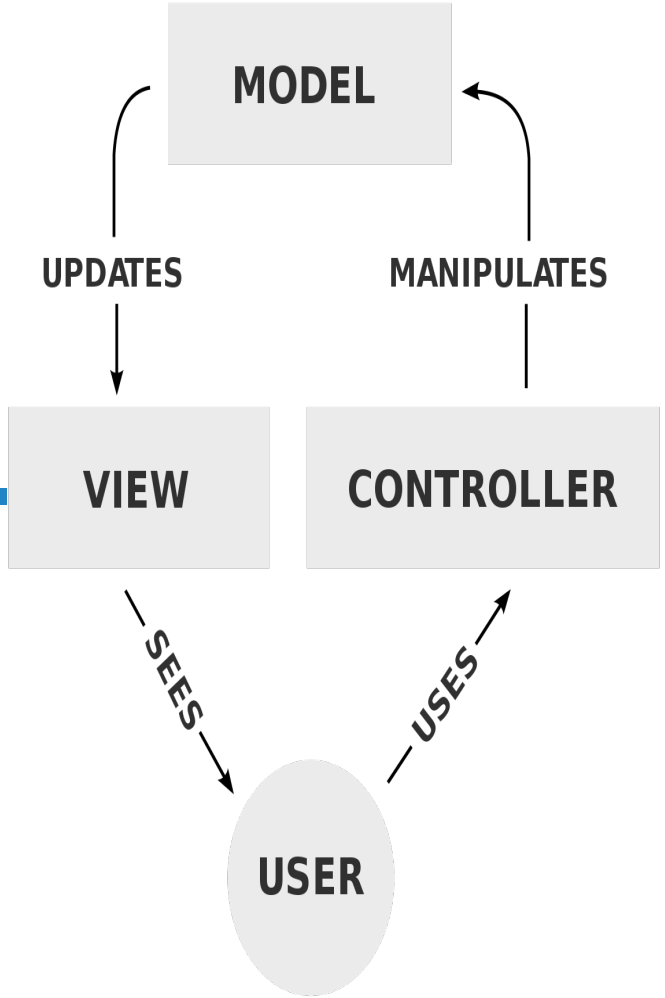
UNIVERSIDAD D CÓRDOBA

PROGRAMACIÓN WEB – BLOQUE II

Fundamentos del desarrollo Web



Dr. José Raúl Romero Salguero
jrromero@uco.es

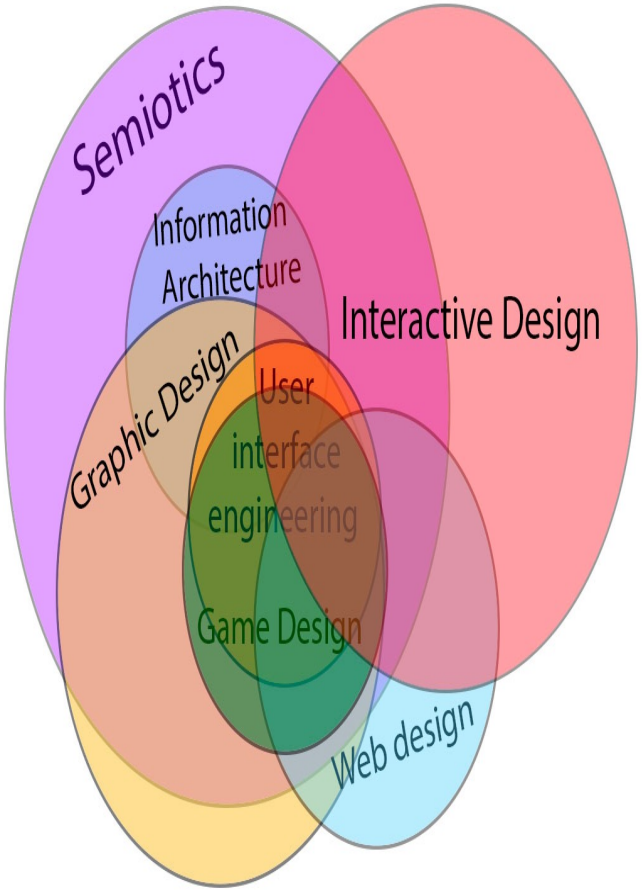
Contenidos del Bloque

1. Marcos tecnológicos
2. Lenguajes para la web
3. Principios de diseño y arquitectura

UNIVERSIDAD D CÓRDOBA

PROGRAMACIÓN WEB – TEMA II-3

Principios de diseño y arquitectura



Dr. José Raúl Romero Salguero
jrromero@uco.es

Contenidos del Tema

1. Principios de diseño
2. Concepto de arquitectura software y arquitectura C/S en capas
3. Modelos basados en Cloud
4. Microservicios

1.

Principios de diseño

”Los dos días más importantes de tu vida son el día en el naces y el día en que averiguas por qué” (M. Twain)

Principios de la W3C (Bert Bos)

- Mantenibilidad. Mantener especificaciones, ficheros o programas de tamaño manejable y con estructura clara. Utilizar las estructuras de los lenguajes que permiten separar elementos, como `@import` en CSS
- Modularidad. Trabajar únicamente en un número de conceptos/objetivos a la vez. Dividir el problema en bloques y delegar en otros módulos
- Mínima redundancia. Mantener al mínimo el solapamiento funcional de distintos módulos, ya que lleva a errores. “Mínima redundancia” ≠ “no redundancia”
- Accesibilidad. En general, referido a la medida de resistencia de la página a dificultades externas o temporales del usuario. También, en datos, se recomienda codificar los datos al mayor nivel de abstracción posible

- Independencia del dispositivo. La especificación no debe depender de ningún (tipo de) dispositivo específico. Se debe prestar atención porque CSS es dependiente en muchos casos, al igual que HTML. También, cuidado con la realización de pruebas y simulaciones (sesgadas)
- Internacionalización. Cuidar los formatos (incluir al menos UTF-8 – ASCII no es el único formato de codificación de texto) y modos de presentación de datos (ej. 7/1/92 puede ser “7 de enero” o “1 de julio”, según país). W3C recomienda el uso del inglés para el código
- Extensibilidad. Todo necesita ser actualizado. Debe planificarse adecuadamente la tecnología y los lenguajes para permitir compatibilidad a largo plazo, evitando degradación. Puede provocar el uso continuado de tecnologías obsoletas (problemas de seguridad, entre otros)
- Facilidad de aprendizaje. La sintaxis es uno de los puntos más importantes, ya que permite escribir el modelo mental del programador

- Legibilidad. Utilizar notaciones legibles, ya que está altamente relacionado con la mantenibilidad y longevidad del programa. Es más importante que otros aspectos, como la eficiencia, y repercute directamente en el coste
- Eficiencia. Según Jakob Nielsen, el usuario es más productivo si su ordenador responde en menos de 1 segundo. La pérdida de velocidad puede ocurrir en el servidor (solicitud HTTP demasiado compleja), en la red (demasiados datos) y en el cliente (visualización muy pesada) – La intuición del programador no suele ser la mejor guía para hacerlo bien
- Formato binario o texto. W3C utiliza mayoritariamente formatos de texto (ej. SVG para imágenes), pero recomienda considerar binarios. Los formatos de texto son más sencillos de desarrollar y depurar, son "auto-descriptivos". Sin embargo, también son propensos a redundancia y extensiones no previstas. Los formatos binarios son más eficientes de procesar y transportar por la red (más pequeños)
- Implementabilidad. Cuanto más sencillo de codificar, más compatible y mantenible será. RECUERDA: la tarea más costosa es la depuración y test

- Simplicidad. Buenas herramientas de soporte (IDEs) pueden esconder malos lenguajes de alguna forma, PERO cuando el modelo mental es demasiado complejo para entenderse, entonces los programas serán difíciles de hacer
- Longevidad. Los documentos web se escriben para la eternidad, lo que implica el control de enlaces rotos. Sin embargo, escribir especificaciones que sobrevivan a las tecnologías es demasiado caro. Se recomienda el uso de estándares y seguir las siguientes reglas:
  - No incluir características que queden obsoletas en las siguientes versiones
  - Utilizar formatos sencillos que puedan decodificarse si se pierde la especificación (formatos de texto, generalmente)
  - Escribir código independiente del dispositivo y, si hay que particularizar, modularizar las reglas específicas para ellos en módulos separados
  - Seguir estos criterios de diseño (W3C)
- Compatibilidad hacia atrás. Dos tipos: (a) compatibilidad con especificaciones anteriores del sitio; (b) compatibilidad con versiones anteriores de la pila de tecnología. Relacionado con extensibilidad y modularidad.

- Interoperabilidad. Una aplicación o documento web escrito conforme a unas especificaciones debería funcionar igual en diferentes ordenadores y sobre distinto software base. Por "igual" se entiende que habrá variaciones pero que la experiencia del usuario será idéntica
- Reformulación. Capacidad de adaptación de alguna porción de datos para un nuevo objetivo.
- Oportunidad. Para que una nueva propuesta funcione, debe venir en el momento adecuado. Si llega mucho antes, se utilizará otras soluciones que será luego difíciles de reemplazar (coste) y será olvidada cuando llegue su momento. Si llega después, el mercado ya estará cubierto
- Utiliza lo que hay. Evitar el síndrome de "si no utilizo la última tecnología, quedo obsoleto". Esta **falacia** puede provocar la recodificación continua de unas pocas funciones, frente a avanzar en el producto. **La tecnología avanza MUY rápido** (por encima de nuestras capacidades) y –aunque ofrezca buenas características– es posible que el coste no lo compense

- Diseño en grupo. Los grupos de trabajo de W3C consisten en 10-20 personas trabajando juntos. Igual filosofía para nuestros sitios/aplicaciones. "2 mejor que 1" para comprobar errores, encontrar soluciones más creativas a problemas, sumar experiencia (¿qué no funcionó en el pasado?)
- Experiencia. Si una especificación es tan larga que necesitamos expertos en sus partes y nadie la puede ver como un todo, entonces la especificación es *demasiado* larga. La especificación debe poder "componentizarse" en grupos de trabajo para obtener un mejor resultado
- Brevedad. La documentación debe ser directa, estructurada, no redundante y focalizada. Evitar explicaciones y razonamientos (escribirlo aparte en notas o documentos específicos). Paradoja: cuanta menos explicación, más fácil de entender
- Estabilidad. El futuro no se puede predecir. Mejor el uso de estándares para mantener la estabilidad deseable en las especificaciones

- Robustez. Las redes fallan, el software tiene fallos, los discos duros (*hardware*) pueden provocar errores:
  - Se debe ofrecer la información adecuada en el momento concreto (ej. excepciones)
  - Si dos partes de un documentos son esenciales para entender el conjunto, entonces deben ir siempre unidas (ej. la red podría romperse tras la solicitud del primero y no enviar el segundo). Se deben evitar datos innecesarios (ej. imágenes muy bonitas pero inútiles) para focalizarse en lo necesario

- Guía de diseño W3C disponible en

  https://www.w3.org/People/Bos/DesignGuide/toc.html

  - Estos criterios se escribieron en 2002 para las propias especificaciones W3C (HTML, CSS, etc.)
  - Se pueden extrapolar a desarrollos web y siguen estando vigentes

Principios de Tim Berners-Lee

- Simplicidad. "*Keep it simple, stupid!*". No confundir con la facilidad de entender el diseño
- Modularidad. Dividir nuestro sistema en un grupo de características cercanas y bien comunicadas
- Ser parte de un diseño modular. Debemos considerar nuestro sistema como parte de otro sistema mayor, ofreciendo las interfaces apropiadas (más complejo que modularidad)
- Tolerancia. "*Be liberal in what you require but conservative in what you do*". No implica llegar a romper más allá de lo necesario (ej. dejar de utilizar estándares)

Fuente: https://www.w3.org/DesignIssues/Principles.html

- Descentralización. Se está diseñando un sistema distribuido para la sociedad.
  - Cualquier punto común que esté involucrado en alguna otra operación tenderá a limitar la forma en que escala el sistema, y producirá un punto único de error total
- Test de invención. "*If someone else had already invented your system, would theirs work with yours?*" Relacionado con la modularidad de dentro hacia afuera
- Principio de la potencia mínima. La elección del lenguaje es un criterio de diseño. El extremo de menor potencia suele ser sencillo de diseñar, implementar y utilizar. El externo de mayor potencia suele ser atractivo y permitir hacer cualquier cosa, sólo limitado por la imaginación del programador. Al principio, los lenguajes era muy potentes. Hoy en día, se recomienda utilizar el lenguaje menos potente posible. Cuanto menos potente el lenguaje, más se podrá hacer con los datos: "*Elegí HTML para que no fuera un lenguaje de programación porque quería que diferentes programas hicieran diferentes cosas con él*" (T. Berners-Lee)

Contenidos del Tema

1. Principios de diseño
2. Concepto de arquitectura software y arquitectura C/S en capas
3. Modelos basados en Cloud
4. Microservicios

2.a

Concepto de arquitectura software

Arquitectura software

- En un sentido amplio, la arquitectura del software es el diseño de más alto nivel de la estructura de un sistema, programa y aplicación.
- Los objetivos de la arquitectura del software son:
  - Identificar los módulos principales del sistema, sin considerar aspectos de implementación.
  - Identificar la funcionalidad y responsabilidades de cada módulo.
  - Definir las interacciones posibles entre los módulos haciendo uso de mecanismos de control y flujos de datos, secuenciación de información, protocolos de interacción y comunicación, etc.

- ¿Por qué es importante la arquitectura del software?
  - Su descripción hace más sencilla la comprensión de sistemas complejos, ya que hacen explícitas las decisiones de diseño de alto nivel.
  - Su representación facilita la comunicación entre las partes interesadas en el desarrollo del sistema.
  - Las descripciones pueden ser reutilizadas a gran escala. No sólo reutilizamos librerías:

¡un buen diseño arquitectónico potencia la reutilización de grandes componentes software!

- El proceso de creación de la arquitectura software se fundamenta y destaca la toma de decisiones tempranas
  - Profundo impacto en el proceso de ingeniería posterior, desde el diseño e implementación hasta el despliegue y mantenimiento

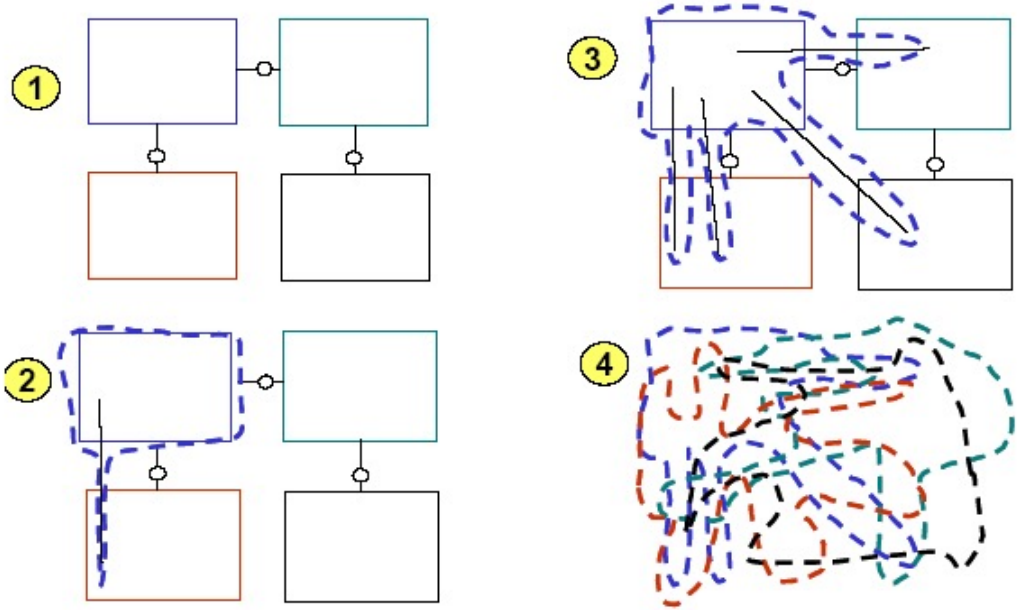


Figure 1: Illustrating how architectures are undone. Each decision to by-pass interfaces creates coupling. The system quickly devolves into a tangled mass of code.

Fuente: Architecture as a Business Competency. Bredemeyer Consulting

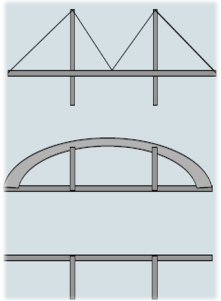
Arquitectura del Software

“la arquitectura del software involucra la descripción de los elementos a partir de los cuales se construyen los sistemas, las interacciones entre dichos elementos, patrones que guían en su composición y restricciones sobre esos patrones”

Shaw & Garlan, 1996

Estilo arquitectónico

- En el diseño arquitectónico, se observan ciertas regularidades en la estructura, estilo y elementos utilizados dando respuesta a demandas similares
- Estas regularidades recurrentes –tomando nomenclatura de arquitectura civil– se denominan estilos arquitectónicos
- Los estilos arquitectónicos representan formas diferentes de estructurar un sistema usando componentes y conectores, siguiendo decisiones esenciales sobre los elementos arquitectónicos y estableciendo restricciones importantes sobre tales elementos y sus posibles relaciones
- Ejemplos: modelo C/S, *peer-to-peer*, arquitectura en capas…



2.b
Arquitectura C/S en capas

Modelo Cliente/Servidor

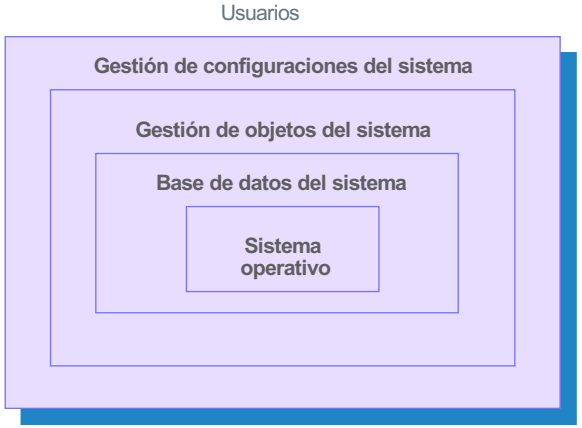
- En el modelo cliente-servidor (C/S), el sistema se organiza como un conjunto de servicios y servidores asociados, más unos clientes que acceden y utilizan los servicios
- Los principales componentes del modelo C/S son:
  - Conjunto de servidores que ofrecen servicios a otros subsistemas: servidores de impresión, servidores de archivos, servidores de bases de datos, …
  - Conjunto de clientes que llaman a los servicios ofrecidos por los servidores:
    - Invocan los servicios ofrecidos por los servidores mediante un protocolo de petición-respuesta (p.ej., HTTP en la WWW)
    - Pueden existir varias instancias de un cliente ejecutándose concurrentemente
    - Tienen que conocer los nombres de los servidores disponibles y los servicios que suministran, pero los servidores no conocen a los clientes
    - Básicamente, un cliente realiza una petición y espera hasta recibir la respuesta

Modelo Cliente/Servidor__ Ventajas e inconvenientes

- Este modelo ofrece notables ventajas:
  - Flexibilidad, pudiéndose ejecutar sobre distintas plataformas hardware y software
  - Fácil integración de elementos
  - Fácil mantenimiento local
- Debemos considerar los inconvenientes que plantea:
  - Difícil mantenimiento a nivel global
  - Los servidores son potenciales cuellos de botella
  - Dificulta el manejo de errores y consistencia de datos si se replican (*mirroring*)
  - Difícil distribución de datos
  - Baja confidencialidad y eficiencia. Empeora si hay duplicidad de datos

Arquitectura en capas

- La arquitectura en capas (también denominada arquitectura estratificada) modela la interacción entre los subsistemas organizando un sistema en una serie de capas (*layers*)
- Cada capa presta servicios a la capa inmediatamente superior y actúa como cliente de la inferior
- Los conectores se definen mediante los protocolos que determinan las formas de la interacción entre cada par de capas
- Este estilo soporta el desarrollo incremental de sistemas:
  - Cada capa desarrollada queda disponible para los usuarios o para incorporar nuevas capas superiores

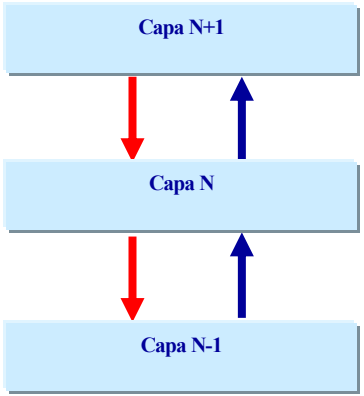


Modelo de capas de un sistema de gestión de versiones.
Fuente: [*Sommerville, 2005: Figura 11.4*]

Arquitectura en capas__ Ventajas e inconvenientes

☺ La arquitectura estratificada presenta las siguientes ventajas:

- La arquitectura es cambiable y portable
- Si la interfaz de una capa se mantiene, la capa es reemplazable por otra
- Si la interfaz se cambia, sólo se afectará la capa adyacente.
- Para implementar el sistema en otras computadoras sólo es necesario recodificar las capas internas (más dependientes de la máquina)



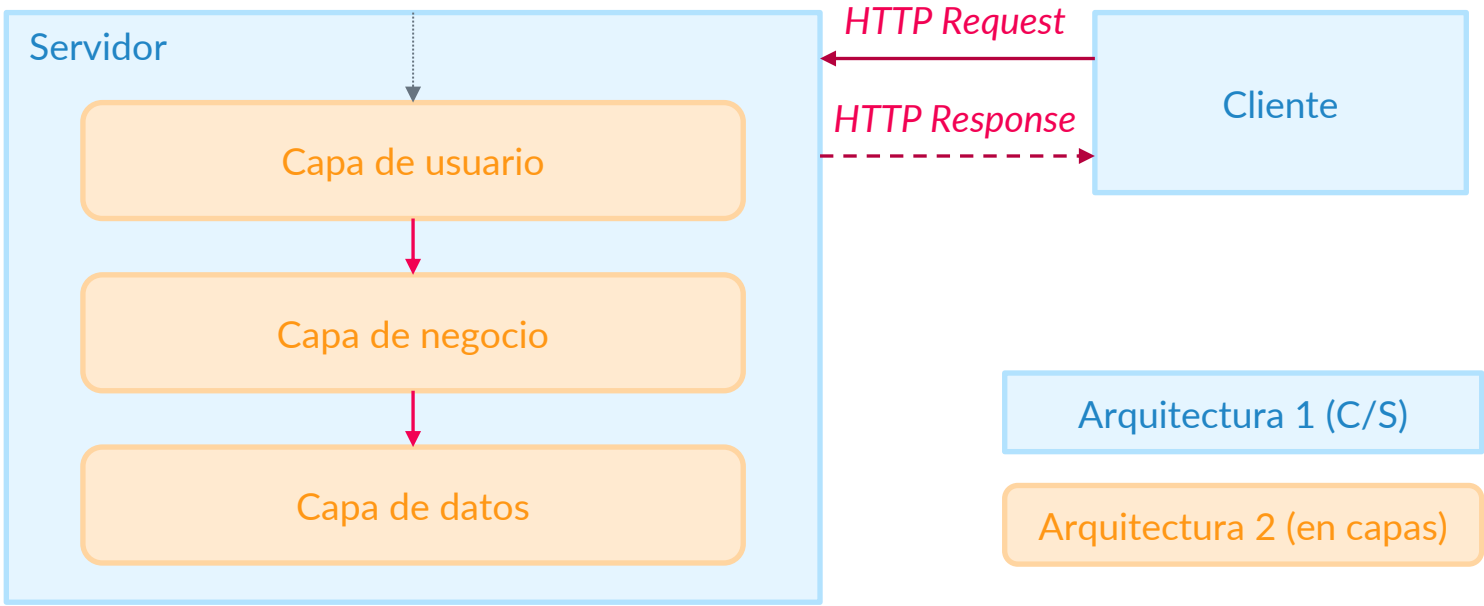
☹ Pero también presenta inconvenientes:

- Es difícil estructurar los sistemas en capas (p.ej., el usuario puede requerir el acceso a capas internas, como la base de datos, lo que pervierte el modelo)
- El rendimiento puede resultar afectado por los múltiples niveles de interpretación de órdenes y protocolos que se requieren

Arquitectura C/S en capas

- En desarrollo web, el estilo C/S en capas es el más utilizado
- Se trata de un estilo heterogéneo que mezcla los elementos y reglas de dos o más estilos arquitectónicos puros:
  - NO es un estilo arquitectónico por sí mismo
- El C/S en capas se trata de un sistema jerárquicamente heterogéneo, en el que la arquitectura interna de los componentes de un sistema conforme (servidor) a un cierto estilo (en C/S) sigue un estilo arquitectónico distinto (en capas)

Arquitectura C/S en capas



Modelo básico C/S en 3 capas

Contenidos del Tema

1. Principios de diseño
2. Concepto de arquitectura software y arquitectura C/S en capas
3. Modelos basados en Cloud
4. Microservicios

3.
Modelos basados en Cloud

Cloud computing o por qué delegar en terceros

- *Cloud Computing* (computación en la nube) es un paradigma de computación que permite ofrecer servicios a través de servidores remotos (tanto recursos computacionales, como aplicaciones y centros de datos)
- Características: modelo de pago por uso; autoservicio; recursos a demanda
- Ventajas para las empresas (delegan el control y configuración):
  - Agilidad y velocidad
  - Reducción de costes
  - Eficiencia de los recursos
  - Escalabilidad
  - Alta disponibilidad
  - Movilidad
  - Tecnología permanentemente actualizada por terceros
  - Mayor capacidad de almacenamiento
  - Seguridad

SaaS__ *Software as a Service*

- Acceso al software basado en *cloud* de un proveedor externo a la organización
- El cliente no instala aplicaciones en local, sino que residen en una red remota accediendo a través de la web o una API
- Los clientes pueden almacenar y analizar datos en remoto, además de colaborar en proyectos

SaaS__ *Software as a Service*

- Funciones principales:
  - Proporciona software y aplicaciones al cliente mediante un modelo de suscripción
  - El cliente no gestiona, instala ni actualiza el software
  - Los datos están seguros en *cloud* (ej. un fallo en el equipo no provoca la pérdida de datos)
  - Puede escalarse el uso de los recursos en función de las necesidades de servicio
  - Aplicaciones accesibles desde casi cualquier dispositivo conectado a Internet
- Ejemplo: Dropbox, Google Drive

PaaS__ *Plarform as a Service*

- Proporciona un entorno *cloud* con herramientas prediseñadas para desarrollar, gestionar, personalizar y distribuir aplicaciones del cliente
- Apropiado para empresas de desarrollo que quieren implementar metodologías Agile
- Plataforma apropiada para la economía de APIs

PaaS__ *Plarform as a Service*

- Funciones principales:
  - Ofrece un entorno unificado y configurado remotamente
  - Permite centrarse en el desarrollo, sin preocuparse por la infraestructura subyacente
  - El proveedor gestiona la seguridad, los sistemas operativos, el software de servidor y las copias de seguridad
  - Facilita la colaboración entre equipos de trabajo en remoto
- Ejemplo: *Google App Engine* (plataforma para el desarrollo y despliegue de aplicaciones en Java, Python, PHP, C#, .Net, Go o Node.js)

IaaS__ *Infrastructure as a Service*

- Un proveedor proporciona acceso a recursos computacionales (servidores, almacenamiento y redes – centro de datos), permitiendo al cliente utilizar sus propias plataformas y aplicaciones
- Funciones principales:
  - El cliente no adquiere el hardware directamente, paga por *IaaS on demand*
  - La infraestructura es escalable, en función de las necesidades de almacenamiento y procesamiento.
  - Ahorra en coste por compra y mantenimiento de hardware propio
  - Si los datos están en el *cloud*, no existe ningún punto único de anomalía
  - Tareas administrativas virtualizadas, liberando tiempo para otros trabajo

IaaS__ *Infrastructure as a Service*

- Ejemplo: Amazon Web Services (servicio de máquinas virtuales en la nube, gestionadas por el desarrollador remoto, con almacenamiento y hardware transparente 100%)

Contenidos del Tema

1. Principios de diseño
2. Concepto de arquitectura software y arquitectura C/S en capas
3. Modelos basados en Cloud
4. Microservicios

4.
Microservicios

Definición de servicio web

- Un **servicio web** (propio de SOA – *Service Oriented Architecture*) es una función software máquina–máquina interoperable, que se aloja en una localización de red direccionable
- Un servicio web está orientado al negocio, permitiendo intercambio de información basada en XML
- Un servicio web ofrece una interfaz, que oculta los detalles de implementación
- Los servicios web son independientes del hardware o plataforma, y del lenguaje de programación
- Los servicios web son apropiados para desarrollo de aplicaciones débilmente acopladas, orientadas a componentes y con implementaciones en múltiples tecnologías
- Los servicios web pueden utilizarse unitariamente o en conjuntos de transacciones de negocio más complejos

Definición de microservicio

- Estilo arquitectónico en el que las aplicaciones complejas son descompuestas en uno o más servicios.
  - A cada servicio se le denomina microservicio
  - Puede desplegarse independientemente del resto
  - Son débilmente acoplados
  - Se focalizan en una única tarea de forma eficaz
  - Son independientes del lenguaje de programación
  - Se comunican mediante APIs (generalmente, API REST)
  - Requieren un contexto muy limitado

Un microservicio es una unidad de descomposición funcional de un sistema en un servicio gestionable e independientemente desplegable

Microservicio - Características

- Pequeños y focalizados
  - Regla de las 2 pizzas
  - Tratados como una aplicación o producto independiente
  - La granularidad depende de las necesidades del negocio
- Despliegue ”zero coordination”
- Un MS no conoce la implementación de base ni la de otros MS vecinos (ni siquiera de su existencia)
- Se pueden ejecutar múltiples copias de un MS en distintas máquinas
- NO es el estilo adecuado para comenzar una aplicación
  - Solución a problemas de escalabilidad, no para nuevos diseños
- El despliegue debe externalizarse

Microservicios – Diferencias con modelo basado en servicios (SOA)

- Múltiples similitudes:
  - Mismo modelo arquitectónico (basado en servicios)
  - Mismo sistema de catálogo y registro
  - Mismo uso de protocolos
- Diferencias en notables en puntos relevantes:
  - Ámbito:
    - **SOA**. Los servicios se ponen a disposición de aquel que necesite usarlos [*un mismo servicio forma parte de múltiples sistemas*]
    - **MS**. Muy restringido y orientado al objetivo, por lo que forma parte de un único sistema distribuido

Microservicios – Diferencias con modelo basado en servicios (SOA)

- Existencia:
  - **SOA**. Se descubren en tiempo de ejecución (→ descubrimiento)
  - **MS**. Su existencia es implícita (no existe descubrimiento)
- Motivación de negocio y ROI:
  - **SOA**. Transformación de negocio (mayor inversión y planificación)
  - **MS**. Realización rápida de funcionalidades locales
- Objetivo, se resume en una palabra:
  - **SOA**. Reutilización
  - **MS**. Escalabilidad

Microservicio - Ventajas

- Cada microservicio (MS) es pequeño:
  - + fácil de desarrollar
  - + fácil ser productivo con IDEs
  - + eficiente el despliegue (inicio más rápido)
- Cada MS se despliega independiente del resto → + fácil actualizar versiones
- Sencillo organizar equipos de proyecto para diseño, desarrollo y despliegue (uso de metodologías ágiles)
- Sencillo aislar errores y comportamientos anómalos (testabilidad)
- Elimina compromiso con tecnología (tech stack) a largo plazo

Microservicio - Inconvenientes

- Mayor consumo de memoria → si se reemplazan N instancias de una aplicación monolítica por MxN instancias de servicios, donde cada M ejecuta su propia VM (docker, JVM, etc.) → sobrecarga M veces
  - Si se utiliza modelo en la nube, la sobrecarga es incluso mayor (y costosa)

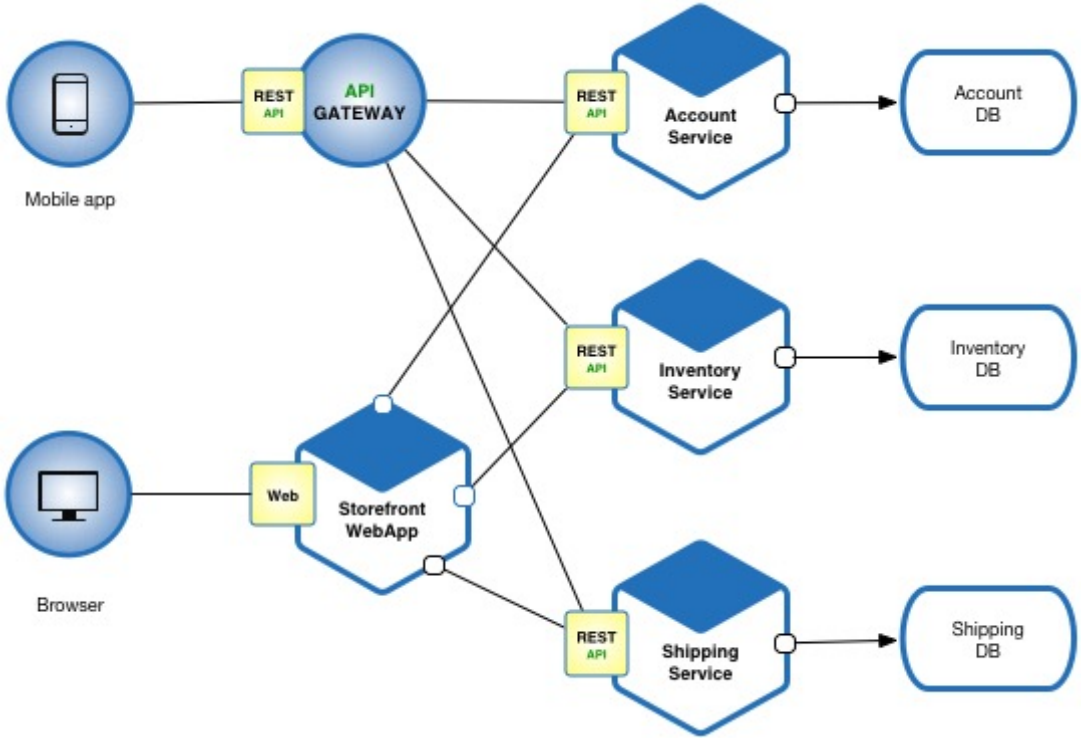
Microservicio - Inconvenientes

- Incurre en problemas asociados al desarrollo de sistemas distribuidos:
  - A nivel de sistema, el testing es complejo
  - No hay IDEs suficientemente preparados para desarrollo/testing a nivel de sistema
  - Requiere desarrollar mecanismos de comunicaciones entre servicios
  - Diseño e implementación de casos de uso complejos requieren transacciones distribuidas y equipos diferentes
- Despliegue y mantenimiento de aplicación costoso en producción

Microservicios - Ejemplo

S. Daya *et al*., “Microservices from theory to practice. Creating applications in IBM Bluemix using the microservices approach”. IBM Redbooks, agosto 2015.

C. Richardson, “Microservices patterns - With examples in Java”. Manning, octubre 2018. ISBN 9781617294549



https://microservices.io/patterns/microservices.html (Fuente)

Programación Web

Fundamentos del desarrollo web__ **Curso 2021/22**