Cualidades de sw + Historia WEB:

El mejor atributo a maximizar es la plata (we live in a society).

Hay parejas de cualidades que se llevan mal (seguridad y usabilidad, portabilidad y mantenibilidad)

Hay otras que se llevan bien (robustez y testeabilidad)

tolerancia a fallo -> si me puedo recuperar me recupero, sino me da una forma feliz de fallar

(se me rompe el login, y no es que todos pueden entrar con cualquier usuario)

robustez -> que ante problemas sobre el sistema, este sigue funcionando bien.

Ambas estan relacionadas.

La web se crea entre universidades para pasarse papers.

El protocolo web es HTTP

hypertext -> enlaces, combino un verbo + un recurso (por ej GET apuntesweb/)

El puerto es algo que esta levantado del otro lado que sabe responder cuando yo le pego (ese algo se llama web server)

Del lado del cliente hay un browser.

yo pongo www.tacs-utn.com:80/apuntes/web -> Se envia una solicitud al ISP

-> hay una traduccion de direcciones (del nombre del dominio o host a la ip a traves del DNS)

-> El recurso escupido por el puerto al cliente a traves del socket se manda en formato HTML (gestionar texto, imagenes y enlaces),

-> El siguiente paso fue tener precargado, levanto y cargo una vez y libero cuando apago el webserver (PHP, ASP, JAVA, .NET).

-> Tener estado conversacional o sesion (poder saber si el que me pide ya esta logueado y recordar que se logueo)

-> En HTTP, el cliente siempre arranca la conexion el server solo puede contestar.

Ahi nace AJAX (javascript y xml asincrono), se pone un timer por atras y cada tanto se le pega al server para ver si hay algo nuevo

(por ej si quiero ver una notificacion nueva sin que el usuario haga ninguna accion).

JS corre del lado del cliente y sirve para hacer validaciones antes de mandar las requests del lado del cliente.

cookies: Pedazito de texto que guarda en el navegador del key y

entonces lo puede devolver al server y asi saber quien es y que esta logueado.

sesion: Parecido a las cookies pero del lado del servidor. Para cada usuario tengo un mapa<string, object/string>

por ej mapa<seba, logueado=true>.

Usamos 1 cookie cuando se loguea, con un token super aleatorio dificil de adivinar y con esa cookie

la manda al servidor por cada request entonces puede saber quien es y acceder a la sesion.

HTTP viaja plano (no hay encriptacion ni nada) -> HTTPS para encriptarlo.

cgi (common gateway interface) fue el primero que permitio mandar parametros al pedir recursos y ejecutar cosas

Contenido estatico -> recurso que no cambia (paper)

Contenido dinamico -> lo puedo ejecutar y hacer cosas (.exe)

La estructura de los recursos es como un arbol.

HTML es declarativo (que hay que hacer no como).

verbo protocolo://host:port/recurso

el puerto es algo que esta levantado del otro lado que sabe responder cuando yo le pego (ese algo se llama web server)

pasos para correr un programa:

- cargarlo en memoria

- ejecutar

- libero recursos

APIs:

Sockets: ???

RPC (llamada a procedimiento remoto): Es una comunicación entre procesos que permite llamar a una función en otro proceso que reside en una máquina local o remota. Todo está definido en una interfaz, con sus respuestas, etc.

RMI (invocación de método remoto): Es un API, que implementa RPC en Java con el apoyo de los paradigmas orientados a objetos.

RPC/RMI:

- Alto acoplamiento

- Llamadas sincrónicas

- Errores como excepciones

- múltiples implementaciones

- Se tienen que poner de acuerdo entre los 2 que se comunican

Serialización: Consiste en un proceso de codificación de un objeto en un medio de almacenamiento con el fin de transmitirlo a través de una conexión en red. Hay mismatch entre lenguajes (uno puede tener un tipo NULL y el otro no por ej.).

Maneras de serializar -> XML, JSON, YAML.

XML/XSD: machine/human reeditable, se usa en SOAP, soporta comentarios.

JSON: human readable, se usa en JS y APIs, esquema de autovalidación.

YAML: human readable, se usa en swagger, se pueden tener referencias, tipos de datos definidos custom, sin esquema de validación.

Binary encoding: Para no mandar todo el choclo de datos de manera plana.

protocol buffers por ej.:

- Rápida serialización/deserialización (es más ligero pasar datos por el caño).

- No importa la plataforma (x86, x64 da igual).

- Serializado binario => not human readable.

- Tiene ciertos tipos de datos definidos que hay que cumplir al momento de comunicarse.

- Me puedo comunicar entre distintos lenguajes, encodeado de manera binaria.

Transporte/Mensajería

SOAP (Simple Object Access Protocol) es un protocolo estándar que define cómo dos diferentes procesos pueden comunicarse por medio de intercambio de datos XML:

- extensible (le puedo agregar seguridad por ej.).

- neutral (puede funcionar sobre http, smtp, etc.).

- independiente del paradigma.

- funciona exclusivamente sobre XML.

JSON es más ligero para pasar por el caño que XML porque tengo que leer más cosas en XML (hay etiquetas que abren y cierran).

SOAP tiene WSDL (Web Services Description Language) que te dice cómo funciona una API (que mensajes podés enviar y que vas a recibir). Es como un contrato escrito en piedra.

REST (representation state transfer): Es un estilo de arquitectura para comunicarte entre objetos (principal competidor de SOAP)

- Arq cliente - servidor

- stateless: el servidor no tiene que mantener estado con respecto al cliente

(por ej no tengo que mantener estado de si el cliente está logueado o no).

Por lo que no podria mantener una sesion de usuario por ej.

Esto es asi por escalablidad (¿si tengo 2 servidores en cual guardo la sesion? deberia pegarle siempre al mismo y se podria caer, etc)

- Se tiene que intentar que los recursos sean altamente cacheables

- interfaz uniforme

- sistema de capas (separo responsabilidades)

- puede tener codigo bajo demanda, o sea el servidor le puede mandar

al cliente codigo para ejecutar (super cuestionable, no se deberia usar)

REST tiene interfaces mediante niveles de madurez (richardson maturity model):

- Nivel 0 Swamp of POX: hiciste todo mal capo.

- Nivel 1 Recursos: Uso de sustantivos para definir recursos.

- Nivel 2 Verbos: GET, POST, etc. y códigos de respuesta (200, 400, etc.)

- Nivel 3 HATEOAS (opcional): El cliente interacciona con una aplicación cuyos servidores de aplicación proporcionan información de la API, dinámicamente a través de hipermedia.

REST funciona sobre HTTP.

REST suele funcionar con JSON, pero no necesariamente.

SOAP mantiene estado, REST no.

Verbos HTTP pueden ser:

- Seguro: sin efecto de lado (read only)

- Idempotente: el server queda en el mismo estado si hacemos **un mismo** request n veces.

Metodo	Seguro	Idempotente
GET	SI	SI
OPTIONS	SI	SI
PUT	NO	SI
PATCH	NO	NO
DELETE	NO	SI
POST	NO	NO

PUT es idempotente porque si yo hago por ej PUT usuarios/usuario1 con la misma request todas las veces, no va a cambiar el estado del recurso **entre** los PUTs. Lo mismo si hago DELETE varias veces.

patch no es impotente, porque se puede usar para agregar campos.

HATEOAS: es una propiedad de rest que indica que el cliente pueda moverse por la aplicación web

únicamente siguiendo a los identificadores únicos URI en formato hipermedia.

Caching (El browser hace automáticamente todo esto):

- Cach-control: header que está en el response del servidor (no--store, private, public, etc).
- If-modified-since: si se modificó devuelve el response, sino un código de que no se modificó.
- ETag: Un código generado a través del response que puedo usar como hash
- If-none-match: Mando el Etag y cuando me vuelven a hacer otra request con if-not-match con este ETag, si el servidor tiene ese mismo hash me devuelve que no se modifico

← Si el etag cambio (se cambió el recurso) no deja modificar sino si (como si fuera una transacción).

Un problema de REST es combinación de recursos (quiero artista = ACDC, año=1998, Genero=Rock)

Se podría solucionar mediante querys params

Si mando songs?orderby=year&artista=dec y songs? artista=dec &orderby=year son dos cosas distintas y por tanto se lleva mal con la cache.

Otro problema de REST es cómo manejar el versionado de los recursos.

Una forma de arreglar es en la url poniendo número de versionado.

Mas problemas:

- Cache aware
- Documentacion obligatoria

FIX N+1: Ej: Si hago una request donde pido usuarios y para cada uno que pida tambien la imagen de perfil, hace directamente esa consulta extra (1 = consulta extra) para los N usuarios.

Under fetching: No tengo suficiente información con la primer request, estoy forzado a hacer una segunda a otro endpoint.

SOAP es común en APIs fuertementes transaccionales (bancos, aerolíneas, etc) porque mantiene estado.

GraphQL:

- Query and mutation language (put, patch, etc)
- Tipado
- Validaciones
- Paginacion
- Autenticacion
- Soluciona Fix over/under fetching
- Soluciona Fix N+1
- Directivas (@include(if:boolean))
- No soluciona problemas de caching, por lo que no asegura buena performance.
- Puedo tener queries en cache, donde dejo un placeholder para que el cliente solo me mande el valor a cambiar en la query (y el hash para que sepa el servidor que se trata de esta query) y no necesito mandar la query entera desde el cliente.

gRPC:

- Version mejorada de RPC
- Load balancer automatico que es el cliente (el elige a quien pegarle si tarda mucho un server)
- Manejo de errores
- Si se interrumpe un request en el medio yo me entero (cascading cancell cancellation)
- Full duplex (no hay cliente-servidor, va para los 2 lados).

API:

- La API es una necesidad interna antes que una externa.
- Debe ser simple y única.
- La API no es solo una interfaz para programas (lo usa la gente por lo que hay que reducir la friccion de su uso).

Arquitectura WEB / Frontend:

Al frontend se le da bola por usabilidad -> $ (¿Qué tal otra broma Murray?)

DOM: Representación interna del browser al leer el HTML.

CSSOM: Representación del CSS

Render tree: junta DOM + CSSOM

JS no solo agrega comportamiento a la página, sino que también permite modificar el DOM.

JS tiende al caos porque es muy flexible.

Cada browser muestra lo que quiere mediante el DOM, entonces hay que probar mi app en todos los browsers en los que quiera que corra mi APP. Hay herramientas que hacen esto (Google analytics por ej.). Hay distintas organizaciones que se encargan de definir estándares para que no tengamos que hacer una página para cada navegador (ECMA por ej. para JS).

Render(): Borro todo de la página y lo vuelvo a generar.

Bounce rate: cantidad de usuarios que se van de mi página sin hacer ninguna acción. A medida que aumenta con el tiempo, pierdo más usuarios. Está relacionado con el tiempo de carga de mi página.

El browser hace lo siguiente:

Procesa HTML y CSS en paralelo → Render tree → Layout → Paint

Si quiero que mi página ande rápido, necesito que el HTML y el CSS se contesten rápido. Si además hago un buen código en JS, el usuario lo va a usar rápido.

CDN: cache para alojar contenido en servers más cercanos a los usuarios finales que el server original.

Lighthouse: Herramienta incluida en las tools de Chrome que nos da un reporte sobre distintos aspectos de un sitio en particular (te dice que esta mal de mi pagina en cuanto a performance, y como mejorarla).

La idea de los frameworks de JS es mejorar la UX haciendo transiciones entre pantallas, evitando volver a cargar otro doc.

ReactJS genera una estructura interna, Virtual DOM, y cada vez que hago un cambio en las propiedades o en el estado de mi clase, se genera uno nuevo, se hace un diff entre estos dos y se dispara solo las actualizaciones al DOM real (asi evitas regenerar el DOM nuevamente con un Render() y actualizar toda la página).

Una facilidad que ofrecen los navegadores es la capacidad de navegar hacia atrás y hacia adelante en la historia. Esto es algo que no se puede manejar directamente desde JS (también por ej cuando cambio de pagina y tengo que cambiar la url en la que estoy). Para esto hay librerías de JS que usan la API de *history*.

Hay que dar soporte a herramientas diseñadas para interpretar solo el contenido de las paginas (el motor de indexing de Google por ej). Para esto SSR (server side rendering), que es un server en el medio que interpreta el JS y devuelve como resultado un HTML tradicional, y además sirve la aplicación para que ante la siguiente request mi app se ejecute del lado del cliente.

SPA (single page application): Tener la página precargada y todo actúa como si fuera una sola página.

Ventajas:

- Transiciones fluidas
- Luce como una app
- Requests dentro de la app son mas livianas
- En el escenario correcto y bien aplicado mejora la UX

- Es útil cuando tenes páginas con formas parecidas, y necesitas cambios de estado de objetos con un ruteo artificial (lo manejas a través del routing).

Desventajas:

- Manejo de routing
- Alta complejidad del lado del cliente
- La request inicial carga gran parte de la app

Un ej. en el que no necesitaría una SPA es un diario, ya que el contenido es estatico (refrescas la pagina y el contenido es el mismo).

polling: preguntandole al servidor cada tanto tiempo si tiene algo.

long polling: dejo la conexion abierta despues de la primera peticion y espero a que me llegue algo.

web sockets: es un protocolo de comunicación donde me dejo una conexión full duplex abierta entre el cliente y el servidor para que se pase información, en vez de hacer requests HTTP.

sse: el cliente se conecta, pero el servidor es el unico que envia.

UX responsive: adaptarse al dispositivo.

BFF (backend for frontend): Es una capa que se encarga de preparar los datos para un dispositivo puntual (Mobile bff, Web bff, etc)

- Desacopla presentación de servicios de lógica del negocio.
- Los devs frontend solo hacen desarrollo front end.

Infra tradicional

Todos los fierros los manejas vos.

Performance: Cantidad de trabajo útil completado por un sistema (una forma de medirlo es por tiempo de rta).

Escalabilidad: Propiedad de un sistema de adaptarse al crecimiento del uso, manteniendo la calidad de servicio. Picos de carga implica mayor uso de recursos, luego están en desuso.

- Horizontal (mas PCs).
- Vertical (mejora la PC que tengo, o sea más memoria y recursos). Tiene un límite físico.

Clustering: Grupo de servidores.

Alta disponibilidad: Habilidad del sistema para continuar funcionando, luego de la falla de uno o más servidores.

Estrategias al deployar:

- Canary deploy: ponemos la versión nueva en un servidor y lo monitoreamos, sino tiene errores groseros deployamos la nueva versión en los demás.
- Blue green deploy: Levantamos un cluster de servidores con la nueva versión lista y luego apagas los clusters anteriores. Asi te ahorras el tiempo de indisponibilidad del sistema.
- A/B TESTING: Se sube una funcionalidad y solo se muestran a un grupo de usuarios.

Load Balancing: Distribuir eficientemente la carga entre servidores. Tengo el problema de como trato las sesiones (esto en stateful, si es stateless no me importa). El tipo de LB más eficiente, es el que implementa Round Robin.

Si uso un IP Hash (que las requests con cierta IP caigan en cierto servidor) tengo el problema de que si se cae un servidor tengo que redireccionar esas ips a otro server. También puede que todas las requests recaigan en el mismo servidor.

Stickiness: Logra una afinidad entre el request y el servidor. Por ej. hago módulo de los últimos números de sesión de donde vino la request entre la cantidad de servidores. El tema es que, si se pierde el servidor, se pierde la sesión.

Session replication: En vez de que caiga en un servidor, lo podemos hacer caer a un pool de servidores y que tengan replicada la sesión. Si se cae uno no pasa nada porque tenes los otros. Esto genera overhead en memoria y red.

- O cada vez que hacemos un cambio vamos a modificar (sincrónica) o hacemos el cambio devolvemos el request y empezamos a propagar la sesión (asincrónica).
- La mejor opción es replicar la sesión en el momento en que se modifique.

Se pueden combinar stickiness y sesión replication (la idea de replicación de sesión se podría hacer en todos los servers directamente, pero ahi es lo mismo que tener un solo servidor por el espacio que ocupa).

CDN (content delivery network): Es para entregar contenido estático (fuera de nuestro server, lo cual les reduce la carga a estos).

- Están más cerca del cliente.
- Reduce carga de servidores, incrementa ancho de banda, reduce latencia, etc.
- Mejora web caching.

Failover: La habilidad de que las conexiones del cliente migren de un servidor a otro, en eventos de falla de servidor, por lo que las apps pueden continuar operando.

Transparent failover: El cliente no se entera de la caída del servidor, ya que el LB redirige el request a otro server disponible.

Si voy a usar clusterizacion, debería cachear recursos remotos (BD, APIs, etc) porque sino se genera un cuello de botella en estos con las requests de los N servidores.

Docker

Docker: Espacio virtual que me sirve para meter en un paquete todas las cosas que necesita mi programa para correr. (codigo, librerias, runtime, etc) y me abstrae de la compatibilidad del mismo con respecto a la pc en la que se está ejecutando (interfaz común). Garantiza que el SW correrá igual independientemente del lenguaje.

Los contenedores de Linux aíslan un entorno del otro. Es una tecnología de virtualización en el nivel del SO para Linux. Permite que un servidor ejecute múltiples instancias de sistemas operativos aislados. No provee una máquina virtual, más bien provee un entorno virtual que tiene su propio espacio de procesos y redes.

Image:

- Es un template read-only desde donde se crean los containers.
- Derivados de una imagen base (Ubuntu por ej).
- Armada por un conjunto de capas que conforman el FS o Image.
- Construida a partir de un Docker file.

Las imágenes son más eficientes en cuanto a espacio en disco, comparadas con un sistema tradicional de imágenes de Linux, ya que las capas se pueden reutilizar para distintos containers. La desventaja es que tiene que ir saltando por n capas a fuerza bruta entonces tiene más redirecciones y tarda más que un sistema tradicional que vas directo a cada imagen (menos performante).

Si quiero modificar un archivo en una imagen, tengo que hacer una copia de la misma al contenedor, y en esa capa R/W editarlo.

Docker registry: Almacena Docker Images, como si fuera un Repositorio GIT.

Container: es un ambiente virtual que te genera Docker, donde se pueden correr procesos aislados de los demás containers. También se podría decir que es una instancia runtime de una Docker image, agrega una capa r/w arriba de la imagen.

La mayor diferencia entre containers y layers es la capa R/W que tiene el container. Múltiples containers pueden compartir una misma imagen.

El sistema de capas es para optimizar espacio.

Los containers que escriban gran cantidad de datos, consumirán más espacio que los containers que no. Esto es porque la mayoría de las operaciones de escritura consumiran nuevo espacio en la capa writable del container. Si tenemos que escribir mucho dentro de un container hay que considerar usar un data volume.

Data Volume: Directorio o archivo en el FS del host, montado directamente dentro de un container.

Los containers nos proveen de un ambiente lo más cercano posible al que obtendríamos en una vm, pero sin el overhead que implica correr kernels por separado y simular todo el hardware.

Varios containers pueden correr sobre un kernel común. Las vms en cambio, cada una tiene su propio kernel.

Obtenemos aislamiento en procesos, redes, recursos (cpu, memoria, etc).

Hay que tratar de minimizar las capas, para que haga menos saltos en cada capa.

Se pueden setear variables de entornos para las imágenes y asi dejar seteados datos sensibles y no subirlos al repo por ej.

docker build -> image

docker run -> container

Las capas de las imagenes estan congeladas lo cual es bueno porque si corrio una vez va a correr siempre, pero la desventaja es que si queres agregar/actualizar una dependencia, tenes que buildear de nuevo.

Pros de Docker:

- Ambientes virtuales, livianos, rápidos y desechables.
- App agnóstica del entorno.
- Escape from dependency hell.
- La arquitectura la podés ver en el código.
- Simplifica configuración del ambiente de desarrollo, todas las apps son iguales desde afuera.

Registry: Es un build freezado (versionado de builds). Almacena docker images, es parecido a un git respoitory.

Container: Instancia runtime de una docker image con una capa read write arriba de la imagen.

Docker compose: Para definir y correr aplicaciones con múltiples containers (uno para el front, otro para la bd, etc).

Los containers están aislados a nivel de SO, pero comparten el kernel con otros containers. Las vm no comparten kernel por eso son más seguras.

Las vm compiten por los recursos, los contenedores están limitados a lo que se les asigne.

Como las BD necesitan persistir y los containers, nacen hacen algo y se mueren => no es buena idea tener un container para una BD.

Infraestructura Cloud

Beneficios:

- Reduce costos de infra
- Mejora la respuesta de IT
- Acelera y simplifica el aprovisionamiento de aplicaciones y recursos
- Soporta disaster recovery
- Manejo centralizado.

Cloud computing -> da acceso on demand a un pool compartido de recursos computacionales.

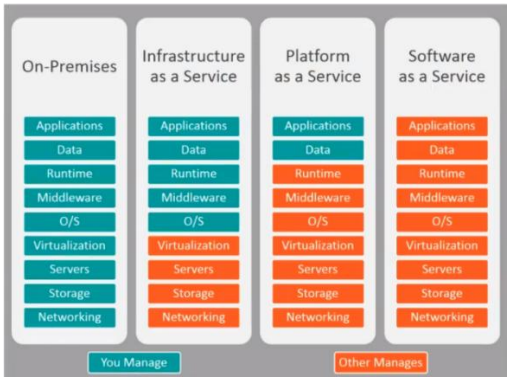
- Es accesible desde cualquier plataforma.
- Los recursos son compartidos entre distintos clientes.
- Capacidad de escalar rápidamente para soportar picos de tráfico.
- La factura es medida. Pago por lo que consumo.

On Premise es mejor que On demand en el caso de que tu carga sea siempre constante (lo cual es poco probable, caso contrario es mejor On Demand). Otro caso es la seguridad, por ej si sos un banco o algo que requiera tener tus datos ya que son sumamente sensible, entonces te conviene tener un data center, o también cuando no te interesa escalar por un tiempo.

IAAS -> VMs, Servers, Load Balancer, etc

PAAS -> bases de datos, development tools, etc

SAAS -> Email, Virtual Desktop, etc



Cloud vs Infra tradicional

Pros:

- Elasticidad
- Puedo empezar con bajo costo e ir creciendo
- Distribuido geográficamente
- Servicios provistos por el proveedor

Contras:

- No tengo los datos en mi poder
- Dependo de un externo (Vendor Locking)
- Me cobran por todo lo que hago
- Tengo menos flexibilidad, no siempre puedo hacer todo lo que quiero

IAAS: Manejas todo el fierro (por ej. Amazon EC2)

Pros:

- Puedo hacer lo que quiero
- Servicios provistos por la plataforma

- Fácil para portar aplicaciones existentes

Contras:

- Escalabilidad manual (más trabajo)
- Tengo que tener determinados perfiles en el equipo

PAAS: Permite la creación de apps webs rápida y fácilmente, sin la necesidad de mantener el SW y la infra detrás de estas, a través de un ambiente de desarrollo (por ej. Heroku)

Pros VS IAAS:

Ventajas:

- Autoscaling
- Menos costo de mantenimiento
- Poco conocimiento de infra

Desventajas:

- App acoplada a la plataforma (Vendor lock-in)
- Requerimientos acotados a la plataforma (hay proveedores que no permiten threads por ej)
- No es aplicable a cualquier framework
- Request con timeouts (en infra no pasa)
- BB. DD. Supeditada a la plataforma

SAAS: El proveedor de este SW licencia la aplicación a los clientes como un servicio on-demand, a través de una suscripción o sin cargos generando ingresos a través de publicidad (por ej. Gmail). Tiene sentido en SW que va a ser usado en un periodo corto o altos picos una vez por mes (Por ej. liquidación de sueldos)

Pricing: En que punto una empresa va de un cloud público a uno privado o a un hibrido.

Microservicios + refactor

Aplicaciones monolíticas: Aplicaciones concebidas para ser “toda la solución”. Quizás el contexto inicial era correcto, y luego cambio agregando cada vez más responsabilidades sobre la app.

Ventajas:

- Simples
- Baja latencia
- Único artefacto deployable
- Es fácil agregar una funcionalidad que haga uso de lo que ya hay

Desventajas:

- Los deploys son muy grandes -> Resta agilidad.
- No escala en gente -> muchas manos en un plato hacen mucho garabato.
- Tiempos altos en Test, build, deploy, etc.
- Un problema en la aplicación puede arrastrar todas las funcionalidades (por ej. un leak de memoria tira toda la app abajo).
- Funcionalidades diferentes pueden requerir infra diferente o configuraciones diferente.
- Al momento de los deploys hay que tener un técnico de cada funcionalidad (en el caso de microservicios puedo saber cuál microservicio falla y volver ese a una versión anterior y no afectar a los demás).
- Si bien tanto en monolito como en microservicios hay que dejar parte del código legacy ante un nuevo cambio hasta adaptar la compatibilidad, en monolito tenes más gente que pudo haber tocado ese código y más difícil es sacarlo.

Deploy disruptivo: rompe con la retrocompatibilidad.

La conversión de monolitico a miroservicios se debe hacer de a pedazos (dejo el código viejo mientras agrego la misma funcionalidad, pero en microservicio). En este transcurso se puede tener overhead por mantener 2 componentes iguales, replicación de bugs, etc.

Una forma de ganar crédito ante un PM para migrar a microservicios, es hacer notar el hecho de que ahora en vez de tener 30 personas encargadas de todo el proyecto, tenes 2 personas encargadas de que la sección de catálogo sea la mejor sección de catálogo.

Microservicios:

Ventajas:

- Puedo desacoplar los módulos, si se cae uno no se cae todo (ej. búsqueda de Spotify).
- Equipos más chicos enfocados a módulos concretos.
- Es más fácil utilizar funcionalidades de diferentes apps desde otras (ej, la app mobile solo se integra con los módulos de destacados y políticas comerciales y no con el resto).
- Cada microservicio en si es más simple.

Desventajas:

- Timeouts acumulativos y agregado de overhead (el primer micro, le pega al segundo y este al tercero, etc).
- Hay que coordinar la comunicación entre los distintos microservicios.
- Algunas responsabilidades no son tan claras como para determinar si van en una caja u otra.
- Si necesito más datos de una aplicación tengo que coordinar con el equipo responsable.
- Algún equipo de trabajo puede estar sobrecargado y otro sin carga.
- Mas puntos de falla.

NoSQL

DB Relacional

La normalización se usa para evitar duplicar operaciones por ej. si el nombre de la provincia la dejo dentro de la tabla de clientes, y cambia el nombre de una provincia, tengo que cambiar ese campo en todos los clientes. También por el espacio, el ID de una tabla de provincias es más barato que el string metido en cada cliente.

Para hacer JOINS, una manera óptima que usan las BD es a través de un hash match join, que utiliza una función de hash para matchear los elementos.

Es más rápido leer secuencialmente de disco, que leer aleatoriamente de RAM debido a la cache. Las computadoras trabajan mejor con cargas de trabajos que son secuenciales porque pueden aprovechar toda la cache. Con la escritura es lo mismo.

Si uso un archivo por cada columna, y todas las columnas ordenadas por el mismo valor se traduce en hacer JOINs de manera eficiente (puedo leer de archivos de manera secuencial). Permite comprimir bien los datos.

Perfect Hashing: es un mecanismo para organizar los servers en forma de anillo, para que si agrego o saco un server no tenga que cambiar toda la data de lugar como la tendría si usara un hash normal y a cada server le asigno las peticiones de cierto key del hash (si agrego o saco un server en el último caso, me cambia la numeración).

Replicacion: Mongo me permite tener réplicas de mis BD en varios servidores, donde una es la master y las demás son secundarias. Si se cae la principal, alguna de las secundarias pasa a ser primaria. Así si se cae una la app puede seguir funcionando.

Sharding: Consiste en dividir una base de datos horizontalmente, generalmente por las filas de una tabla. Así, se divide la responsabilidad de guardar la información entre diferentes nodos, reduciendo el procesamiento y el espacio de almacenamiento de cada nodo individualmente considerado.

- Al tener BD más chicas, tenes una escalabilidad mucho más rápida y accedes más rápido a los datos.
- Un problema podría ser si tengo una computadora a la que le caen todas las requests y las demás ninguna.
- Otra desventaja es si quiero sacar por ej el top 10, no puedo hacerlo por cada sharding tengo que hacerlo por cada uno y entre los que dieron juntarlos y volver a hacer el top 10.

NoSQL es útil cuando tenes muchos datos y muchas requests, caso contrario => SQL

En NoSQL primero pensas tus casos de uso y a partir de ellos sale el esquema de dominio (en SQL es al revés)

SQL	NoSQL
Optimizado para almacenamiento	Optimizado para computo
Normalizado	Desnormalizado/Jerarquico
Queries Ad Hoc	Queries fijas
Escalamiento vertical	Escalamiento horizontal
ACID	BASE
OLAP? *	OLTP? *

*Si yo tengo muchas transacciones pequeñas de lectura, escritura, etc => NOSql funciona muy bien.

BASE (el equivalente ACID de NoSQL):

- Basic Available: La BD está disponible la mayor parte del tiempo.
- Soft State: El estado de la BD puede cambiar incluso sin entradas. Esto es debido al siguiente punto.
- Eventual Consistent: Eventualmente los datos son los mismos en todos los servers (yo puedo leer un dato de una réplica secundaria que esta desactualizado con respecto al master, y en algún momento se actualizara).

Teorema CAP: Solamente puedo elegir 2 de las siguientes 3 propiedades:

- Consistencia: Todos los clientes ven lo mismo, incluso en la presencia de insert/update. Notar que no es la misma consistencia que ACID (ante una transacción paso de un estado consistente a otro también consistente).
- Disponibilidad: Todos los clientes pueden acceder a los datos, incluso ante la presencia de fallas.
- Tolerancia a partición: El sistema continúa funcionando, aunque existan particiones de red.

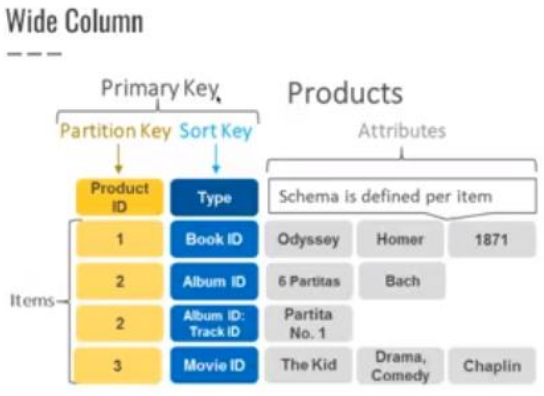
Esto es en la teoría. En la realidad, si vos tenes una BD distribuida en distintas maquinas, no podés elegir no ser tolerante a particiones (tendrías que tener todo en la misma maquina y se pierde la gracia de sharding o replicación por ej.).

En el caso de que tomemos CT (consistencia y tolerancia), es necesario que el pedido se aplique y confirme en todos los nodos, como uno de ellos no está disponible => el pedido no se puede realizar (puesto que tendría que esperar a que se levante para confirmar el pedido en esta y tenerla consistente).

Si tomamos DT (disponibilidad y tolerancia), en cuanto se caiga un nodo, va a seguir tomando requests pero puede mostrar datos temporalmente Inconsistentes (ya que no todas las BD van a tener el dato más reciente).

Tipos de DB NoSQL (**complementar con el otro resumen**):

- Key Value: Los datos se almacenan como un par clave – valor (como un hashmap). Eso hace que tenga flexibilidad nula (solo busco por la clave) pero es altamente performante.
- Wide column: como Key value pero el valor a su vez es otra estructura. Es poco flexible y altamente performante. La key se puede repetir, pero la key en conjunto con la sort key no.
- Columnares: Los datos se almacenan en un archivo por columna. Comprimen eficientemente los datos y son altamente performante en agregaciones.
- Orientadas a documentos: Colecciones de documentos, y cada documento a su vez tiene una estructura jerárquica (como si fuera un JSON). Son flexibles. MongoDB es orientada a documentos.
- Orientadas a grafos: Las entidades base son nodos y sus relaciones. Te permite hacer cosas que en SQL no (buscar relaciones entre los amigos de los amigos de mis amigos). No se puede partir. Altamente flexibles.



Service Mesh + Serverless

Service registry: Tengo una aplicación que conoce todas las instancias de B que están OK para que A se pueda comunicar. De alguna forma puede configurar/activar/desactivar instancias.

Health-Check: A le pregunta a B cada x tiempo si está ahí.

HeartBeat: B le dice cada x tiempo a A que todavía sigue vivo.

Service mesh: Capa de infraestructura dedicada para facilitar las comunicaciones de servicio a servicio entre microservicios.

Server side: Tengo un load balancer que routea los pedidos:

/B/servicio1

client side: Código embebido, de alguna forma en A conozco las instancias de B:

/B-01/servicio1

/B-02/servicio1 (le pego a cada instancia individual)

- Mucho trabajo repetido (poco eficiente).
- Si lo hago como una biblioteca me tengo que acoplar a un lenguaje/framework
- Si tengo un problema en la biblioteca, tengo que tirar abajo la app

sidecar proxy o data plane: Parecido a server side en el sentido de que no conozco las instancias, pero tengo un segundo proceso que se encarga de hablar con A (a través de un sidecar proxy en B y otro en A).

- Se complementa con un componente de arquitectura llamado “Control Plane”.
- Se diferencia de un load balancer, ya que este es un single point of failure. En cambio, acá es uno por cada instancia del cliente.

- Si llega a haber un problema en el sidecar, el código es uno => no tengo que codearlo en todos los microservicios, solo en el sidecar proxy.

Server side: A -> LB -> B

Control plane: Da apoyo para la comunicación. Sabe las instancias que existen de cada app. Se encarga de Routing, Balancing, Health Check.

Ventajas:

- Independiente del lenguaje, framework o aplicacion.
- Puedo deployar una nueva versión sin intervención de la app
- Vision global de mi arquitectura (si quiero poner los logs en un lugar centralizado, puedo).
- Como es algo que maneja a nivel centralizado, todos los SP, puedo agregar features para todos mis componentes (por ej servicio de monitoreo).

Service mesh: A|SP -> SP|B (menos impacto en la red pues hay menos salida a la red)

Ventajas:

- No requiere casi cambios a la app (salvo la comunicación con el SP)
- No hay un single point of failure
- Features out of the box (por ej. configuración de servicios, este no le puede pegar a este, etc.).
- Eliminamos capas de alto tráfico.

Desventajas:

- Dependencia implícita de la aplicación (se te cae el SP => el microservicio no es accesible).
- Agrega complejidad (si tengo un bug puede estar también en el SP o en la interacción).
- Proxies descentralizados (es más difícil decirles a todas las instancias que se cayó otra).

serverless: Es un concepto en el que el proveedor de la nube ejecuta el servidor y administra dinámicamente la asignación de recursos de la máquina (a través de containers efímeros). Yo solo me encargo de la función específica de mi código que quiero que se ejecute sobre este servidor (Faas).

Ventajas:

- Simplifico la operatoria IT.
- Alta escalabilidad desde el momento 0.
- Bajo costo (pagamos por lo que usamos, ideal para startups).

Desventajas:

- Vendor locking.
- Limitaciones (memoria, time out definido por el proveedor, etc.).
- Cold start problem (la primer request tarda más porque está en frio).

En serverless no se comparten procesos con otras funciones

Serverless es para casos concretos (necesito algo chiquito como por ej. mandar un mail)

SRE (Service Reliability Engineering)

Service management: Todo lo que no es features (como voy a administrar ese servicio, mantenimiento, deployar, etc).

Time to market: Tiempo que tardo en deployar y que funcione todo bien.

DevOps: Conjunto de prácticas para involucrar gente de desarrollo con gente de operaciones. Desarrolladores más comprometidos con la parte de deployar.

- Aceptar el fallo como normal.

- Implementar cambios graduales. Mayor frecuencia de deploy.
- Medir todo.
- Automatización.

SRE es una implementación de DEVOps. El objetivo principal es desarrollar un sistema altamente confiable desde la perspectiva del usuario.

Reliability es el feature más importante. Los usuarios son los que miden la confiabilidad del sistema, no su monitoreo/logs.

La confiabilidad se mide en cuanto tiempo está funcionando. 100% es el approach equivocado ya que es imposible y sumamente costoso (a no ser cosas como aeronáutica o cosas que impliquen que si falla se muere alguien).

Error Budget: Tiempo indisponible. 100% - Avilability target. Cuanto más chico (o más alto el tiempo que necesito de disponibilidad), más caro es meter un nuevo feature o cambio, porque tengo que meter toda una planificación para testear, meter un ambiente para testear, etc.

SLI: Indiciador del nivel del servicio. Mide que una request sea lo suficientemente buena. Request latency, error rate, etc.

SLO: Definir un objetivo de nivel de servicio, para no invertir de menos ni de más. Por ej. que el servicio ande super rápido puede degradar la performance (si de repente sube el tiempo de rta a un nivel normal, los usuarios que se acostumbraron a lo rápido lo toman como algo malo), entonces defino un umbral. Otro ej. si tengo una disponibilidad de 99,999999999% entonces quizás estoy usando muchos recursos en algo que no lo amerita. El SLO se basa en base al daño que se puede soportar (se define a través del SLI).

SLA: Es un contrato donde yo me comprometo a esto y si no tengo estas consecuencias. Por ej. AWS me promete 99,999% de disponibilidad y si no lo cumple, me recompensa de alguna manera (un descuento).

Error budgets y SLO son indicadores de cuando parar la pelota y analizar. El SLO se acuerda no solo con el equipo de desarrollo sino con otros equipos, como finanzas.

Si supero el error Budget --> Freeze (no meto más features hasta arreglar esto, arreglo los bugs y veo como no volver a tropezar con la misma piedra).

Para implementar SRE:

- Tiene que haber soporte ejecutivo y voluntad de que va a mejorar a largo plazo esto.
- Una aplicación a la vez.
- Empezar definiendo el error Budget (se desprende del SLO). Nos indica si podemos acelerar o frenar la salida de features del sistema.
- Alertas y monitoreo. Alertar cosas que afecten al usuario, no alertar de más.
- No culpar a las personas por los errores.

Principios:

- Aumentar la confianza requiere un costo (de HW y de tiempo).
- Consideraciones de SLO (que nivel espera el usuario, que SLA provee la compañía, etc.)
- Error Budget para obtener la máxima velocidad de entrega de features.
- Tengo que aceptar que va a haber bugs.
- Automatizar.

Monitoreo

Ver cómo se comporta mi app en runtime (Memoria usada, espacio utilizado, uso de cpu, threads, etc.)

Tiempo de respuesta:

- Tiempo de respuesta del usuario (browser)
- Percentiles (el promedio es mentiroso, si tengo la mitad de usuarios que les respondo super rápido y al otro super lento => en promedio estaría bien, lo cual no es cierto)

JMX: Protocolo que permite conectarse a una VM viva y ver diferentes aspectos (memoria, hilos, etc.)

- Comando “verbose” me va diciendo que hace el GC (garbage collector).

Vegeta: APP para simular volumen.

Memory leak: No se liberó memoria y quedaron objetos que ya no deberías acceder, pero el GC no los puede liberar. Puede hacer que mi app se caiga.

Manejo de memoria:

- Reservar espacios de memoria para su uso
- Liberar espacios de memoria previamente reservados
- Compactar espacios libres y llevar cuenta de espacios libres y ocupados

Alertas de que estamos usando mal la memoria (la veo en la VM y no baja):

- Se te caen instancias
- Alarma de que hay pocas memorias

Maneras de solucionarlo:

- Reinicio la app (dejamos 1 instancia de testigo para testearla).
- Heap dump: Vuelca la memoria ocupada por la JVM en un archivo para ser analizada (la más utilizada es jmap). Una vez que tengo el archivo lo analizo con MAT (memory analizer tool) que permite analizar un dump (el archivo que creo el heap dump).

Serializacion (Synchronized): define una zona de exclusión mutua (solo puede tomarlo el primero que lo toma). La contra es que es muy lento (solo 1 por vez puede operar).

JStack: Thread Dump (puedo ver deadlocks y locks). Otro seria fastthread.io