Introducción

jueves, 16 de noviembre de 2023
20:14

Esta materia se centra en proporcionar a los estudiantes los conocimientos y habilidades necesarios para construir, analizar y evolucionar software de manera efectiva.

Partiendo de lo que son buenas practicas como evitar los Code Smells

Code Smells

martes, 14 de noviembre de 2023
6:49

Recordar: Notación Asintótica

Contexto

Cuando implementamos la programación en Fundamentos de Programación, Programación Orientada a Objetos , aprendimos que hay ciertos patrones de buenas prácticas que debemos aprender desde el inicio.

Definición

Es un término utilizado en el desarrollo de software para describir patrones de código que pueden indicar la presencia de problemas o malas prácticas en el diseño del software.

Estos no son errores específicos, sino más bien señales de que puede haber problemas en el código que podrían afectar su mantenibilidad, legibilidad o rendimiento.

NUETRO CODIGO DEBE SER EFICIENTE MAS NO EFICAZ

WHY?

Debemos tener buenas prácticas para que el código que escribamos sea legible, entendible y mantenible para nosotros y para otros.

Debemos comprender la eficiencia de los códigos que creemos siempre pensando en la escalabilidad, porque con pocos datos los defectos no se notan pero a grandes datos se nota la deficiencia.

Clean Code

"El Clean Code hace una cosa bien"

Es la práctica de escribir código de software de manera que sea fácil de entender, mantener y modificar. Se centra en producir código legible y elegante, siguiendo principios y convenciones que favorecen la claridad y la simplicidad.

- Cuando creamos el Diseño de Software, buscamos equilibrar las características de
  - **Cohesión**
  - **Acoplamiento**

  que son relacionadas a las dependencias mínimas. Cuando realicemos cambios debemos ver que no solo cambia en esa parte del código sino como puede tener efectos colaterales.

- El código debe estar preparado para cualquier escenario que se enfrente nuestro código.

Algunos principios son:

Nombres significativos para variables y funciones

- Son aspectos y puntos que ya hemos tratado:

- Nombres que revelen intenciones
- Evitar desinformación
- Distinciones con sentido
- Trabajar con nombres que se puedan entender
- Nombres que se puedan pronunciar
- Nombres fáciles de localizar
- Evitar asignaciones mentales
- Evitar que los nombres tengan que entenderse con comentarios :ej //ppp significa prestamos por pagar, pudiendo usar prestamosPagar

  La parte de Nombres que revelen intenciones tiene ciertas excepciones, pero esto excepto en las excepciones ya que se usan variables que no tienen relevancia ni influencia más allá del bucle de donde están.

  Un claro ejemplo:

  i j k

  Otras reglas referente a nombrar:
- Reglas de nombrar clases
- Reglas para nombrar métodos

Funciones - Métodos

En las funciones deben cumplir

- Tamaño reducido
- Bloques - sangría
- Enfocadas en una cosa
- Número de argumentos : que no te pases de tantos argumentos
- Separación: consultas - comandos
- Efectos secundarios
- Uso de Excepciones
- Manejo de bloques Try - Catch

- ## Formato vertical

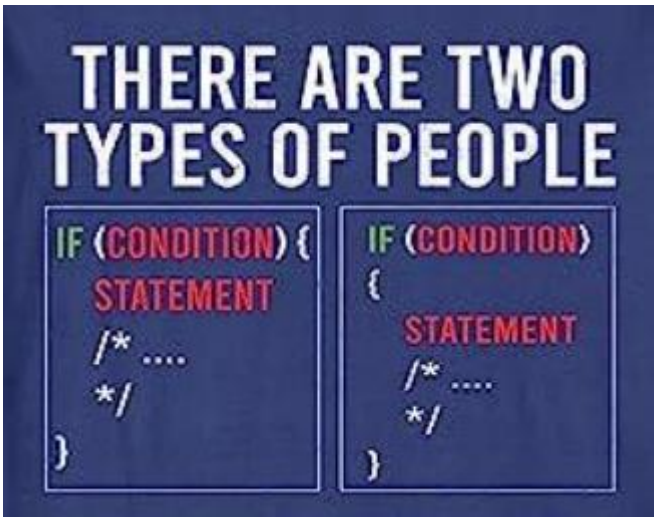
La extensión del código de forma vertical

- Densidad: cuantas líneas de código
- Distancia: distancia entre líneas
- Dependencia entre funciones: las funciones que están estrechamente relacionadas deben estar cerca.
- Afinidad de concepto
  - El agregar comentarios a lo menso
  - El orden de como colocamos las variables y métodos

- ## Formato horizontal

La extensión del código de forma horizontal
Algunos IDEs tienen una línea horizontal que indica el límite de donde debemos escribir

- Apertura
- Densidad: correcta sangría
- Alineación
- Indentación



- Ya es hilar fino esto así que no es relevante but …

- ## Comentarios

- Comentarios legales: el uso del código
- Comentarios informativos: autor, fechas
- Explicar la intención: explicación
- Evitar código comentado: no es de ahuevo comentar, o dejar código comentado así /* */
- Claros pero no extensos: resume

- ## La minimización de la complejidad
- ## Adhesión a estándares de formato consistentes.

Esto nos da un Plus de **Calidad** Funcional, el código puede valer sin implementar ninguna de las reglas que dimos pero tendrá desventajas como no ser mantenible ni entendible.

Refactorización

Pero y ahora qué hacemos si cuando se implementó el código no se usaron estas buenas praxis, aquí es cuando entra la refactorización que es una técnica disciplina para reestructurar el código existente, alterando su estructura interna para que sea más fácil de entender sin cambiar su comportamiento externo.

Objetivos:

- Mejora el diseño del software
  - Reducir el acoplamiento
  - Entendible
- El software es más fácil de entender
- Permite a encontrar bugs
- Ayuda a programar más rápido
  - Ayuda a expandir y evolucionar nuestra app

Una herramienta fundamental para poder hacer la refactorización es la utilización de un buen IDE en el programa que queremos refactorizar

Cosas a lo que se enfrenta la refactorización

Mysterious Name

Problemas:

- El nombre del método no explica lo que hace.

```
function circum(radius) {...}
```

```
function circumference(radius) {...}
```

- El nombre de una variable no expresa lo que la variable almacena.

```
let a = height * width;
```

```
let area = height * width;
```

- El nombre de un campo no expresa lo que almacena.

```
class Organization {
  get name() {...}
}
```

```
class Organization {
  get title() {...}
}
```

Para abordar esto es con la ayuda de un IDE es utilizar REFACTOR que esto nos permite renombrar a las variables

Duplicado de código o || Funciones largas

Problemas

- Esto tiene que ver más con el apartado del tamaño de las funciones, y que las funciones o bien pueden unirse ya que cumplen el mismo objetivo o se relaciona o por el contrario dividir el código de una función en 2.

  Aquí debemos buscar un balance

- Aquí un problema muy común visto en Fundamentos de Programación que es código que suele ser idéntico y no nos damos cuenta

```
let result;
if (availableResources.length === 0) {
  result = createResource();
  allocatedResources.push(result);
} else {
  result = availableResources.pop();
  allocatedResources.push(result);
}
return result;
```

```
let result;
if (availableResources.length === 0) {
  result = createResource();
} else {
  result = availableResources.pop();
}
allocatedResources.push(result);
return result;
```

Lo que podemos es haceres cambiar esa parte del código duplicado a una posición lógica donde no afecte su fin por el cual fue codificado

Una buena praxis aplicando los conocimientos de Programación Orientada a Objetos es aplicar Herencia.

Aunque aquí también existe el inconveniente de que algunos autores no aprueban la herencia ya que esto aumenta el acoplamiento

- Los métodos contienen un grupo de parámetros que se repiten

```
function amountInvoiced(startDate, endDate) {...}
function amountReceived(startDate, endDate) {...}
function amountOverdue(startDate, endDate) {...}
```

```
function amountInvoiced(aDateRange) {...}
function amountReceived(aDateRange) {...}
function amountOverdue(aDateRange) {...}
```

Solución Reemplazar estos parámetros con un objeto, o usar la lógica de los enrutadores

- Varios atributos de un objeto son usados como parámetros a los métodos.

Solución Usar todo el objeto como atributo

- Pero cuando tenemos variables locales por ende ya no puede venir empaquetadas en un objeto

Solución transformar el método en una clase separada para que las variables locales se conviertan en campos de la clase, Luego puede dividir el método en varios métodos dentro de la misma clase

- Condicional complejo

  Descomponer las partes complicadas en métodos separados: condición, then y else

```
if (!aDate.isBefore(plan.summerStart) && !aDate.isAfter(plan.summerEnd))
  charge = quantity * plan.summerRate;
else
  charge = quantity * plan.regularRate + plan.regularServiceCharge;
```

```
if (summer())
  charge = summerCharge();
else
  charge = regularCharge();
```

Evitar anidar a lo menso

- Problemas donde varios objetos ejecutan diferentes acciones pero dependiendo del tipo de objeto realizan cosas diferentes

  Esto es básicamente para lo que se creó él Polimorfismo por ende comprenderlo nos ayudará a discernir y resolver problemas como este.

- Bucles hacen más de una cosa

  Dividir en varios bucles en donde cada uno realice una función.

Variables

Problemas con

- Mutación de los datos

Se refiere a la modificación que almacenan las variables, ejemplo los métodos Setter

- Locales
  - Siempre que se las declaren deben estar lo más cerca del inicio del bloque en el que vayan a estar

- Es usada para almacenar varios valores intermedios dentro de un método, porque así cambiamos la lógica de lo que guarda la variable dando esto hincapié a confusiones

  Solución Usar variables diferentes cada una para un valor, no temer a crear las variables locales que se requieran.

```
let temp = 2 * (height + width);
console.log(temp);
temp = height * width;
console.log(temp);
```

- Fragmentos de código que asignan el valor a una variable se encuentran dentro de un método.
  Solución Mover el código a un nuevo método y reemplazar el código con una llamada al método.

Shotgun Surgery

(cirugía de escopeta)

Es un término que se utiliza en programación para describir una situación en la que un cambio pequeño en el código requiere hacer modificaciones en muchos lugares diferentes del sistema.

Es decir, si necesitas realizar una modificación en una parte específica de tu código, terminas teniendo que realizar cambios en varias clases, métodos o archivos dispersos por todo el proyecto.

Situaciones que generan Shotgun Surgery:

- Cambios en Requisitos:
  - Cuando hay cambios en los requisitos del sistema, especialmente si afectan a funcionalidades compartidas por múltiples partes del código.
- Dependencias Excesivas:
  - Si hay muchas dependencias entre clases o módulos, un cambio en una parte puede requerir cambios en todas las clases que dependen de ella.
- Falta de Cohesión:
  - Cuando las responsabilidades y funcionalidades relacionadas no están agrupadas de manera cohesiva en las clases, lo que lleva a la dispersión de la lógica relacionada.
- Diseño Monolítico:
  - En sistemas con un diseño monolítico, donde las funcionalidades no están bien separadas y los cambios pueden tener efectos colaterales en varias partes.
- Varias funciones modifican parte de una misma estructura
  Para evitar que las pequeñas funciones se dispersen por todo el código, debemos hacer que los métodos sean privados.

```
function base(aReading) {...}
function taxableCharge(aReading) {...}
```

```
function enrichReading(argReading) {
  const aReading = _.cloneDeep(argReading);
  aReading.baseCharge = base(aReading);
  aReading.taxableCharge = taxableCharge(aReading);
  return aReading;
}
```

Buenas Prácticas para Evitar Shotgun Surgery:

- Principio de Responsabilidad Única (SRP):
  - Aplica el principio SRP, donde cada clase debe tener una única responsabilidad. Esto ayuda a limitar el impacto de los cambios a una sola área del código.
- Principio de Abierto/Cerrado (OCP):
  - Diseña tus clases para que sean abiertas para la extensión pero cerradas para la modificación. Esto significa que puedes agregar nuevas funcionalidades sin cambiar el código existente.
- Inyección de Dependencias:
  - Utiliza la inyección de dependencias para reducir las dependencias directas entre clases y facilitar la sustitución de implementaciones.
- Interfaces y Abstracciones:
  - Define interfaces y abstracciones que permitan cambiar las implementaciones sin afectar el código que las utiliza.
- Patrones de Diseño:
  - Aplica patrones de diseño como el patrón Observer, el patrón Strategy, o el patrón Composite para estructurar tu código de manera que los cambios impacten de manera limitada.
- Desacoplamiento:
  - Minimiza el acoplamiento entre diferentes partes del código, evitando dependencias directas siempre que sea posible.
- Pruebas Unitarias:
  - Implementa pruebas unitarias sólidas para detectar rápidamente cualquier impacto no deseado de cambios en el código.

Cambios divergentes

Se requiere cambiar muchos métodos **no relacionados** cuando realiza cambios en una clase, principalmente pasa cuando no hicimos un diseño correcto.

Problemas

- Un código ejecuta varias cosas diferentes no relacionadas.
  Solución Dividir el comportamiento del código en fases secuenciales (modularidad).

- **Feature Envy** Poner las funciones donde esté más junto a los elementos que haga referencia lo que permite que otras partes del software sean menos dependiente de los detalles de este módulo esto siempre y cuando sea posible

  - Esto mismo puede pasar con las variables sí se están usando más en otra clase que en la que se declaró creamos un atributo en una clase nueva y redirigirla al campo anterior

    Soluciones:
    - Move Function
    - Extract Function

Problemas puntuales

Data Clumps

Diferentes partes del código contienen grupos idénticos de variables.
Estos grupos deben convertirse en su propia clase.

Si al eliminar uno de los valores los otros valores no tienen sentido es una buena señal de que este grupo de variables debe combinarse en un objeto.

Soluciones:
- Extract Class
- Introduce Parameter Object
- Preserve Whole Object

Primitive Obsession

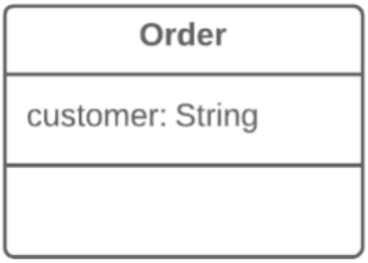
Uso de primitivos en lugar de objetos pequeños.

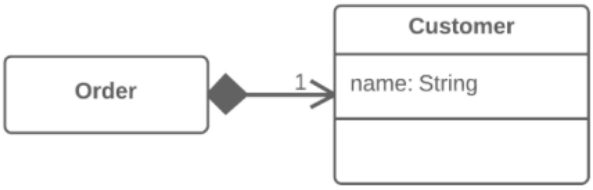
Algunas soluciones:
- Extract Class
- Introduce Parameter Object

Primitive Obsession - Replace Primitive with Object

Una clase (o grupo de clases) contiene un atributo y este atributo tiene su propio comportamiento y datos asociados.

Solución Crear una nueva clase, colocar el campo anterior y su comportamiento en la clase y almacenar el objeto de la clase en la clase original.





- Si un primitivo es parte fundamental de una estructura condicional, mejor usar un Switch
  - La lógica del enrutador.

Loops - Replace Loop with Pipeline

- Uso de bucles para iterar un conjunto de objetos

Solución Usar Collection pipeline para organizar los cálculos como una secuencia de operaciones (filter, map, reduce, etc) en donde el resultado de una operación es la entrada de otra.

Lazy Element

- Una clase padre y su hijo no son los suficientemente diferentes como para mantenerlas separadas en una jerarquía.

Solución Unir las dos clases en una sola, aplicar:

- Pull Up Field
- Push Down Field
- Pull Up Method
- Push Down Method

para mover los elementos a una clase y finalmente modificar las referencias.

Speculative Generality

El anticiparse mucho al futuro no hace crear clases y herencias innecesarias, por lo cual empezamos a crear Código Muerto.

Para herencia y clases abstractas:

▪Collapse Hierarchy

Para funciones se puede aplicar, Eliminar la delegación innecesaria::

▪Inline Function

▪Inline Class

Para funciones con parámetros sin utilizar o innecesarios:

▪Change Function Declaration

Problema
Una variable, parámetro, campo, método o clase ya no se usa (generalmente porque está obsoleta).

Solución Una vez que el código ya no se usa se debe eliminar si es algo que a futuro podría necesitarse, mediante **control de versiones** se podrá tener acceso a esa información.

Temporary Field

En base a un valor específico el comportamiento es el mismo. Algunos métodos devuelven null en lugar de objetos reales, existen muchas comprobaciones para null en el código.

Solución Crear un elemento de caso especial que captura todo el comportamiento común. En lugar de null retornar un objeto null que muestre el comportamiento predeterminado.

```
Estudiante e = new Estudiante ()
Estudiante e1 = new EstudianteEgresado

if (e1 instanceof EstudianteEgresado)

solicitarExpediendeGrado ();}
```

El caso especial en este ejemplo es el EstudianteEgresado, este tipo de

Se puede consolidar aplicando:

▪Extract Class

▪Move Function

Message Chains

Ocurre cuando un cliente solicita otro objeto, ese objeto solicita otro más, y así sucesivamente. Estas cadenas significan que el cliente depende de la navegación a lo largo de varias clases. Cualquier cambio en estas relaciones requiere modificar el cliente.

▪Extract Function

▪Move Function

Solución

Crear un nuevo método en la clase A que delegue la llamada al objeto B. Ahora el cliente no conoce ni depende de la clase B.

Replace Superclass with Delegate

- Una subclase usa solo una parte de los métodos de su superclase (o no es posible heredar los datos de la superclase).

  Solución
  Crear un campo y colocar un objeto de superclase en él, delegar métodos al objeto de superclase y eliminar la herencia.

- La herencia introduce una relación muy estrecha entre las clases

  Solución Crear una clase para delegar y agregar un campo en la superclase para la clase delegada.

Favorece la composición sobre la herencia

▪Una subclase no puede reducir la interfaz de la superclase.
  Solo usar el 50 o 60% de una clase no tiene sentido, las subclases deben usar la mayoría.

▪Al sobrescribir métodos es necesario asegurar que el nuevo comportamiento sea compatible con el de base.

▪La herencia rompe la encapsulación de la superclase.

▪Las subclases están fuertemente acopladas a superclases.

▪Intentar reutilizar código mediante la herencia puede conducir a la creación de jerarquías de herencia paralelas.

Insider Trading

Si tenemos que hacer cambios, debemos ver cuales son los efectos colaterales, para evitar el acoplamiento y la Cohesión, para evitar

Para que las cosas funcionen, se debe realizar algún intercambio, pero debe ser el mínimo.

▪Move Function
▪Move Field
▪Hide Delegate
▪Replace Subclass with Delegate
▪Replace Superclass with Delegate

Large Class

Para evitar tener clases muy grandes

- Extract Class
- Extract Superclass
- Replace Type Code with Subclasses

Alternative Classes with Different Interfaces

- Change Function Declaration
- Move Function
- Extract Superclass

Data Class

- Remove Setting Method

  Si no queremos que se modifiquen los datos

- Move Function
- Extract Function
- Split Phase

Encapsulate Record

Constantes usadas a lo largo del código.

Solución: Tener una clase para almacenar estas constantes.
Esto para evitar que si se cambia una constate estarla buscando en todo el codigo sino que se pueda cambiar en un solo punto

Refused Bequest
Cuando una subclase hereda de la superclase pero solo usa un
subconjunto de los métodos definidos en el padre.

- Push Down Method
- Push Down Field
- Replace Subclass with Delegate
- Replace Superclass with Delegate

Comments - Introduce Assertion

Problema Para que una parte del código funcione correctamente, ciertas condiciones o valores deben ser verdaderos.

Solución Reemplazar estas suposiciones con controles de específicos.

Reflexión

Llegar al 100% de algo casi siempre es imposible por ende estar siempre por ciento con cohesión o con acoplamiento

no siempre va a ser beneficioso, parte de que somos ingenieros es tratar de ver el equilibrio entre estos: y ver la forma más eficiente que se convierta nuestro programa.

- No reinventes la rueda, una parte fundamental de la construcción del software es el análisis de que herramientas, IDEs y lenguajes de programación usaremos, así que es bueno implementar nuestros proyectos con lenguajes que ya tengan APIs que nos ayuden a resolver problemas sin que nosotros lo hagamos de 0, eso es ser parte de una comunidad.

- Hemos escuchado mucho sobre el Dominio del problema, esto nos vuelve expertos en un cierto ambito o tema y nos da experiencia en dicho campo, pero ante problemas como Temporary Field eso ya es experiencia de nuestra carrera, podremos dominar el problema pero ya en la construcción vienen estas buenas praxis y saber identificar los problemas vistos.

- La refactoriacion es una solucion a un problema que claramente se puede evitar, si seguimos buenas praxis

Docker

domingo, 19 de noviembre de 2023
17:37

Conceptos previos a dominar:

- contenedores
- Image

Razón

Te ha pasado que tú creas un proyecto en cualquier lenguaje de programación y en tu computadora si funciona pero la trasladas a la computadora de un compañero y ya no funciona pues puede haber varios motivos ya sea por:

- las versiones de un JK
- el tipo de sistema operativo
- Bibliotecas
- Dependencias
- Drivers

y por ello es que existen estas plataformas como Dockers, pudiendo usarse en un entorno de desarrollo y de pruebas para la producción.

Qué es?

Es una plataforma de código abierto que facilita la creación, implementación y ejecución de aplicaciones en contenedores.

Los contenedores son entornos ligeros y portátiles que encapsulan una aplicación y sus dependencias, lo que

permite ejecutarla de manera consistente en diferentes entornos.

Objetivo

- Proporcionar una forma eficiente y consistente de empaquetar, **distribuir y ejecutar aplicaciones**, independientemente del entorno de desarrollo o implementación. Con Docker, puedes asegurarte de que una aplicación funcione de la misma manera en el entorno de desarrollo, en pruebas y en producción.

Ecosistema

Docker CLI (Command-Line Interface) Permite ejecutar comandos para construir, gestionar y desplegar contenedores. Así interactuamos con Docker Herramienta para enviar instrucciones o comandos a Docker Server	**Docker Server** También conocido como "Docker Daemon", es un proceso que se ejecuta en segundo plano en el sistema operativo y gestiona la construcción, ejecución y distribución de contenedores.	**Docker Machine** Es una herramienta que permite crear y gestionar máquinas virtuales Docker en diferentes entornos, facilitando la instalación y configuración de Docker en sistemas remotos. Útil para crear servidores local o en la nube
Docker Image Es una plantilla de **solo lectura** que contiene las intrusiones que se utiliza como base para la creación de contenedores.	**Docker Swarm** Es una característica de Docker que permite la creación y gestión de clústeres de contenedores. Facilita la orquestación y escalabilidad de aplicaciones distribuidas.	**Docker Container** Es una instancia en ejecución de una imagen de contenedor. Contiene la aplicación y sus dependencias, ejecutándose de manera aislada en el sistema operativo anfitrión.
Docker Compose	**Docker Volume** Es un mecanismo para persistir y	**Docker Network** Permite la creación y gestión de redes

Es una herramienta que permite definir y gestionar aplicaciones Docker multi-contenedor. Se utiliza para configurar servicios, redes y volúmenes de forma declarativa.	gestionar datos fuera del ciclo de vida del contenedor. Los volúmenes facilitan el almacenamiento de datos de forma duradera y compartida entre contenedores.	virtuales para conectar contenedores. Facilita la comunicación entre contenedores y servicios en un entorno Docker.
Docker Hub Servicio en la Nube para compartir aplicaciones que proporciona un repositorio centralizado para almacenar y compartir imágenes de contenedores.		**Docker Registry** Es un servicio que almacena y distribuye imágenes de contenedores. Puede ser público o privado, y Docker Hub es un ejemplo de un registro público. Los registros privados permiten almacenar imágenes de forma segura dentro de una organización.

Elementos

- **Images**

  Los contenedores se crean a partir de imágenes.

  Una imagen es la base que define el entorno y la configuración que el contenedor utilizará durante su ejecución.

- **Contenedor**

  Se crean a partir de imágenes y representan la ejecución real de una aplicación.

  Puedes tener múltiples contenedores basados en la misma imagen, y cada uno actúa de forma independiente.

- **Volúmenes**

  Son mecanismos de persistencia de datos que permiten almacenar y compartir información entre contenedores y el sistema operativo anfitrión.

Los volúmenes son independientes del ciclo de vida del contenedor.

- Facilitan la persistencia de datos, lo que significa que los datos almacenados en un volumen persistirán incluso si el contenedor se detiene o se elimina. Son útiles para almacenar bases de datos, archivos de configuración, o cualquier dato que necesite sobrevivir al ciclo de vida del contenedor.

- **Redes**

  Facilita la comunicación entre contenedores y servicios en un entorno

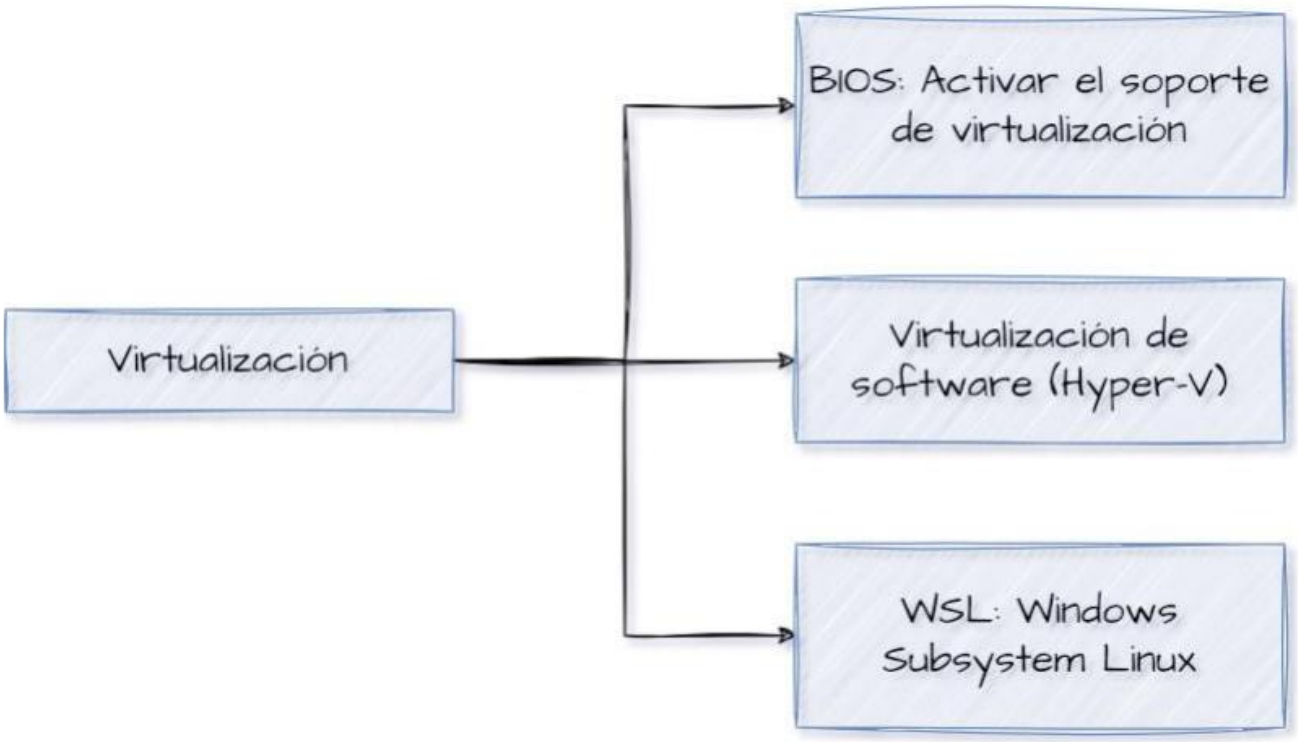
  Puedes definir redes personalizadas para tus aplicaciones y gestionar cómo se conectan y comunican los contenedores.

Relaciones entre los Elementos:

Imagen → Contenedor: Los contenedores se crean a partir de imágenes.
Contenedor ↔ Volumen: Los volúmenes se pueden montar en contenedores para persistir datos.
Contenedor ↔ Red: Los contenedores pueden estar en la misma red y comunicarse entre sí.

Virtualización

Se apoya el Docker de la virtualización



Docker vs. Máquinas Virtuales (VMs)

Tener en cuenta que no es lo mismo que tener una máquina virtual

Docker con imágenes y contenedores proporciona una forma más ligera y eficiente de virtualización, mientras que las máquinas virtuales ofrecen un aislamiento más completo pero con una mayor sobrecarga de recursos.

- Las VMs requiere que se indiquen cuales recursos son para la VMs
- Los dockers pueden usar todos los recursos de nuestra maquina

Creación y Ejecución del contenedor

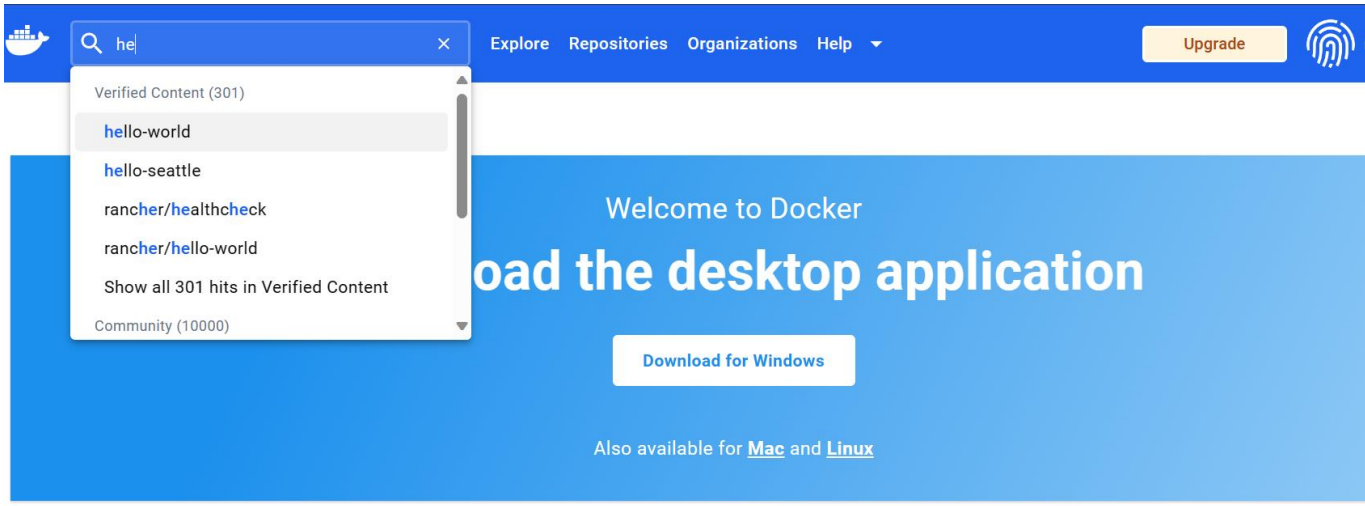
En la terminal de visual studio code hacer el siguiente comando:

docker	run	hello-word

El ejecutar este comando activa los siguientes pasos

1. Escribimos a través de Docker CLI (Command-Line Interface)
2. el cual busca en Docker Server
3. El cual busca la imagen localmente
4. o descargo la imagen desde Docker Hub: Docker Hub

    En este sitio web nosotros podemos buscar la imagen que nosotros acabamos de crear



5. Creó y ejecutó un contenedor usando la imagen descargada
6. Se apaga el contenedor

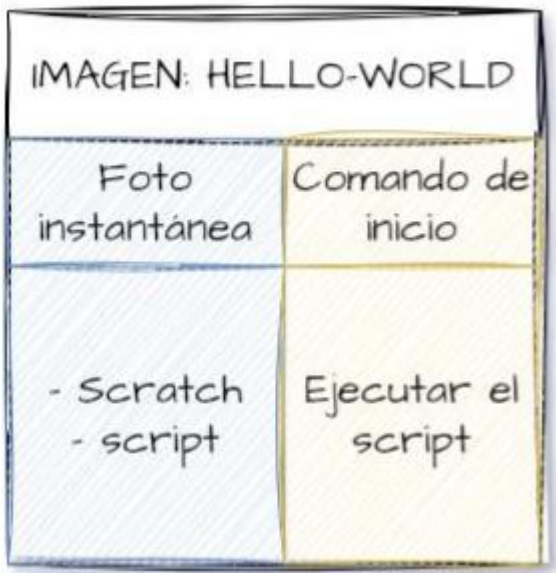
Y como paso todo eso?

Como es que llegamos a ver el siguiente mensaje

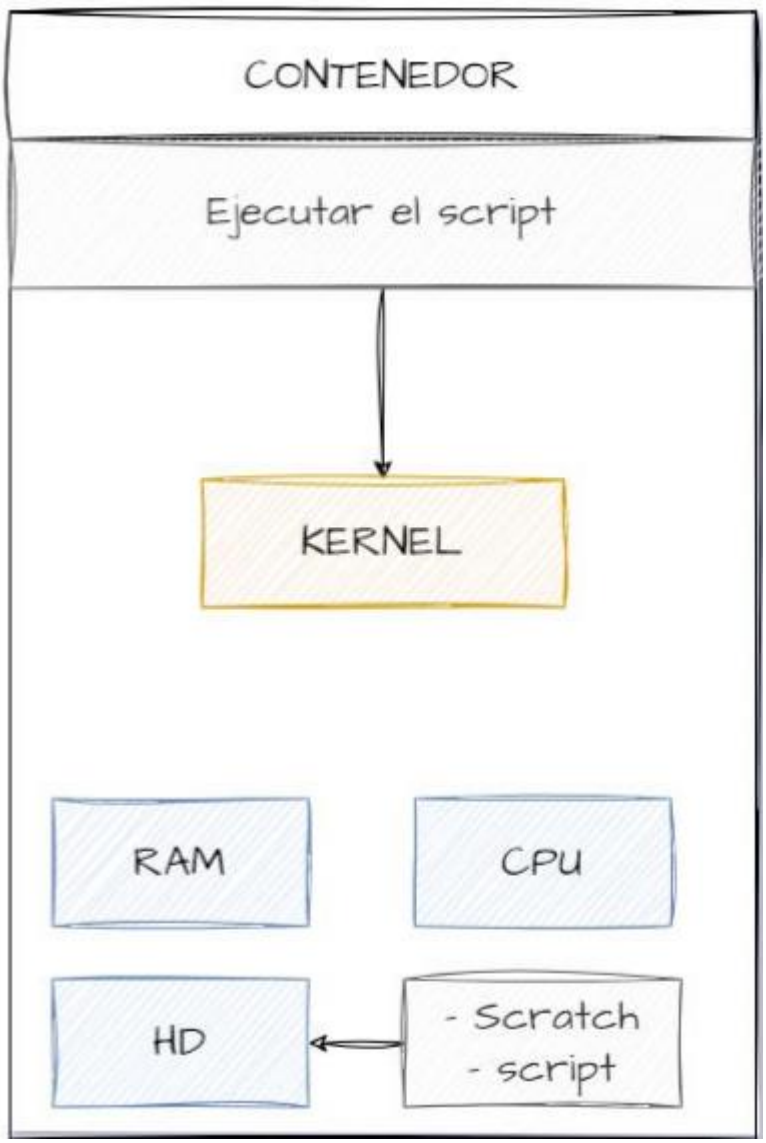
```
PS C:\Users\david> docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
```

Primero que nada hay 2 partes

1. Image hello world

Foto instantánea		Comando de inicio
Qué contiene no de aplicaciones y demás que necesitamos y un script de inicio • No se puede modificar		Tiene el comando de inicio que sirve para ejecutar el script • Si se puede modificar

2. Contenedor

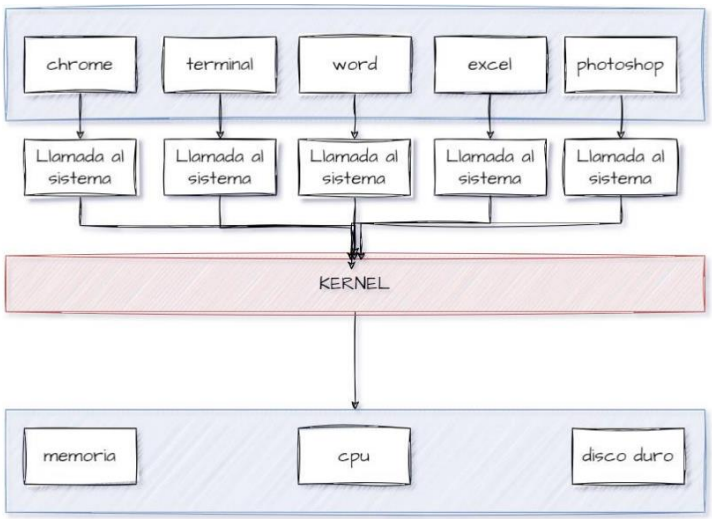


Lo que pasa es:

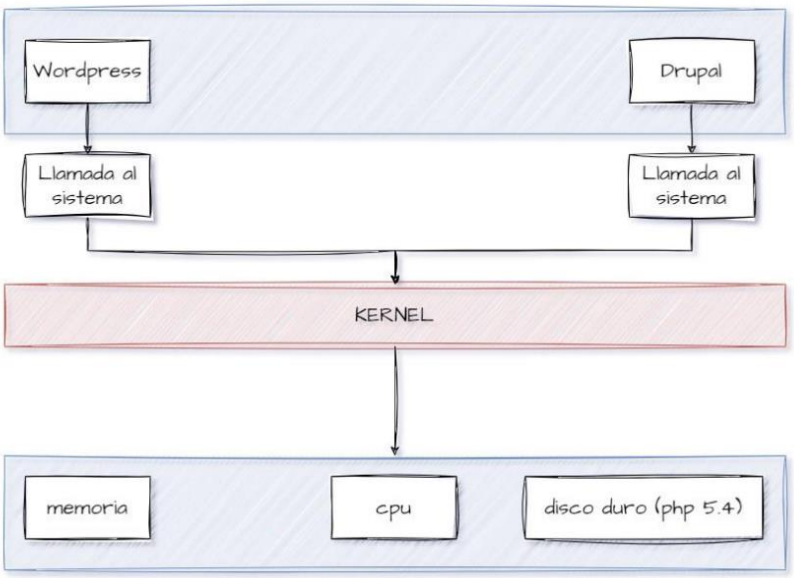
1. Foto instantánea se copia al disco duro
2. Se ejecuta el Script que interactúa con el Kernel
3.

Contenedores y Sistemas de Gestión de Contenidos

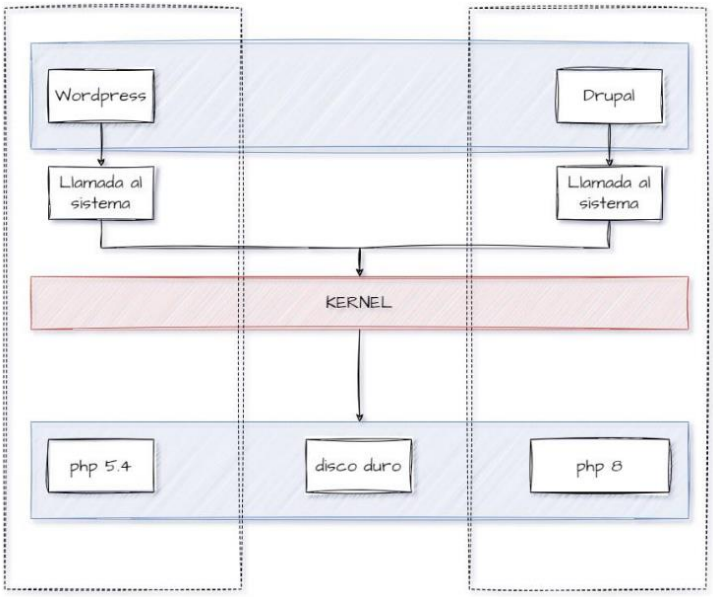
Para que las aplicaciones que siempre usamos funcionen estas llaman al sistema que van al kernel, se reparten memoria, cpu y disco duro



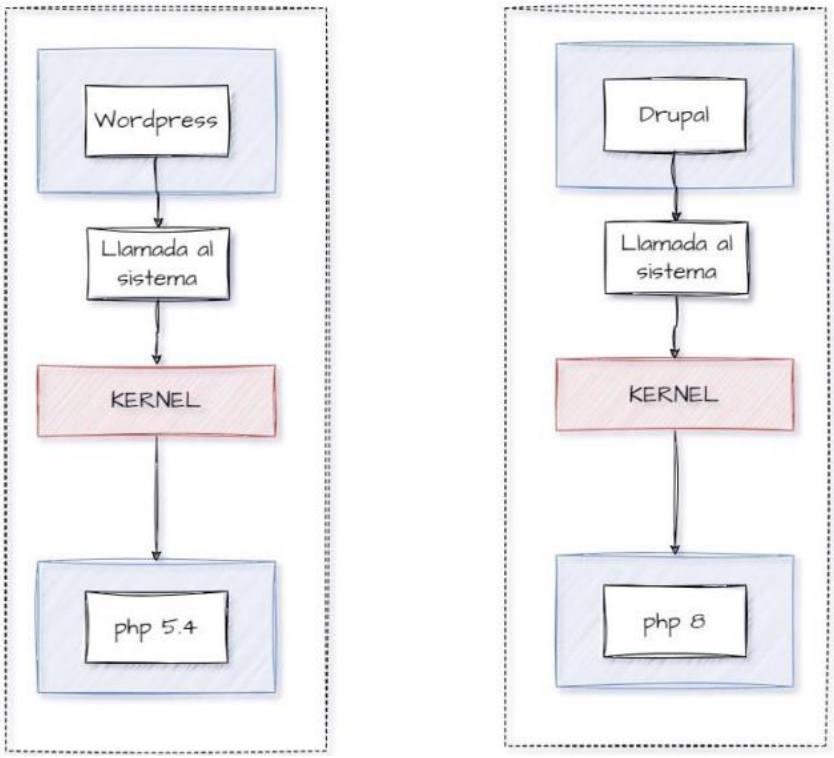
Imaginemos que queremos tener estos Sistemas de Gestión de Contenidos estos hacen lo mismo que cualquier otra aplicación



Pero ahora si tenemos una vulnerabilidad en un CMS (Drupal) donde en nuestro disco duro debe actualizarse al php 8 para que siga funcionando
Pero ahora en este supuesto caso vemos que nuestra computadora está trabajando su disco duro con php 5.4 php 8,

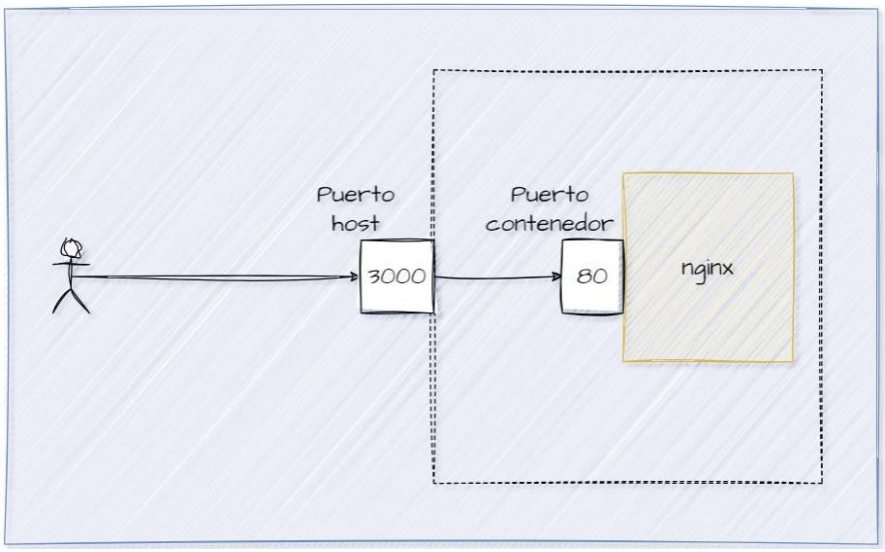


Aquí es donde entran los contenedores con los cuales podemos encapsular cualquiera de estos Sistemas de Gestión de Contenidos y así evitarnos que nuestra computadora principal tenga descargada demasiadas versiones de aplicaciones.

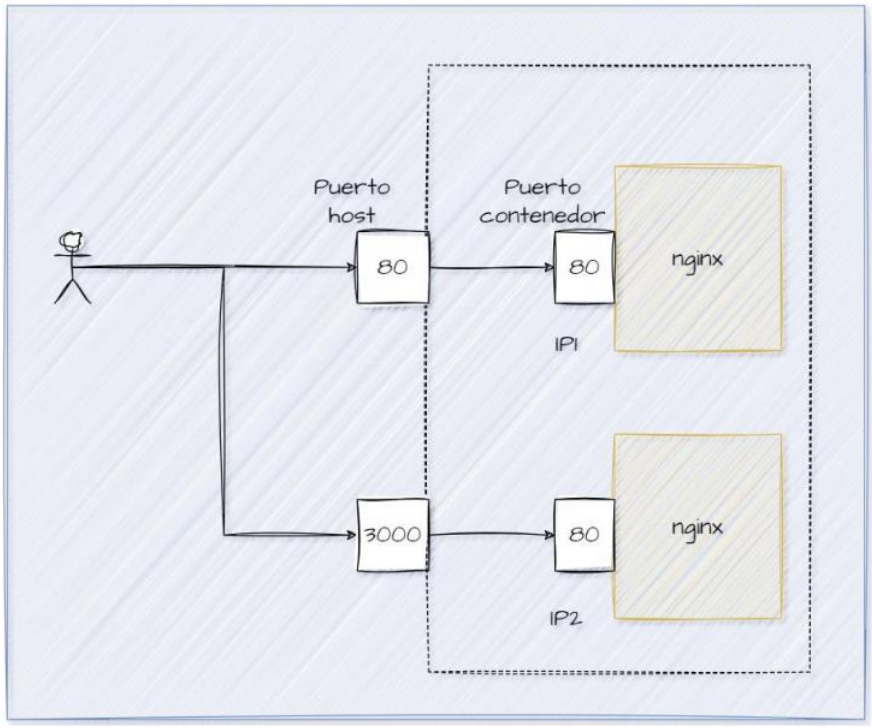


El cuadrado con líneas entrecortadas es una forma de representar a los contenedores, algunas aplicaciones exponen puertos.

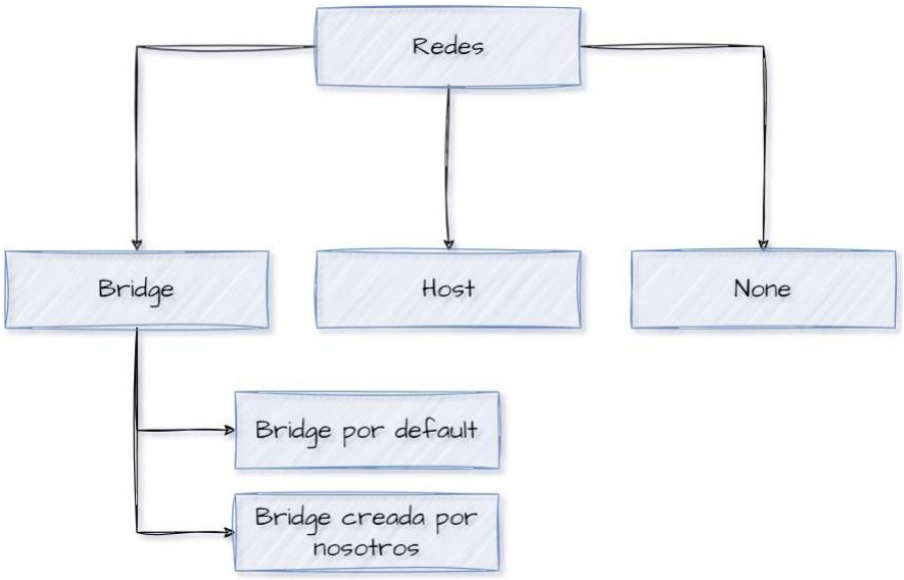
Cada contenedor va a tener una ip, para obtener esa dirección utilizamos dentro de un contenedor veremos en Redes



Si pueden existir dentro de nuestra Docker Network varios contenedores que compartan la misma imagen que tiene el mismo puerto, pero los dockers tienen diferente IP, pero el puerto host si debe ser diferente



Creando Redes



- Docker ofrece diferentes tipos de redes, como **bridge (por defecto)**, host, none

1. Red Bridge por Defecto:
   - Todos los contenedores creados pertenecen por defecto a la red bridge.
   - Esto significa que pueden comunicarse entre sí.

2. Definición de Redes Personalizadas:
   - Cuando queremos que ciertas redes se comuniquen.
   - Esto permite que un grupo de contenedores se vea entre sí, excluyendo a otros.

3. Contenedores en Redes Personalizadas:
   - Los contenedores pueden pertenecer a más de una red personalizada sin restricciones. Esto facilita la organización de contenedores según las necesidades del usuario.

Creación y Vinculación de Contenedores y Redes

- Se pueden crear contenedores primero y luego vincularlos a una red específica. No es necesario configurar la red al crear el contenedor, A diferencia del mapeo de puertos, no es obligatorio configurar la red al crear el contenedor. Puede realizarse posteriormente.

- Desvinculación de Contenedores de Redes:
  - Si se desea que un contenedor deje de pertenecer a una red, se puede desvincular posteriormente.

Para ver esto podemos ver la práctica Organizando el Entorno

Co

f

Relación con la materia?
Podría ser relevante en el desarrollo y despliegue de software.

Por ejemplo, si estás discutiendo temas relacionados con la construcción y evolución de software, Docker podría ser mencionado en el contexto de la gestión de entornos de desarrollo, la creación de imágenes de contenedores para aplicaciones, o la implementación de soluciones DevOps.

Docker facilita la creación de entornos aislados y reproducibles, lo que puede ser beneficioso para la construcción, prueba y despliegue de software. Además, Docker se utiliza comúnmente en la implementación de soluciones modernas de contenerización, lo que contribuye a la eficiencia y consistencia en el ciclo de vida del desarrollo de software.

Linux account
linuxuser
P@sswOrd123

Documentación

domingo, 3 de diciembre de 2023
22:19

Importancia de la documentación

- **Referencia y Registro**
  Proporciona un registro histórico de los procesos, decisiones y desarrollo de proyectos.
  Se necesita para que el producto software evolucione y tengamos un historial del porque.

- **Comunicación**
  Facilita la comunicación entre diferentes partes, como equipos de trabajo o entre una empresa y sus clientes.

- **Cumplimiento de Normativas**
  Ayuda a cumplir con normativas legales o estándares de la industria.

- **Mejora y Mantenimiento**
  Permite el análisis y la mejora continua de procesos y productos.

- **Base para la Toma de Decisiones**
  Ofrece la información necesaria para tomar decisiones informadas.

Tipos de documentación

Documentación de Requisitos:

- **Especificaciones de Requisitos de Software (SRS)**
  Detalla lo que el sistema debe hacer, incluyendo requisitos funcionales y no funcionales.

- **Historias de Usuario (Ágil) y Casos de Uso (Tradicional)**
  Describen las características del software desde la perspectiva del usuario.

Documentación de Diseño

- Diseño de Alto Nivel (Arquitectura)
  Proporciona una visión general de la arquitectura del sistema, incluyendo la estructura de los componentes principales y su interacción.

- **Diseño Detallado**
  Incluye diagramas de clases, diagramas de secuencia, y otros documentos que detallan cómo se implementarán las características.

Documentación Técnica

- **Documentación del Código Fuente**
  Comentarios en el código, descripciones de funciones, clases, y módulos.
  Esto se hace a través de anotaciones

- **Guías de API**
  Documentación para desarrolladores que explica cómo utilizar las APIs que ofrece el software.

  Una conocida en java son Documentación with Javadoc

Documentación del Usuario

- **Manuales de Usuario**
  Instrucciones y guías para los usuarios finales sobre cómo utilizar el software.

- FAQs y Ayuda En Línea
  Preguntas frecuentes y documentación de ayuda para asistir a los usuarios en la resolución de problemas comunes.

Documentación de Pruebas:

- Planes de Pruebas
  Detalla cómo se probará el software, incluyendo estrategias, recursos y cronograma.

- Casos de Prueba y Escenarios
  Especificaciones detalladas de las pruebas a realizar, incluyendo pasos, condiciones y resultados esperados.

- Informes de Pruebas
  Resultados de las pruebas realizadas, incluyendo fallos y su análisis.

Documentación de Implementación y Despliegue

- Guías de Instalación y Configuración
  Instrucciones para instalar y configurar el software en el entorno del usuario.

- Notas de la Versión
  Información sobre las versiones del software, incluyendo nuevas características, mejoras, correcciones de errores y problemas conocidos.

Documentación de Mantenimiento y Soporte

Manuales de Mantenimiento
Provee información para soportar y mantener el software, incluyendo diagnóstico de problemas y corrección de errores.

Documentación de Operaciones
Guías y procedimientos para el personal de operaciones, incluyendo monitoreo del sistema y procedimientos de respaldo.

Documentación de Proyecto

Esta materia es una vistazo previo a Gestion de proyectos de software

- Planes de Proyecto
  Detalles sobre la gestión del proyecto, incluyendo cronograma, recursos, y presupuesto.

- Informes de Estado y Actas de Reuniones
  Resúmenes periódicos del progreso del proyecto y decisiones tomadas durante las reuniones.