

Esta clase va a ser

- grabada

**Clase 5.** Testing y Escalabilidad Backend

# Clusters & Escalabilidad

# Temario

4

## Logging y Testing de performance

- ✓ Loggers
- ✓ Testing de performance
- ✓ Testing avanzado de performance

5

## Clusters & Escalabilidad

- ✓ [Módulo Cluster](#)
- ✓ [Docker](#)
- ✓ [Docker como PM](#)

6

## Orquestación de contenedores

- ✓ DockerHub
- ✓ Orquestación de contenedores
- ✓ Orquestación con Kubernetes

# Objetivos de la clase

- Entender y aplicar el módulo de Cluster de Nodejs
- Conocer Docker
- Implementar Docker como un Process Manager

# Glosario

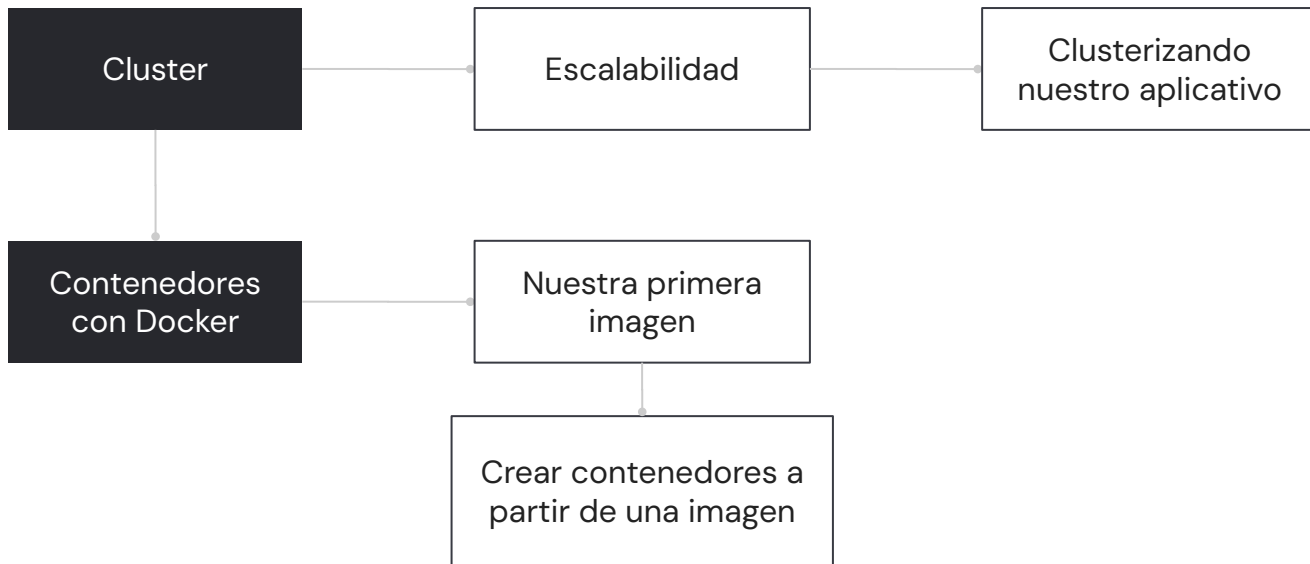
**Artillery:** Es un toolkit de performance que prueba nuestro servidor y corrobora su fiabilidad en un entorno real.

**Niveles:** permitirá priorizar “qué es importante mostrar” y “qué podría ignorar en algunos casos

**Transportes:** Los transportes nativos de Winston permiten que nuestros logs puedan salir de la consola y enviarse por otros medios.

**Winston Logger:** Winston es un logger diseñado para poder trabajar con multitransportes.

## MAPA DE CONCEPTOS



# Cluster

# Recordemos...

¿Recuerdas cómo intentamos hacer un test de performance con Artillery, aplicando “una operación sencilla” y “una operación compleja”?

Al hacer las operaciones complejas, se rechazaron muchas peticiones debido a que el servidor se saturaba y demoraba mucho en responder.

Seguramente en ese momento te enfrentaste a un reto de la realidad:  
**El servidor no puede atender todas las peticiones recurrentes, mucho menos estando con las manos tan ocupadas con operaciones bloqueantes.**

¡Pero las cosas no se pueden quedar así! ¡Deberíamos poder hacer algo! 🚀



# Estrategia: Escalabilidad

Cuando hablamos de “escalar” un servidor, lo hacemos a partir de dos conceptos:

✓ **Escalamiento vertical:** Mi servidor necesita ser más potente y necesito mejorar el hardware para tener un servidor más potente.

✓ **Escalamiento horizontal:** Dividamos las tareas en multi-instancias de servidores que alojen el aplicativo y se apoyen en las tareas complejas.

# Escalabilidad vertical

Básicamente, significa **mejorar el hardware** del servidor, para que sea más potente, mucho más rápido y pueda atender una mayor cantidad de peticiones y, por lo tanto, mejorar el performance de los aplicativos.

👉 El escalamiento vertical requiere de grandes inversiones de recursos por parte de las empresas para poder contar con los equipos más actualizados posibles en el mundo de la tecnología.

Además, llegará un punto en el que alcanzaremos un **tope tecnológico**, y tendremos que esperar a que se desarrollen mejores soluciones de hardware para poder comprarlas (un tiempo de espera que una empresa difícilmente puede contener).

# Escalabilidad horizontal

Este modelo es más complejo, pero mucho más interesante y eficiente.

La escalabilidad horizontal significa utilizar múltiples servidores, conocidos como **nodos**, los cuales trabajarán en equipo para resolver un problema en particular.

A esta red de **nodos** trabajando juntos, se le conoce como **cluster**, haciendo referencia a que estos múltiples servidores se encuentran en un contexto general donde todos conocen cómo ayudarse a las tareas más complejas.

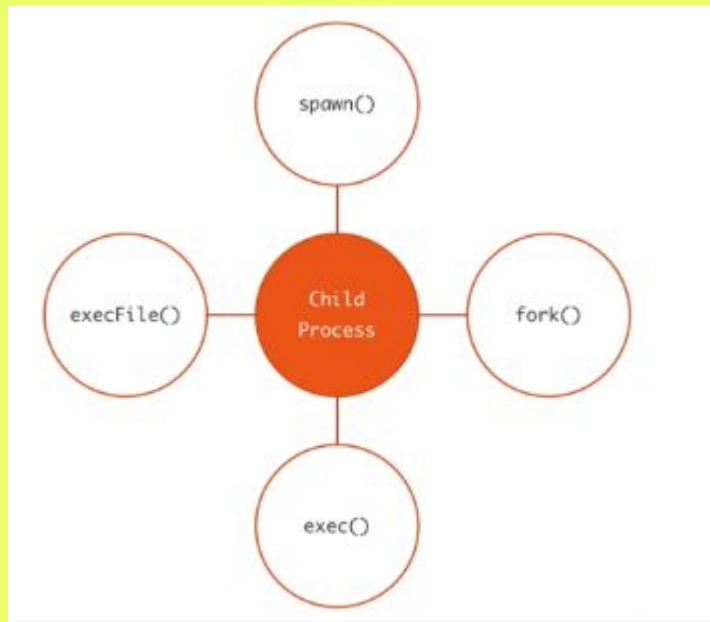
Así, la diferencia radica en que, cuando necesitamos más recursos, **no hace falta tirar el servidor que ya tenemos a la basura para comprar uno mejor**, sino que podemos conectar otra instancia de otro servidor para que se una a la red de **nodos** y forme parte del **cluster**.



# ¿Cómo clusterizar nuestro aplicativo?

Para poder configurar satisfactoriamente nuestro servidor a partir de un modelo horizontal, tenemos que recordar cómo funcionaba la gestión de los child process vistos en la clase 0.

**¿Recuerdas cómo funcionaba?**

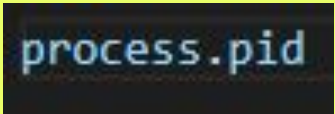


# Sobre el process id

Cuando un proceso se ejecuta, este tiene dentro de sus características principales una propiedad conocida como **pid**.

Este processId es muy importante para poder trabajar con otros procesos.

Cuando un proceso padre instanciaba un proceso hijo, este mantiene una referencia a partir del **pid**, haciéndole saber que ese proceso es parte de él.



```
process.pid
```

# Sobre el forkeo

Ahora, el proceso global podía generar el nuevo proceso a partir de 4 métodos principales, donde nosotros tuvimos la posibilidad de hablar sobre el forkeo.

La palabra **fork** será clave para hacer referencia a que un proceso nuevo surgirá, pero se mantendrá ligado al proceso que lo generó.

Anteriormente, llamábamos global process al proceso padre que **forkeaba** al proceso hijo.

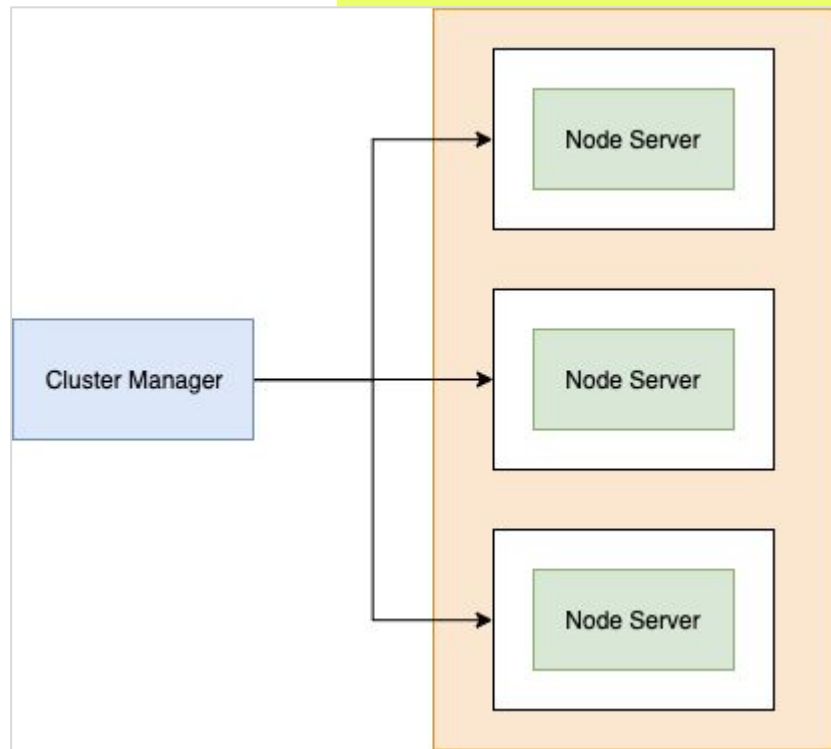
Sin embargo, esta vez conoceremos al proceso principal como **Primary process** (anteriormente llamado **Master**), mientras que a las múltiples instancias que se generen se llamarán **workers**.

# Clusterizando nuestro aplicativo

# Módulo nativo cluster

Cluster es un módulo nativo de node js que nos permitirá ejecutar este concepto de clusterización que recién comentamos, donde podremos tener a un proceso principal contando con un grupo de procesos trabajadores.

Estos trabajadores van a trabajar en conjunto para resolver el problema de las situaciones de las peticiones.



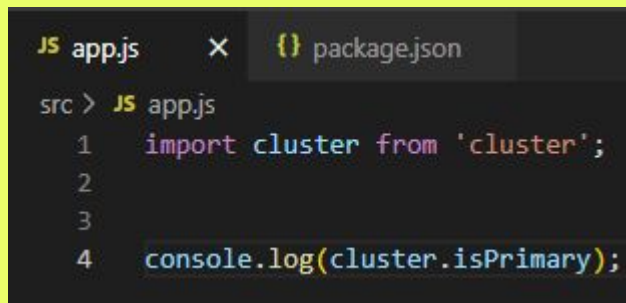


# isPrimary?

Vamos a importar el módulo nativo `cluster` en nuestro archivo `app.js` y analizaremos primero nuestra primera propiedad principal: **isPrimary**

Esta propiedad nos ayuda a corroborar si el proceso es el principal, o viene forkeado de algún proceso superior.

**Cuidado:** si tu versión de node es 16+, podrás utilizar `isPrimary` sin problema, si es anterior deberás utilizar **isMaster**



```
JS app.js  X  {} package.json
src > JS app.js
1  import cluster from 'cluster';
2
3
4  console.log(cluster.isPrimary);
```



true

# Realizamos un forkeo desde cluster

Para poder generar nuestro primer proceso trabajador, vamos a hacerlo solo desde el proceso principal, entonces, la lógica es:

- ✓ Si eres un proceso primario, entonces indica que eres el principal y forkea a un trabajador.
- ✓ Si eres un proceso trabajador, entonces indica que eres trabajador y procede a realizar las tareas que corresponden.

# Diferenciando primary de workers

```
JS app.js  X  {} package.json
src > JS app.js
1  import cluster from 'cluster';
2
3  if(cluster.isPrimary){
4      console.log("Proceso primario, generando proceso trabajador");
5      cluster.fork();
6  }
7  else{
8      console.log("Al ser un proceso forkeado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!");
9  }
10 }
```

Notamos cómo primero se comunica el proceso primario, pero al realizar el forkeo, se comunica el proceso secundario.

```
Proceso primario, generando proceso trabajador
Al ser un proceso forkeado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!
```

# Aprovechando la capacidad de un computador

Sabemos que el procesamiento de nuestro servidor será siempre single-threaded. Al realizar nuestros primeros forkeos, estamos comenzando a romper el paradigma que implica.

¿Cómo levantar múltiples instancias, sin que afecte demasiado en tamaño?

Para ello, lo primero debería ser determinar el número de hilos que podrán procesar estos multiprocesamientos, esto lo conseguiremos con las siguientes líneas

```
import { cpus } from 'os';  
  
const numeroDeProcesadores = cpus().length;  
console.log(numeroDeProcesadores);
```

En el caso de este ejemplo, trabajaremos con

16

# Generando múltiples trabajadores

```
JS app.js  x  {} package.json
src > JS app.js > ...
1  import cluster from 'cluster';
2  import { cpus } from 'os';
3
4  const numeroDeProcesadores = cpus().length;
5
6  if(cluster.isPrimary){
7    console.log("Proceso primario, generando proceso trabajador");
8    for( let i = 0; i<numeroDeProcesadores;i++){
9      cluster.fork()
10    }
11  }
12  else{
13    console.log("Al ser un proceso forkeado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!")
14    console.log(`Me presento, soy un proceso worker con el id : ${process.pid}`)
15  }
16 }
```

```
Proceso primario, generando proceso trabajador
Al ser un proceso forkeado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!
Me presento, soy un proceso worker con el id : 18344
Al ser un proceso forkeado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!
Me presento, soy un proceso worker con el id : 14824
Al ser un proceso forkeado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!
Me presento, soy un proceso worker con el id : 17392
Al ser un proceso forkeado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!
Me presento, soy un proceso worker con el id : 14836
Al ser un proceso forkeado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!
Me presento, soy un proceso worker con el id : 18272
Al ser un proceso forkeado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!
Al ser un proceso forkeado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!
Me presento, soy un proceso worker con el id : 17812
Me presento, soy un proceso worker con el id : 15872
Al ser un proceso forkeado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!
```

# ¿Por qué tener múltiples procesos?

```
const numeroDeProcesadores = cpus().length;

if(cluster.isPrimary){
  console.log("Proceso primario, generando proceso trabajador");
  for( let i = 0; i<numeroDeProcesadores;i++){
    cluster.fork()
  }
}
else{
  console.log("Al ser un proceso forkeado, no cuento como primario, por lo tanto isPri");
  console.log(`Me presento, soy un proceso worker con el id : ${process.pid}`)
  const app = express();

  app.get('/',(req,res)=>{
    res.send({status:"success", message:"Petición atendida por un proceso worker"})
  })

  app.listen(8080,()=>console.log("Listening on 8080"))
}
```

```
Listening on 8080
Me presento, soy un proceso worker con el id : 18132
Listening on 8080
Al ser un proceso forkeado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!
Me presento, soy un proceso worker con el id : 10212
Listening on 8080
Listening on 8080
Al ser un proceso forkeado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!
Me presento, soy un proceso worker con el id : 12276
Listening on 8080
Listening on 8080
Al ser un proceso forkeado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!
Me presento, soy un proceso worker con el id : 1476
Listening on 8080
Al ser un proceso forkeado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!
Me presento, soy un proceso worker con el id : 17248
Al ser un proceso forkeado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!
Me presento, soy un proceso worker con el id : 17468
Listening on 8080
Listening on 8080
```

¡Entonces viene lo maravilloso! ¿Recuerdas que comentamos la importancia de que todos estuvieran conectados en un contexto general? ¡Nota cómo lo utilizamos para que cada quien cuente con una instancia del servidor!

# ¿Qué ocurrió?

En este momento hay múltiples procesos escuchando y trabajando sobre el servidor, en realidad, cada uno es una instancia de servidor.

¿Cómo reconocemos que esto sea cierto?

Vamos a utilizar un comando muy útil en windows conocido como

**tasklist /fi "imagenname eq node.exe"**

El comando en negritas significa que encuentre una imagen (instancia de proceso) del proceso **node.exe**

Al ejecutarlo, podemos ver los procesos que corresponden a nuestros respectivos contextos.

Nombre de imagen	PID	Nombre de sesión	Núm. de ses	Uso de memor
node.exe	5344	Console	1	31,692 KB
node.exe	14224	Console	1	31,504 KB
node.exe	9412	Console	1	31,532 KB
node.exe	3740	Console	1	31,388 KB
node.exe	16480	Console	1	31,188 KB
node.exe	4036	Console	1	31,460 KB
node.exe	10784	Console	1	31,748 KB
node.exe	15708	Console	1	31,520 KB
node.exe	18412	Console	1	31,596 KB
node.exe	18132	Console	1	31,500 KB
node.exe	12216	Console	1	31,488 KB
node.exe	10212	Console	1	31,788 KB
node.exe	312	Console	1	31,376 KB
node.exe	12276	Console	1	31,852 KB
node.exe	1476	Console	1	31,616 KB
node.exe	17248	Console	1	31,520 KB
node.exe	17468	Console	1	31,680 KB

# ¡Hay más!

No solo generamos múltiples instancias del servidor, sino que podemos escuchar a eventos desde el proceso primario.

Entonces, seguro te recuerda al proceso de comunicación de socket.io, ya que en el proceso padre podremos colocar `.on` para saber qué pasa con alguno de sus trabajadores.

```
cluster.on(['',  
e{  
  console.log(  
  console.log(  
  const app =  
  app.get('/',  
  disconnect  
  exit  
  fork  
  listening  
  message  
  online  
  setup
```

```
cluster.on('message', worker=>{  
  console.log("Mensaje recibido desde el worker: "+worker.process.pid)  
})
```



# ¡A esto nos referíamos!

Nota que el proceso primario no solo se está encargando de levantar múltiples procesos, sino que también nos permitirá tener un control sobre estos. Ahora vamos a ponerlo a prueba con las operaciones que ya conocemos.

```
else{
  console.log("Al ser un proceso forkeado, no cuento como primario, por lo tanto isPrimary=false. ¡Entonces soy un worker!")
  console.log(`Me presento, soy un proceso worker con el id : ${process.pid}`)
  const app = express();

  app.get('/operacionsencilla',(req,res)=>{
    let sum = 0;
    for(let i = 0; i<10000;i++){
      sum+=i;
    }
    res.send({status:"success", message:`El worker ${process.pid} ha atendido esta petición, el resultado es ${sum}`})
  })

  app.get('/operacioncompleja',(req,res)=>{
    let sum = 0;
    for(let i = 0; i<5e8;i++){
      sum+=i;
    }
    res.send({status:"success", message:`El worker ${process.pid} ha atendido esta petición, el resultado es ${sum}`})
  })
}
```

# Ejecutando el proceso de Artillery para las operaciones sencillas

```
artillery quick --count 40 --num 50 "http://localhost:8080/operacionsencilla" -o resultadosSencillos.json
```

```
{  
  "histograms": {  
    "http.response_time": {  
      "min": 0,  
      "max": 13,  
      "count": 2000,  
      "p50": 7,  
      "median": 7,  
      "p75": 7.9,  
      "p90": 8.9,  
      "p95": 10.9,  
      "p99": 12.1,  
      "p999": 13.1  
    }  
  },  
}
```

¡Impresionantes resultados! Nota que ahora el tiempo mínimo de resolución de la operación sencilla es tan bajo que Artillery lo ha dejado en 0. y el tiempo máximo es de 13 milisegundos (casi nada)

La operación sencilla era fácil de afrontar, pero ahora falta el verdadero reto: La operación compleja sólo nos permitió finalizar 10 de la meta de 2000 peticiones, veamos cómo se comporta en esta ocasión

# Ejecutando el proceso de Artillery para las operaciones complejas

```
artillery quick --count 40 --num 50 "http://localhost:8080/operacioncompleja" -o resultadosComplejos.json
```

```
JS app.js {} resultadosComplejos.json X {} package.json
{} resultadosComplejos.json > {} aggregate > {} counters
1  {
2  "aggregate": {
3  "counters": {
4    "vusers.created_by_name.0": 40,
5    "vusers.created": 40,
6    "http.requests": 2000,
7    "http.codes.200": 2000,
8    "http.responses": 2000,
9    "vusers.failed": 0,
10   "vusers.completed": 40
11  },
12  "rates": {
13    "http.request_rate": 5
14  },
15  }
```

Voilà! Muy seguramente tu computadora ha tenido que enfrentar una dura y larga batalla para resolver las operaciones complejas, sin embargo, notamos que lograron resolver las 2000 peticiones y no se devolvió ningún TIMEOUT como lo hacía cuando sólo había una instancia del servidor tratando de cargar con todo el trabajo.

# Los console.log demuestran cómo los diferentes procesos trabajan en conjunto

```
El worker 19232 ha atendido esta petición, el resultado es 124999999567108900
El worker 18596 ha atendido esta petición, el resultado es 124999999567108900
El worker 13988 ha atendido esta petición, el resultado es 124999999567108900
El worker 18308 ha atendido esta petición, el resultado es 124999999567108900
El worker 17660 ha atendido esta petición, el resultado es 124999999567108900
El worker 10212 ha atendido esta petición, el resultado es 124999999567108900
El worker 6756 ha atendido esta petición, el resultado es 124999999567108900
El worker 11564 ha atendido esta petición, el resultado es 124999999567108900
El worker 17268 ha atendido esta petición, el resultado es 124999999567108900
El worker 4392 ha atendido esta petición, el resultado es 124999999567108900
El worker 12636 ha atendido esta petición, el resultado es 124999999567108900
El worker 16872 ha atendido esta petición, el resultado es 124999999567108900
El worker 11460 ha atendido esta petición, el resultado es 124999999567108900
□
```

# Para pensar

El ejemplo presentado en estas diapositivas fue realizado con un computador de 16 núcleos, lo cual significa que tuvimos un equipo de 16 servidores resolviendo el problema de las operaciones complejas.

Si tu computador tenía más o menos núcleos, **¿qué tanto cambió el resultado?**



# Estabilizador de workers

Duración: 5 – 10 minutos



## ACTIVIDAD EN CLASE

# Estabilizador de workers

### Ahora que comprendemos sobre los clusters

- ✓ Crear un servidor de express que levante **n** workers según sea el número de cpus() de tu computador.
- ✓ Confirmar en tu consola cuántos procesos con el nombre de imagen **node.exe** existen.
- ✓ Configurar los listeners del proceso primario para que, si alguno de sus workers muere en alguna operación o falla, el proceso primario cree una nueva instancia para siempre tener **n** número de workers estables.
- ✓ Hacer pruebas matando un worker con el comando **taskkill /pid PID -f** `taskkill /pid 14400 -f`
- ✓ Confirmar que el proceso primario haya creado el nuevo proceso y que el número de workers se mantenga estable.





# Break

¡10 minutos y volvemos!



# Contenedores con Docker

# ¡Atención!

Recuerda instalar **curl** para la próxima clase.

[¿Qué estoy por instalar?](#)



[Página oficial](#)

# Docker

Docker es una plataforma que permitirá crear, probar e implementar aplicativos en unidades de software estandarizadas llamadas **contenedores**.

Con docker, podremos “virtualizar” el sistema operativo de un servidor con el fin de realizar ejecuciones de aplicaciones con la máxima compatibilidad.

Gracias a tener nuestro aplicativo en un contenedor que corra un software con exactamente las especificaciones que necesita esta app, evitamos el típico problema del desarrollador **“en mi computadora sí funcionaba”**

- En mi computadora si funciona
- Si pero no le vamos a dar tu computadora al cliente



# ¿Por qué mi aplicativo puede no funcionar al llegar al cliente?

El desarrollo de un aplicativo no es tan “ideal” como pensamos. Estamos hablando de múltiples desarrolladores haciendo código por su parte, para que al final “todo se una en un único código final.

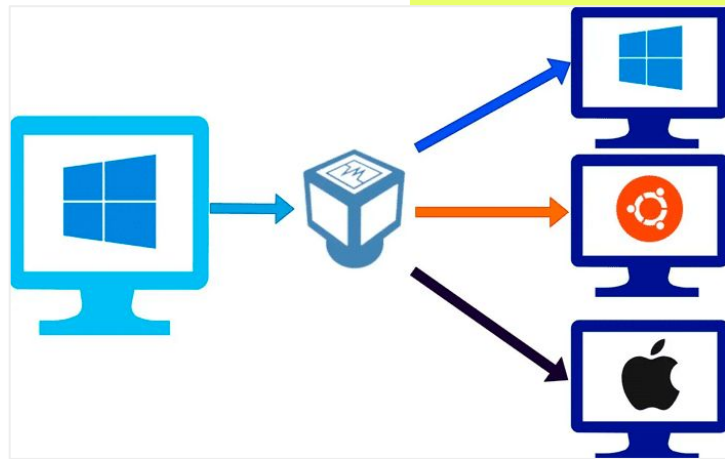
¿Cuáles podrían ser los puntos de dolor?

Cada computadora es un mundo, donde existe exactamente una versión específica de alguna librería, o exactamente una configuración particular del sistema, o exactamente una versión del entorno en general donde se ejecuta.

**Necesitaríamos que todas las computadoras fueran exactamente iguales para asegurar una extrema compatibilidad en lo que ejecutamos.**

# ¿Qué son las máquinas virtuales?

Como primera solución se plantearon las máquinas virtuales. **Grosso modo**, podemos decir que es cuando nuestra computadora utiliza sus recursos para **simular otro sistema operativo**. Eso significa que estamos “engañando” a una computadora, haciendo creer que está corriendo sobre un entorno o sistema particular (y, en esencia, lo hace), sin embargo, todo esto en realidad está dentro de nuestra computadora principal, utilizando recursos de la computadora principal.



# El problema de las máquinas virtuales



El cloud computing se basa en máquinas virtuales. Sin embargo, para desarrollo de aplicativos, realmente no parece ser la mejor opción contar con **todo un sistema operativo**, solo para una simple aplicación.

La idea debe estar entonces en ejecutar entornos que tengan **únicamente las configuraciones** necesarias para ejecutar una aplicación, y nada más. Esto es llamado **contenedor**.

# Contenedor

Un contenedor es un **entorno de ejecución para un aplicativo en particular**, el cual tiene todas las dependencias que necesita dicha aplicación para poder correr sin problemas de compatibilidad.

La clave de un **contenedor es el concepto del aislamiento**, esto indicando que podemos tener múltiples contenedores, con diferentes entornos, con diferentes dependencias, y nunca habrá conflictos porque la instalación y uso de las dependencias se hace de manera interna.

Además, ya que el entorno no ocupa utilizar todo el sistema operativo (sólo el **kernel**), se vuelven realmente livianos en comparación con mover todo un sistema operativo en cada aplicativo.

# El papel de Docker en el mundo de contenedores

Docker es una plataforma gestora de contenedores. Nos permitirá entonces empaquetar en un contenedor nuestro aplicativo, y posteriormente compartirlo a algún lado, para que al momento en el que tenga que ejecutarse, este pueda hacerlo dentro del contenedor aislado y asegurar que la ejecución será satisfactoria siempre.

La lógica de Docker se basa en tres pasos generales:





1

### Paso 1

Un dockerfile:  
Este cuenta con las  
instrucciones paso a  
paso para que  
nuestro proyecto  
genere una imagen.

2

### Paso 2

Una imagen es el  
equivalente de una **clase**,  
pero con un proyecto  
completo. Cuando  
generamos la imagen de  
una aplicación, significa  
que podemos generar  
múltiples contenedores a  
partir de esa aplicación  
(como **instancias**)

3

### Paso 3

Contenedor:  
El punto final en el  
que ejecutamos el  
aplicativo, pero esta  
vez desde un entorno  
cerrado.

# **Primer acercamiento a Docker: nuestra primera imagen**

# 1. Descarga de la página oficial

El paso más simple, vamos a descargar Docker Desktop a partir del sistema que necesitemos (En este caso la instalación será para Windows).

**Develop faster. Run anywhere.**

The most-loved Tool in Stack Overflow's 2022 Developer Survey.

Download Docker Desktop

Windows



Apple Chip



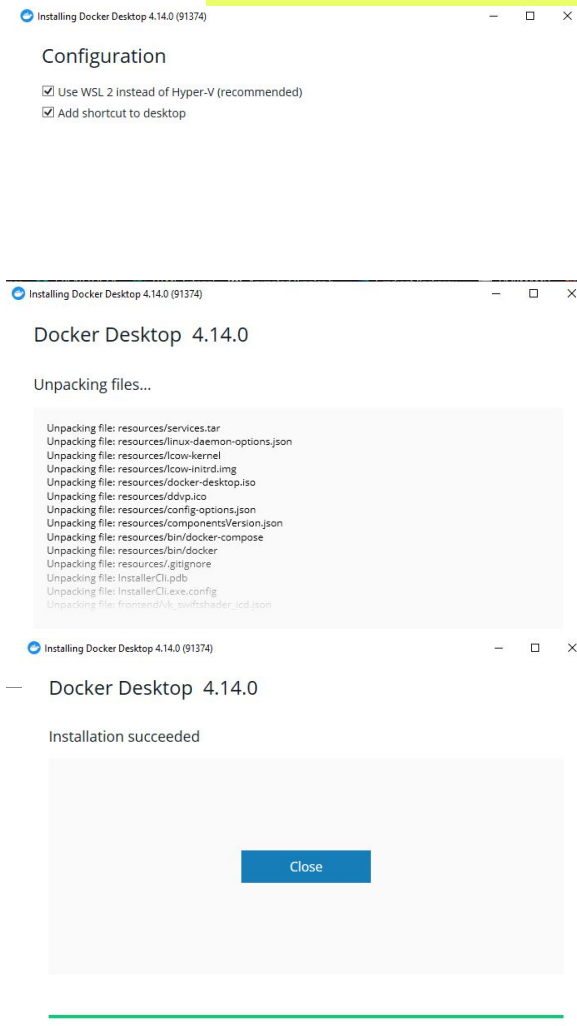
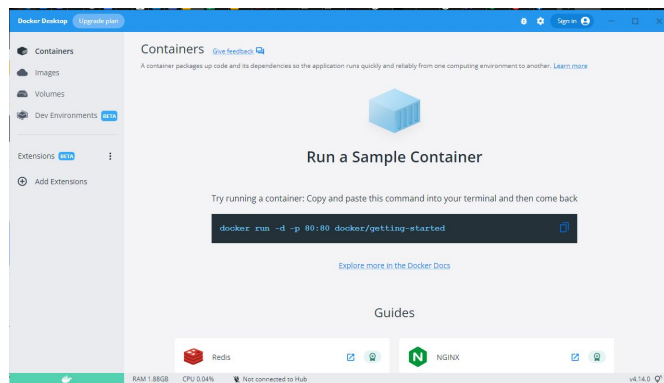
Linux



Intel Chip

# 2. Instalando

El instalador es bastante sencillo, al final Docker revisará que nuestro computador tenga **activada la opción de virtualización de Hardware**. Esta configuración se realiza desde el BIOS y es variable en cada computadora. Una vez que docker reconoce que podemos virtualizar, nos mostrará una pantalla como ésta:



# ¡Importante!

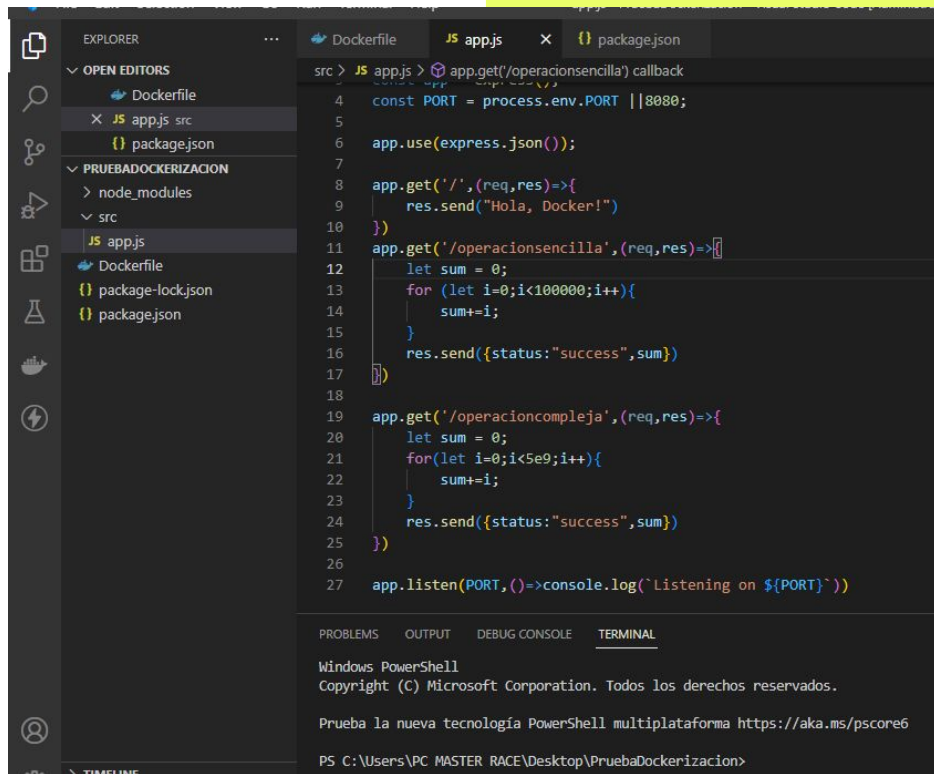
Si al momento de la instalación obtienes un error de Docker al querer inicializar los contenedores, significa que **necesitas activar la virtualización de hardware desde tu BIOS.**

Ya que es una configuración que requiere el reinicio del computador y entrar a la BIOS, es por ello que se solicitó su instalación previamente

# 3. Crear un Dockerfile en un proyecto

Tomaremos el proyecto que hace nuevamente la operación sencilla y la operación compleja, agregando el saludo de docker desde el endpoint de la ruta base. En dicho proyecto, crearemos un Dockerfile, que será el punto de partida de nuestra imagen.

**CoderTip:** También se recomienda contar con una extensión de Docker instalada en tu Visual Studio Code.



The screenshot shows the Visual Studio Code interface. On the left, the Explorer sidebar displays the project structure: 'OPEN EDITORS' (Dockerfile, JS app.js src, package.json), 'PRUEBA DOCKERIZACION' (node\_modules, src), and 'src' (JS app.js, Dockerfile, package-lock.json, package.json). The main editor area shows the 'Dockerfile' file with the following content:

```
src > JS app.js > app.get('/operacionsencilla') callback
4  const PORT = process.env.PORT || 8080;
5
6  app.use(express.json());
7
8  app.get('/', (req, res) => {
9    res.send("Hola, Docker!");
10 })
11 app.get('/operacionsencilla', (req, res) => {
12   let sum = 0;
13   for (let i = 0; i < 100000; i++) {
14     sum += i;
15   }
16   res.send({ status: "success", sum });
17 })
18
19 app.get('/operacioncompleja', (req, res) => {
20   let sum = 0;
21   for (let i = 0; i < 5e9; i++) {
22     sum += i;
23   }
24   res.send({ status: "success", sum });
25 })
26
27 app.listen(PORT, () => console.log(`Listening on ${PORT}`))
```

At the bottom, the TERMINAL panel shows the Windows PowerShell prompt with the following text:

```
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Prueba la nueva tecnología PowerShell multiplataforma https://aka.ms/pscore6

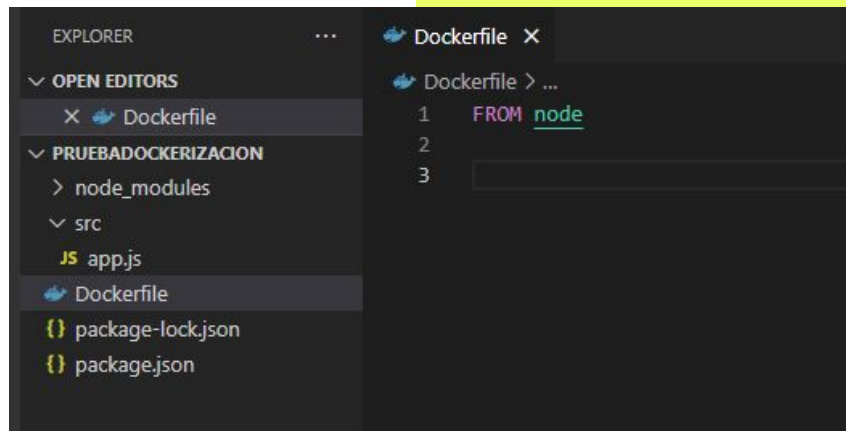
PS C:\Users\PC MASTER RACE\Desktop\Prueba Dockerizacion>
```

# 4. ¿Qué es una imagen base?

Escribimos en nuestro código:  
**FROM node.**, pero ¿qué significa?

Recuerda que una imagen es una plantilla para generar contenedores, es decir, hay muchas plantillas en las cuales podemos basarnos, para tener la configuración base de un entorno (que en este caso es node), y a partir de este comenzar a correr configuración más específica hacia nuestro proyecto.

FROM node, entonces, significa que estaremos tomando una imagen base del entorno de node, para poder configurar nuestra app.





## Para pensar

Las imágenes base ya existen, deben tomarse de algún lado. ¿De dónde se descarga nuestra imagen para instalarse?

**¡Existe un repositorio de imágenes!**

Te invitamos a pasar por [DockerHub](#)



# 5. Comenzamos a escribir en nuestro Dockerfile

Después del **FROM**, escribiremos el resto de la configuración:

**WORKDIR** Será nuestro directorio de trabajo principal, donde comenzaremos a crear todo. Las operaciones que hagamos más abajo se harán sobre este directorio

**COPY** permitirá copiar archivos de la carpeta donde estamos ejecutando el Dockerfile, y pegarlos en la carpeta que hayamos creado con **WORKDIR**

```
Dockerfile x {} package.json
Dockerfile > ...
1 #Primero definimos una imagen base: node
2 FROM node
3
4 #Después creamos una carpeta interna donde vamos a guardar nuestro proyecto (usualmente es app)
5 WORKDIR /app
6
7 #Con esto, copiamos el package.json de nuestra carpeta actual, a la carpeta dockeroperations
8 COPY package*.json ./
9
10 #Una vez copiado el package.json, procedemos a ejecutar un npm install interno en esa carpeta.
11 RUN npm install
12
13
14 #Después de la instalación, procedemos a tomar todo el código del aplicativo
15 COPY . .
16
17 #Exponemos un puerto para que éste escuche a partir de un puerto de nuestra computadora.
18 EXPOSE 8080
19
20 #Una vez realizado, se deberá ejecutar "npm start" para iniciar la aplicación (ten listo el comando en tu package.json)
21 CMD ["npm", "start"]
```

# 5. Comenzamos a escribir en nuestro Dockerfile

**RUN** nos permitirá ejecutar comandos. Al usar la imagen base **node**, significa que el entorno podrá correr comandos de node y npm sin problema.

**CMD** al final es la ejecución del comando final que se utilizará al momento de echar a andar el servidor cuando hagamos **docker run**

```
Dockerfile x package.json
Dockerfile > ...
1 #Primero definimos una imagen base: node
2 FROM node
3
4 #Después creamos una carpeta interna donde vamos a guardar nuestro proyecto (usualmente es app)
5 WORKDIR /app
6
7 #Con esto, copiamos el package.json de nuestra carpeta actual, a la carpeta dockeroperations
8 COPY package*.json ./
9
10 #Una vez copiado el package.json, procedemos a ejecutar un npm install interno en esa carpeta.
11 RUN npm install
12
13
14 #Después de la instalación, procedemos a tomar todo el código del aplicativo
15 COPY . .
16
17 #Exponemos un puerto para que éste escuche a partir de un puerto de nuestra computadora.
18 EXPOSE 8080
19
20 #Una vez realizado, se deberá ejecutar "npm start" para iniciar la aplicación (ten listo el comando en tu package.json)
21 CMD ["npm","start"]
```

# 6. Ejecutamos el build

Una vez configurado nuestro respectivo dockerfile, podemos poner a prueba éste ejecutando el comando build.

El comando build leerá el archivo y comenzará con la construcción de la imagen para nuestro aplicativo.

Una vez que tenga la imagen del aplicativo, necesita colocarle un nombre, la flag **-t** significa "tag" y es para nombrar la imagen.

el punto **.** Sirve para indicarle que el **dockerfile** que necesitamos que lea está en la misma ubicación donde estamos corriendo el comando

```
docker build -t dockeroperations .
```

```
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 32B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/node:latest
=> [1/5] FROM docker.io/library/node@sha256:743707dbaca64ff4ec8673a6e0c4d03d048e32b4e8ff3e89e
=> [internal] load build context
=> => transferring context: 27.69kB
=> CACHED [2/5] WORKDIR /app
=> CACHED [3/5] COPY package*.json ./
=> CACHED [4/5] RUN npm install
=> CACHED [5/5] COPY . .
=> exporting to image
=> => exporting layers
=> => writing image sha256:a13f7eb192efbdf0d43b55a2ab79d0032a5d10c6e8ec1d76ac6fba834e6ce32c
=> => naming to docker.io/library/dockeroperations
```

# ¿Funcionó?

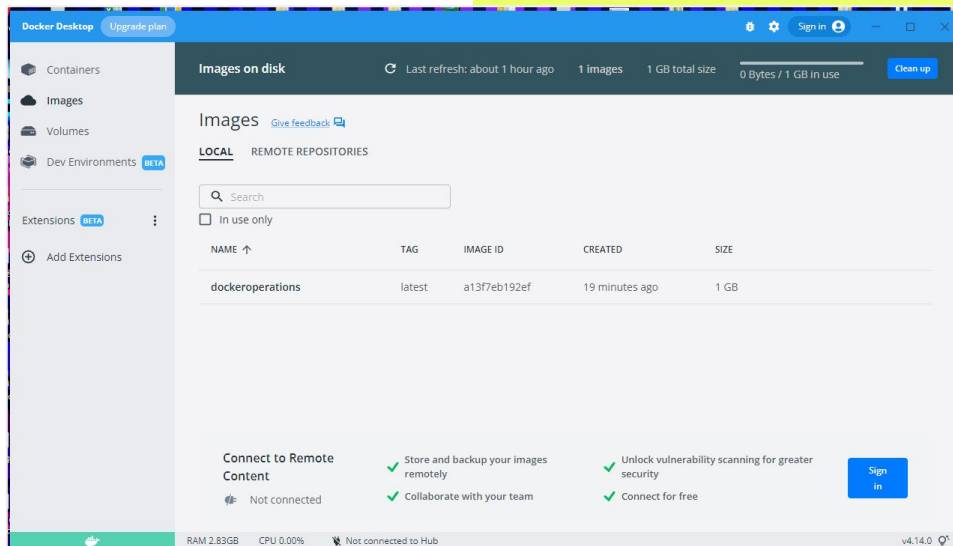
Para poder corroborar que el build se haya realizado correctamente, podemos ejecutar el comando `docker images`

Si ejecutamos el comando, deberíamos ver la imagen que construimos en la consola.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
dockeroperations	latest	a13f7eb192ef	10 hours ago	1GB

También podemos ver la imagen desde **Docker Desktop**, en la sección images

Solo nos queda hacer una cosa más: instanciar un contenedor a partir de esta imagen.



# Creando contenedores a partir de una imagen

# Creando contenedor desde CLI

```
> pruebaDockerizacion@1.0.0 start  
> node src/app.js
```

```
Listening on 8080
```

Hola, Docker!

La primera forma para poder crear un contenedor es desde un comando, podemos ejecutar:

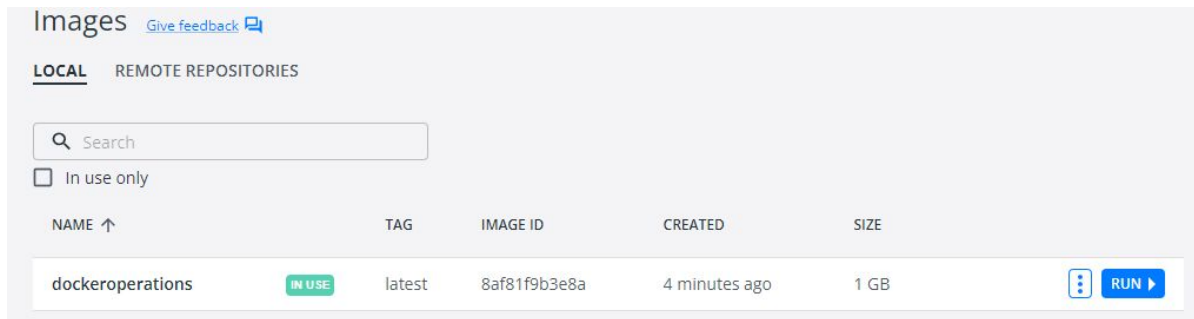
```
docker run -p 8080:8080 dockeroperations
```

¿Por qué esa sintaxis del puerto? Recuerda que docker es un contenedor aislado, de manera que su “puerto interno 8080 (al que le hicimos EXPOSE)”, en realidad no existe en “el mundo real de nuestra computadora”. Entonces, necesitamos proveer un puerto para que realmente funcione en el exterior.

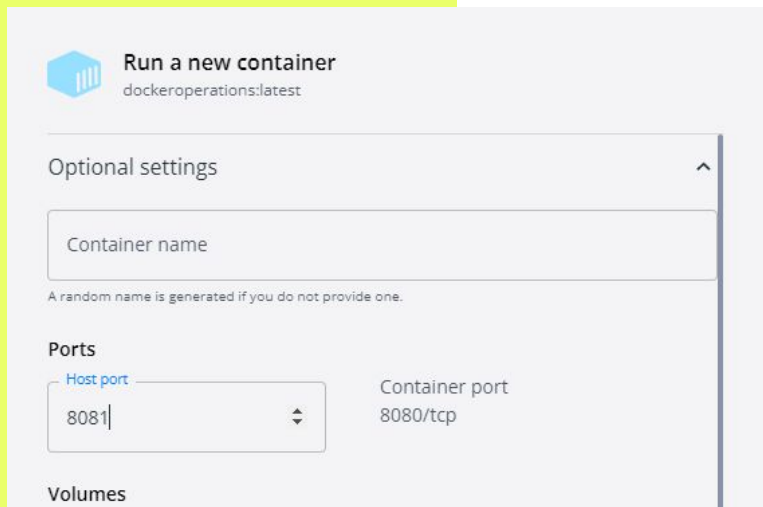
Es decir, de manera interna, el contenedor escucha en 8080, pero nosotros decidimos a qué puerto real lo conectamos.

# Creando contenedor desde Docker Desktop

También podemos ir a Docker Desktop y seleccionar la opción "Run"



# Creando contenedor desde Docker Desktop



The screenshot shows the 'Run a new container' window in Docker Desktop. At the top left is the Docker logo and the text 'Run a new container' with 'dockeroperations:latest' below it. The main section is titled 'Optional settings' with an upward arrow. It contains a 'Container name' input field with a note: 'A random name is generated if you do not provide one.' Below this is the 'Ports' section, which has a 'Host port' input field containing '8081' and a 'Container port' field containing '8080/tcp'. At the bottom is a 'Volumes' section.

Es importante tener conectado también un puerto de escucha al contenedor que estamos por crear, asignamos el 8081 (porque el otro lo utilizamos desde el CLI).



# Tenemos dos contenedores, ejecutando el servidor

Cada servidor es independiente, pero ahora tenemos escuchando estos aplicativos en dos puertos: 8080 y 8081,

<input type="checkbox"/>	NAME	IMAGE	STATUS	PORT(S)	STARTED	ACTIONS
<input type="checkbox"/>	 agitated_euler 87cc29c9e03c 	<a href="#">dockeroperations:latest</a>	Running	<a href="#">8080:8080</a> 	19 minutes ago  	
<input type="checkbox"/>	 inspiring_keldysh 09afef2e82c3 	<a href="#">dockeroperations:latest</a>	Running	<a href="#">8081:8080</a> 	8 minutes ago  	

# Los contenedores como instancias de la imagen

Recordando algo de Programación Orientada a Objetos (POO), sabemos que una clase permite generar múltiples objetos idénticos, pero con su propia identidad al final.

Cuando levantamos un contenedor, prácticamente estamos haciendo lo mismo. La imagen puede replicar múltiples veces el mismo proyecto, y así tener múltiples instancias del servidor.

¿Por qué es esto útil? Cuando hablamos de escalabilidad, tenemos que comenzar a pensar en múltiples procesos que puedan apoyarse entre sí, con el fin de poder atender un mayor número de peticiones. Con el modelo que recién creamos, si en algún momento necesitamos mayor potencia de procesamiento, podemos levantar un contenedor adicional para que se una al equipo.

Recuerda que esto es conocido como **escalamiento horizontal**.

¿Preguntas?

# ¡Atención!

Recuerda instalar **curl** para la próxima clase.

[¿Qué estoy por instalar?](#)



[Página oficial](#)

**Muchas gracias.**

# Resumen de la clase hoy

- ✓ Clusterización
- ✓ Pruebas de performance en servidor clusterizado.
- ✓ Imágenes y contenedores con Docker

**Opina y valora**  
esta clase

**Educación digital  
para el mundo real.**