**Notas para la presentación de Kubernetes, Docker y Helm (maybe)**

(Introducción) Contar un poco el hecho de que no se vea en la UCA pero sí en el trabajo etc. Link del proyecto. La charla será un tanto práctica. Podéis interrumpirme en cualquier momento para preguntar dudas.

From Monolithic apps to Microservices

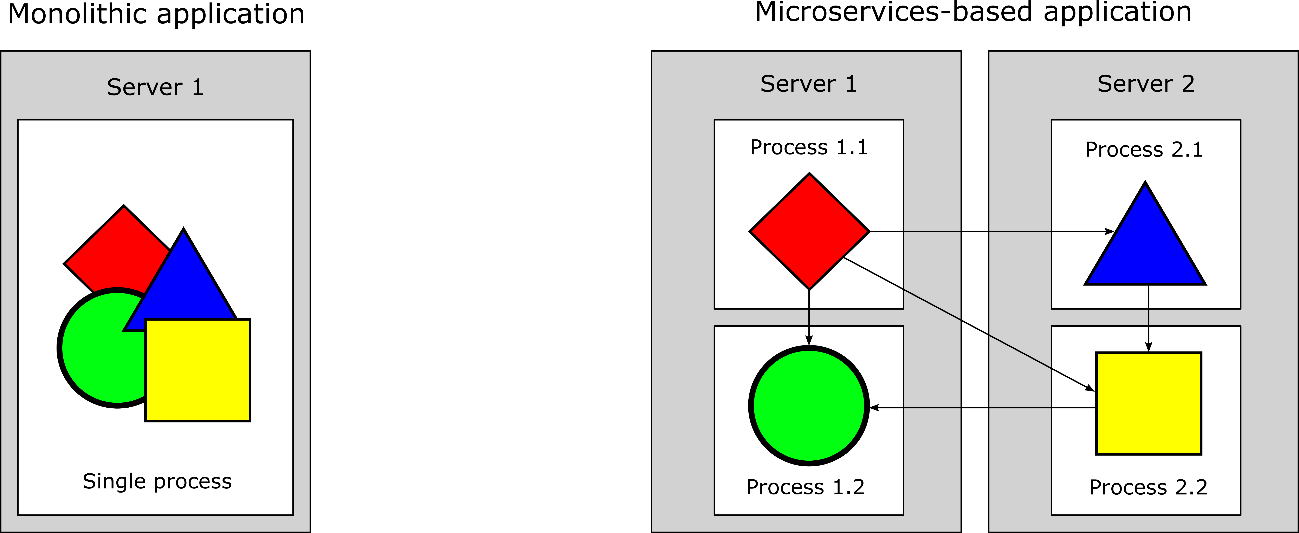
Hace unos años, la mayoría de las aplicaciones eran monolíticas. Esto significa que corrían como un único proceso en el sistema operativo y tenían que ser desarrolladas, desplegadas y mantenidas como una única entidad. Cambios en alguna parte de la aplicación significaba un redespliegue de la aplicación entera.

Correr una aplicación monolítica usualmente requiere de pocos servidores potentes, para proporcionar los recursos necesarios a la aplicación. Para solucionar problemas de falta de recursos, tendrías que escalar verticalmente estos servidores (aumentar número de CPUs, RAM, ..) o escalar horizontalmente, a base de añadir más servidores y correr múltiples copias (réplicas) de la aplicación.

Escalar verticalmente no requiere cambios en la aplicación, se vuelve caro y tiene un límite. Escalar horizontalmente es más barato en cuanto al hardware, pero requeriría de grandes cambios a nivel de software. De hecho, hay veces que escalar horizontalmente es imposible (bases de datos relacionales, por ejemplo).

Dados estos problemas, se empezaron a dividir estas grandes y complejas aplicaciones monolíticas en partes más pequeñas e independientes llamadas microservicios. Cada microservicio corre como un proceso independiente y se comunica con los demás microservicios vía APIs usando protocolos como el HTTP, por ejemplo.

Escalar microservicios, a diferencia de los sistemas monolíticos, está hecho a nivel de servicios. No tienes que escalar la aplicación entera como en un sistema monolítico, si no que tienes la opción de escalar aquellos servicios que necesitan más recursos solamente. Esto se consigue a base de replicar componentes.



Docker y contenedores

La idea detrás de Docker es crear contenedores ligeros y portables para que las aplicaciones puedan ejecutarse en cualquier máquina con Docker instalado, independientemente del sistema operativo que tenga por debajo, facilitando así los despliegues.

Lo bueno de esto es que los desarrolladores pueden despreocuparse de si su aplicación funcionará en la máquina en la que se ejecutará.

Os pongo un ejemplo:

Pepe y José están trabajando en distintas aplicaciones por separado. Pepe tiene instalado Java 7 en su ordenador, que es lo necesario para correr su aplicación.

Por otro lado, José está trabajando en un módulo de su aplicación que requiere Java 8. José quiere que Pepe pruebe su aplicación, pero esta no funcionará a no ser que Pepe se instale Java 8.

Este escenario desaparece con Docker. Para ejecutar la aplicación, José podría crear un contenedor Docker con la aplicación dentro, la versión 8 de Java y los demás recursos que necesite su aplicación. De esta forma, Pepe solo necesitará Docker para correr la aplicación en su ordenador.

**Contenedores**

Un contenedor es simplemente un único proceso aislado corriendo en el sistema operativo en el que está encima.

Comparados con las máquinas virtuales, son mucho más ligeros. Esto te permite correr más componentes en el mismo hardware. Esto se debe a que una máquina virtual tiene que correr sus propios procesos del sistema, etc.

**Vagrant**

Para entender Docker y Kubernetes, vamos a ejecutar esta charla de forma práctica. Antes de nada, debo explicar otra plataforma más para entender dónde estamos haciendo las pruebas. En el enlace os he descrito bien en el README.md cómo ejecutar la máquina virtual.

Vagrant es una herramienta para crear y administrar entornos de máquinas virtuales en un solo flujo de trabajo. Las máquinas se aprovisionan sobre VirtualBox, VMware, AWS o cualquier otro proveedor.

Vagrant aislará las dependencias y su configuración en un solo entorno desechable y consistente.

El Vagrantfile es un documento con las instrucciones para montar tu imagen.

Explicar Xming para exportar las X.

First steps with Docker

Crear, correr y compartir una imagen de contenedor

Lo que vamos a hacer es crear, correr y compartir una imagen de contenedor.

Una vez instalado Docker, podemos probar a correr algunos comandos para familiarizarnos con él. Por ejemplo, podemos empezar por descargar y correr una imagen de Docker Hub. Docker Hub (hub.docker.com) es un sitio público que contiene imágenes listas para ser usadas. Vamos a usar la imagen **busybox** para imprimir un simple “Hola mundo”.

**$ docker run busybox echo “Hello world”**

**$ docker run hello-world**

**$ docker run Ubuntu ls -Al**

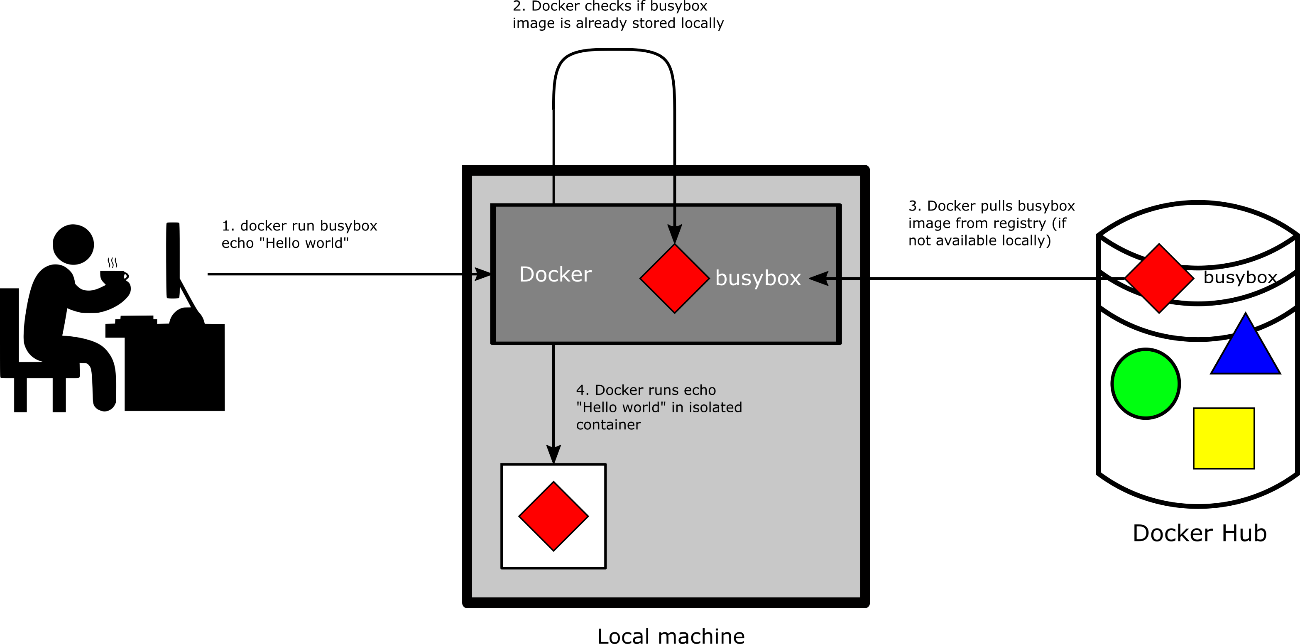
**\*Correr de nuevo un comando para mostrar que no se pullean las imágenes de nuevo\***

**$ docker images**

**\*Mostrar imágenes descargadas\***

Vale, pero ¿qué estamos haciendo exactamente? Realmente esto es increíble porque estamos descargando y ejecutando una aplicación en un simple comando, sin tener que instalarla ni nada por el estilo. También es importante destacar que esta aplicación que estamos corriendo se ejecuta en un contenedor completamente aislado de todos los demás procesos corriendo en nuestra máquina virtual.

**Entendiendo qué pasa realmente (Figura 1)**



Primero, Docker comprueba si la imagen **busybox:latest** se encuentra en local. Si no es así, Docker se la descarga del Docker Hub registry (en docker.io). Después de que la imagen se descargara en tu ordenador, Docker ha creado un contenedor a partir de esa imagen y corrió el comando dentro de él.

**Versionado de las imágenes**

Para distinguir la versión de una imagen, se añaden dos puntos tras el nombre y luego se escribe la versión. Para obtener la última versión de esta, se pone “latest”. Si no especificas el tag, Docker asumirá que te refieres a “latest”.

**$ docker run <image>:<tag>**

**Creando una aplicación sencilla en Node.js**

Para explicar los siguientes conceptos, como por ejemplo el Dockerfile, vamos a crear una aplicación bastante simple. Será una aplicación web (Un simple print) desplegada sobre un contenedor con ayuda de Docker. Esta aplicación va a aceptar peticiones HTTP y contestará a cada una de ellas con un estado 200 (un OK por parte del server) y un print del hostname de la máquina en la que está corriendo. De esta forma, como la máquina estará corriendo en un contenedor solo verá su propio hostname y no el de la máquina que está levantando ese contenedor. En un futuro, cuando usemos Kubernetes y escalemos horizontalmente la aplicación (esto es, ampliar el número de réplicas a base de usar varios contenedores), veremos como las peticiones HTTP alcanzan diferentes hostnames (de los diferentes contenedores que hay).

La aplicación consistirá en un único fichero: app.js. **(Enseñar código y explicarlo)**

El código de la aplicación lanza un servidor HTTP en el puerto 8080, y responde con un status code 200 OK a cada petición recibida, además del print antes mencionado.

Para correr la aplicación podrías descargar e instalar Node.js, porque no lo he metido en la máquina virtual. Pero esto no es necesario, porque Docker es el encargado de instalar todo lo necesario en nuestra imagen para abstraernos de tener instaladas cosas en nuestra máquina local.

**Creando el Dockerfile**

Para empaquetar tu aplicación en la imagen que queremos crear, debemos crear lo que se llama como Dockerfile. Es la hoja de instrucciones que leerá Docker para crear nuestra imagen. **(Enseñar Dockerfile)**

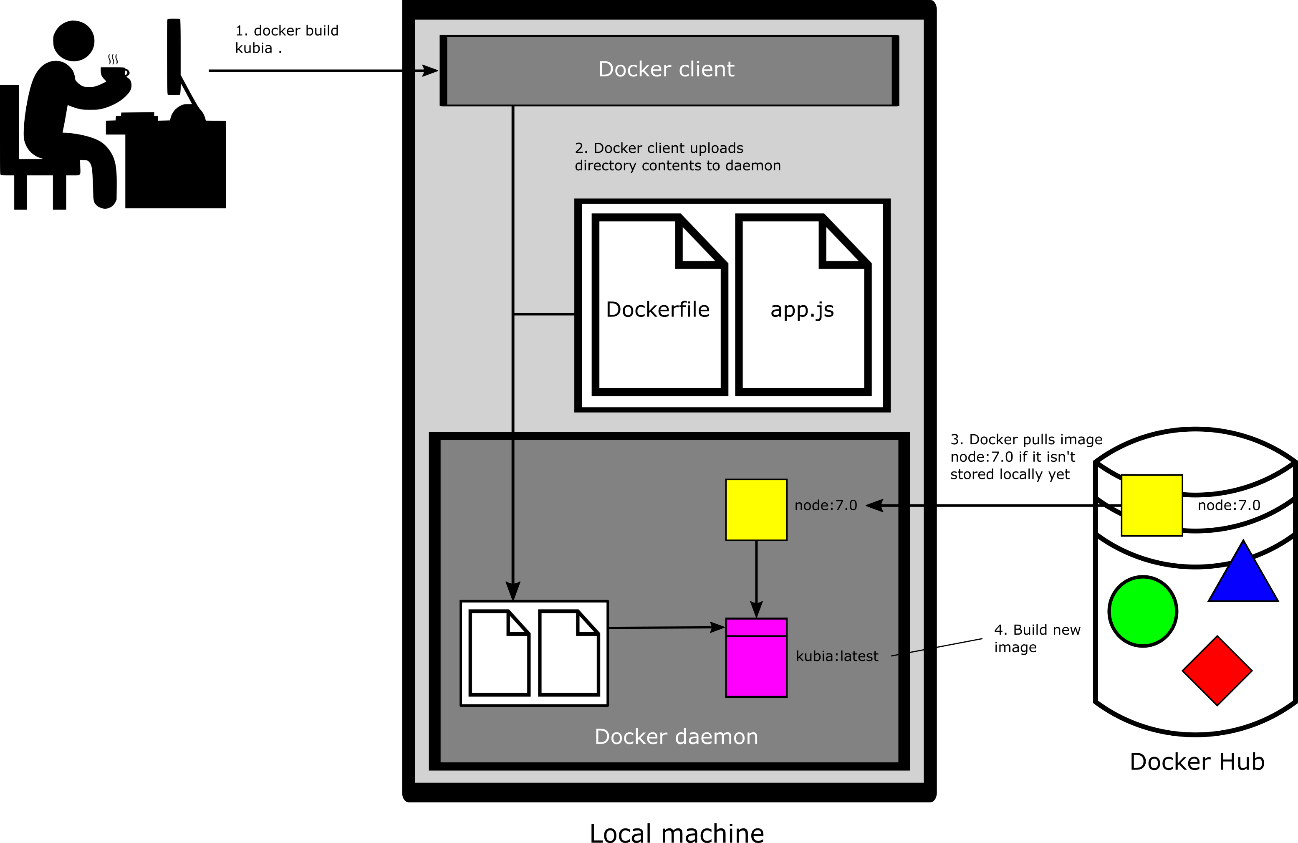
En la parte del **FROM** se define la imagen sobre la cuál vas a partir, junto con su tag. En la segunda línea, estás añadiendo tu aplicación desde tu directorio local al directorio root de tu imagen, con el mismo nombre. Por último, en la tercera línea le estás diciendo qué comando debe ejecutar cuando alguien vaya a correr la imagen. En este caso es **node app.js**.

**Construyendo la imagen**

Ahora que tenemos el Dockerfile, corremos el comando:

**$ docker build -t kubia .**

Con este comando le estamos diciendo a Docker que construya una imagen llamada kubia. Docker buscará el Dockerfile que se encuentra en la dirección especificada para ello. **(Explicar gráfico)**



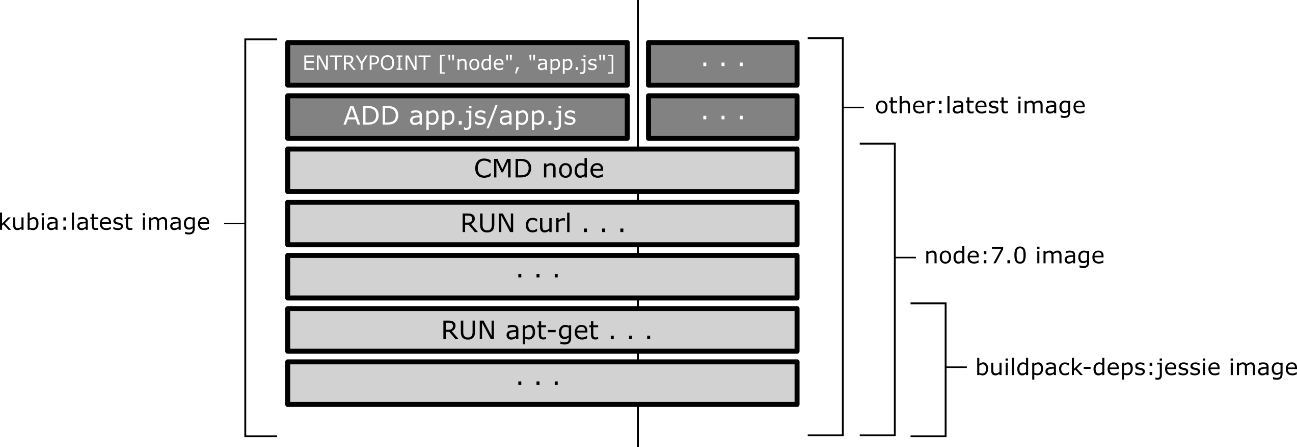
**Entendiendo cómo Docker construye una imagen**

Docker daemon es el encargado de construir la imagen, no el cliente de Docker. Nuestros ficheros se suben al demonio de Docker y este construye la imagen. De hecho, no tienen ni por qué estar en la misma máquina. Si estás construyendo la imagen desde un sistema operativo que no es Linux, el cliente de Docker lo vas a tener en ese sistema operativo pero el demonio estará corriendo en la máquina virtual dentro de ese sistema operativo. En este caso, lo normal es que estén en la misma máquina dado que estoy ejecutando Docker dentro de esta misma máquina virtual.

**Entendiendo las capas de la imagen**

Una imagen no es un solo binario gigante, está compuesto por varias capas. Quizás os habéis dado cuenta al hacer varios pulls antes cuando Docker se estaba bajando busybox. Diferentes imágenes pueden tener capas en común, haciendo el proceso de descarga más eficiente. Por ejemplo, si creas varias imágenes que se basan en la misma imagen base (por ejemplo, node:7), esa imagen solo se guardará una vez y será usada por todas las demás imágenes.

Aparte de eso, cuando construyes una imagen Docker crea una nueva capa por cada comando del Dockerfile justo encima de las de la imagen de la que parte. En nuestro caso, la última capa será etiquetada como **kubia:latest.** También se puede ver cómo otra imagen cualquiera **other:latest** basada en **node:7.0** aprovecharía esas capas comunes.



Una vez construida la imagen, la tenemos localmente. Podemos comprobarlo corriendo el comando **$ docker images.**

**Ejecutando nuestra imagen**

Vamos a ejecutar nuestra imagen con el siguiente comando:

**$ docker run --name kubia-container -p 8080:8080 -d kubia**

Este comando ejecutará nuestra imagen **kubia** en un contenedor al que le pondrá de nombre **kubia-container** y lo hará en modo detached (-d). Esto significa que podremos seguir poniendo comandos y la aplicación se ha lanzado en segundo plano (separada de la consola). La opción **-p 8080:8080** significa que el puerto de nuestra máquina local 8080 será mapeado al puerto 8080 dentro del contenedor. Con esto conseguiremos acceder a nuestra aplicación a través de **http://localhost:8080** desde nuestra máquina local.

Si estáis usando Docker con un Mac o Windows, no estáis corriendo el Docker daemon dentro de la misma máquina virtual así que necesitaréis el nombre o IP de la máquina virtual en la que está corriendo este demonio de Docker. Para mirarlo, usad la variable de entorno **DOCKER\_HOST**.

Ahora podemos acceder a nuestra aplicación. Podemos hacer curl o abrir un navegador.

**$ curl localhost:8080**

El número hexadecimal que recibimos es el ID del contenedor de Docker. Si hacemos **$ docker ps** veremos una lista de los contenedores que está ejecutando Docker ahora mismo.

**Explorando el interior de un contenedor**

¿Qué pasa si queremos ver cómo es el entorno dentro de nuestro contenedor? Dado que nuestra imagen se basa en la imagen de Node.js que contiene un bash shell, puedes correr un shell dentro de él así:

**$ docker exec -it kubia-container bash**

-it -> -i: Se asegura que STDIN se mantiene abierto. -t: Pseudo terminal (TTY)

Si ejecutamos **$ ps aux** solo vemos tres procesos. No vemos ningún proceso de nuestra máquina local. Pero lo curioso de esto es que si salimos a nuestra máquina local y ejecutamos **$ ps aux | grep app.js** encontraremos también este proceso corriendo en nuestro host OS. Esto demuestra que los procesos que corren en el contenedor también corren en nuestro host OS.

**Parar y eliminar un contenedor**

Para parar tu aplicación, ejecutamos lo siguiente:

**$ docker stop kubia-container**

Aún podemos ver el contenedor, aunque no está ya corriendo, con **$ docker ps -a**. Para eliminar el contenedor, usamos **$ docker rm kubia-container**.

**Haciendo push de nuestra imagen a un image registry**

La imagen que acabamos de hacer solo está disponible en nuestra máquina local. Para permitir que más gente pueda descargársela y correrla, necesitamos subirla a un registro externo de imágenes. Usaremos Docker Hub para almacenar nuestra imagen.

Para poder subir nuestra imagen al Hub de Docker, tenemos que retaggear nuestra imagen de acuerdo a las reglas de Docker. Hacemos:

**$ docker tag kubia docker\_id/kubia**

Como podéis ver, no retaggea la imagen, sino que se crea un tag adicional para la misma imagen. Se puede comprobar ejecutando **$ docker images**.

Para hacer push de nuestra imagen, debemos logearnos con **$ docker login**. Una vez logeados, ejecutamos:

**$ docker push docker\_id/kubia**

Ahora que la imagen está pusheada a Docker Hub, podemos correrla haciendo el comando que vimos al principio. Ahora está disponible para todo el mundo.

**$ docker run -p 8080:8080 -d docker\_id/kubia**

**OPCIONAL (ponerle nombre): $ docker run --name stolen-image -p 8080:8080 -d docker\_id/kubia**

KUBERNETES

Con la suma de más y más componentes para el despliegue y de grandes bases de datos, se vuelve muy difícil de configurar y mantener. Necesitamos una automatización que permita la organización de esos componentes a los servidores y mantenimiento automático. Aquí es donde entra Kubernetes.

**Preparando un cluster de Kubernetes**

Una vez empaquetada una imagen usando Docker , podemos desplegarla en un cluster de Kubernetes en vez de usar Docker directamente. Pero antes de nada hay que preparar ese cluster.

La forma más simple de preparar un cluster de Kubernetes es usando Minikube. Minikube es una herramienta que prepara un cluster de un único nodo que está bastante bien para testear Kubernetes y también para desarrollar aplicaciones en local.

Tal y como tengo configurada esta máquina, minikube ya ha preparado el cluster de Kubernetes. Esto se puede comprobar corriendo el siguiente comando:

**$ kubectl cluster-info**

**Corriendo la primera aplicación en Kubernetes**

La forma más simple de correr nuestra aplicación usando kubernetes es con un comando al estilo Docker. Ejecuta:

**$ kubectl run kubia --image=docker\_id/kubia --port=8080 --generator=run/v1**

Este comando creará los componentes necesarios para nuestra aplicación. No tendremos que lidiar con JSON o YAMLs. Aquí estamos ejecutando la imagen antes pusheada al Docker Hub y le estamos indicando el puerto de escucha también. En cuanto al último flag, le estamos indicando a Kubernetes que cree un replication controller en vez de un despliegue (deployment). Hablaré de los replication controllers más tarde (**¿???????**).

Si ejecutamos **$ kubectl get replicationcontroller** podremos ver que tenemos al nuestro.

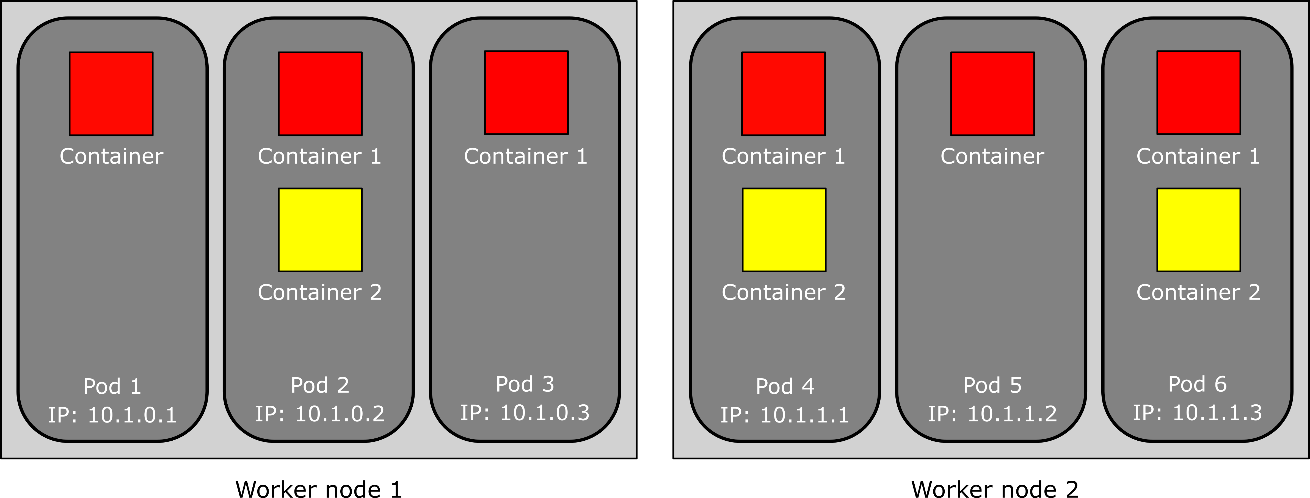
Ahora vamos a irnos a un nivel bastante bajo para partir de la base necesaria para entender Kubernetes.

**Introduciendo los Pods**

Quizás os preguntéis dónde podéis ver los contenedores en los que está corriendo nuestra aplicación. Quizás **$ kubectl get containers ???.** La cosa es que no es así como funciona Kubernetes exactamente. La unidad mínima que maneja Kubernetes se llama Pod.

Un pod es una agrupación de contenedores. Contiene uno o más contenedores. La cosa es que los pods suelen contener un solo contenedor. En caso de contener varios, estos estarán bastante relacionados.

Cada pod es como una máquina virtual aislada. Tiene su propia IP, hostname, procesos, etc corriendo una aplicación. Esta aplicación puede ser un único proceso (corriendo en un container) o una aplicación principal y subprocesos apoyándola (varios contenedores).



Los pods se distribuyen en los worker nodes, como podéis ver. No podemos listar los contenedores, pero si los pods.

**$ kubectl get pods**

Este es nuestro pod. En él se está corriendo nuestra aplicación ahora mismo. Ahora mismo está en estado **Pending** porque el worker node que se le ha asignado está descargando la imagen aún. Cuando la descarga finalice, empezará a correr nuestra aplicación.

Para saber más información acerca del pod, podemos correr el siguiente comando:

**$ kubectl describe pod pod-name**

**Entendiendo qué ha sucedido al correr la aplicación**

