Componentes ordenadores: CPU, Memoria (RAM), tarjetas de red (conexión) y un disco duro interno. Los MAC tienen todos la misma arquitectura (x86). También está el sistema operativo (OS/SO). Y luego están las aplicaciones.

Problemas: el ordenador está infrautilizado, es más caro de lo que debería. El task manager sirve para ver cuánto usas del ordenador. Estamos “malgastando” nuestros ordenadores. Ejemplo de Amazon y el Black Friday, ni infra ni sobreutilización, para optimizar los recursos. La virtualización (VMs) soluciona estos problemas, aprovechando lo máximo posible la máquina.

Las empresas antes de la virtualización aprovechaban menos del 5% de toda la “granja de servidores” de que disponían. Esa granja es muy cara, tanto por el hardware como por el personal. También, si tenemos múltiples aplicaciones en un solo SO (una sola máquina) ello entraña un problema de seguridad grave. O un problema de acoplamiento, que si un SO se cae se caen todas las apps.

Ahora tenemos en un mismo dispositivo varias VM, cada una de las cuales con su propio SO (uno puede ser Linux, otro Windows y así) y sus aplicaciones y su parte del hardware del total. Lo que hacemos es aislarlos. Por ejemplo, si tengo en mi ordenador entero 8 gigas de RAM entonces puedo sacar dos VMs de 4 gigas cada una. Puedo tener tantas como quiera, con límite en los recursos existentes, juego de suma cero, obviamente.

Capa o software de virtualización permite crear las VMs. Un programa instalado dentro el SO que se encarga de gestionar y de crear todas esas VMs\*\*\*. Hacer muchos a partir de uno, con objetivo empresarial principal de reducir costes. También para evitar averías, pues con las VMs hay un sistema de snapshots (fotos de cada VM en un momento dado) que permite que en cuestión de minutos se repare fácilmente o se pueda cambiar a otro sitio para seguir trabajando mientras se restaura.

Es parecido al servicio Cloud, pero sabiendo cómo es la máquina física subyacente (en la nube no lo sabremos nunca), pero el funcionamiento es parecido.

HyperV y Docker

* Server Virtualization: poner varios SOs corriendo al mismo tiempo en el mismo ordenador físico.
* Network Virtualization: virtualización de redes, las diferentes máquinas comparten ancho de banda, tarjetas de red. Útil para temas de seguridad, protección de datos o lo que sea. Las VPNs funcionan así.
* Desktop Virtualization: el escritorio, las aplicaciones en concreto.
* Storage: varias unidades virtuales de almacenamiento sobre el mismo disco.

\*\*\* A esa pieza software se le conoce como **hypervisor**. Separa el SO de la máquina física y sus aplicaciones. La gracia de esto es que puedes tener tirando y ejecutando a la vez varios SOs. Si la arquitectura del hardware subyacente es común es más fácil hacer esa virtualización.

Virtualbox es como HyperV pero para casa, amigable, sencillo y compatible con todos los SOs, pero no lo usaremos en clase.

**Continuous Integration (CI)**: que la rueda no pare de girar, que el mecanismo de testing y validación sea automático y en bucle, desde la mente del desarrollador hasta producción. Detectan fallos en el sistema de los desarrolladores. (Sistema de control de código puede ser GIT; la compilación traduce de tu lenguaje al lenguaje de la máquina). Subo el contenido a mi GIT, se compila, se testea, se comprueba si funciona y en función de eso se actúa (notificaciones). Servidor de integración continua.

**Continuous Delivery (CD)**: la continuación de CI, cuando acaba CI, el CD hace el delivery de las aplicaciones en las estructuras donde digamos.

Pasos desde que se chequea una línea de código hasta que llega a producción. Y mientras, se produce esa automatización de CI y CD. La virtualización viene genial para CI y para CD, gracias a las VMs. Para probar con varios SOs o para no parar esa rueda, unas VMs estarán probando el software que está ya para producción y otras estarán con el software que se está desarrollando, por ejemplo. La virtualización sirve para simular la infraestructura necesaria para que esa cadena no se pare.

El testing puede ser muy diverso, desde errores simples de código hasta incumplimiento de los estándares de código (más hilo fino). Hay muchas herramientas de virtualización.

Vagrant es una manera de **automatizar la creación de VMs**, pero a través de ficheros de texto. Por detrás está utilizando Virtualbox en realidad. Está ahí la configuración concreta de la VM para por si se estropea el ordenador tener uno nuevo enseguida exactamente igual al estropeado y poder trabajar sin perder el tiempo (tengo la “receta” a través de un fichero de texto para ejecutarlo y levantarlo enseguida en cuanto se necesite).

Para crear VMs en **Google Cloud** (cuenta EDEM) hacíamos Compute Engine + Instancias de VM. Hay regiones > zonas > CDNs (estos últimos son la infraestructura de los pobres, allí donde no llegan zonas siquiera sí habrá algún CDN, -*content distribution network*-). Una imagen es una foto del disco en un momento dado que contiene la información de ese disco, para poder copiarlo igual. Para abrir una terminal en un equipo que está en remoto uso protocolos como SSH (puerto 22). Cada IP tiene varios puertos asociados y se va al puerto, no al IP (“dirección postal” de tu ordenador). Http es el puerto 80 y el https (igual pero en cifrado) es el 443. La máquina virtual que creas tiene una IP diferente, no puede haber en el mundo dos IPs iguales. Hay que abrir solo los puertos que necesitas, ninguno más, por temas de seguridad. Las etiquetas pueden servir para filtrar por puertos, o por direcciones de IP de origen/destino, por ejemplo (cortafuegos).

En **Amazon** se llaman EC2 las máquinas virtuales. Amazon es quien mayor cuota de mercado tiene de las 3 grandes (Azure, Google Cloud y AWS). Hay subasta (“mercado negro”) aparte del precio fijo de las VMs, donde tú pones el precio máximo que quieres pagar y si está por encima luego el precio pues te tumban las máquinas. Servicio de almacenamiento de datos de Amazon es S3.

Arquitectura de microservicios: tengo una pieza para cada función (especialización). Tengo muchas piececitas muy pequeñas. Bueno para optimizar recursos en una empresa. Se tiene todo mucho más aislado al separarlo en múltiples. Algo positivo de esto: conseguir una **base de datos distribuida**, para mejorar el rendimiento. Al ser tan pequeños son tan sencillos que van muy rápido los microservicios. Y cada servicio se levanta en una VMi distinta con una imagen asociada para poder replicarlas sin tener que crearlo desde cero de nuevo en el caso de que se caiga una, por ejemplo.

Un contenedor es un empaquetado de software. Hay subsistemas operativos (puede ser cualquiera y diferentes) que permite comunicarse a través del *container engine* (**DOCKER**) con el SO de mi máquina (que es uno único). En las VMs también hay SO (falta en la diapositiva). Se gana mayor rapidez y eficiencia con respecto a las VMs normales. La cosa aquí está en que hay subsistemas operativos pequeñitos más eficientes que se crean gracias a Docker en cada VM. El contenedor está dentro de una VM. Cada contenedor tiene un subsistema muy pequeño que incluye cada una de las aplicaciones que usaremos. Para las apps es como si estuvieran corriendo en un SO completo, pero con mucha menos carga (se gana agilidad y velocidad). Y cada contenedor es completamente independiente de los otros, como las VMs. Pero no todas las apps se pueden meter en un contenedor.

En Google Cloud se hace crear una VM con Imagen como Container Optimized OS y se activan las http y https. Y una vez creada se da a SSH para que salga el terminal.

Docker run se ha bajado una **imagen** de Internet porque no la ha encontrado localmente, la ha metido dentro de un contenedor y la está ejecutando. 80:80 es el puerto externo e interno respectivamente, podrían ser diferentes.

Http://IP externa:puerto externo

Si ponemos puerto 8000:80 no nos deja porque no está habilitado. Pues para abrir un puerto, en Google Cloud se pone Red de VPC y crear cortafuegos. Se pone nombre, se crea etiqueta (después de las casillas de http), se permite entrada a todas las máquinas (0.0.0.0/0) y se pone tcp: 8000 (se permite desde ese puerto). Luego en instancia VM se edita y se pone etiqueta, la misma que había puesto en cortafuegos.

Docker File permite crear la imagen que habíamos sacado de Internet: es un documento de texto que contiene todos los comandos necesarios para construir una imagen. Se ejecutan las instrucciones secuencialmente, con un orden, una detrás de otra. Cada línea de instrucciones es una capa distinta, cada una superpuesta sobre la anterior. El conjunto de todas las capas es mi nueva imagen que quiero construir a partir de ese fichero. Docker **cachea**, es otra ventaja, recuerda lo que tenías, lo guarda en una memoria de acceso rápido (como la RAM) en vez de en una lenta (disco duro). Cada capa es pues un trozo de instrucciones cacheado por Docker que en su conjunto creará la imagen.

Una variable de entorno es aquella que puede ser accedida dentro del SO en cualquier momento.

En terminal:

Touch Dockerfile

Ls

Vi Dockerfile

Copiamos todo eso

Se da a ESC y luego :wq para guardar (w) y salir (q) de vi

En local:

Se crea un archivo cualquiera con el nombre Dockerfile y se copia todo eso

El docker run es el que ejecuta el contenedor.

--name nombrecontenedor nombreimagen: contenedor que incluye la imagen que previamente habíamos creado.

Ejecuto el intérprete bash (que abra una interfaz para que pueda ejecutar comandos en el sistema, el que entiende los comandos que escribes y lo traduce) en el contenedor. Accedemos dentro del ordenador con este último comando, le pedimos que ejecute X dentro del contenedor creado.

[…]

nginx es un servidor web

Vagrant va solo con VMs, no con contenedores

Hands on 4: creo una página web con ese contenido. Docker Hub es un repositorio universal de imágenes Docker, de ahí hemos creado la nuestra (FROM ubuntu …), y se descarga en local. Se crea imagen y a partir de él un contenedor. Las imágenes vienen del registro de imágenes (DockerHub, local o cualquier otro sitio).

La imagen se sobreescribe con la nueva info del COPY y es la misma con mismo nombre, pero el contenedor tiene que ser diferente y por tanto hay que abrir un nuevo puerto porque el de antes lo está ocupando el otro contenedor.

Por **orden**:

Docker build: crea la imagen de Dockerfile (archivo “touch”)

Docker pull: descarga la imagen sin ejecutarla

Docker run: coge la imagen y crea el contenedor

Primero, parar y eliminar los contenedores y luego las imágenes.

Docker container stop nombreID: lo paras primero. Para saber el ID pongo docker ps o docker container ls.

Docker container rm nombreID: lo borras después.

Para imagen directamente hacer docker image rm nombrenormaldelaimagen

Dockerhub para descargarse imágenes ya hechas, tenemos ahí en concreto la de Wordpress, pero podría ser cualquier otra.

Hacemos pull: docker pull wordpress

Y ejecutamos con el run como dice la diapo

Los contenedores nos proporcionan un aislamiento (como las VMs), pero son más ágiles.

Cuando le das a docker exec ya estás dentro del contenedor.

Para crear la imagen a partir del contenedor (que está modificado por la línea añadida, hemos añadido algo al fichero index.html) hago commit contenedor imagennueva. Este commit se hace fuera del contenedor (darle a exit si estamos dentro de exec).

2ABallybrit@

docker tag (etiqueta) IDimagennueva dockerhubID(cadoca)/loquesea:latest (la última versión).

Y hago push de la etiqueta de la imagen (dockerhubID(cadoca)/loquesea) para subirlo a Dockerhub. Hemos subido la imagen a un registro de imagen público. Y el público ya puede descargarse esa imagen con un pull.

Un volumen es una unidad externa al contenedor que sirve para desacoplar el contenedor de los datos (es como un pen drive). Si tiro el contenedor sigo teniendo los datos porque tengo el volumen, evitas que se pierdan los datos. También hay carpetas mapeadas desde el pc (bind mounts)??. El mismo volumen puede tener información de múltiples contenedores.

Docker Compose: una aplicación con **múltiples contenedores provenientes de un mismo fichero**, no me tengo que crear diferentes Dockerfiles para cada contenedor.

Creo un contenedor db a partir de una imagen (que es una base de datos), y se ejecuta un contenedor para la base de datos, y aparte un contenedor para wordpress.

Touch **docker-compose.yml**

Vi (…) y rellenar con todo el código del link

Lanzo varios contenedores de manera fácil y rápida. El docker compose cogerá a cada uno y hará el docker run para cada uno.

Para descargar en cloud el docker compose:

<https://cloud.google.com/community/tutorials/docker-compose-on-container-optimized-os>

(añadir –d al final donde pone up)

En local sería: docker compose up (creo)

Docker-compose up **–d** (añadir la –d siempre): es como un docker build que buscaba directamente al fichero concreto que se llamaba Dockerfile, pues aquí lo mismo pero con docker-compose.yml

En Kitematic salen los contenedores y se pueden eliminar directamente (se ve todo con una interfaz gráfica) pero solo si lo haces desde local, no desde cloud, obviamente.

Kubernetes administra y gestiona los contenedores, quien orquesta todo, los recursos asignados a cada uno, dónde va cada uno, etc. Kubernetes y Docker irán de la mano, por tanto. Pero este existirá ya cuando la aplicación esté desarrollada, no para el desarrollo propiamente dicho (para eso está Docker).