores de eventos** que generan un flujo de eventos, y **consumidores de eventos** que escuchan los eventos.

- Los eventos se entregan casi en tiempo real, de modo que los consumidores pueden responder inmediatamente a los eventos cuando se producen. Los productores se desconectan de los consumidores — un productor no sabe que los consumidores están escuchando. Los consumidores también se desconectan entre sí, y cada consumidor ve todos los eventos. Esto difiere de un patrón Competing Consumers, donde los consumidores extraen los mensajes de una cola y los mensajes solo se procesan una vez (suponiendo que no haya errores). En algunos sistemas, como IoT, los eventos se deben ingerir en volúmenes muy altos.



Patrón de arquitectura Event-driven

Componentes de la arquitectura

Una arquitectura basada en eventos puede usar un modelo pub/sub o un modelo de flujo de eventos.

•**Pub/sub**: la infraestructura de mensajería mantiene un seguimiento de las suscripciones. Cuando se publica un evento, se envía el evento a cada suscriptor. Después de que se recibe un evento, no se puede reproducir, y los nuevos suscriptores no ven el evento.

•**Flujo de eventos**: los eventos se escriben en un registro. Los eventos siguen un orden estricto (dentro de una partición) y son duraderos. Los clientes no se suscriben al flujo, sino que un cliente puede leer desde cualquiera de sus partes. El cliente es responsable de avanzar su posición en el flujo. Esto significa que un cliente puede unirse en cualquier momento y puede reproducir los eventos.

En el lado del consumidor, hay algunas variaciones comunes:

•**Procesamiento sencillo de eventos**. Un evento desencadena inmediatamente una acción en el consumidor. Por ejemplo, podría usar Azure Functions con un desencadenador de Service Bus para que una función se ejecute cada vez que se publica un mensaje en un tema de Service Bus.

•**Procesamiento de eventos complejos**. Un consumidor procesa una serie de eventos, en busca de patrones en los datos de eventos, mediante una tecnología como Azure Stream Analytics o Apache Storm. Por ejemplo, podría agregar las lecturas de un dispositivo insertado durante una ventana de tiempo y generar una notificación si la media móvil supera un umbral determinado.

•**Procesamiento de secuencias de eventos**. Use una plataforma de flujo de datos, como Azure IoT Hub o Apache Kafka, como canalización para ingerir eventos y suministrarlos a procesadores de flujo. Los procesadores de flujos sirven para procesar o transformar el flujo. Puede haber varios procesadores de flujo para diferentes subsistemas de la aplicación. Este enfoque es una buena opción para las cargas de trabajo de IoT.

Patrón de arquitectura Event-driven

Cuándo utilizar esta arquitectura

•Varios subsistemas deben procesar los mismos eventos.

•Procesamiento en tiempo real con retardo mínimo.

•Procesamiento de eventos complejos, como coincidencia de patrones o agregación durante ventanas de tiempo.

•Gran volumen y alta velocidad de datos, como IoT.

Ventajas

•Se desvinculan productores y consumidores.

•No hay integraciones punto a punto. Es fácil agregar nuevos consumidores al sistema.

•Los consumidores pueden responder a eventos inmediatamente a medida que llegan.

•Muy escalable y distribuida.

•Los subsistemas tienen vistas independientes del flujo de eventos.

Desafíos

•Entrega garantizada. En algunos sistemas, especialmente en escenarios de IoT, es fundamental garantizar la entrega de los eventos.

•Procesamiento de eventos en orden o exactamente una vez. Cada tipo de consumidor normalmente se ejecuta en varias instancias, a fin de conseguir resistencia y escalabilidad. Esto puede suponer un desafío si se deben procesar los eventos en orden (dentro de un tipo de consumidor), o si la lógica de procesamiento no es idempotente.

Patrón de Arquitectura

Event Driven

Patrón de arquitectura Event-driven

Componentes de la arquitectura

Una arquitectura guiada por eventos puede estar basada tanto en un modelo pub/sub como en un modelo event stream.

- **Modelo de Pub/Sub (Publicación/Suscripción):**

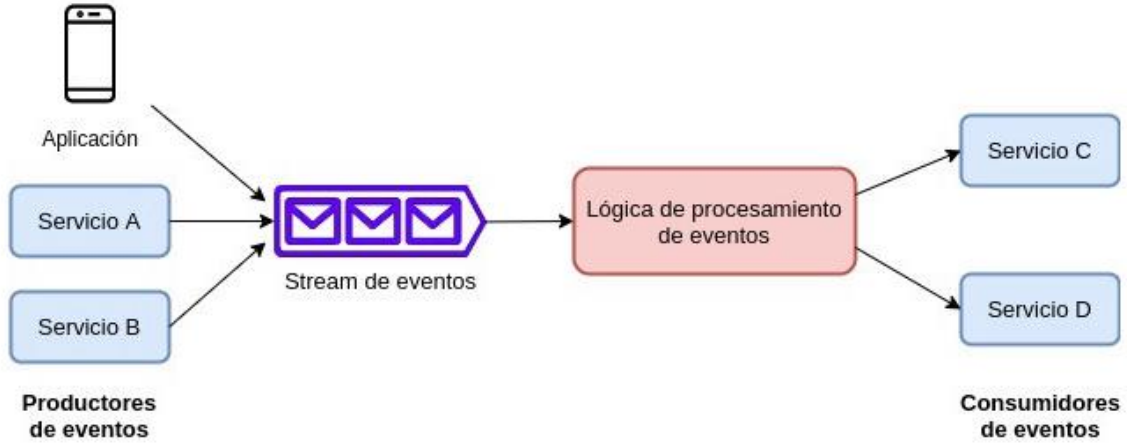
Es una infraestructura de mensajería que se basa en subscripciones a un flujo de eventos. Con este modelo, una vez que se genera o se publica un evento, este se envía a los suscriptores que necesitan estar informados al respecto.

- **Modelo de Event-Streaming (Transmisión de eventos):**

Con este modelo event streaming, los eventos se escriben en un registro. Los consumidores de eventos no se suscriben a un event streaming. en vez de esto, pueden leer cualquier parte del stream y unirse en cualquier momento.

La arquitectura sin bloqueo (non-bloking) también se conoce como arquitectura reactiva o controlada por eventos(event-driven). Las arquitecturas basadas en eventos son bastante populares en los desarrollos modernos de aplicaciones web.

Son capaces de manipular una gran cantidad de conexiones concurrentes con un consumo mínimo de recursos. Las aplicaciones modernas necesitan un modelo completamente asíncrono para escalar. Estos marcos web modernos proporcionan un comportamiento más confiable en un entorno distribuido.



Patrón de arquitectura Event-driven

Características de la Arquitectura

- La Arquitectura Dirigida por Eventos se centra en la generación y entrega de notificaciones de eventos. Este concepto define arquitecturas flexibles, en las cuales los elementos que generan las notificaciones de eventos no necesitan los componentes del receptor.
- Este paradigma convierte en una realidad la creación de arquitecturas con respuesta en tiempo real. Las notificaciones de eventos implican modificaciones en el estado actual del sistema. Las notificaciones se pueden disparar bien por fuentes externas como pueden ser inputs de usuarios, o necesidades del mercado. Sin embargo, también hay notificaciones de eventos internas, como el envío de información por el pipeline de la cadena de trabajo,
- multidifusión de parámetros para procesos heterogéneos, disparadores internos para ciertos servicios y generación de outputs. Al final, los eventos se pueden entender como una especie de mensajes entre distintos módulos del sistema, conteniendo información relevante acerca del funcionamiento general y particular del sistema y sus servicios.

- Una arquitectura guiada por eventos utiliza eventos para dispararse y comunicarse entre servicios desacoplados y es común en aplicaciones modernas construidas con microservicios. Un evento es un cambio de estado, o una actualización, como un elemento cuando se coloca en un carrito de la compra en un e-commerce. Los eventos pueden llevar el estado (el elemento comprado, su precio y la dirección de entrega) o pueden ser identificadores (una notificación de que el pedido se ha enviado).

Patrón de arquitectura Event-driven

Capas de flujo de eventos de la Arquitectura

Capas de flujo de eventos

Una arquitectura de evento disparado se basa en cuatro capas lógicas. Se inicia con la detección de un hecho, su representación técnica en la forma de un evento y termina con un conjunto no vacío de reacciones a ese evento.

● **Generador de eventos:** La primera capa lógica es el generador de eventos, que detecta un hecho y representa el hecho de en un evento. Dado que un hecho puede ser casi cualquier cosa que se puede detectar, por lo que puede un generador de eventos también serlo. La conversión de los diferentes datos recogidos por los sensores de una forma estandarizada que se pueda evaluar es un problema importante en el diseño e implementación de esta capa .

● **Canal de eventos:** Un canal de evento es un mecanismo mediante el cual la información a partir de un generador de eventos se transfiere al motor de eventos o en el fregadero. Esto podría ser una conexión TCP / IP o cualquier tipo de archivo de entrada (text plano, formato XML, correo electrónico, etc.) Varios canales de eventos se pueden abrir al mismo tiempo.

● **Motor de procesamiento de eventos:** El motor de procesamiento de eventos es donde se identifica el evento, y la reacción adecuada se selecciona y se ejecuta.

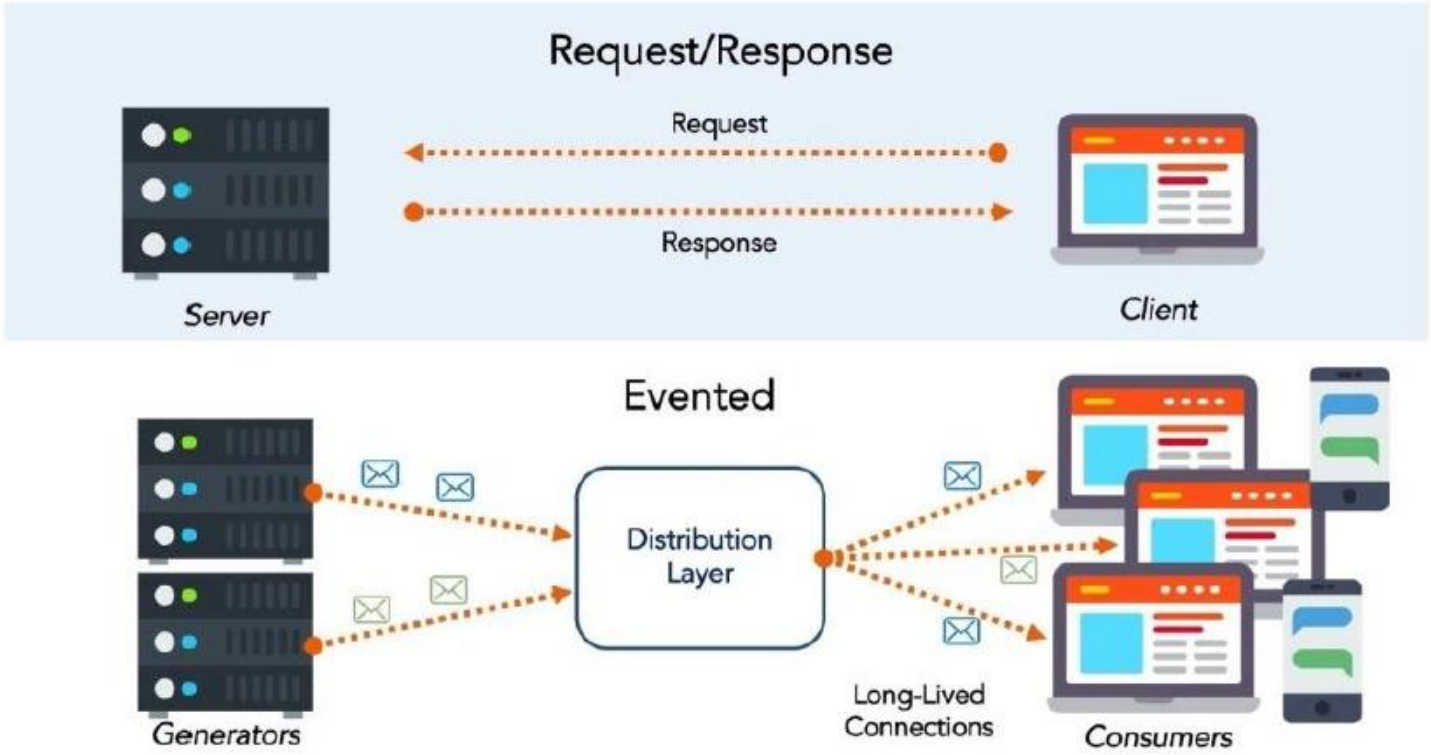
Esto también puede dar lugar a una serie de afirmaciones que se producen. Es decir, si el evento que entra en el motor de procesamiento de eventos es un "identificador de producto bajo en la acción", esto puede desencadenar reacciones tales como, "ID de pedido del producto" y "Notificar al personal".

● **Actividad de respuesta dirigida por evento:** Aquí es donde se muestran las consecuencias del suceso. Esto se puede hacer de muchas maneras y formas diferentes, Por ejemplo, un correo electrónico se envía a alguien y una aplicación puede mostrar algún tipo de advertencia en la pantalla Dependiendo del nivel de automatización proporcionada por el receptor (el motor de procesamiento de eventos) la actividad aguas abajo puede no ser necesaria.

Patrón de arquitectura Event-driven

Casos de uso de la arquitectura (ejemplo 1)

Los ejemplos de sistemas que utilizan esta arquitectura son numerosos. El estilo se utiliza en ambientes de integración de herramientas, en sistemas de gestión de base de datos para asegurar las restricciones de consistencia (bajo la forma de disparadores, por ejemplo), en interfaces de usuario para separar la presentación de los datos de los procedimientos que gestionan datos, y en editores sintácticamente orientados para proporcionar verificación semántica incremental.



Patrón de arquitectura Event-driven

Ventajas y desventajas de la arquitectura

Patrón de Arquitectura

Microservicios

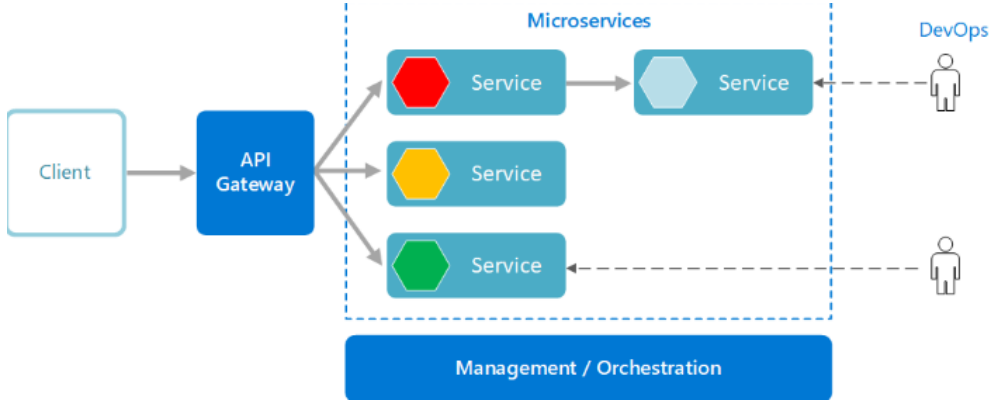
Componentes que conforman la arquitectura

Los microservicios (también conocidos como arquitectura de microservicios) son un estilo arquitectónico que estructura una aplicación como un conjunto de servicios sin conexión directa, que implementan capacidades empresariales. La arquitectura de microservicios permite la entrega/despliegue continuo de aplicaciones grandes y complejas. También permite a una organización evolucionar su pila tecnológica.

Los microservicios son pequeños e independientes, y están acoplados de forma imprecisa.

Un único equipo reducido de programadores puede escribir y mantener un servicio.

- Cada servicio es un código base independiente, que puede administrarse por un equipo de desarrollo pequeño.
- Los servicios pueden implementarse de manera independiente. Un equipo puede actualizar un servicio existente sin tener que volver a generar e implementar toda la aplicación.
- Los servicios son los responsables de conservar sus propios datos o estado externo.
- Esto difiere del modelo tradicional, donde una capa de datos independiente controla la persistencia de los datos.
- Los servicios se comunican entre sí mediante API bien definidas. Los detalles de la implementación interna de cada servicio se ocultan frente a otros servicios.
- No es necesario que los servicios compartan la misma pila de tecnología, las bibliotecas o los marcos de trabajo.



Una arquitectura de microservicios consta de una colección de servicios autónomos y pequeños. Los servicios son independientes entre sí y cada uno debe implementar una funcionalidad de negocio individual

Además de los propios servicios, hay otros componentes que aparecen en una arquitectura típica de microservicios:

● Administración e implementación. Este componente es el responsable de la colocación de servicios en los nodos, la identificación de errores, el equilibrio de servicios entre nodos, etc. Normalmente, este componente es una tecnología estándar, como Kubernetes, en lugar de algo creado de forma personalizada.

● Puerta de enlace de API La puerta de enlace de API es el punto de entrada para los clientes. En lugar de llamar a los servicios directamente, los clientes llaman a la puerta de enlace de API, que reenvía la llamada a los servicios apropiados en el back-end.

Deberíamos utilizar una arquitectura de microservicios para cualquier producto o proyecto con estos dos enfoques:

Únicamente monolítico o con un primer enfoque monolítico. Normalmente, cuando una aplicación monolítica tiene éxito o necesita ayuda para escalar y mejorar el rendimiento, podemos optar por una arquitectura de microservicios de dos maneras:

● Extender los componentes modulares bien diseñados de la aplicación monolítica.

● Crear la aplicación de microservicios desde cero y volcar en ella la aplicación monolítica existente.

Un primer enfoque de microservicios. Debemos optar por la primera aproximación a los microservicios cuando:

● La modularidad y la descentralización son un aspecto importante desde el inicio de cualquier proyecto.

● Prevemos que la aplicación tendrá un alto volumen de transacciones o de tráfico.

● Tenemos preferencia por beneficios a largo plazo en comparación con los de corto plazo.

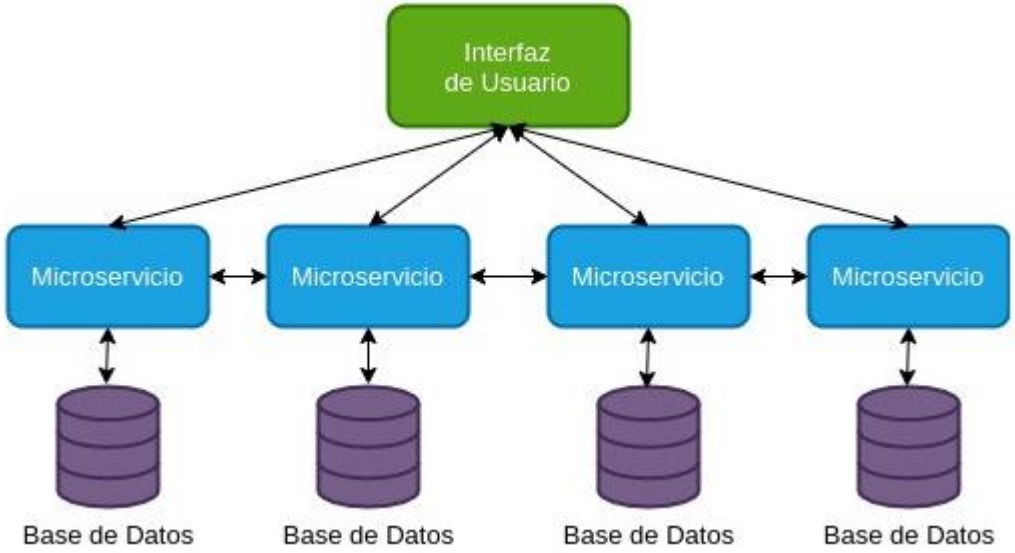
● Disponemos de un conjunto adecuado de personas para diseñar, desarrollar e implementar aplicaciones rápidamente, sobre todo durante la fase inicial.

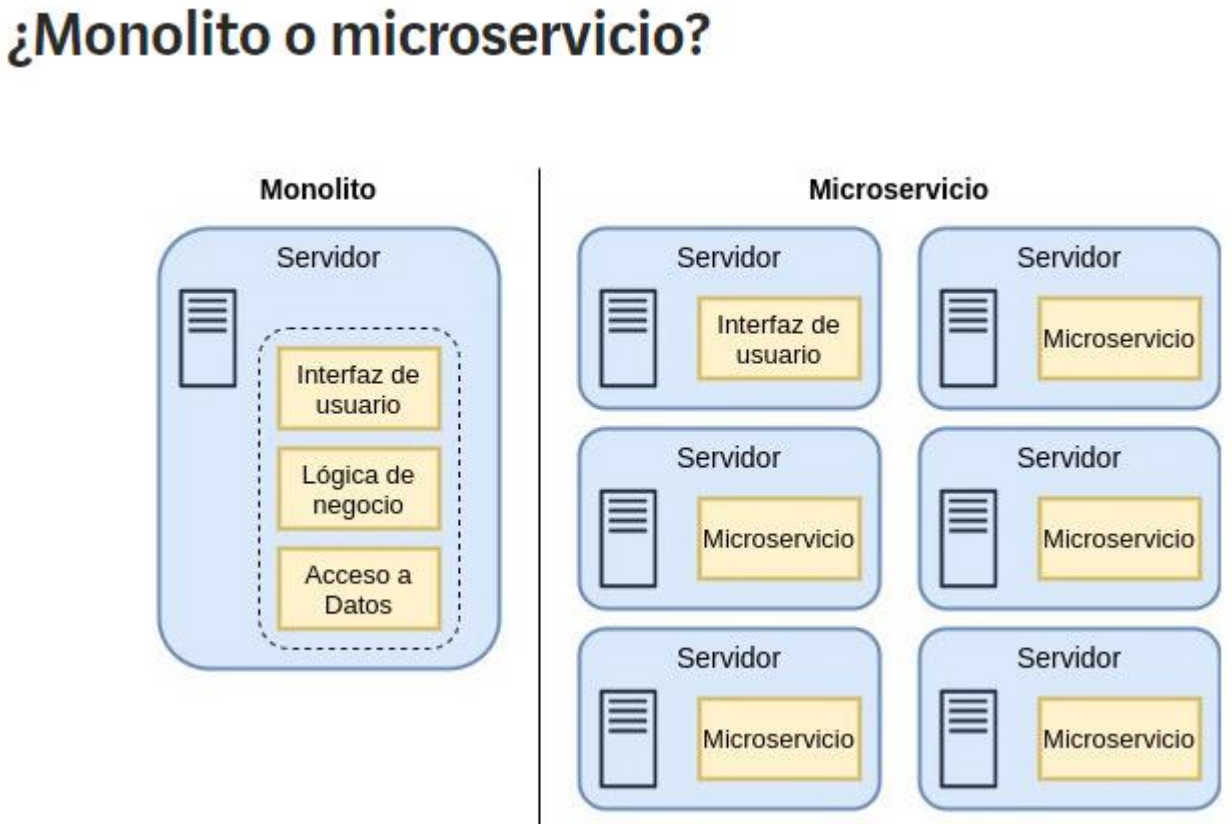
● Tenemos el compromiso de utilizar herramientas y tecnologías de vanguardia

Características de la Arquitectura

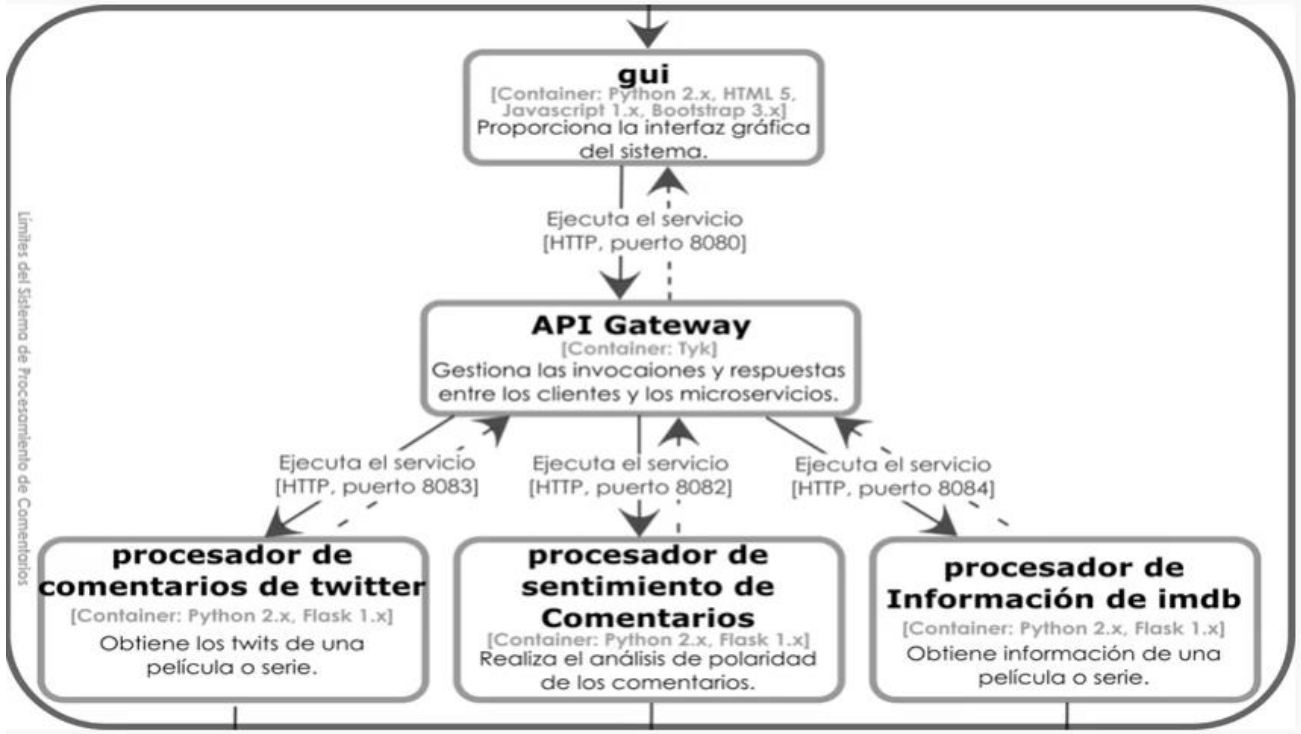
En una arquitectura de microservicios, diferentes características/tareas se dividen en módulos/bases de códigos, que en conjunto con cada uno de estos funcionan para formar un gran servicio completo.

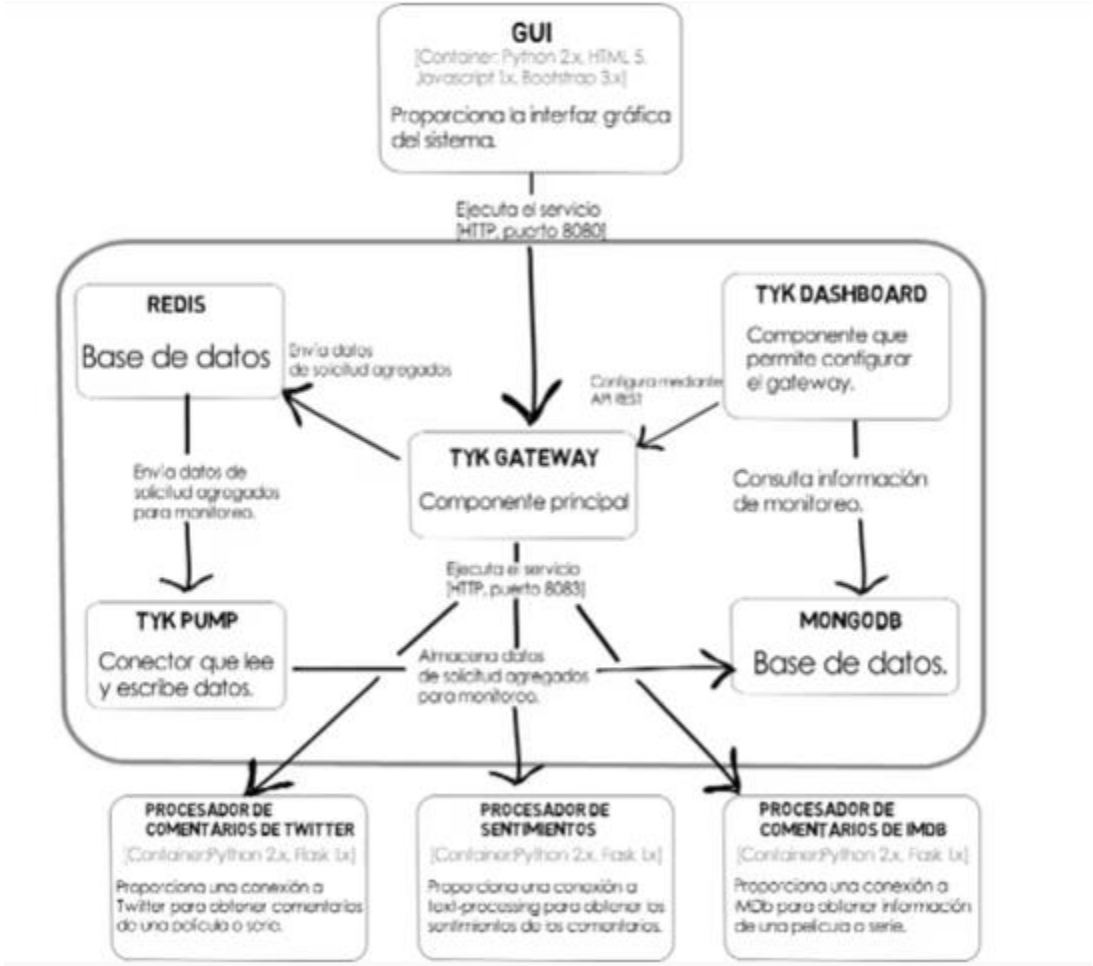
Esta arquitectura facilita el mantenimiento de una aplicación, el desarrollo de características, las pruebas y la implementación en comparación con una arquitectura monolítica.



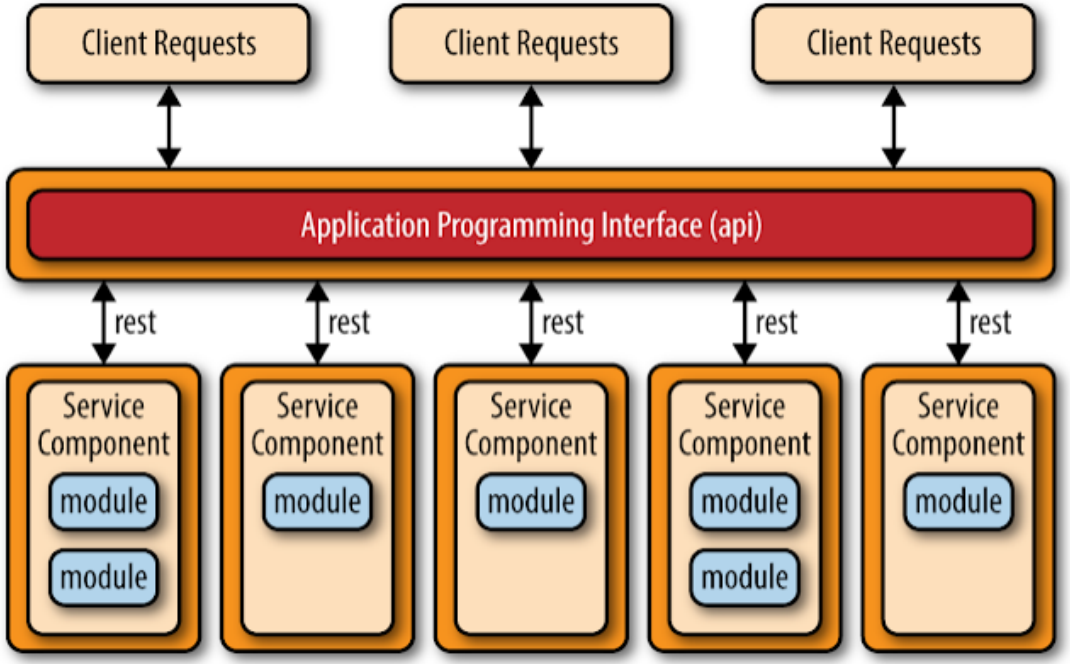


Casos de uso de la arquitectura (ejemplo 1)

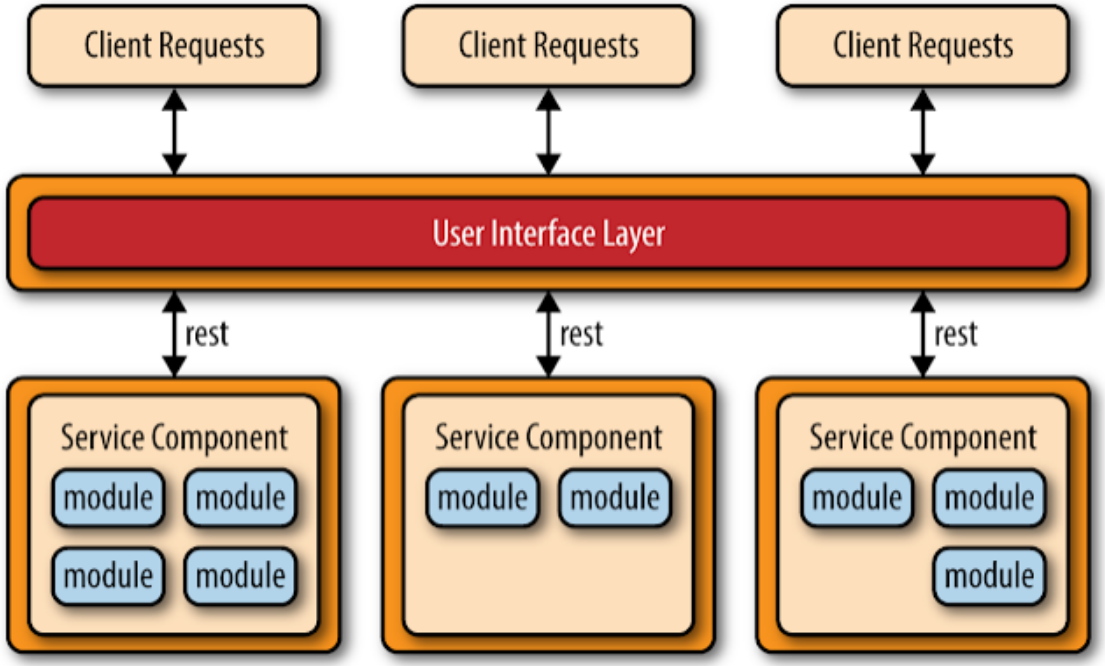




Casos de uso de la arquitectura (ejemplo 2)



Casos de uso de la arquitectura (ejemplo 3)



Ventajas de la arquitectura

- **Agilidad** . Dado que microservicios se implementan de forma independiente, resulta más fácil de administrar las correcciones de errores y las versiones de
- características. Puede actualizar un servicio sin volver a implementar toda la aplicación y revertir una actualización si algo va mal.
- **Equipos pequeños y centrados** . Un microservicio debe ser lo suficientemente pequeño como para que un solo equipo de características lo pueda compilar, probar e implementar. Los equipos pequeños favorecen la agilidad. Los equipos grandes suelen ser menos productivos, porque la comunicación es más lenta, aumenta la sobrecarga de administración y la agilidad disminuye.
- **Base de código pequeña** . En las aplicaciones monolíticas, con el paso del tiempo se da la tendencia de que las dependencias del código se acaben enredarse, por lo que para agregar una nueva característica, es preciso tocar el código en muchos lugares. Al no compartir el código ni los almacenes de datos, la arquitectura de microservicios minimiza las dependencias y resulta más fácil agregar nuevas características.
- **Mezcla de tecnologías** . Los equipos pueden elegir la tecnología que mejor se adapte al servicio de una combinación de pilas de tecnología, según corresponda.
- **Aislamiento de errores** . Si un microservicio individual no está disponible, no interrumpe toda la aplicación, siempre que los microservicios de nivel superior estén diseñados para controlar los errores correctamente.
- **Escalabilidad** . Los servicios se pueden escalar de forma independiente, lo que permite escalar horizontalmente los subsistemas que requieren más recursos, sin tener que escalar horizontalmente toda la aplicación.
- **Aislamiento de los datos** . Al verse afectado solo un microservicio, es mucho más fácil realizar actualizaciones del esquema. En una aplicación monolítica, las actualizaciones del esquema pueden ser muy complicadas, ya que las distintas partes de la aplicación pueden tocar los mismos datos, por lo que realizar modificaciones en el esquema resulta peligroso.

Desventajas de la arquitectura

- **Complejidad** . Una aplicación de microservicios tiene más partes en movimiento que la aplicación monolítica equivalente. Cada servicio es más sencillo, pero el sistema como un todo es más complejo.
- **Desarrollo y pruebas** . La escritura de un servicio pequeño que utilice otros servicios dependientes requiere un enfoque que no sea escribir una aplicación tradicional monolítica o en capas. Las herramientas existentes no siempre están diseñadas para trabajar con dependencias de servicios.
- **Falta de gobernanza** . El enfoque descentralizado para la generación de microservicios tiene ventajas, pero también puede causar problemas. Puede acabar con tantos lenguajes y marcos de trabajo diferentes que la aplicación puede ser difícil de mantener.
- **Congestión y latencia de red** . El uso de muchos servicios pequeños y detallados puede dar lugar a más comunicación interservicios. Además, si la cadena de dependencias del servicio se hace demasiado larga (el servicio A llama a B, que llama a C...), la latencia adicional puede constituir un problema. Tendrá que diseñar las API con atención. Evite que las API se comuniquen demasiado, piense en formatos de serialización y busque lugares para utilizar patrones de comunicación asincrónica.
- **Integridad de datos** . Cada microservicio es responsable de la conservación de sus propios datos. Como consecuencia, la coherencia de los datos puede suponer un problema. Adopte una coherencia final cuando sea posible.
- **Administración** . Para tener éxito con los microservicios se necesita una cultura de DevOps consolidada. El registro correlacionado entre servicios puede resultar un desafío. Normalmente, el registro debe correlacionar varias llamadas de servicio para una sola operación de usuario.
- **Control de versiones** . Las actualizaciones de un servicio no deben interrumpir servicios que dependen de él. Es posible que varios servicios se actualicen en cualquier momento; por lo tanto, sin un cuidadoso diseño, podrían surgir problemas con la compatibilidad con versiones anteriores o posteriores.
- **Conjunto de habilidades** . Los microservicios son sistemas muy distribuidos. Evalúe cuidadosamente si el equipo tiene los conocimientos y la experiencia para desenvolverse correctamente.

Procedimientos recomendados

- **Adapte** los servicios al dominio empresarial.
- **Descentralice** todo. Los equipos individuales son responsables de diseñar y compilar servicios. Evite el uso compartido de código o esquemas de datos.
- El **almacenamiento** de datos debería ser **privado** para el servicio que posee los datos. Use el almacenamiento recomendado para cada tipo de servicio y de datos.
- Los servicios se comunican a través de **API** bien diseñadas. Evite la pérdida de detalles de la implementación. Las API deben modelar el dominio, no la implementación interna del servicio.
- Evite el acoplamiento entre servicios. Entre las causas de acoplamiento se encuentran los **protocolos de comunicación** rígidos y los esquemas de bases de datos compartidos.
- Descargue a la puerta de enlace de cuestiones transversales, como la **autenticación** y la terminación SSL.
- Conserve el conocimiento del dominio fuera de la puerta de enlace. La puerta de enlace debe controlar y enrutar las solicitudes de cliente sin ningún conocimiento de **las reglas de negocios** o la lógica de dominio. En caso contrario, la puerta de enlace se convierte en una dependencia y puede provocar el acoplamiento entre servicios.
- Los servicios deben tener un acoplamiento flexible y una alta cohesión funcional. Las funciones que es probable que cambien juntas se deben **empaquetar** e implementar en conjunto. Si residen en distintos servicios, estos terminan estrechamente acoplados, porque un cambio en un servicio requerirá la actualización del otro servicio. La comunicación demasiado intensa entre dos servicios puede ser un síntoma de un acoplamiento estrecho y una cohesión baja.
- **Aísle los errores**. Utilice estrategias de resistencia para impedir que los errores dentro de un servicio se reproduzcan en cascada. Consulte Patrones de resistencia y Diseño de aplicaciones confiables.

Patrón de arquitectura

publicador / suscriptor

Patrón de arquitectura publicador y suscriptor

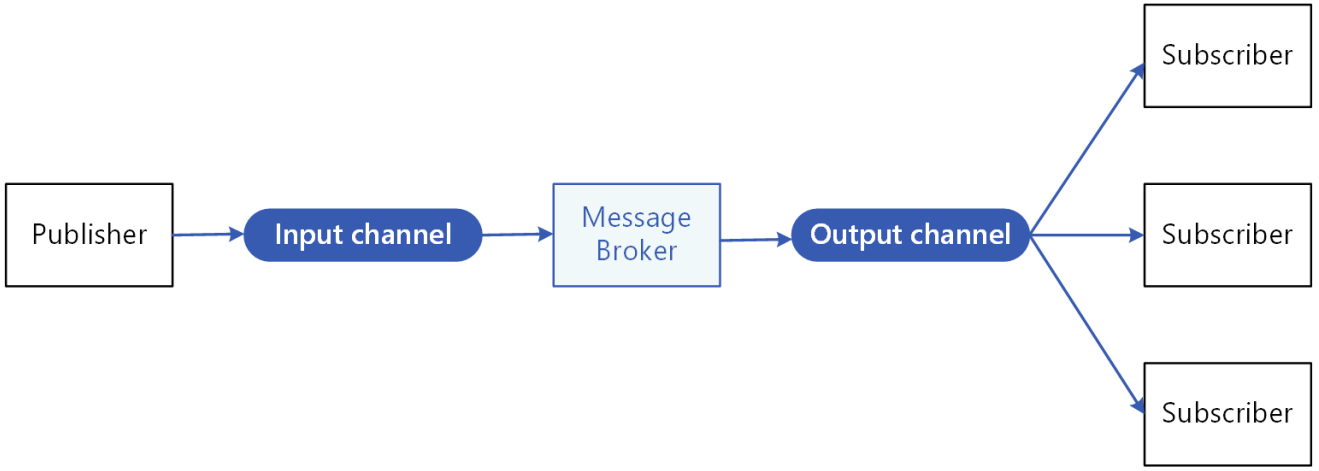
Características de la arquitectura

- Permite que una aplicación anuncie eventos de forma asincrónica a varios consumidores interesados, sin necesidad de emparejar los remitentes con los receptores.
- En las aplicaciones distribuidas y basadas en la nube, los componentes del sistema a menudo necesitan proporcionar información a otros componentes a medida que suceden los eventos.
- La mensajería asincrónica es una forma eficaz de desacoplar a los remitentes de los consumidores y evitar bloquear al remitente para que espere una respuesta. Sin embargo, el uso de una cola de mensajes dedicada para cada consumidor no es efectivo para muchos consumidores. Además, algunos de los consumidores podrían estar interesados solo en un subconjunto de la información. ¿Cómo puede el remitente anunciar eventos a todos los consumidores interesados sin conocer su identidad?

Patrón de arquitectura publicador y suscriptor

Componentes de la arquitectura

- Se introduce un subsistema de mensajería asincrónica que incluya lo siguiente:
- Un canal de mensajería de entrada utilizado por el remitente. El remitente empaqueta los eventos en mensajes, mediante un formato de mensaje conocido, y envía estos mensajes a través del canal de entrada. El remitente en este patrón también se denomina *publicador*.
- Un canal de mensajería de salida por consumidor. Los consumidores se conocen como *suscriptores*.
- Un mecanismo para copiar cada mensaje del canal de entrada a los canales de salida para todos los suscriptores interesados en ese mensaje. Esta operación suele ser manejada por un intermediario, como un agente de mensajes o un bus de eventos.



Patrón de arquitectura publicador y suscriptor

Ventajas

- La mensajería de publicación y suscripción tiene las siguientes ventajas:
- Desacopla los subsistemas que aún necesitan comunicarse. Los subsistemas se pueden administrar de forma independiente y los mensajes se pueden administrar correctamente incluso si uno o más receptores están desconectados.
- Aumenta la escalabilidad y mejora la capacidad de respuesta del remitente. El remitente puede enviar rápidamente un solo mensaje al canal de entrada y luego volver a sus responsabilidades de procesamiento principales. La infraestructura de mensajería es responsable de garantizar que los mensajes se entreguen a los suscriptores interesados.
- Mejora la confiabilidad. La mensajería asincrónica ayuda a que las aplicaciones continúen funcionando sin problemas bajo cargas cada vez mayores y controlen los errores intermitentes con mayor eficacia.
- Permite el procesamiento diferido o programado. Los suscriptores pueden esperar para recoger los mensajes hasta las horas de menor consumo, o los mensajes pueden enrutarse o procesarse de acuerdo a un horario específico.
- Permite una integración más sencilla entre sistemas que utilizan diferentes plataformas, lenguajes de programación o protocolos de comunicación, así como entre sistemas locales y aplicaciones que se ejecutan en la nube.
- Facilita flujos de trabajo asincrónicos en toda la empresa.
- Mejora la capacidad de prueba. Los canales pueden supervisarse y los mensajes se pueden inspeccionar o registrar como parte de una estrategia general de pruebas de integración.
- Proporciona separación de preocupaciones para sus aplicaciones. Cada aplicación puede centrarse en sus funcionalidades principales, mientras que la infraestructura de mensajería controla todo lo necesario para enrutar de forma fiable los mensajes a múltiples consumidores.

Patrón de arquitectura publicador y suscriptor

Problemas y consideraciones

- Tenga en cuenta los puntos siguientes al decidir cómo implementar este patrón:
- **Tecnologías existentes.** Se recomienda encarecidamente utilizar los productos y servicios de mensajería disponibles que admiten un modelo de publicación y suscripción, en lugar de crear uno propio. En Azure, considere el uso de Service Bus o Event Grid. Otras tecnologías que se pueden utilizar para la mensajería de publicación y suscripción incluyen Redis, RabbitMQ y Apache Kafka.
- **Control de suscripciones.** La infraestructura de mensajería debe proporcionar mecanismos que los consumidores puedan utilizar para suscribirse o darse de baja de los canales disponibles.
- **Seguridad.** La conexión a cualquier canal de mensajes debe estar restringida por una directiva de seguridad para evitar que usuarios o aplicaciones no autorizados resulten interceptados.
- **Subconjuntos de mensajes.** Por lo general, los suscriptores solo están interesados en el subconjunto de los mensajes distribuidos por un publicador. Los servicios de mensajería a menudo permiten a los suscriptores reducir el conjunto de mensajes recibidos por:
    - **Temas.** Cada tema tiene un canal de salida dedicado, y cada consumidor puede suscribirse a todos los temas pertinentes.
    - **Filtrado de contenido.** Los mensajes se inspeccionan y se distribuyen según el contenido de cada mensaje. Cada suscriptor puede especificar el contenido que le interesa.
- **Suscriptores con caracteres comodín.** Considere la posibilidad de permitir que los suscriptores se suscriban a varios temas mediante caracteres comodín.
- **Comunicación bidireccional.** Los canales en un sistema de publicación y suscripción se tratan como unidireccionales. Si un suscriptor específico necesita enviar el acuse de recibo o comunicar el estado al editor, considere la posibilidad de utilizar el patrón de solicitud/respuesta. Este patrón utiliza un canal para enviar un mensaje al suscriptor y un canal de respuesta separado para comunicarse con el publicador.

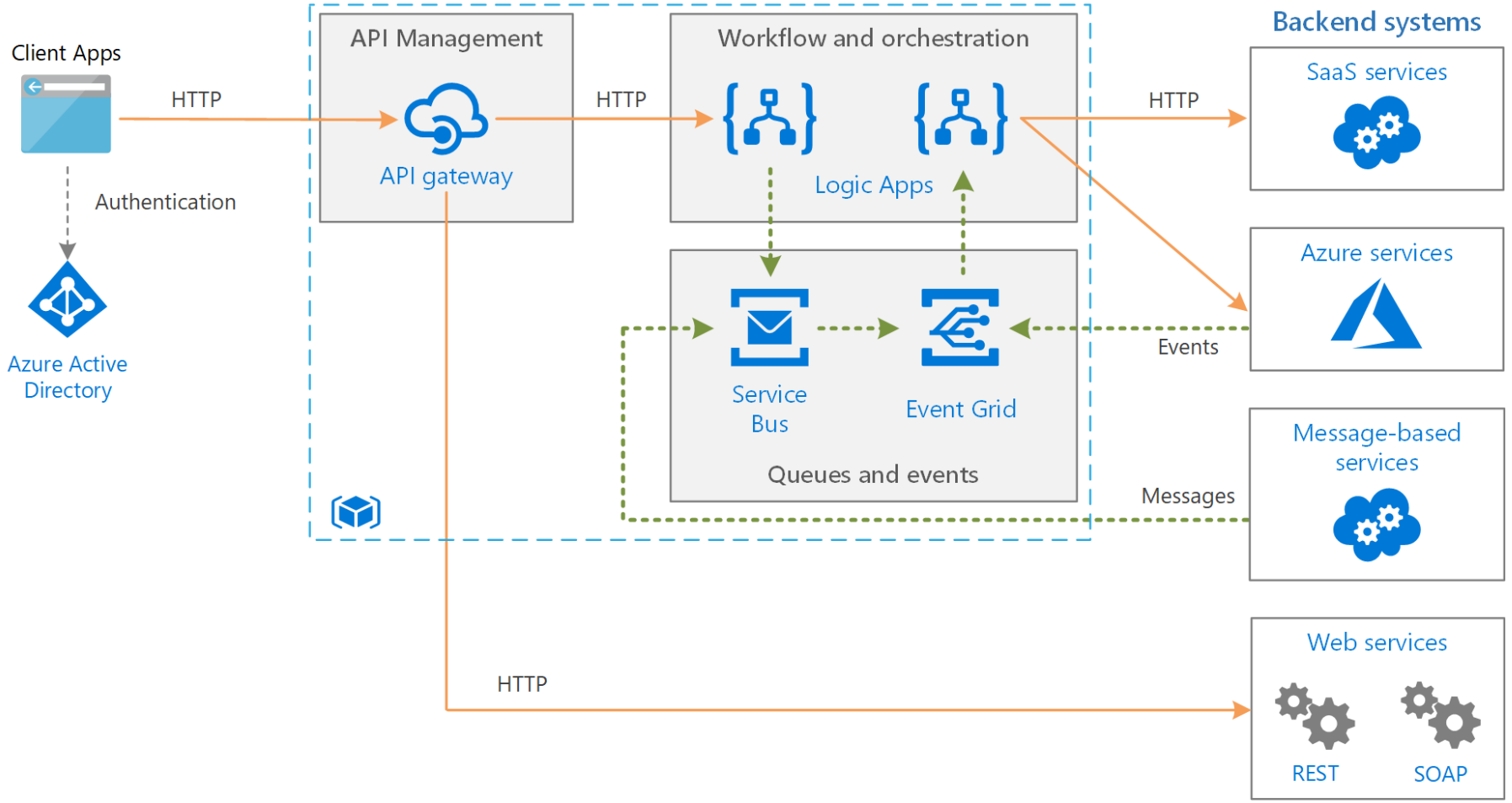
Patrón de arquitectura publicador y suscriptor

Problemas y consideraciones

- **Comunicación bidireccional.** Los canales en un sistema de publicación y suscripción se tratan como unidireccionales. Si un suscriptor específico necesita enviar el acuse de recibo o comunicar el estado al editor, considere la posibilidad de utilizar el patrón de solicitud/respuesta. Este patrón utiliza un canal para enviar un mensaje al suscriptor y un canal de respuesta separado para comunicarse con el publicador.
- **Orden de los mensajes.** El orden en que las instancias de consumidor reciben los mensajes no está garantizado y no refleja necesariamente el orden en que se crearon los mensajes. Diseñe el sistema para asegurarse de que el procesamiento de mensajes sea idempotente para eliminar cualquier dependencia en el orden en el que se administran los mensajes.
- **Prioridad del mensaje.** Algunas soluciones pueden requerir que los mensajes se procesen en un orden específico. El patrón de cola de prioridad proporciona un mecanismo para garantizar que determinados mensajes se entreguen antes que otros.
- **Mensajes repetidos.** El mismo mensaje se puede enviar varias veces. Por ejemplo, puede producirse un error después de que el remitente publique un mensaje. A continuación, una nueva instancia del remitente podría iniciarse y repetir el mensaje. La infraestructura de mensajería debe implementar la detección y eliminación de mensajes duplicados (también conocida como deduplicación) basada en identificadores de mensajes para proporcionar la entrega de mensajes de una sola vez.
- **Expiración de mensajes.** Un mensaje puede tener una duración limitada. Si no se procesa dentro de este período, es posible que ya no sea pertinente y se debe descartar. Un remitente puede especificar un tiempo de expiración como parte de los datos del mensaje. Un receptor puede examinar esta información antes de decidir si desea realizar la lógica de negocios asociada con el mensaje.
- **Programación de mensajes.** Un mensaje puede estar prohibido temporalmente y no debe procesarse hasta una fecha y hora específicas. El mensaje no debe estar disponible para un receptor hasta ese momento.

Patrón de arquitectura publicador y suscriptor

Casos de uso de la arquitectura (ejemplo 1)



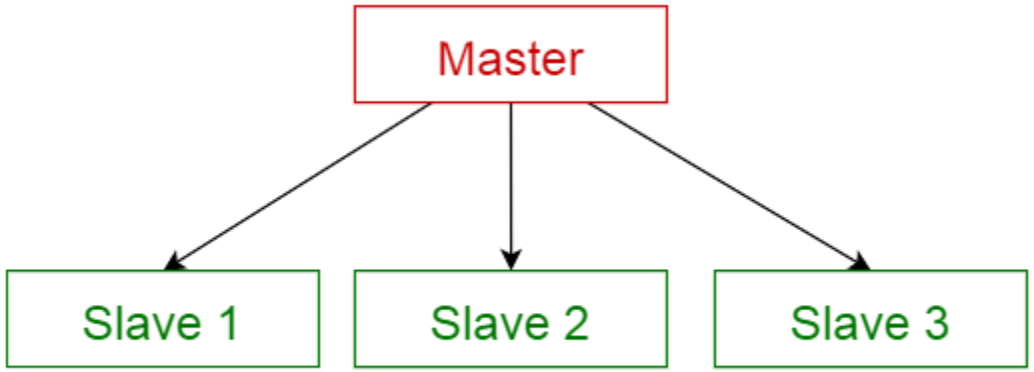
En el diagrama siguiente se muestra una arquitectura de integración empresarial que usa Service Bus para coordinar flujos de trabajo y Event Grid para notificar a los subsistemas los eventos que se producen. Para más información, consulte Integración empresarial en Azure mediante colas de mensajes y eventos.

Otros patrones

Patrón de arquitectura maestro-esclavo

Componentes que conforman la arquitectura

Este patrón consiste en dos partes; **maestro** y **esclavos** . El componente maestro distribuye el trabajo entre componentes esclavos idénticos y calcula el resultado final de los resultados que devuelven los esclavos.



Patrón maestro-esclavo

Uso

•En la replicación de la base de datos, la base de datos maestra se considera como la fuente autorizada y las bases de datos esclavas se sincronizan con ella.

•Periféricos conectados a un bus en un sistema informático (unidades maestra y esclava)

Características de la arquitectura

También llamado maestro- trabajador o patrón de reducción de mapa, es un patrón usado cuando se tienen dos o más procesos que necesitan ejecutarse simultánea y continuamente pero a diferentes velocidades

El patrón de diseño Maestro/Esclavo es muy ventajoso cuando creamos aplicaciones multi-tarea. Le da un enfoque más modular al desarrollo de la aplicación debido a su funcionalidad multi-bucle, pero más importante, le da un mejor control de la gestión de tiempo en su aplicación. Cada bucle paralelo es tratado como una tarea o hilo separado. Un hilo se define como la parte de un programa que se puede ejecutar independientemente de las otras partes. Si tiene una aplicación que no utiliza hilos separados, esa aplicación se interpreta por el sistema como un hilo. Cuando separa su aplicación en múltiples hilos, cada uno comparte el tiempo de procesamiento por igual entre ellos.

Implementación

PASO 1: DIVIDE EL TRABAJO

Es importante el saber como el cómputo se puede dividir en subtareas iguales. Para ello se puede basar la división en el tamaño de la memoria de la tarea o el tiempo de ejecución esperado. A veces, el trabajo se divide según la cantidad de elementos que analizarán las subtareas.

Al definir subtareas,es importante considerar el entorno que las procesa. Es posible definir tareas que son demasiado precisas para algunas arquitecturas de procesador y que requieren una sobrecarga adicional en el nivel maestro para administrar las subtareas.

PASO 2: COMBINA LAS SUBTAREAS

En el Paso 1, se decidió cómo se va a definir la división del trabajo. En este paso, es preciso decidir cómo se combinarán los resultados de las subtareas.

PASO 3: DEFINIR CÓMO EL MAESTRO Y LOS ESCLAVOS COOPERARÁN

Este paso define una interfaz para la división de tareas identificadas en el Paso 1. Las subtareas se pueden pasar a los esclavos como llamadas o parámetros, o se pueden ubicar en un repositorio que contiene asignaciones de tareas para el acceso de los esclavos. Del mismo modo, las respuestas de los esclavos pueden ser en forma de parámetros o llamadas a funciones, o los resultados pueden colocarse en un repositorio.

Con esto se puede decidir cómo manejar los datos que necesitan los esclavos. Los esclavos pueden usar estructuras de datos compartidas, o incluso cada esclavo puede tener sus propias estructuras de datos.

Los factores a considerar al decidir un enfoque son los costos de pasar subtareas a los esclavos, duplicar estructuras de datos y crear estructuras de datos compartidas.

Otra cosa a considerar es definir si los esclavos modifican los datos originales. Si modifican los datos que otros esclavos comparten, necesitan su propia copia de los datos.

Implementación

PASO 4: IMPLEMENTAR LOS COMPONENTES ESCLAVOS

Dentro de este paso se realiza la construcción de los esclavos reales para realizar las subtareas del Paso 1 con las interfaces definidas en el Paso 3.

PASO 5: CREA EL COMPONENTE PRINCIPAL

En general, las tareas se pueden dividir en un número fijo de subtareas. Master-Slave es más aplicable cuando se entrega una tarea completa a los esclavos para su procesamiento. Esto ayuda a aumentar la tolerancia a fallas de una aplicación. Una aplicación es difícil de construir, probar y mantener cuando el maestro realiza parte de una tarea y los esclavos realizan el resto.

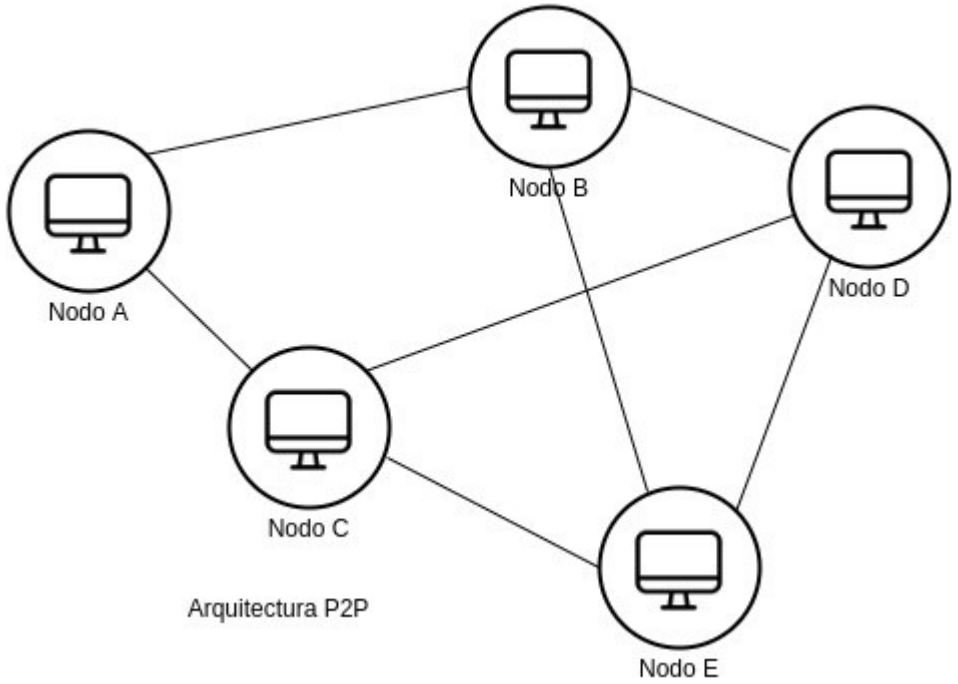
Otra opción es dividir la aplicación general en tantas tareas como sea posible. Esto es especialmente útil al dividir el trabajo en una gran cantidad de procesadores.

El maestro debe tener el código que necesita para iniciar a los esclavos, administrar su procesamiento, recopilar los resultados, matar a los esclavos y combinar los resultados de los muchos esclavos en el producto final. El maestro también debe manejar errores, tales como fallas de esclavos o fallas de hilos, y proporcionar un manejo elegante de los errores.

Patrón de arquitectura P2P

Componentes que conforman la arquitectura

- Una red P2P es una red en la cual las computadoras, también conocidas como nodos, pueden comunicarse entre sí sin la necesidad de un servidor central. La ausencia de un servidor central descarta la posibilidad de un único punto de falla.
- Todas las computadoras en la red tienen los mismos derechos. Un nodo actúa como seeder y leecher al mismo tiempo. Con lo cual, incluso si algunos de las nodos se caen, la red y la comunicación seguirán activas. P2P es la base de la tecnología blockchain.



Características de la Arquitectura

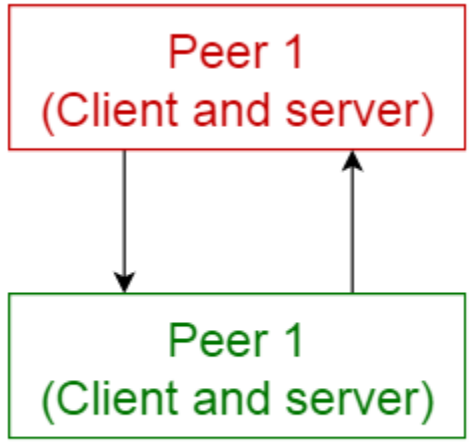
Casos de uso de la arquitectura (ejemplo 1)

Ventajas y desventajas de la arquitectura

Patrón de arquitectura de igual a igual

Componentes que conforman la arquitectura

- En este patrón, los componentes individuales se conocen como **pares** . Los pares pueden funcionar tanto como un **cliente** , solicitando servicios de otros pares, y como un **servidor** , proporcionando servicios a otros pares. Un par puede actuar como un cliente o como un servidor o como ambos, y puede cambiar su rol dinámicamente con el tiempo.



Patrón de igual a igual

Uso

•Redes de intercambio de archivos como **Gnutella** y **G2**)

•Protocolos multimedia como **P2PTV** y **PDTP**

Características de la Arquitectura

Casos de uso de la arquitectura (ejemplo 1)

Ventajas y desventajas de la arquitectura

Bibliografía

Pressman, R. S. (2010). Ingeniería de Software, Un enfoque practico Séptima Edición. Ciudad de México: Mc Graw Hill.

Sommerville. (2011). Ingeniería de Software 9 Edición. Estado de México: Pearson.

UNID Universidad Interamericana para el desarrollo. (2018). Ingeniería de software. Ciudad de México: UNID.