



Programando con Python

POO



POO introducción

- ¿Qué aprenderemos? (a la manera Python) :
 - Encapsulación
 - Abstracción de datos
 - Polimorfismo
 - Herencia



POO introducción

- **Clase:** El esquema o plano de construcción de donde se originan los objetos.
- En Python, todo es un objeto
- Se usa la función `type()` para determinar que clase origina un objeto
- Clase mínima:

```
class Robot:  
    pass
```



POO introducción

- **Atributos:** habilidad específica o características que algo o alguien posee.
- Suele llamársele como propiedades
- En Python propiedades y atributos son algo diferentes
- Aunque los atributos se definen internamente en las clases. en Python, los atributos se puede definir dinámicamente
- Internamente en la clase se pueden verificar con `__dict__`



POO introducción

- **Métodos:** en Python son esencialmente funciones.
- Definen el comportamiento de los objetos
- Aunque pueden ser definidos fuera de la clase, no es una buena práctica
- El primer parámetro es usado como referencia de la instancia llamada. Es nombrado *self*
- *self* ¡no es una palabra reservada de Python!, usuarios de C++ o Java lo pueden llamar *this*, pero puede ser contraproducente.

POO introducción

- **El método `__init__`:** es llamado inmediata y automáticamente después que una instancia ha sido creada.
- Ejm:

```
Class A:
    def __init__(self):
        print("¡¡__init__ ha sido ejecutado!!")
```

POO introducción

```
def __init__(self, radius):  
    self.radius = radius
```

- El método **__init__** es conocido como el constructor
- Es invocado cada vez que un objeto es creado en memoria.
- Aparte del parámetro self, se pueden agregar tantos parámetros como se necesite para inicializar un objeto.
- Definir el constructor no es requerido y si no se hace, Python proporciona uno vacío: `__init__()`

POO introducción

➤ Definir una clase (ejemplo):

```
import math

class Circle:

    def __init__(self, radius):
        self.radius = radius

    def get_area(self):
        return math.pi * self.radius ** 2

    def get_perimeter(self):
        return 2 * math.pi * self.radius
```

- En Python el parámetro **self** es requerido en cada método y refiere al objeto que invocó tal método
- Al llamar el método no se necesita pasar ningún valor a self.
- Dentro de la clase se usa self, para tener acceso a los atributos y métodos propios del objeto



POO introducción

- **Conceptos:**
- **Encapsulación:** Ocultar o proteger los datos, de manera que solo puedan ser accedidos usando funciones especiales, es decir los métodos
- **Ocultamiento de información:** principio de OOP por medio del cual los datos no pueden ser cambiados accidentalmente.
- **Abstracción de datos:** Los datos están ocultos y se usó la encapsulación, es decir:
Abstracción de datos = Encapsulación + Ocultamiento



POO introducción

- **Abstracción de datos:**
- **Métodos getter:** permiten obtener o tener acceso a los valores de los atributos, no cambian el valor del atributo.
- **Métodos setter:** son usados para cambiar los valores de los atributos.

POO introducción

➡ Ejemplo:

```
class Robot:
    def __init__(self, name= None):
        self.name = name
    def decir_hola(self):
        if self.name:
            print("Hola, yo soy " + self.name)
        else:
            print("Hola, soy un robot sin nombre")
    def get_name(self):
        return self.name
    def set_name(self, name):
        self.name = name
```



POO introducción

➤ **El Zen de Python:**

- se invoca usando `import this`
- indica que: “Debe haber una, y preferiblemente solo una, forma obvia de hacer algo”
- Ya lo trataremos más adelante

POO introducción

- **Los métodos `__str__` y `__repr__`**
- Si son definidos en la clase son usados para obtener una representación personalizada del objeto.
- Si solo se define `__str__`, `__repr__` retornará el valor predefinido
- Si solo se define `__repr__`, funcionará para llamadas de `__repr__` y de `__str__`
- Considere siempre usar `__str__`, ya que `__repr__` puede que no funcione para objetos diferentes de string, se debe cumplir:

```
obj == eval(repr(obj))
```
- `__repr__` se usa para representación interna, `__str__` para el usuario final

POO introducción

➤ Ejemplo:

```
class Robot:
    def __init__(self, name, build_year):
        self.name = name
        self.build_year = build_year
    def __repr__(self):
        return "Robot(\"" + self.name + "\", "
            + str(self.build_year) + ")"
    def __str__(self):
        return "Name: " + self.name + ", Build Year: "
            + str(self.build_year)
```



POO introducción

➤ **Atributos Public, Protected y Private**

➤ In POO significan:

- Private, sólo puede ser usado por el propietario
- Protected, úselo a su propio riesgo, solo por alguien que escriba una subclase
- Public, puede ser usado sin restricción

➤ En Python, se pueden definir así:

- `name` , Public
- `_name`, Protected,
- `__name`, Private

(ver el ejemplo reescrito)

POO introducción

➤ Atributos de clase y de instancia

➤ In POO significan:

- de instancia: son particulares a cada instancia
- de clase: pertenecen a la clase

➤ En Python, atributo de clase:

```
Class A:  
    a = "atributo de clase"
```

```
x = A()  
y = A()  
x.a
```

- OJO: Si se desean cambiar se debe hacer con el NombreClase.atributo, si se hace con un objeto, se crea un atributo de instancia.
(ver ejemplo)

POO introducción

- **Métodos estáticos**

- In POO significan:

- se acceden desde la propia clase, no de una instancia

- En Python:

```
Class Robot:
    __counter = 0
    def __init__(self):
        type(self).__counter += 1

    @staticmethod          # <- ojo con el decorador
    def RobotInstances():
        return Robot.__counter
```

- Simplemente no llevan *self*

POO introducción

➤ Métodos de clase

- Son como métodos estáticos, pero están atados a la clase, la que se pasa como referencia

➤ En Python:

```
Class Robot:
    __counter = 0
    def __init__(self):
        type(self).__counter += 1

    @classmethod
    def RobotInstances(cls):
        return cls, Robot.__counter
```



POO introducción

- **Métodos de clase, cuando usarlos:**
- En la definición de métodos llamados factory, no se tratará el tema de patrones de diseño
- En conjunto con métodos estáticos, que deben llamar a otros métodos estáticos.
(ver ejemplo)
- Se volverán a tratar mas adelante en herencia.



POO introducción

- **Uso de propiedades:**
- Cuando se desee usar la forma de acceso objeto.atributo, pero respetando el principio de encapsulamiento.
- Se eliminan los prefijos get_ y set_ y se sustituyen por @property (para el get_) y @atributo.setter por el setter, cada método simplemente lleva el nombre del atributo.

POO introducción

➤ Uso de propiedades (ejemplo)

```
class P:
    def __init__(self, x):
        self.__x = x

    @property
    def x(self):
        return self.__x

    @x.setter
    def x(self, x):
        if x < 0:
            self.__x = 0
        elif x > 1000:
            self.__x = 1000
        else:
            self.__x = x
```



POO introducción

Fin parte 1





Programando con Python

Clases y Objetos (Herencia y Polimorfismo)



POO introducción

- **Herencia:**
- Python soporta herencia y herencia múltiple
- Sintáxis: `class SubclassName(SuperclassName):`
`pass`
- Funciones `type()` y `isinstance()`, la primera nos dice el tipo, la segunda si el objeto es o no derivado de una clase en particular
(ver ejemplos)

Clases y Objetos: Herencia y Polimorfismo

➤ Herencia:

- En Python se usa la función **super()**, para llamar algún método de la super clase

➤ Ejemplo:

```
class Rectangle(Shape):
```

```
    def __init__(self, length, width):  
        super().__init__()  
        self.__length = length  
        self.__width = width
```

Ver ejemplo inheritance.py / herencia_simple.py

Clases y Objetos: Herencia y Polimorfismo

► Herencia múltiple:

► En Python es posible derivar una clase desde otras clases, esto se llama herencia múltiple.

► Sintaxis:

```
class ParentClass_1:  
    # body de ParentClass_1
```

```
class ParentClass_2:  
    # body de ParentClass_2
```

```
class ParentClass_3:  
    # body de ParentClass_1
```

```
class ChildClass(ParentClass_1, ParentClass_2, ParentClass_3):  
    # body de ChildClass
```

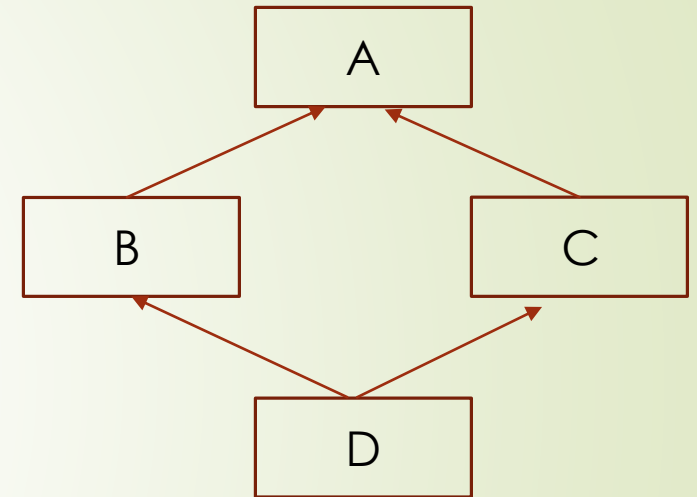
Ver ejemplo: [herencia_multiple.py](#) / [herencia_multiple2.py](#)

POO introducción

- **Herencia múltiple:**

- Sintaxis: class
SubclassName(Superclass1,
SuperClass2, ...,
SuperClassN):
pass

- Para evitar el problema del “mortal diamante de la muerte”, Python usa el llamado MRO (method resolution order), el cual le permite resolver el problema. Ver ejemplo.





POO introducción

- **Herencia – sobreescritura (overriding):**
- Cuando un método de la superclase se implementa de diferente forma en la subclase
- Es posible llamar a métodos de la superclase
- La subclase puede sus propios métodos (especialización)
(ver ejemplos)



POO introducción

- **Diferencia entre overwriting, overloading y overriding:**
- Overwriting, cuando una función se declara nuevamente con diferentes parámetros
- Overloading, no existe en Python, pero se puede simular con parámetros predefinidos. O mejor aún, usando la forma `def(*x):`, lo cual indica que la función `f` puede recibir un número indeterminado de parámetros.
- Overriding, cuando un método de la subclase tiene el mismo nombre de otro en la superclase.



Clases y Objetos: Herencia y Polimorfismo

- Polimorfismo:

- En Python el polimorfismo es definido de manera que un método de la clase hija tenga el mismo nombre que el de la clase padre.
- Es en esencia una sobre-escritura
- Ver ejemplos:
 - `method_overriding.py`
 - `method_overriding_2.py`

Clases y Objetos: Herencia y Polimorfismo

- object – la clase base:
 - En Python todas las clases heredan de la clase **object** de forma implícita.
 - Eso quiere decir que estas dos expresiones son equivalentes:

```
class MyClass:  
    pass
```



```
class MyClass(object):  
    pass
```

Clases y Objetos: Herencia y Polimorfismo

■ object – la clase base:

- La clase object posee métodos que son heredados por todas las clases. Algunos importantes son:

1. `__new__()` -> crea el objeto
2. `__init__()` -> después `__new__()`, se llama para inicializar los atributos del objeto
3. `__str__()` -> retorn a una representación en string del objeto (ver ejemplo `__str__method.py`), por lo general se sobrescribe según la necesidad



Clases y Objetos: Herencia y Polimorfismo

- Sobreescribiendo la funcionalidad de los operadores:
 - Python permite redefinir la forma como los operadores incluidos definen sus operaciones
 - De allí que el operador `+` sirva tanto para sumar números como para concatenar dos o más strings
 - Los métodos especiales que definen los operadores comienzan y terminan con doble guion bajo, así el de `+` es `__add__()`
 - Las clases `int` y la clase `str`, lo definen cada uno de acuerdo a lo que necesiten hacer y de allí que se pueda usar el mismo signo `+` de dos maneras diferentes.
 - Aunque empiezan con doble guion bajo no son privados, porque también terminan con doble guion bajo

Clases y Objetos: Herencia y Polimorfismo

➤ Operador y su método especial:

Operador	Método Especial	Descripción
+	<code>__add__(self, object)</code>	Suma
-	<code>__sub__(self, object)</code>	Resta
*	<code>__mul__(self, object)</code>	Multiplicación
**	<code>__pow__(self, object)</code>	Exponenciación
/	<code>__truediv__(self, object)</code>	División
//	<code>__floordiv__(self, object)</code>	División Entera
%	<code>__mod__(self, object)</code>	Modulo
==	<code>__eq__(self, object)</code>	Igual a

Clases y Objetos: Herencia y Polimorfismo

➤ Operador y su método especial:

Operador	Método Especial	Descripción
!=	<code>__ne__(self, object)</code>	Diferente de
>	<code>__gt__(self, object)</code>	Mayor que
>=	<code>__ge__(self, object)</code>	Mayor o igual que
<	<code>__lt__(self, object)</code>	Menor que
<=	<code>__le__(self, object)</code>	Menor o igual que
in	<code>__contains__(self, value)</code>	Operador de membresía
[index]	<code>__getitem__(self, index)</code>	Elemento en índice
len()	<code>__len__(self)</code>	Calcula número de elementos
str()	<code>__str__(self)</code>	Convierte objeto a string

Ver ejemplo: `special_methods.py`



Clases y Objetos: Herencia y Polimorfismo

- Sobreescribiendo la funcionalidad de los operadores:
 - Ejemplos:
 - `special_methods.py`
 - `point.py`



Clases y Objetos: Herencia y Polimorfismo

➤ Clases abstractas:

- Las clases abstractas son clases que contienen uno o más métodos abstractos.
- Un método abstracto es un método que se declara, pero que no contiene ninguna implementación.
- Las clases abstractas no pueden ser instanciadas, y requieren subclases para proporcionar implementaciones para los métodos abstractos.
- Aunque los métodos abstractos tengan implementación en la superclase, deben ser implementados en la subclase
- Una clase que se deriva de una clase abstracta no puede ser instanciada a menos que todos sus métodos abstractos sean sobrescritos.
- Un método abstracto puede tener una implementación en la clase abstracta! Incluso si se implementan, los diseñadores de subclases se verán obligados a sobrescribir la implementación.

Clases y Objetos: Herencia y Polimorfismo

➤ Clases abstractas:

- Python viene con un módulo que proporciona la infraestructura para definir las Clases Base Abstractas (ABCs). Este módulo se llama - por razones obvias - abc.
- El siguiente código Python utiliza el módulo abc y define una clase base abstracta:

```
from abc import ABC, abstractmethod  
class AbstractClassExample(ABC):  
    def __init__(self, value):  
        self.value = value  
        super().__init__()  
    @abstractmethod  
    def do_something(self):  
        pass
```



Clases y Objetos: Herencia y Polimorfismo

- Clases abstractas:

- Ejemplos:

- Ver `abstract_class_example_1.py`

- `abstract_class_example_2.py`

- `abstract_class_example_3.py`



Clases y Objetos: Herencia y Polimorfismo

FIN





Programando con Python

Estructuras de datos



Estructuras de datos

- Son *estructuras* que mantienen algunos *datos* a la vez. En otras palabras, son usadas para almacenar una colección de datos relacionados.
- Hay cuatro estructuras de datos incorporadas en Python:
 - list
 - Tuple
 - set
 - dictionary



Estructuras de datos - List

- Es una estructura que mantiene una colección ordenada de elementos, por ejemplo, se puede almacenar una *secuencia* de elementos en una list
- Su formato viene dado por una secuencia de elementos separados por coma, entre corchetes:
variable = [item1, item2, item3, ..., itemN]
- Una vez creada una list, se pueden agregar, remover o buscar entre sus elementos. De allí que se diga que es un tipo de datos *mutable*

Estructuras de datos - List

➤ Métodos de la clase list:

Nombre(args)	Qué hace
<code>list.append(x)</code>	Agrega un elemento al final de la list
<code>list.extend(iterable)</code>	Extiende la list agregando todos los elementos del iterable
<code>list.insert(i, x)</code>	Inserta un elemento <i>i</i> en la posición <i>x</i> dada
<code>list.remove(x)</code>	Remueve el primer elemento de la list cuyo valor es <i>x</i>
<code>list.pop([i])</code>	Remueve y retorna el elemento <i>i</i> especificado. Si este no es declarado, remueve y retorna el último elemento de la list
<code>list.clear()</code>	Remueve todos los elementos de la list
<code>list.index(x [, start[, end]])</code>	Retorna un índice sobre base cero del primer elemento en la list cuyo valor es igual a <i>x</i> . Los argumentos opcionales <i>start</i> y <i>end</i> son interpretados como en la notación de un slice de la list y son usados para buscar una subsecuencia particular de la list
<code>list.count(x)</code>	Retorna el número de veces que <i>x</i> aparece en la list

Estructuras de datos - List

➤ Métodos de la clase list:

Nombre(args)	Qué hace
<code>list.sort(key = None, reverse = False)</code>	Ordena los elementos de la lista
<code>list.reverse()</code>	Invierte el orden de los elementos de la list
<code>list.copy()</code>	Retorna una copia de la list

➤ Funciones para trabajar con list:

Nombre(args)	Qué hace
<code>len(list)</code>	Retorna el número de elementos de la list
<code>sum(list)</code>	Retorna la suma de los elementos de la list
<code>max(list)</code>	Retorna el elemento de mayor valor en la list
<code>min(list)</code>	Retorna el elemento de menor valor en la list

Estructuras de datos - List

► Ejemplos:

```
>>> frutas = ['naranja', 'manzana', 'pera', 'cambur', 'kiwi',  
'manzana', 'cambur']
```

```
>>> frutas.count('manzana')
```

2

```
>>> frutas.count('mandarina')
```

0

```
>>> frutas.index('cambur')
```

3

```
>>> frutas.index('cambur', 4)
```

6



Estructuras de datos - List

► Ejemplos:

```
>>> frutas.reverse()
```

```
>>> frutas
```

```
['cambur', 'manzana', 'kiwi', 'cambur', 'pera', 'manzana', 'naranja']
```

```
>>> frutas.append('fresa')
```

```
>>> frutas
```

```
['cambur', 'manzana', 'kiwi', 'cambur', 'pera', 'manzana', 'naranja',  
'fresa']
```

```
>>> frutas.sort()
```

```
>>> frutas
```

```
['cambur', 'cambur', 'fresa', 'kiwi', 'manzana', 'manzana', 'naranja',  
'pera']
```

```
>>> frutas.pop()
```

```
'pera'
```

Estructuras de datos - List

■ Lists usadas como **stacks**:

- Los métodos de list permiten usar una lista como un stack (pila), donde el último elemento es agregado es el primero retribuido (LIFO):

```
>>> stack = [3, 4, 5]
```

```
>>> stack.append(6)
```

```
>>> stack.append(7)
```

```
>>> stack
```

```
[3, 4, 5, 6, 7]
```

```
>>> stack.pop()
```

```
7
```

```
>>> stack.pop()
```

```
6
```

```
>>> stack.pop()
```

```
5
```

```
>>> stack
```

```
[3, 4]
```



Estructuras de datos - List

- Lists usadas como **queues (colas)**:
 - También podemos implementar una queue donde el primer elemento ingresado es el primer elemento que sale (FIFO)
 - Si bien los métodos appends y pops se ejecutan rápido en una list, son ineficientes para crear la queue, ya que insertar o extraer del principio ocasiona que haya que correr todos los elementos de su posición.
 - Para implementar una queue, se usa **collections.deque**, la cual se diseñó para hacer appends y pops de cualquier extremo, de una forma rápida

Estructuras de datos - List

- Lists usadas como **queues (colas)**, ejemplo:

```
>>> from collections import deque
```

```
>>> queue = deque(["Juan", "María", "Alberto"])
```

```
>>> queue.append("Rosa")
```

```
>>> queue.append("José")
```

```
>>> queue.popleft()
```

```
'Juan'
```

```
>>> queue.popleft()
```

```
'María'
```

```
>>> queue
```

```
deque(['Alberto', 'Rosa', 'José']) ← mantiene el orden de llegada
```

```
>>>
```


Estructuras de datos - List

- Creando una list usando la función **range()**

- La función `range()` retorna un objeto *iterable*

- Simplemente se pasa el objeto a la list

- Ejemplo1:

```
>>> list1 = list(range(8))
```

```
>>> list1
```

```
[0, 1, 2, 3, 4, 5, 6, 7]
```

Estructuras de datos - List

- Creando una list usando la función **range()**

- Ejemplo2:

```
>>> list2 = list(range(20, 35))
```

```
>>> list2
```

```
[20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34]
```

- Ejemplo 3:

```
>>> list3 = list(range(30, 60, 3))
```

```
>>> list3
```

```
[30, 33, 36, 39, 42, 45, 48, 51, 54, 57]
```

Estructuras de datos - List

- Las lists son **mutables**. Podemos modificarla sin crear un nuevo objeto

- Ejemplo:

```
>>> lista = ["carro", "pelota", "niño", "casa"]
```

```
>>> id(lista)
```

```
50947536
```

```
>>> lista[1] = "animal"
```

```
>>> lista
```

```
['carro', 'animal', 'niño', 'casa'] ← cambió el índice 1
```

```
>>> id(lista)
```

```
50947536 ← pero el objeto sigue siendo el mismo
```

Estructuras de datos - List

- Iterando los elemento de una list
- Se usan los bucles: for

```
>>> marcas = [10, 15, 33, 95, 44]
```

```
>>> for m in marcas:
```

```
...     print(m)
```

```
...
```

```
10
```

```
15
```

```
33
```

```
95
```

```
44
```

```
>>>
```



Estructuras de datos - List

- Iterando los elemento de una list
- Se usan los bucles: while

```
>>> marcas = [10, 15, 33, 95, 44]
```

```
>>> i = 0
```

```
>>> while i < len(marcas):
```

```
...     print(marcas[i])
```

```
...     i += 1
```

```
...
```

```
10
```

```
15
```

```
33
```

```
95
```

```
44
```



Estructuras de datos - List

- Al igual que con strings, con la list podemos:
 - Hacer **slicing**: lista[start: end]
 - Aplicar el operador **in** y **not in**
 - Concatenar: listC + listD
 - Aplicar el operador de repetición: listA = listB * 3



Estructuras de datos - List

- **Comprensión:** Cuando se desea crear una list donde cada elemento es el resultado de una operación, o cuando cumple con alguna condición
- Sintaxis: [expresión **for** elemento **in** iterable]
- Ejemplo: se desea obtener los cubos de una secuencia de 2 a 7:

```
>>> cubos = [ i ** 3 for i in range(1,8)]
```

```
>>> cubos
```

```
[1, 8, 27, 64, 125, 216, 343]
```


Estructuras de datos - List

- También se puede colocar una condición **if**
- Sintaxis: [expresión **for** elemento **in** iterable **if** condición]
- Ejemplo: se desea obtener los cubos de una secuencia de 2 a 7:

```
>>> cubos = [ i ** 3 for i in range(1,8) if i % 2 == 0]
```

```
>>> cubos
```

```
[8, 64, 216]
```

Estructuras de datos - Tuples

■ Tuples

- Forma parte de la familia de tipos de datos llamados secuencias, al igual que las **list** y el **range**
- Al igual que las list, son una agrupación de datos separados por coma, pero encerrados entre paréntesis

```
>>> t = 1, 3, 5, 7, 9
```

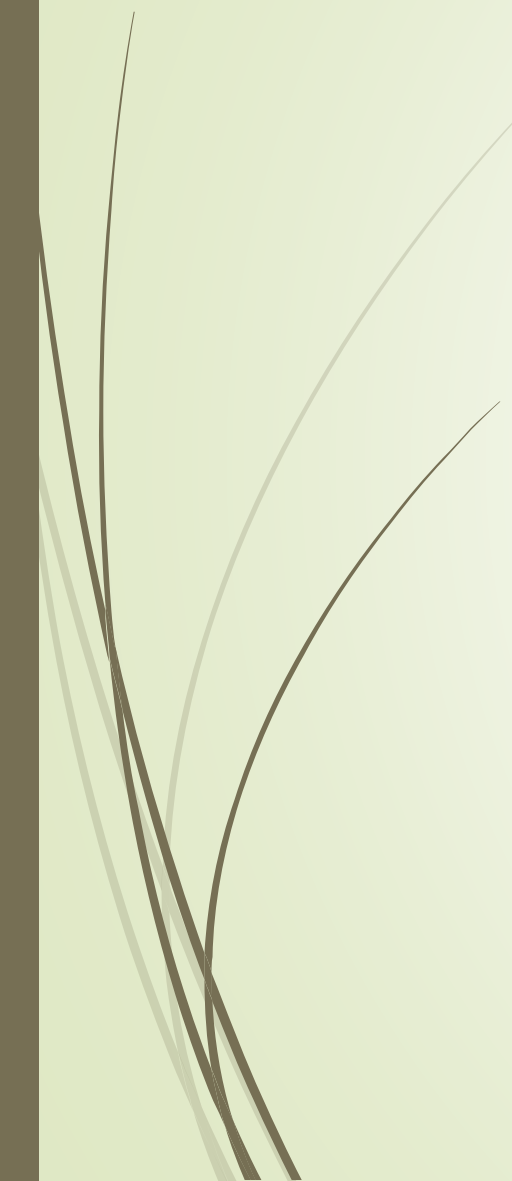
```
>>> t
```

```
>>> (1, 3, 5, 7, 9)
```

- Son inmutables
- Pero pueden contener múltiples objetos
- Se usan en situaciones y propósitos diferentes de list



Estructuras de datos - Tuples

- Operaciones con Tuples
 - Se puede tener acceso a un elemento o hacer slicing con el operador `[]`
 - Las operaciones incluidas **min()**, **max()**, **sum()** son válidas
 - Los operadores de membresía **in** o **not in**
 - Los operadores de comparación
 - Los operadores **+** y *****
 - Hacer iteraciones con **for** o **while**
- 

Estructuras de datos - Sets

➤ Sets

- Es una colección de elementos sin orden que no permite elementos repetidos
- Los datos se encuentran encerrados entre llaves “{ }”

```
>>> s = {1, 3, 5, 7, 9}
```
- Son mutables, para agregar elementos se usa el método **add()**
- Permiten eliminar elementos usando los métodos **remove()** y **discard()**. La diferencia radica en que **discard()** no genera error si el elemento no existe
- No se puede tener acceso a sus elemento usando un índice

Estructuras de datos - Sets

➤ Sets – métodos (algunos):

Método	Descripción
add()	Agrega un elemento al set
clear()	Remueve todos los elementos del set
copy()	Retorna una copia del set
difference()	Retorna un set con la diferencia entre uno o más sets
discard()	Remueve el elemento especificado
intersection()	Retorna un set que es la intersección de otros dos sets
issubset()	Retorna un boolean indicando si otro set contiene a este
issuperset()	Retorna un boolean indicando si este set contiene a otro
pop()	Remueve el elemento indicado
update()	Actualiza el set con la unión de este y otros sets

Estructuras de datos - Dictionarios

➤ Dictionarios

- Son conjuntos de datos agrupados en pares clave-valor
- A diferencia de las secuencias los diccionarios son indexados por claves
- Las claves pueden ser de cualquier tipo inmutable: strings, números o tuples. Si un tuple contiene un objeto mutable, directo o indirecto, no puede ser usado como clave
- Las claves deben ser únicas
- Se usan {} para representarlos, de la manera {clave: valor}
- Las principales operaciones de un dictionary es almacenar un valor con una clave y luego extraer el valor usando la clave.

Estructuras de datos - Dictionarios

➤ Dictionarios

➤ Pueden ser cambiados, no son inmutables

➤ Para eliminar un par *clave:valor* se usa **del**

```
>>> dic = {'juan': 1234, 'maria': 5678, 'rosa': 2468}
```

```
>>> del dic['maria']
```

```
>>> dic
```

```
{'juan': 1234, 'rosa': 2468}
```

➤ **list(dic)**, retorna una lista de todas las claves del diccionario

➤ El constructor **dict()**, crea diccionarios desde secuencias *clave:valor*

Estructuras de datos - Dictionaries

➤ Operaciones con dictionaries (algunas)

Método	Descripción
clear()	Remueve todos los elementos del diccionario
copy()	Retorna una copia del diccionario
fromkeys()	Retorna un diccionario con las claves y valores especificados
get()	Retorna el valor especificado por la clave
items()	Retorna una list conteniendo un tuple por cada clave:valor
keys()	Retorna una list con las claves del diccionario
pop()	Remueve el elemento con la clave especificada
popitem()	Remueve el ultimo par clave valor insertado
setdefault()	Retorna el valor de la clave especificada. Si esta no existe, inserta la clave con el valor indicado
update()	Actualiza el diccionario con la clave valor especificada
values()	Retorna una lista con todos los valores del diccionario

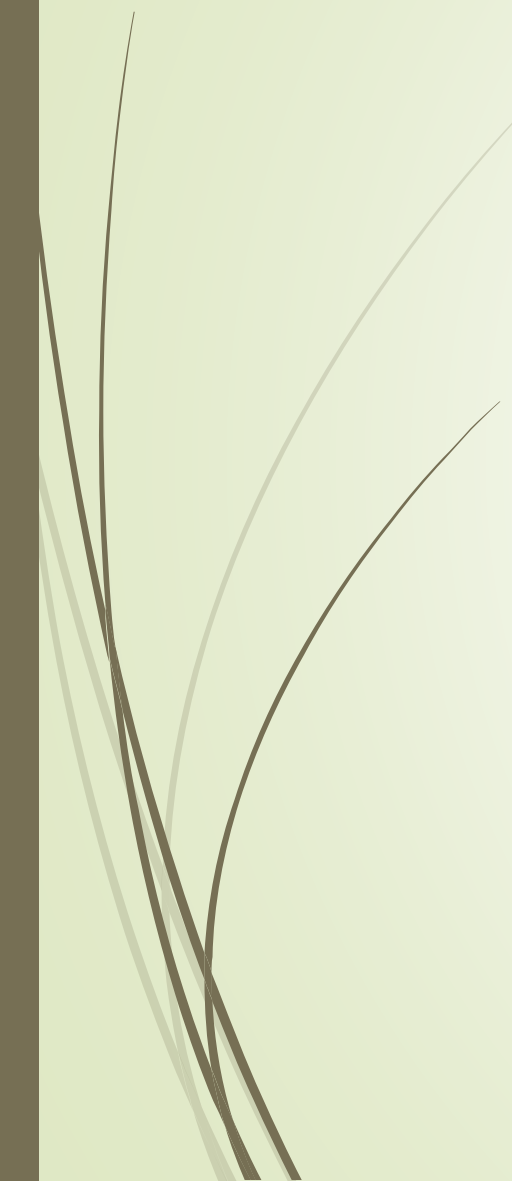


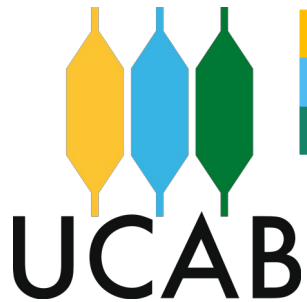
Técnicas de looping

- Al hacer un lazo a través de diccionarios, la clave y el correspondiente valor pueden ser obtenidos de una sola vez usando el método **items()**
- Al hacer un lazo en una secuencia, el índice de posición y su valor se pueden obtener usando la función **enumerate()**
- Para hacer loop sobre dos o mas secuencias al mismo tiempo, las entradas pueden se apareadas usando la función **zip()**
- Para hacer loop en forma inversa sobre una secuencia use la función **reversed()**



Técnicas de looping

- Para hacer loop sobre una secuencia de forma ordenada, use la función **sorted()**, la cual retorna una nueva lista ordenada, dejando la original intacta
 - Si alguna vez se trata de cambiar una lista mientras está haciendo un loop sobre ella; es más simple y seguro hacer una copia de la lista.
- 



Programación de APIs con Python



Pydantic

Profesor:
José Gregorio Castillo Pacheco

Decoradores – Ambientes Virtuales

Temario

- Pydantic

Pydantic

¿Qué es Pydantic?

- Pydantic es una librería que aprovecha la potencia de las type hints de Python para realizar la validación y serialización de datos.
- También tiene la capacidad de generar esquemas JSON a partir de los modelos, permitiendo una fácil integración con otras herramientas.

Pydantic

Ventajas de Pydantic:

- Seguridad de tipos: Pydantic aplica type hints en tiempo de ejecución, asegurando que los datos cumplan con los tipos y formatos esperados. También soporta tipos de datos personalizados, como enums, UUIDs, direcciones IP, etc.
- Errores sencillos: Cuando la validación falla, Pydantic proporciona mensajes de error informativos y fáciles de entender, incluyendo la ubicación, el tipo y el valor incorrecto. También proporciona enlaces a la documentación de cada tipo de error.
- Rendimiento: El núcleo de la lógica de validación de Pydantic está escrito en Rust, lo que la convierte en una de las bibliotecas de validación de datos más rápidas para Python. También soporta validación lazy y almacenamiento en caché para mejorar la eficiencia.
- Facilidad de uso: Pydantic es simple e intuitivo de usar, requiere un mínimo de código repetitivo y configuración. Funciona bien con muchos IDEs populares y herramientas de análisis estático, como PyCharm, VS Code, mypy, etc.

Pydantic

¿Cómo usar Pydantic?

- Para usar Pydantic, necesitas instalarlo usando
`pip install pydantic` o
`conda install pydantic -c conda-forge`.
- La principal forma de usar Pydantic es crear clases personalizadas que hereden de **BaseModel**, que es la clase base para todos los modelos de Pydantic. Los atributos del modelo usando type hints, y opcionalmente proporcionar valores por defecto o validadores.
- Pydantic valida y analiza automáticamente los datos de entrada de acuerdo con las anotaciones de tipo. También convertirá los datos al tipo correcto cuando sea apropiado. Por ejemplo, convertirá la cadena '2024-04-01 12:22' a un objeto datetime, y los bytes b'3' y el str '2' a enteros.

Ver ejemplo_1.py y ejemplo_2.py

Pydantic

¿Cómo usar Pydantic?

- Si los datos de entrada son inválidos o algún valor requerido no está presente, Pydantic levantará un `ValidationError` con un mensaje detallado de qué es lo que estaba errado. Ver ejemplo_3.py
- Pydantic permite aceptar únicamente valores predefinidos: Muchas veces se desea es permitir sólo un subconjunto de cadenas. Ver ejemplo_4.py
- También es posible definir propiedades como modelos Pydantic anidados. Es decir, podemos usar modelos pydantic como campos y podemos tener subobjetos
Ver ejemplo_5.py

Pydantic

¿Cómo usar Pydantic?

- Obteniendo valores dinámicos en tiempo de ejecución: El ejemplo más sencillo sería añadir una propiedad llamada **created_at** en el blog. Esta propiedad debería crearse dinámicamente cada vez que se crea un objeto blog. Se podría estar tentado a pensar que esto es fácil, todo lo que tenemos que utilizar es inicializar la propiedad con **datetime.now()**. Ver ejemplo_7.py
- ¡Ambas salidas tienen el mismo tiempo a pesar del delay!!
- Sin embargo, esto no funciona, ya que la instanciación de **datetime** se realiza sólo en el momento en que se importa el módulo. Para hacer frente a este tipo de situaciones Pydantic nos proporciona un argumento **default_factory** en una función **Field**. Ver ejemplo_8.py

Pydantic

¿Cuáles son algunas de las aplicaciones de Pydantic?

Pydantic puede ser utilizado para diversas aplicaciones que implican la validación y serialización de datos, tales como:

- Desarrollo web: Pydantic puede usarse para validar y analizar peticiones y respuestas en frameworks web, como FastAPI, Django Ninja y Starlette. También puede generar esquemas JSON y especificaciones OpenAPI a partir de sus modelos, lo que facilita la documentación y las pruebas de sus API.
- Análisis de datos: Pydantic se puede utilizar para validar y limpiar datos de diversas fuentes, como archivos CSV, bases de datos o web scraping. También puede ayudar con la exploración y visualización de datos proporcionando prácticos métodos y atributos, como `model_dump()`, `schema()`, `fields`, etc.

Pydantic

¿Cuáles son algunas de las aplicaciones de Pydantic?

- Aprendizaje automático: Pydantic puede utilizarse para validar y serializar modelos y datos de aprendizaje automático, como modelos TensorFlow, estimadores Scikit-learn o tensores PyTorch. También puede ayudar con el despliegue y la inferencia de modelos proporcionando métodos para exportar e importar modelos como archivos JSON o pickle.
- Gestión de la configuración: Pydantic puede utilizarse para gestionar los ajustes y archivos de configuración de tus aplicaciones, como variables de entorno, archivos INI o archivos YAML. También puede ayudar con la recarga dinámica y la validación de sus ajustes utilizando la clase `BaseSettings`.

Estos son sólo algunos de los ejemplos de cómo Pydantic puede ser utilizado en diferentes dominios y escenarios. Pydantic es una librería versátil y potente que puede ayudarte con cualquier tarea relacionada con datos en Python.

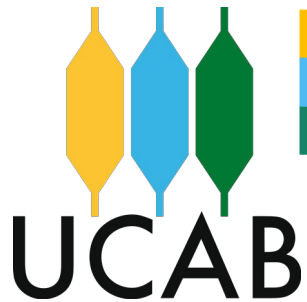
Pydantic

¿Cómo funcionan las validaciones a partir de la versión 2?

- La nueva versión de Pydantic tiene varias formas de implementar validadores propios:
 - Annotated validators: Pydantic usa la forma especial “Annotated” del módulo typing para agregar metadata a una declaración de un tipo.
 - Field validator: Pydantic permite agregar un validador particular a un campo específico
 - Model validator: puede validar el modelo completo
 - Para mayor información ver: <https://docs.pydantic.dev/latest/concepts/validators/>

FIN (parte 1)





Programación de APIs con Python



FastAPI

Profesor:
José Gregorio Castillo Pacheco

Decoradores – Ambientes Virtuales

Temario



FastAPI

¿Qué es FastAPI?

- FastAPI es un framework web moderno y de alto rendimiento diseñado específicamente para la creación de APIs, Fue creado por Sebastián Ramírez y liberado en 2018.
- Se basa en las anotaciones de tipos estándar de Python, lo que facilita la validación, serialización y deserialización de datos.
- Su velocidad de desarrollo y rendimiento lo hacen adecuado para proyectos de cualquier tamaño.

FastAPI

Componentes:

- **Pydantic:** Se utiliza para la validación de esquemas y serialización de datos mediante anotaciones de tipo (type hints), proporcionando sugerencias de tipo en el IDE.
- **Starlette:** es un framework liviano tipo ASGI (Asynchronous Server Gateway Interface) para soportar la funcionalidad asíncrona en Python.
- **Uvicorn:** es un servidor web de aplicaciones mínimo de bajo nivel para trabajar con frameworks asíncronos, que sigan la especificación ASGI

FastAPI

Características:

- **Rendimiento:** Altamente optimizado para un rápido desarrollo y ejecución
- **Validación de datos:** Integración nativa con Pydantic para la validación robusta de datos
- **Documentación automática:** Genera documentación interactiva de la API a partir del código
- **Soporte para múltiples métodos HTTP:** Facilita la creación de endpoints para diversos métodos HTTP
- **Esquemas de datos:** Define esquemas de datos para representar la estructura de las respuestas y peticiones
- **Dependencia de inyección:** Simplifica la gestión de dependencias en la aplicación

FastAPI

Instalación:

- De muy fácil instalación simplemente se ejecuta:

```
pip install fastapi
```

```
pip install "uvicorn[standard]"
```

- También puede usar la forma

```
pip install fastapi[standard]
```

- Y listo, ya podemos desarrollar nuestras API usando FastAPI

FastAPI

Primera aplicación (“Hola mundo”):

- Creamos un archivo llamado main.py

```
from typing import Union
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root():
    return {"Hola": "Mundo"}

@app.get("/items/{item_id}")
def read_item(item_id: int, q: Union[str, None] = None):
    return {"item_id": item_id, "q": q}
```


FastAPI

Ejecución:

- En la consola ejecutamos el siguiente comando

```
$ uvicorn main:app --reload
```

```
INFO:      Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

```
INFO:      Started reloader process [28720]
```

```
INFO:      Started server process [28722]
```

```
INFO:      Waiting for application startup.
```

```
INFO:      Application startup complete.
```

FastAPI

Probamos:

- En el navegador abrimos la dirección:
`http://127.0.0.1:8000` y observamos la respuesta
- Luego colocamos este query:
`http://127.0.0.1:8000/items/5?q=somequery` y observamos la respuesta

FastAPI

Vemos la documentación:

- En el navegador abrimos la dirección:
`http://127.0.0.1:8000/docs`
- Y nos muestra la documentación automática generada por FastAPI, basada en swagger
- Y si queremos otra forma de ver la documentación, colocamos:
`http://127.0.0.1:8000/redoc`

FastAPI

Usando Pydantic para validación:

- Ver la aplicación de main2.py

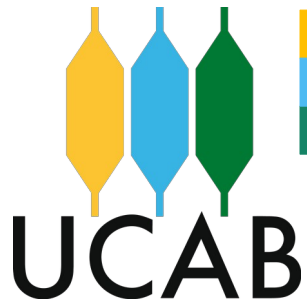
FastAPI

Conclusiones:

- **Rápido desarrollo:** Su simplicidad y flexibilidad permiten crear APIs de forma rápida y eficiente.
- **Alto rendimiento:** Está optimizado para un rápido desarrollo y ejecución, ideal para aplicaciones de alto tráfico.
- **Validación robusta de datos:** La integración con Pydantic garantiza la integridad de los datos de entrada y salida.
- **Documentación automática:** Genera documentación interactiva de la API a partir del código, mejorando la comprensión y usabilidad.
- **Escalabilidad:** Permite crear APIs escalables que pueden adaptarse a las necesidades cambiantes.

FIN (parte 2)





Programación de APIs con Python



Docker

Profesor:
José Gregorio Castillo Pacheco

Docker

Temario

- Docker

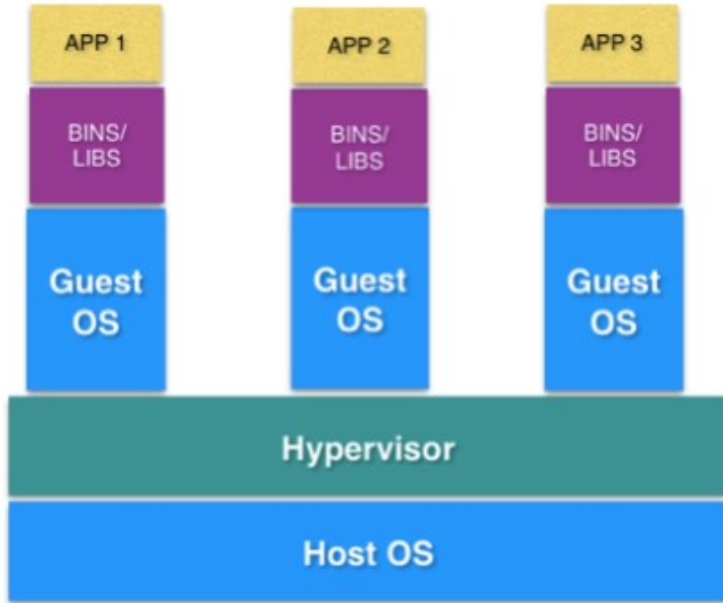
Docker

¿Qué es Docker?

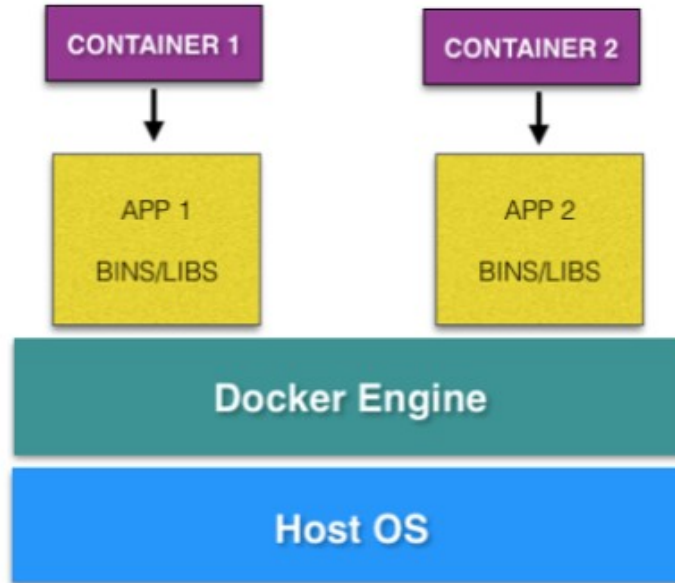
- Docker es una plataforma que permite a los desarrolladores crear, compartir y ejecutar aplicaciones en contenedores.
- Los contenedores son paquetes que utilizan la virtualización a nivel de sistema operativo para ofrecer un conjunto de productos de plataforma como servicio (PaaS)
- Los contenedores además son unidades estandarizadas que contienen todo lo que el software necesita para ejecutarse, incluidas las bibliotecas, las herramientas del sistema, el código y el tiempo de ejecución.
- En otras palabras Docker es un sistema operativo para contenedores.

Docker

¿Qué es Docker?



Arquitectura de una Máquina Virtual



Arquitectura Docker

Docker

¿Qué es Docker?

- Los contenedores Docker se basan en imágenes Docker, que se construyen a partir de archivos Docker (Dockerfile).
- Una imagen de contenedor Docker es un paquete portátil de software que incluye todo lo necesario para ejecutar una aplicación: código, tiempo de ejecución, herramientas del sistema, bibliotecas del sistema y configuraciones.
- Un Dockerfile es un archivo de texto que contiene instrucciones sobre cómo construir una imagen Docker, especifica:
 - el sistema operativo que subyacerá al contenedor,
 - los lenguajes, variables de entorno,
 - ubicaciones de archivos, puertos de red y otros componentes que necesita,
 - y lo que hará el contenedor una vez que lo ejecutemos.

Docker

Imagen Docker

- Una imagen Docker es una plantilla ligera y autónoma que contiene todo lo necesario para ejecutar una aplicación, incluidas bibliotecas, dependencias, código y configuraciones.
- Las imágenes son de solo lectura y están compuestas de múltiples capas.
- Cada capa representa una instrucción del Dockerfile durante la construcción de la imagen.
- Las imágenes se almacenan en Docker Hub o en un registro privado, y pueden ser compartidas y reutilizadas por otros.

Docker

Contenedor Docker

- Los contenedores se pueden iniciar, detener, mover y eliminar de manera independiente, y representan un entorno aislado y ejecutable para una aplicación.
- Un contenedor es una instancia en tiempo de ejecución de una imagen Docker, que se ejecuta en un sistema operativo host.
- Los contenedores son livianos y comparten el mismo núcleo del sistema operativo host, pero tienen sus propios sistemas de archivos y recursos aislados.

Diferencias Imagen - Contenedor Docker

	Imagen	Contenedor
Naturaleza	Es una imagen estática de solo lectura	Es una instancia en tiempo de ejecución de una imagen
Inmutabilidad	Es inmutable, no se puede modificar después de creada	Es mutable, se puede modificar su estado y datos mientras está en ejecución.
Almacenamiento	Se almacenan en el sistema de archivos del host y, Se componen de múltiples capas	Tienen su propio sistema de archivos y almacenamiento en tiempo de ejecución
Uso	Se utilizan como base para crear contenedores	Son ejecutables y se utilizan para ejecutar aplicaciones y procesos
Persistencia de Datos	Son de sólo lectura, no tienen datos persistentes	Pueden almacenar datos utilizando volúmenes o <i>bind mounts</i>
Escalabilidad y Manejo	Son versiones estáticas y no pueden escalar individualmente	Pueden escalar y administrar contenedores de forma dinámica

Docker

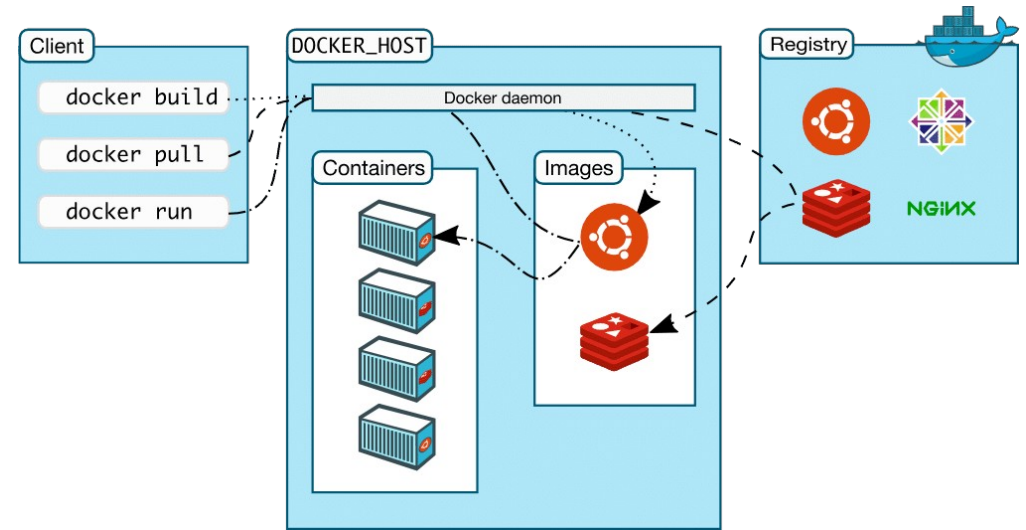
... más conceptos

- **Docker Registry:** es un sistema de almacenamiento y distribución de imágenes Docker. Es una aplicación del lado del servidor que almacena y permite distribuir imágenes Docker.
- **Docker Engine:** es la tecnología subyacente que ejecuta los contenedores Docker. Es un tiempo de ejecución ligero que se ejecuta en la máquina host y gestiona los contenedores.
- **Docker Compose:** es una herramienta para definir y ejecutar aplicaciones Docker multicontenedor. Permite definir los servicios que componen la aplicación, cómo interactúan y cómo deben ejecutarse.
- **Docker Swarm:** es una herramienta nativa de clustering y orquestación para Docker. Le permite crear y gestionar un clúster de nodos Docker, y desplegar y escalar servicios a través del clúster.

Docker

Arquitectura Docker

- Docker usa una arquitectura cliente-servidor.
- El cliente Docker habla con el **daemon Docker**, que hace el trabajo pesado de construir, ejecutar y distribuir sus contenedores Docker.
- El cliente Docker y el daemon pueden ejecutarse en el mismo sistema, o se puede conectar un cliente Docker a un daemon Docker remoto.
- El cliente Docker y el daemon se comunican utilizando una **API REST**, sobre sockets UNIX o una interfaz de red.



Docker

Arquitectura Docker

- El **Docker daemon** (dockerd) escucha las solicitudes de la API de Docker y gestiona los objetos de Docker, como imágenes, contenedores, redes y volúmenes. Un daemon también puede comunicarse con otros daemons para administrar los servicios de Docker.
- El **cliente Docker** (docker) es la forma principal en que muchos usuarios de Docker interactúan con Docker. Cuando se utilizan comandos como docker run, el cliente envía estos comandos a dockerd, que los lleva a cabo. El comando docker usa la API Docker. El cliente de Docker puede comunicarse con más de un daemon.
- **Registros de Docker:** un registro de Docker almacena imágenes de Docker. (Docker Hub o Docker Cloud). Cuando se utilizan los comandos docker pull o docker run, las imágenes necesarias se extraen del registro configurado. Cuando se utiliza el comando docker push, la imagen se envía a el registro configurado.

Docker

Instalación Docker

- Ver los requerimientos y pasos según el SO en la guía que se anexa en Módulo 7



- Una vez instalado verifique que todo funciona bien con:
\$ docker run hello-world

Docker

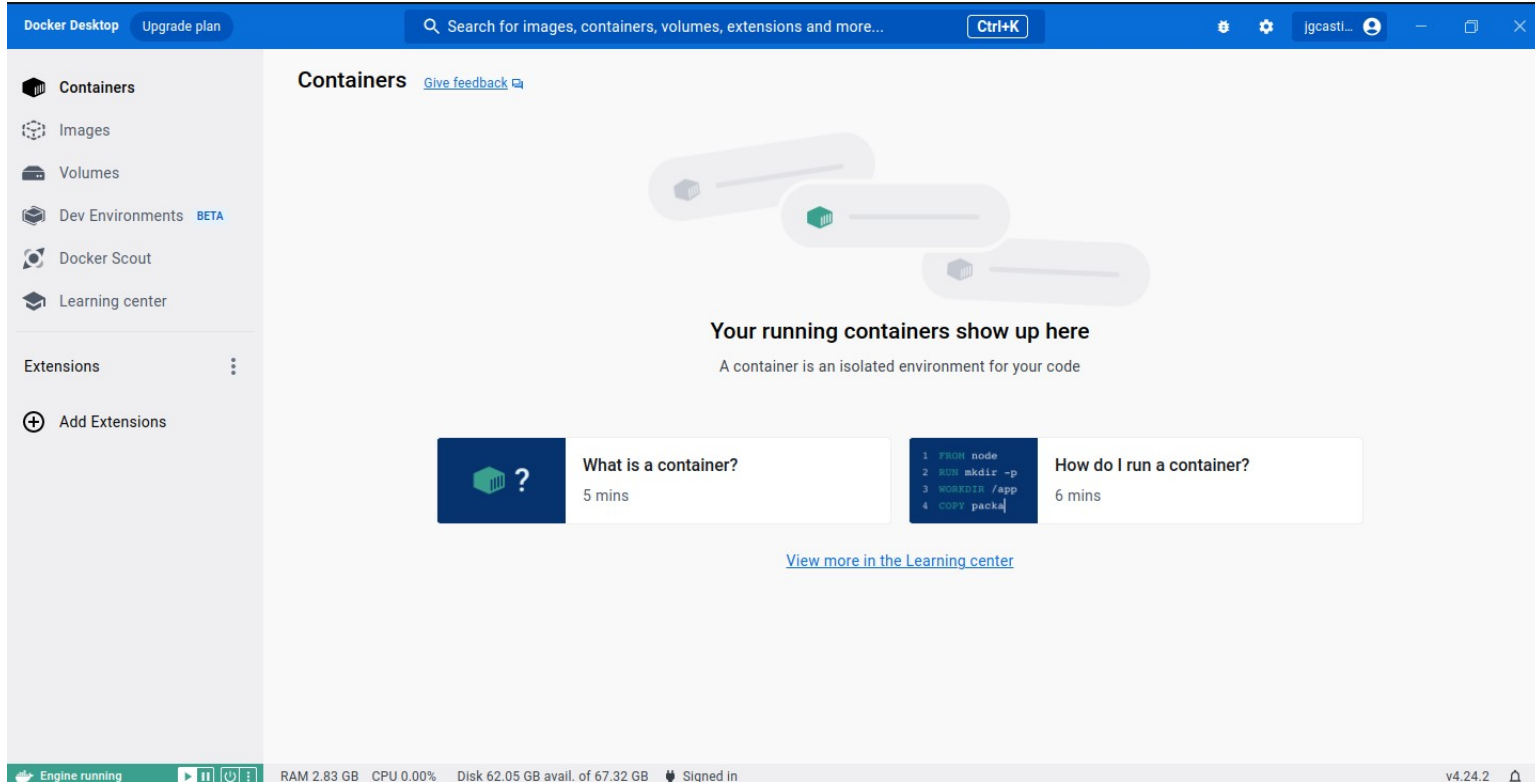
Gestión de imágenes con Docker

- Docker Hub es donde están esas imágenes.
- ¿que es Docker Hub?: **Docker Hub** es un servicio proporcionado por Docker para compartir imágenes de contenedores con el resto del mundo en general. Aunque también es posible compartir tus imágenes de forma privada.
- ¿Y como se comparten esas imágenes? Las imágenes se encuentran en repositorios, almacenes. Estos pueden ser personales o de una empresa, y ser tanto tanto públicos como privados, y es donde se pueden guardar tales imágenes.
- En Docker Hub, con una cuenta gratuita solo se puede tener un repositorio privado.

Docker

Practica

- Docker Desktop



Docker

Practica

- Aunque Docker Desktop funciona muy bien vamos a trabajar mejor desde la terminal del sistema. Esto nos permitirá entender mejor la herramienta
- Usa el comando `$ docker image --help`, esto te dará una lista de comando que son necesarios para el trabajo con Docker:
 - **build**, es la herramienta que permite construir nuestras propias imágenes a partir de un Dockerfile (más adelante trataremos que es el Dockerfile).
 - **history**, muestra la historia de esa imagen, es decir, como se construyó esa imagen. Los pasos que se siguieron para construir la imagen.
 - **inspect**, permite ver los detalles y entrañas de una imagen. Por ejemplo, el número identificador, las etiquetas, la fecha de creación, las variables de entorno, los comandos, la versión de Docker, y otros aspectos más.

Docker

Practica

- **ls**, te muestra todas las imágenes que tienes descargas en el equipo, usando `docker image ls`. Si se desea mostrar también las imágenes intermedias se debe utilizar **`docker image ls -a`**. Si solo se desea ver los números de identificación, ID se utiliza la opción **`-q`**.
- **pull**, permite descargar una imagen de un repositorio. Así, por ejemplo, para descargar una imagen de Ubuntu, se ejecuta **`docker image pull ubuntu`**. Si se desea descargar una versión concreta, se debe añadir la etiqueta. Por ejemplo, si se desea descargar la versión de Ubuntu Xenial, se debe ejecutar **`docker image pull ubuntu:xenial`**. Si no se especifica nada se estará descargando la última, es como si se ejecutara **`docker image pull ubuntu:latest`**.
- **push**, permite subir una imagen a un repositorio.
- **rm**, permite borrar una o varias imágenes. También se puede utilizar **`docker rmi <nombre de la imagen>`**.

Docker

Practica

- Nuestra primera imagen busybox, un UNIX reducido, pero funcional.
- **Una observación:** verás que hay como dos tipos de imágenes a la hora de descargarlas de Docker Hub,
 - Las que se descargan como **docker pull busybox**
 - Otras que tienen la forma **docker pull atareao/baseimage**
- Las primeras son imágenes oficiales, y son la recomendación oficial de imágenes que tienes que utilizar en tus proyectos.
- Las segundas, son imágenes que ha creado una persona como tu o yo, o una empresa. Esto no quita con que no sean fiables. Sin embargo, es mejor conocer la persona o equipo que está detrás de esa imagen.

Docker

Practica:

- Descargar una imagen:
\$ docker pull busybox
- Verificar la imagen:
\$ docker images
- ¿Cómo se hizo la imagen?
\$ docker history busybox
- Detalles:
\$ docker inspect busybox
- Aquí puedes ver:
 - Lo primero es el reducido tamaño de la imagen. Con apenas 4.2 MB ya tienes un UNIX sobre el que empezar a construir tus propias imágenes.
 - En Env encuentras las variables de entorno. En este caso se tiene definido el Path.
 - Por otro lado, en Cmd está la instrucción que se ejecutará al iniciar el contenedor.
- Para eliminar la imagen ejecuta: **\$ docker rmi busybox**

Docker

Gestionar contenedores:

- Anteriormente vimos la gestión de imágenes, ahora se tratará lo referente a la gestión de repositorios.
- Lo concerniente a la ejecución de repositorios se tratará luego.
- Usamos **`$docker container --help`**, lo cual nos da una lista aún mayor que la de imágenes.

Docker

Practica:

- Descargar una imagen: `$ docker pull frodo/alpine-python3`, un Python 3.7 de ~57MB
- Iniciar el contenedor: `$ docker container run frodo/alpine-python3`, o mejor simplemente: `$ docker run frodo/alpine-python3`
- ¿Cuántos contenedores tenemos?: `$ docker ps` O `$ docker container ls`
- ¿No aparece ninguno?, ya que no se están ejecutando, usamos: `$ docker ps -a`
- Algunos detalles de la respuesta:
 - **COMMAND** se refiere al comando que es ejecutado para el arranque.
 - **CREATED** como se observa, indica el tiempo que ha transcurrido desde que se creó el contenedor.
 - **STATUS**, indica en que estado se encuentra, y el tiempo que lleva en ese estado.
 - **NAMES**, indica el nombre del contenedor. Es mas sencillo identificar un contenedor por su nombre que por su **CONTAINER ID**. En el caso de que tu no le pases el nombre, docker generará un nombre de forma aleatoria.

Docker

Practica:

- Colocar nombre al contenedor:
`$ docker run -d --name ejemplo frolovlad/alpinepython3`
- Revisemos la lista de contenedores nuevamente: `$ docker ps -a`
- También es posible renombrar: `$ docker container rename b07882e9b521 perico`
- Donde el número “b0788...” representa el ID del container
- **Arranque y parada del contenedor:**
 - `$ docker run frolovlad/alpine-python3 sleep 5` ← se arranca y espera 5 seg antes de pararse
 - Para tener acceso al contenedor desde la terminal se usa la opción **-d**, así:
`$ docker run -d frolovlad/alpine-python3 sleep 100`
 - Ejecuta `$ docker ps` para verificar que está corriendo.
 - Para detenerlo se usa: `$ docker stop b07882e9b521`
 - Para iniciar de nuevo se usa: `$ docker start b07882e9b521`

Docker

Practica:

- **Otras acciones en el contenedor:**
 - `$ docker kill b07882e9b521`, mata el contenedor.
 - La diferencia entre **stop** y **kill** es que el primero ejecuta una serie de procesos antes de detener el contenedor, mientras que kill lo detiene al instante.
 - Se puede detener e iniciar el contenedor en un solo comando: `$ docker restart b07882e9b521`
 - Para pausar los procesos dentro del contenedor: `$ docker pause b07882e9b521`
 - Para ponerlo nuevamente en funcionamiento: `$ docker unpause b07882e9b521`

Docker

Practica:

- **Ejecución de contenedores Docker:**

- Ya sabemos cómo gestionar imágenes y cómo gestionar contenedores. Ha llegado el momento de empezar a ejecutar los contenedores.

- Debido a que cuando se ejecuta un contenedor por debajo se ejecuta una versión muy sencilla de Linux, conviene conocer un poco de los comandos *bash* . En este sitio existe un tutorial muy completo:

<https://www.freecodecamp.org/espanol/news/la-guia-definitiva-de-linea-de-comandos-de-linux-tutorial-completo-de-bash/>

- Vamos a usar un contenedor de Nginx sobre Alpine, Este es una versión muy reducida de Linux, de apenas 5MB y Nginx, es un servidor web de código abierto, también usado como proxy inverso, cache de HTTP y balanceador de carga.

Docker

Practica:

- **El paso a paso:**

- Ejecutamos el contenedor:

```
$ docker run -d --name test01 nginx:alpine
```

- La opción **-d** es para que el contenedor se ejecute en segundo plano (detached). De otro modo no se podría interactuar con él.
- Peso de la imagen: `$ docker images | grep nginx`
- Más información: `$ docker history nginx:alpine`
- Entrar en el servidor que se está ejecutando: `$ docker exec -it test01 sh`
- Ejecutamos `/ # ps -ef`, observamos que hay unos pocos procesos ejecutándose
- Ya que no podemos ver el Nginx en funcionamiento, no nos sirve de gran cosa así que vamos a detenerlo y eliminarlo:

```
$ docker stop test01  
$ docker rm test01
```

Docker

Practica:

- Vamos a verlo en funcionamiento, expongamos su puerto:
`$ docker run -d --name test01 -p 81:80 nginx:alpine`
- La opción **-p 81:80** no indica que el puerto se expone al exterior por el puerto 81, mientras que dentro del contenedor está en funcionamiento por el 80, lo demos ver en **`http://localhost:81`**
- Se puede exponer mas de un puerto. si se desea tener una conexión segura, se puede exponer el puerto 443. Así:
`$ docker stop test01`
`$ docker rm test01`
`$ docker run -d --name test01 -p 80:80 -p 443:443 nginx:alpine`
- Ahora podemos verlo por **`http://localhost`** o por **`https://localhost`**, aunque esté último nos dará un error por no contar con un certificado de seguridad.
- En la guía hay una forma de obtener un certificado, queda como ejercicio probarlo.

Docker

Practica:

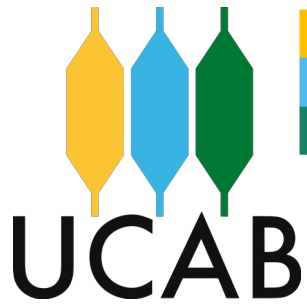
- Cada vez que queramos modificar la configuración de Nginx, o cada vez que queramos cambiar los certificados se deberá modificar la imagen.
- No sólo eso. Cada vez que queramos cambiar la web, vamos a tener que cambiar la imagen.
- Esto no es ni productivo ni mucho menos. Es necesario buscar otra solución.
- La solución es algo similar a lo que se hizo con los puertos, pero esta vez con directorios. Se trata de engañar al contenedor, haciéndole pensar que un directorio del equipo es un directorio del contenedor.

```
$ docker run -d -p 80:80 -p 443:443 -v "$(pwd)"/crt:/etc/ssl/certs \
-v "$(pwd)"/conf:/etc/nginx/conf.d -v "$(pwd)"/web:/var/www/html \
--name test01 jgcastillo/nginx:autocertificado
(ver en /etc/nginx/conf.d/default.conf)
```

- Esto modifica la configuración del Nginx: hace que se vea el puerto 443, pero además en el directorio **crt** se han colocado los certificados, en el directorio **conf** la configuración del Nginx y en el directorio **web**, el contenido de la web.
- Detengamos el contenedor y eliminemos la imagen.

FIN





Programación de APIs con Python



Docker

Profesor:
José Gregorio Castillo Pacheco

Docker

Temario

- Docker Hub
- Docker Compose

Docker

¿Qué es Docker Hub?

- Docker Hub es un servicio en la nube que actúa como un repositorio centralizado para imágenes Docker.
- Permite a los usuarios almacenar, distribuir y compartir imágenes y contenedores Docker de forma pública o privada.

Características y beneficios de Docker Hub

- Registro centralizado de imágenes
- Almacenamiento de imágenes y contenedores
- Acceso a imágenes públicas y oficiales
- Colaboración y compartición
- Seguridad y verificación de imágenes
- Automatización de builds
- Etiquetado y versionado
- Integración con Docker CLI y Docker Compose
- Repositorios privados
- Estadísticas y Monitoreo

Docker

¿Qué hay en los repositorios de Docker Hub?

- En cada repositorio tienes diferentes versiones de una misma imagen.
- Por ejemplo, si vas al repositorio de Ubuntu. encontrarás imágenes para distintas versiones de Ubuntu, eoan, disco, xenial, etc.
- Además para cada una de estas versiones existen imágenes para diferentes arquitecturas, 386, amd64, arm/v7, arm64/v8, etc.

Docker

Trabajar con Docker Hub

- Lo primero es crear una cuenta, si no se tiene.
- Las opciones pagas ofrecen más características que la gratuita, pero para nuestro caso es suficiente con lo que ofrece la gratuita.
- Ahora lo siguiente es usar la imagen de la clase anterior y subirla a nuestro repositorio:
 - Busca en la guía del curso los pasos de creación de la imagen y súbela al repositorio de tu cuenta.

Docker

Trabajar con Docker Hub

- Con Dockerfile, podemos crear nuestra propia imagen desde cero y luego subirla al repositorio de Docker Hub y usarla desde allí.
- ¿Qué es Dockerfile?
 - Es un archivo que está compuesto por una serie de comandos que son los responsables de la construcción de la imagen.

Docker

Trabajar con Docker Hub

- Comandos del Dockerfile:
 - ADD copia un archivo del host al contenedor
 - CMD el argumento que se pasa por defecto
 - ENTRYPOINT el comando que se ejecuta por defecto al arrancar el contenedor
 - ENV permite declarar una variable de entorno en el contenedor
 - EXPOSE abre un puerto del contenedor
 - FROM indica la imagen base que se utilizará para construir la imagen personalizada. Esta opción es obligatoria, y además debe ser la primera instrucción del Dockerfile.

Docker

Trabajar con Docker Hub

- Comandos del Dockerfile:
 - MAINTAINER es un valor opcional que permite indicar quién es el que se encarga de mantener el Dockerfile
 - ONBUILD permite indicar un comando que se ejecutará cuando la imagen sea utilizada para crear otra imagen.
 - RUN ejecuta un comando y guarda el resultado como una nueva capa.
 - USER define el usuario por defecto del contenedor
 - VOLUME crea un volumen que es compartido por los diferentes contenedores o con el host
 - WORKDIR define el directorio de trabajo para el contenedor.

Docker

Trabajar con Docker Hub

- Ejemplo:
 - Nos guiaremos por los ejemplos de la guía, páginas: 52 ~ 55

Docker

Docker Compose

- Docker Compose es una herramienta de Docker que orquesta contenedores en un mismo cliente.
- Consiste en un archivo de texto en formato YAML, que define de forma declarativa los contenedores que se van a desplegar, así como las dependencias entre ellos.
- Al utilizar el paradigma declarativo, en este documento sólo se especifican las características de los contenedores deseados, y no cómo se despliegan.
- Para el despliegue, se basa en la definición de servicios, que referencian imágenes Docker de un registro y las características de los contenedores que se desean desplegar.
- Dentro del archivo **docker-compose.yml** podemos definir diferentes recursos que desplegar. A la hora de desplegar nuestras aplicaciones, lo primero es definir los servicios que vamos a desplegar.

Docker

Docker Compose

- Servicios y recursos:
 - Estos servicios son los que conforman nuestra aplicación y, normalmente, cada uno despliega un contenedor, asociado a una imagen Docker.
 - Dentro de cada servicio podemos definir las características de cada contenedor, como el nombre, la imagen, los puertos que expone, volúmenes y redes a las que se conecta, etc.

Docker

Dockcer Compose

- Para definir un servicio en el archivo `docker-compose.yml`, debemos definir un bloque con el nombre del servicio, y dentro una serie de propiedades que definen las características del contenedor:
 - `image`: Nombre de la imagen Docker usada para desplegar el contenedor.
 - `[container_name]`: Nombre del contenedor a desplegar.
 - `[build]`: Ruta al directorio donde se encuentra el Dockerfile para construir la imagen, si no está construida.
 - `[command]`: Comandos que ejecutar al iniciar el contenedor.
 - `[ports]`: Puertos que exponer al exterior del contenedor. Tienen el formato `puerto_host:puerto_contenedor`, donde, como host, podremos acceder al servicio desde `localhost:puerto_host`.
 - `[volumes]`: Volúmenes que montar en el contenedor.

Docker

Dockcer Compose

- [environment]: Variables de entorno que definir en el contenedor. Es recomendable configurar nuestros contenedores con variables de entorno y no valores fijos, para poder cambiar la configuración de los servicios sin tener que reconstruir las imágenes.
- [depends_on]: Servicios que deben desplegarse antes que este.
- [networks]: Redes a las que pertenece el contenedor.
- [restart]: Política de reinicio del contenedor. Puede tomar los valores *no*, *always*, *on-failure*, *unless-stopped*, por defecto *no*.

Docker

Dockcer Compose

- Volúmenes: Cuando hablamos de volumen nos referimos a los volúmenes de Docker, el mecanismo que tienen los contenedores para persistir y compartir datos. Para ello, consumen o generan archivos en un directorio del sistema de archivos del host.
- Para definir un volumen en el archivo `docker-compose.yml`, debemos definir un bloque con el nombre del volumen, y luego referenciarlo en los servicios que lo necesiten. Dentro de este bloque podemos definir las siguientes propiedades:
 - `[driver]`: Driver del volumen. Por defecto, *local*.
 - `[driver_opts]`: Opciones del driver del volumen.
 - `type`: Tipo de volumen. Puede ser *none*, *bind*, *volume* o *tmpfs*.
 - `device`: Ruta al directorio del host que se va a montar.
 - `o`: Opciones de montaje del volumen. Puede tomar valores *bind*, *private*, *ro*, *rw* y *shared*, entre otros.
 - `[external]`: Indica si el volumen es externo o no. Por defecto, *false*.
 - `[labels]`: Etiquetas del volumen.
 - `[name]`: Nombre del volumen.
 - `[scope]`: Alcance del volumen. Por defecto, *local*.

Docker

Dockcer Compose

- **Redes:** Cuando desplegamos aplicaciones con múltiples módulos, por ejemplo, una aplicación web con un servidor web y una base de datos, las alojamos en contenedores separados. Para que estos contenedores puedan comunicarse entre sí, necesitamos definir una red en común, siguiendo el modelo **Container Network Model**:
 - **SandBox:** Aisla el contenedor del resto del sistema, limitando el acceso a esta red al tráfico que llega por los endpoints.
 - **Endpoints:** Puntos de comunicación entre las redes aisladas de los contenedores y la network que los conecta con el resto del sistema.
 - **Network:** Red que comunica las sandbox de los diferentes contenedores por medio de los endpoints.

Dockcer Compose

- **Redes:** Siguiendo este modelo existen varias implementaciones en Docker:

Nombre	Alcance	Descripción
bridge	Local	Red por defecto de Docker. Crea una red virtual en el host, que conecta los contenedores por medio de un bridge virtual.
host	Local	Deshabilita el aislamiento entre los contenedores y el host, no hace falta exponer puertos
overlay	Global	Permite conectar múltiples daemons Docker entre sí y habilitar la comunicación entre servicios distribuidos.
ipvlan	Global	El usuario obtiene el control total del direccionamiento IPv4 e IPv6 de la red.
macvlan	Global	Permite asignarle una dirección MAC a un contenedor, haciendo que aparezca como un dispositivo físico en la red. El tráfico se direcciona por MAC.
none	Local	Dehabilita toda la gestión de red, normalmente usado en conjunto con un driver de red propio.

Estas implementaciones se definen en el campo *driver* de la sección *networks* del archivo *docker-compose.yml*. Por defecto, si no se especifica, se utiliza la *red bridge*.

Docker

Docker Compose, ejemplo:

- Vamos a crear un ejemplo sencillo, que consiste en un servicio web que utiliza una base de datos. Para ello, el archivo docker-compose.yml quedaría de la siguiente manera:

```
version: '3.7' # Versión de docker-compose, depende de Docker
services:
  web:
    image: nginx
    ports:
      - 80:80
    volumes:
      - web_data:/usr/share/nginx/html
    networks:
      - web_net
  db:
    image: postgres
    volumes:
      - db_data:/var/lib/postgresql/data
    networks:
      - web_net
volumes:
  web_data:
  db_data:
networks:
  web_net:
    driver: bridge
```

Docker

Docker Compose, ejemplo:

- Dependencias entre servicios:
 - Ahora que ya sabemos cómo definir los servicios de nuestra aplicación y sus recursos asociados, vamos a ver cómo podemos definir las dependencias entre ellos.
 - Un servicio depende de otro cuando necesita que el segundo esté desplegado antes de poder desplegarse.
 - Por ejemplo, si tenemos un servicio que despliega una API REST conectada a una base de datos, necesitamos que la base de datos esté desplegada antes de desplegar el servicio que la utiliza.
- El ejemplo quedaría cómo se muestra en la lámina siguiente:

Dockcer Compose, ejemplo:

```
version: '3.7' # Versión de docker-compose, depende de Docker
services:
  web:
    image: nginx
    ports:
      - 80:80
    volumes:
      - web_data:/usr/share/nginx/html
    networks:
      - web_net
    depends_on: # Ahora web depende de db
      - db
  db:
    image: postgres
    volumes:
      - db_data:/var/lib/postgresql/data
    networks:
      - web_net
volumes:
  web_data:
  db_data:
networks:
  web_net:
    driver: bridge
```

Docker

Docker Compose, comandos:

- Una vez definido el archivo `docker-compose.yml`, los siguientes comandos son útiles para gestionar el despliegue de nuestra aplicación y sus recursos asociados:
 - **docker-compose up**: Sirve para desplegar la aplicación. Si no se especifica ningún servicio, se desplegarán todos los servicios definidos en el fichero `docker-compose.yml`. Si se especifica un servicio, se desplegará sólo ese servicio y sus dependencias.
 - **docker-compose down**: Aunque no hayamos utilizado el parámetro `-d` en el comando `docker-compose up`, para detener la ejecución de los contenedores correctamente debemos ejecutar este comando
 - **docker-compose ps**: Sirve para listar los servicios desplegados, y sus contenedores asociados, y ver su estado actual junto con los puertos que tienen expuestos. Es muy útil para comprobar si el despliegue definido en el fichero `docker-compose.yml` es el esperado.
 - **docker-compose logs**: Sirve para ver los logs de los contenedores desplegados.
 - **docker-compose exec**: Sirve para ejecutar un comando en un contenedor desplegado. Es muy útil si queremos conectarnos al contenedor de un servicio para depurar algún tipo de errores.
 - **docker-compose build** o **docker-compose pull**: `build` sirve para reconstruir las imágenes de los servicios definidos en el fichero `docker-compose.yml`.

Docker Compose, comandos:

- Para un ejemplo práctico recomendar realizar el que se encuentra en:
<https://www.adictosaltrabajo.com/2022/12/19/despliegue-de-aplicaciones-con-docker-compose/>

Ejemplo práctico

Ahora que ya somos expertos en docker-compose y tenemos una chuleta de comandos que podemos utilizar, vamos a desplegar una aplicación de ejemplo para poner a prueba nuestros nuevos conocimientos.

1. Aplicaciones de ejemplo

Para este ejemplo vamos a utilizar dos aplicaciones de ejemplo:

- **Servidor web** : Una API REST, en nodejs, que nos permite consultar el estado de una base de datos de coches.
- **Base de datos** : Una base de datos MySQL que contiene la tabla con información sobre coches.

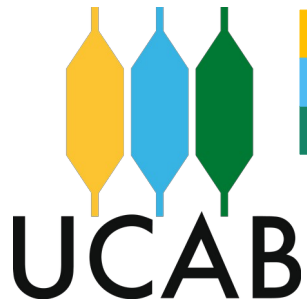
En un proyecto real programaréis vuestro propio servicio web, pero en este caso vamos a clonar la aplicación del siguiente [repositorio](#). Para ello, ejecutamos el siguiente comando:

```
1 # Podemos clonar por https
2 https://github.com/nicolaeolnar/dockerizeNode.git
```



FIN





Programación de APIs con Python



Desarrollo Orientado a Pruebas (TDD)

Profesor:
José Gregorio Castillo Pacheco

TDD

Temario

- TDD:
 - Introducción
 - Beneficios TDD
 - Ciclo de desarrollo TDD
 - Mejores prácticas TDD
 - Pytest y FastAPI

TDD

- Introducción:
 - El Desarrollo Orientado a Pruebas (TDD) es una metodología fundamental en el mundo del desarrollo de software, que se centra en escribir pruebas antes de comenzar a implementar el código.
 - Su importancia radica en la creación de software más robusto, fiable y fácilmente mantenible. Al seguir el ciclo de desarrollo "Red-Green-Refactor", los desarrolladores escriben pruebas automatizadas primero, implementan la funcionalidad necesaria para que las pruebas pasen y luego refactorizan el código según sea necesario.
 - Este enfoque proactivo garantiza que el código esté constantemente validado y mejora la calidad general del producto.

TDD

- Introducción, objetivos:
 - Comprender los conceptos básicos de TDD y su papel en el desarrollo de APIs con Python.
 - Reconocer la importancia de las pruebas en el proceso de desarrollo y sus beneficios a largo plazo.
 - Conocer los pasos clave del ciclo TDD (Red-Green-Refactor) y cómo aplicarlos en proyectos reales.
 - Familiarizarse con la herramienta pytest y su integración en proyectos FastAPI.
 - Desarrollar una aplicación simple de blog utilizando TDD y un ambiente virtual, con énfasis en la estructura del proyecto.

TDD

Beneficios de TDD:

- Mejora de la Calidad del Código
 - El TDD es una práctica esencial que contribuye significativamente a la mejora de la calidad del código.
 - Al escribir pruebas antes de implementar la funcionalidad, los desarrolladores establecen estándares claros y objetivos de comportamiento para cada componente.
 - Esto asegura que cada parte del código cumpla con las expectativas definidas, reduciendo la probabilidad de errores y facilitando la identificación rápida de posibles problemas.

TDD

Beneficios de TDD:

- Identificación Temprana de Errores:
 - Una de las principales ventajas de TDD es la capacidad de identificar errores de manera temprana en el proceso de desarrollo.
 - Las pruebas automáticas ejecutadas de manera regular permiten descubrir y abordar problemas potenciales antes de que se integren profundamente en el código.
 - Esto no solo ahorra tiempo a largo plazo, sino que también mejora la confianza en la estabilidad del software.

TDD

Beneficios de TDD:

- Facilita el Mantenimiento y la Escalabilidad:
 - El TDD también juega un papel crucial en el mantenimiento y la escalabilidad del código.
 - Al tener un conjunto completo de pruebas, los desarrolladores pueden realizar cambios y actualizaciones con confianza, sabiendo que las pruebas actuarán como un mecanismo de verificación.
 - Además, las pruebas proporcionan documentación viva, facilitando la comprensión de la lógica del código y permitiendo a los equipos crecer y escalar proyectos de manera más eficiente.


TDD

Ciclo de Desarrollo TDD (Red-Green-Refactor):

- El TDD sigue un ciclo de desarrollo conocido como "Red-Green-Refactor". Este ciclo consta de tres fases distintas que se repiten a lo largo del proceso de codificación:
- Fase Red: ●
 - En la fase "Red", los desarrolladores escriben pruebas automatizadas que describen la funcionalidad que están a punto de implementar.
 - Estas pruebas inicialmente fallan ya que la funcionalidad aún no existe.
 - El objetivo es tener pruebas que representen las expectativas del comportamiento deseado.

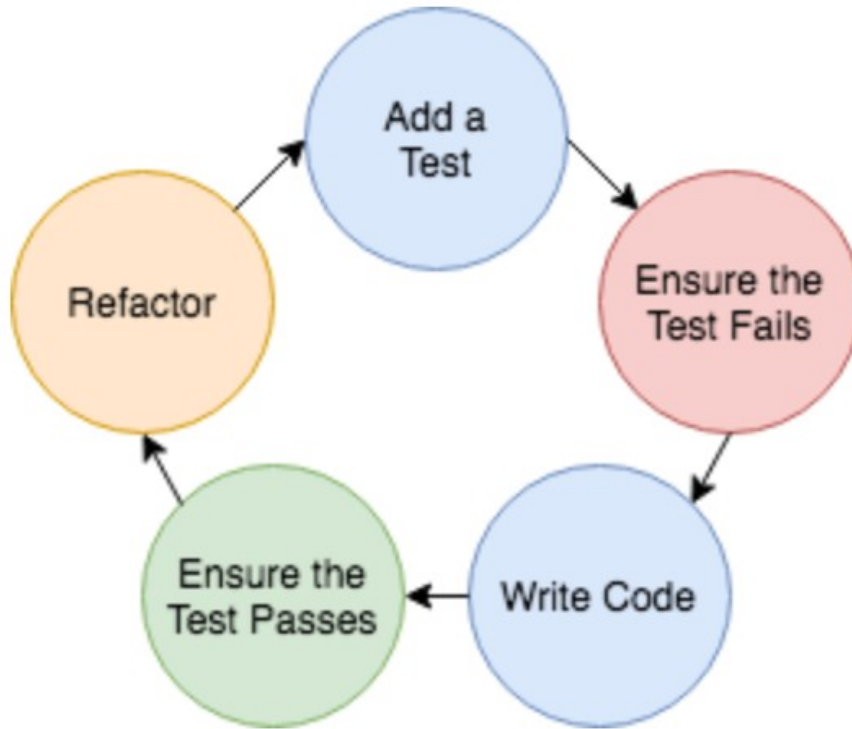
TDD

Ciclo de Desarrollo TDD (Red-Green-Refactor):

- Fase Green: ●
 - En la fase "Green", los desarrolladores implementan la funcionalidad mínima necesaria para que las pruebas pasen.
 - Este enfoque incremental asegura que solo se implementa lo estrictamente necesario para que las pruebas tengan éxito, evitando la creación de código innecesario.
- Fase Refactor 
 - La fase "Refactor" implica mejorar el código sin cambiar su comportamiento.
 - Después de que las pruebas han pasado, los desarrolladores pueden optimizar y limpiar el código, asegurando que cumpla con los estándares de calidad sin afectar la funcionalidad.
 - Este paso es esencial para mantener un código claro y mantenible.

TDD

Ciclo de Desarrollo TDD (Red-Green-Refactor):



1. Agregue un test al grupo de pruebas
2. (**Red**) Ejecute todos los test y asegurese que el nuevo falla.
3. (**Green**) Escriba el mínimo código que hace que este último pase bien.
4. Ejecute todos los tests
5. (**Refactor**) Mejore el código inicial mientras mantiene los test en verde
6. Repita

TDD

Mejores Prácticas en TDD:

- Nombrar Pruebas de Manera Significativa:
 - Nombrar las pruebas de manera significativa es una práctica clave en TDD.
 - Los nombres de las pruebas deben describir claramente el escenario que están probando.
 - Un nombre bien elegido actúa como documentación viva, permitiendo que otros desarrolladores comprendan la funcionalidad probada sin necesidad de revisar el código.

TDD

Mejores Prácticas en TDD:

- Pruebas Atómicas y Pruebas Aisladas
 - Las pruebas atómicas se centran en probar una única funcionalidad o unidad de código, mientras que las pruebas aisladas se ejecutan independientemente de otras pruebas.
 - Estas prácticas aseguran que cada prueba sea específica y no dependa de otras para su ejecución.
 - La ejecución de pruebas atómicas y aisladas facilita la identificación de problemas específicos sin afectar otras partes del código.

TDD

Mejores Prácticas en TDD:

- Iteraciones Pequeñas y Frecuentes
 - Realizar iteraciones pequeñas y frecuentes es esencial en TDD.
 - En lugar de implementar grandes cantidades de código antes de ejecutar pruebas, los desarrolladores realizan cambios incrementales y prueban cada ajuste por separado.
 - Este enfoque permite una retroalimentación constante, facilitando la identificación temprana de problemas y asegurando un desarrollo más ágil y adaptativo.
- Con estas mejores prácticas, los equipos pueden maximizar la eficiencia de sus pruebas y mejorar la calidad general del código, facilitando el mantenimiento y la evolución del software de manera sostenible.

TDD

Introducción a pytest:

- ¿Qué es pytest y por qué usarlo?
 - Pytest es un framework de pruebas en Python que se destaca por su simplicidad y potencia.
 - Su sintaxis clara y expresiva facilita la escritura de pruebas, y su capacidad para manejar una amplia variedad de escenarios hace que sea una elección popular entre los desarrolladores.
 - La flexibilidad de pytest permite abordar tanto pruebas unitarias simples como pruebas de extremo a extremo complejas, haciendo que sea una herramienta versátil y eficaz para el desarrollo y el mantenimiento de código.

TDD

Introducción a pytest:

- Configuración Básica de pytest:
 - La configuración de pytest es sencilla y se integra fácilmente en proyectos existentes.
 - Basta con tener archivos de prueba que sigan la convención de nombres (por ejemplo, `test_*.py`), y pytest los identificará automáticamente.
 - Esto elimina la necesidad de configuraciones extensas, permitiendo a los desarrolladores centrarse en escribir pruebas significativas.

TDD

Introducción a pytest:

- Instalación de pytest:
 - La instalación de pytest es rápida y directa.
 - Se puede realizar mediante el uso de herramientas de gestión de paquetes como pip. La comunidad activa de pytest proporciona una amplia documentación y recursos, facilitando su adopción en proyectos nuevos y existentes.:
- Estructura Inicial de Pruebas con pytest
 - La estructura inicial de las pruebas con pytest es minimalista.
 - Al seguir la convención de nombres y ubicar pruebas en archivos específicos, los desarrolladores pueden organizar fácilmente sus casos de prueba.
 - La simplicidad de la estructura inicial permite una rápida implementación y ejecución de pruebas, fomentando la adopción ágil de pytest en el proceso de desarrollo.

TDD

Pytest y FastAPI:

- Configuración Específica para Proyectos FastAPI usando pytest:
 - La integración de pytest en proyectos FastAPI es sencilla y potente.
 - FastAPI es compatible con pytest de manera nativa, lo que facilita la ejecución de pruebas en el entorno FastAPI.
 - La configuración con FastAPI implica agregar algunos detalles adicionales, como importar la aplicación FastAPI y configurar un cliente de prueba para simular solicitudes HTTP.

TDD

Pytest y FastAPI:

- Significado y Uso de Fixtures en pytest:
 - Las **fixtures** en pytest son funciones especiales que proporcionan datos o configuraciones predefinidas para las pruebas.
 - En el contexto de FastAPI, las fixtures son particularmente útiles para:
 - inicializar y limpiar bases de datos temporales,
 - configurar el cliente de prueba, y
 - gestionar otros recursos necesarios para las pruebas.
 - Al utilizar fixtures, los desarrolladores pueden escribir pruebas más eficientes y centrarse en la lógica específica que están probando.

TDD

Pytest y FastAPI, ejemplos:

```
# En el archivo conftest.py (ubicado en el directorio raíz de las pruebas)
from fastapi.testclient import TestClient
import pytest
from myapp.main import app # Importa la instancia de la aplicación del proyecto

@pytest.fixture
def client():
    """Fixture para proporcionar un cliente de prueba para las pruebas."""
    return TestClient(app)
```

client: Es un fixture que proporciona un cliente de prueba de FastAPI para simular solicitudes HTTP en tus pruebas.

TDD

Pytest y FastAPI, ejemplos:

```
# Otro ejemplo de fixture para inicializar y limpiar una base de datos temporal
@pytest.fixture(scope="module")
def database():
    """Fixture para configurar una base de datos temporal."""
    # Configuración de la base de datos, inicialización, migraciones, etc.
    # Aquí se puede utilizar alguna librería como SQLAlchemy o cualquier ORM

    # Proporciona la conexión a la base de datos a las pruebas
    yield some_database_connection

    # Limpieza después de que las pruebas se hayan ejecutado
    cleanup_database()
```

database: Es un fixture que puede ser utilizado para configurar una base de datos temporal antes de las pruebas y limpiarla después.

TDD

Pytest y FastAPI, ejemplos:

```
# Otro fixture que depende de 'database' para probar la funcionalidad de tu API
@pytest.fixture
def item_with_database(database):
    """Fixture que depende de la base de datos."""
    # Crea un objeto en la base de datos para usar en las pruebas
    item_id = create_item_in_database()
    return item_id
```

item_with_database: Es un ejemplo de un fixture que depende del fixture database. Se puede crear fixtures más complejos dependientes de otros para satisfacer las necesidades específicas de las pruebas.

TDD

Respuestas de Pruebas en FastAPI:

- En FastAPI, las respuestas de las pruebas se centran en los códigos de estado HTTP, indicando el resultado de la solicitud. Algunos códigos comunes son:
 - **200 OK**: La solicitud fue exitosa.
 - **201 Created**: La solicitud ha tenido éxito y se ha creado un nuevo recurso.
 - **400 Bad Request**: La solicitud no pudo ser procesada debido a un error del cliente.
 - **404 Not Found**: El recurso solicitado no se ha encontrado.
 - **500 Internal Server Error**: Indica un error interno en el servidor.

Estos códigos proporcionan información valiosa sobre el estado de la aplicación y son esenciales para garantizar que las API funcionen como se espera.

TDD

Uso de Assertions para Verificar Respuestas:

- Las assertions (afirmaciones) son fundamentales para verificar las respuestas en las pruebas.
- En FastAPI, se utilizan assertions para asegurarse de que la respuesta de una solicitud coincide con las expectativas.
- Por ejemplo, se pueden utilizar assertions para verificar el código de estado, el contenido de la respuesta, o cualquier otra característica relevante.

```
# Ejemplo de una assertion en una prueba con pytest y FastAPI  
def test_read_item():  
    response = client.get("/items/42")  
    assert response.status_code == 200  
    assert response.json() == {"item_id": 42, "description": "Some Item"}
```

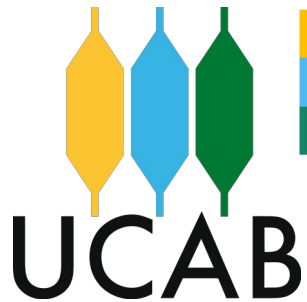

TDD

referencias:

- <https://docs.pytest.org/en/latest/>
- <https://testdriven.io/>
- <https://pytest-with-eric.com/>
- <https://www.jeffastor.com/>
- <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>

FIN





Programación de APIs con Python



Domain Driven Design (DDD)

Profesor:
José Gregorio Castillo Pacheco

DDD

Temario

- ¿Qué es DDD?
- Cómo funciona DDD
- Conceptos claves DDD
- Patrones DDD

- **¿Qué es DDD?**
 - El Desarrollo Orientado al Diseño (DDD) es una metodología de software que busca modelar el dominio del problema de forma cercana y precisa utilizando el lenguaje del código.
- **¿Para qué sirve DDD?**
 - DDD nos ayuda a crear software escalable, mantenible y adaptable a los cambios del negocio. Esto se logra mediante la identificación de los conceptos clave del dominio y su representación en el código de forma clara y consistente.

- **¿Cómo funciona DDD?**
- DDD se basa en principios como:
 - **Centrarse en el dominio:** El software debe modelar el dominio del problema con precisión.
 - **Ubiquidad del lenguaje:** El lenguaje del dominio debe ser utilizado tanto en el código como en la comunicación entre los equipos.
 - **Diseño por dominios:** El software debe estructurarse en torno a dominios autónomos.
 - **Modelo de dominio anémico:** El modelo de dominio debe ser rico en comportamiento.

- **Conceptos clave de DDD:**
 - **Dominio:** descripción del problema que se resuelve.
 - **Modelo de dominio:** representación del dominio en términos de software.
 - **Entidades (Entities):** objetos que representan conceptos importantes del dominio.
 - **Valor (Value Objects):** objetos que encapsulan información importante del dominio.
 - **Agregados (aggregates):** grupos de entidades que mantienen la coherencia interna.
 - **Servicios de dominio:** operaciones que se realizan sobre agregados.

- **Conceptos clave de DDD, con un ejemplo:**
 - **Dominio:** Imagina que estás creando un videojuego de rol. El "dominio" sería todo lo que engloba este juego: desde los personajes, sus habilidades y objetos, hasta las reglas del mundo y la historia que se desarrolla.
 - **Modelo de dominio:** Es como un mapa del videojuego, pero hecho con código. Representa los elementos del dominio (personajes, objetos, etc.) y cómo se relacionan entre sí. Es la forma en que el software "entiende" el juego.
 - **Entidades:** Las entidades son como los personajes principales del juego. Cada entidad tiene atributos (nombre, nivel, vida) y un comportamiento (atacar, defender, usar habilidades). Por ejemplo, un personaje podría ser una entidad llamada "Guerrero" con atributos como "fuerza" y "armadura".

- **Conceptos clave de DDD, con un ejemplo:**
 - **Valor:** Los valores son como las características o detalles de las entidades. No tienen identidad propia, pero sí aportan información importante. Por ejemplo, el "arma" del Guerrero podría ser un valor con atributos como "daño" y "tipo".
 - **Agregados:** Imagina que el Guerrero y su arma son como un equipo inseparable. Un "agregado" es un grupo de entidades y valores que deben mantenerse juntos y tratarse como una unidad. El Guerrero y su arma forman un agregado porque su funcionamiento está interrelacionado.
 - **Servicios de dominio:** Los servicios de dominio son como las acciones que se pueden realizar en el juego. Son operaciones que se ejecutan sobre agregados para lograr un objetivo específico. Por ejemplo, un servicio de dominio podría ser "atacar" que permite al Guerrero usar su arma para dañar a un enemigo.

- **Conceptos clave de DDD, otro ejemplo**
- Tienda online:
 - El dominio sería la gestión de productos, pedidos y clientes.
 - El modelo de dominio representaría estos elementos en código, con entidades como "Producto", "Pedido" y "Cliente".
 - Los valores podrían ser detalles como el precio de un producto o la dirección de un cliente.
 - Los agregados podrían ser un pedido con sus productos asociados o un cliente con su historial de compras.
 - Los servicios de dominio serían operaciones como "agregar un producto al carrito" o "procesar un pedido".

- **Conceptos clave de DDD, otro ejemplo**
- Tienda online:
 - El dominio sería la gestión de productos, pedidos y clientes.
 - El modelo de dominio representaría estos elementos en código, con entidades como "Producto", "Pedido" y "Cliente".
 - Los valores podrían ser detalles como el precio de un producto o la dirección de un cliente.
 - Los agregados podrían ser un pedido con sus productos asociados o un cliente con su historial de compras.
 - Los servicios de dominio serían operaciones como "agregar un producto al carrito" o "procesar un pedido".

DDD

- **Patrones de DDD:**
 - Los conceptos básicos se apoyan en patrones para llevar a cabo un diseño eficaz
 - No todos los diseños son rigurosamente obligatorios, por lo que se sugiere usar los que realmente sean necesarios
 - Mientras más grandes los proyectos, más necesarios se hacen estos patrones

- **Patrones de DDD:**

- **Patrón Entidad**, Ejemplo: En un sistema de gestión de pedidos, una entidad podría ser "Pedido". Esta entidad tendría atributos como "idPedido", "fechaPedido", "cliente", "productos", "estado", etc.
- **Patrón Valor**, Ejemplo: En el mismo sistema de gestión de pedidos, un valor podría ser "Dirección". Este valor tendría atributos como "calle", "número", "ciudad", "código postal", "país", etc. La dirección se asocia a la entidad "Cliente".
- **Patrón Fábrica**, Ejemplo: En el sistema de gestión de pedidos, una fábrica podría ser "PedidoFactory". Esta fábrica sería responsable de crear objetos "Pedido" con los datos necesarios. Por ejemplo, podría recibir como entrada el id del cliente, los productos seleccionados y la dirección de envío, y devolver un nuevo objeto "Pedido" con estos datos.

- **Patrones de DDD:**
 - **Patrón Repositorio**, Ejemplo: En el sistema de gestión de pedidos, un repositorio podría ser "PedidoRepository". Este repositorio sería responsable de almacenar y recuperar objetos "Pedido" de una base de datos. Por ejemplo, podría permitir guardar un nuevo pedido, obtener un pedido por su ID, o eliminar un pedido.
 - **Patrón Servicio de Dominio**, Ejemplo: En el sistema de gestión de pedidos, un servicio de dominio podría ser "CalcularTotalPedidoService". Este servicio sería responsable de calcular el total de un pedido, incluyendo los precios de los productos, los impuestos y los descuentos.
 - **Patrón Evento de Dominio**, Ejemplo: En el sistema de gestión de pedidos, un evento de dominio podría ser "PedidoCreado". Este evento se dispararía cada vez que se crea un nuevo pedido. Los suscriptores al evento, como un servicio de notificación por correo electrónico, podrían ser notificados cuando se crea un nuevo pedido.

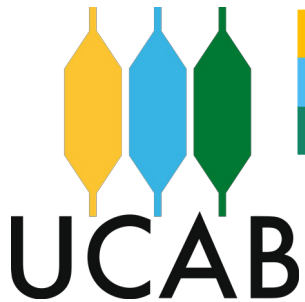
- **Ejemplo:**
 - Desarrollo de un API sencillo para probar el concepto

Referencias

- "Architecture Patterns with Python", Harry Percival, Bob Gregory. Puede ser leído en: <https://www.cosmicpython.com/book/preface.html>
- "Domain-Driven Design: Tackling Complexity in the Heart of Software", Eric Evans
 - Descripción: Este libro es una lectura fundamental para comprender los principios y prácticas de Domain Driven Design. Eric Evans proporciona una guía completa y práctica para aplicar DDD en proyectos de software.
- "Implementing Domain-Driven Design", Vaughn Vernon
 - Descripción: Vaughn Vernon explora la implementación práctica de DDD en este libro, ofreciendo consejos detallados y ejemplos concretos. Es una continuación valiosa después de haber entendido los conceptos básicos de DDD.
- Domain-Driven Design Community, URL: <https://www.dddcommunity.org/>
 - Descripción: Este sitio es una fuente rica en recursos relacionados con DDD. Incluye artículos, presentaciones y foros donde los profesionales comparten sus experiencias y conocimientos.
- "Strategic Design - Context Mapping", Alberto Brandolini, <https://www.infoq.com/articles/ddd-contextmapping/>
 - Descripción: Alberto Brandolini explora el concepto de mapeo de contextos, una técnica esencial en la estrategia de diseño dentro de DDD. Este artículo ofrece una perspectiva práctica y casos de uso.
- FastAPI CRUD Application, <https://www.gormanalysis.com/blog/building-a-simple-crud-application-with-fastapi/>
 - Descripción: GormAnalysis proporciona un tutorial paso a paso para construir una aplicación CRUD (Crear, Leer, Actualizar, Eliminar) con FastAPI. Es una excelente manera de aplicar los conceptos de DDD en un escenario práctico.

FIN





Programación de APIs con Python



Alembic, SQLAlchemy, Async

Profesor:
José Gregorio Castillo Pacheco

SQLModel, ASYNC y Alembic

Temario

- ¿Qué es SQLModel?
- Ejecución en modo asíncrono
- ¿Qué es Alembic?

SQLModel, ASYNC y Alembic

SQLModel:

- **SQLModel** es un potente Object Relational Mapper (ORM) diseñado específicamente para simplificar la interacción con bases de datos relacionales en entornos Python. Desarrollado por el creador de FastAPI, SQLModel ofrece una experiencia de desarrollo intuitiva y eficiente al combinar las fortalezas de SQLAlchemy y Pydantic en un único framework.

SQLModel, ASYnc y Alembic

SQLModel:

- Integración nativa con FastAPI: SQLModel se integra a la perfección con FastAPI, permitiendo crear APIs RESTful de manera rápida y segura, con validación de datos automática y serialización/deserialización de objetos Python.
- Basado en Pydantic: Aprovecha las capacidades de Pydantic para definir modelos de datos con validación de tipos, lo que garantiza la integridad de los datos y reduce la posibilidad de errores.
- Extensión de SQLAlchemy: Se construye sobre SQLAlchemy, proporcionando una capa de abstracción más alta que simplifica la escritura de consultas y la gestión de las relaciones entre objetos.

SQLModel, ASYNC y Alembic

SQLModel – características clave:

- **Modelos declarativos:** Define los modelos de la base de datos de forma concisa y legible, utilizando Python puro.
- **Validación de datos:** Garantiza la integridad de los datos gracias a la validación de tipos y restricciones definidas en los modelos.
- **Consultas eficientes:** Realiza consultas a la base de datos de manera sencilla y eficiente, utilizando la sintaxis de Python.
- **Relaciones:** Maneja relaciones entre objetos de forma natural, como uno a uno, uno a muchos y muchos a muchos.
- **Migraciones:** Gestiona las actualizaciones de la estructura de la base de datos de forma segura y controlada.

SQLModel, ASync y Alembic

SQLModel – beneficios:

- **Aumento de la productividad:** Reduce significativamente la cantidad de código necesario para interactuar con la base de datos.
- **Mejora de la legibilidad:** El código se vuelve más conciso y fácil de entender.
- **Mayor seguridad:** La validación de datos ayuda a prevenir errores comunes y ataques de inyección SQL.

SQLModel, ASync y Alembic

SQLModel – Definición de Modelos Básicos en SQLAlchemy:

En SQLAlchemy, un modelo representa una tabla en una base de datos relacional. Al definir un modelo, estamos describiendo la estructura de esa tabla, incluyendo los campos que contendrá y sus tipos de datos.

```
from sqlalchemy import Field, SQLAlchemy, create_engine

class Usuario(SQLAlchemy, table=True):
    __tablename__ = "usuarios"

    id: int = Field(default=None, primary_key=True)
    nombre: str
    apellido: str
    email: str
```


SQLModel, ASYNC y Alembic

SQLModel – Consultas sencillas usando SQLModel:

Podemos crear un CRUD muy básico para ver las diferentes operaciones que podemos realizar con SQLModel.

Ver ejemplo: `sqlmodel_basico`

SQLModel, ASync y Alembic

SQLModel – Tipos de Relaciones en SQLModel:

SQLModel soporta los tipos de relaciones más comunes

- 1. Uno a Uno:** Una instancia de un modelo se relaciona con una y solo una instancia de otro modelo.
- 2. Uno a Muchos:** Una instancia de un modelo se relaciona con muchas instancias de otro modelo.
- 3. Muchos a Muchos:** Una instancia de un modelo se puede relacionar con muchas instancias de otro modelo, y viceversa.

SQLModel, ASync y Alembic

SQLModel – Tipos de Relaciones en SQLAlchemy:

Ejemplo de **Relación Uno a Uno**: Imaginemos una aplicación de blogs. Un blog tiene un solo autor, y un autor puede tener un solo blog principal.

```
from sqlmodel import Field, SQLModel, create_engine

class Autor(SQLModel, table=True):
    id: int = Field(default=None, primary_key=True)
    nombre: str
    email: str

class Blog(SQLModel, table=True):
    id: int = Field(default=None, primary_key=True)
    titulo: str
    contenido: str
    autor_id: int | None = Field(default=None, foreign_key="autor.id")
```

SQLModel, ASync y Alembic

SQLModel – Tipos de Relaciones en SQLModel:

Ejemplo de Relación Uno a Uno: Aprovechemos el atributo Relationship.

```
from sqlmodel import Field, Relationship, SQLModel, create_engine

class Autor(SQLModel, table=True):
    id: int = Field(default=None, primary_key=True)
    nombre: str
    email: str
    blogs: Blog | None = Relationship(back_populates="autor")

class Blog(SQLModel, table=True):
    id: int = Field(default=None, primary_key=True)
    titulo: str
    contenido: str
    autor_id: int | None = Field(default=None, foreign_key="autor.id")
    autor: Autor | None = Relationship(back_populates="blog")
```

SQLModel, ASYnc y Alembic

SQLModel – Tipos de Relaciones en SQLModel:

¿Qué es atributo Relationship?

Estos atributos no son lo mismo que los campos, ellos no representan una columna directamente en la base de datos y su valor no es un valor singular como un entero. Su valor es el objeto completo al que es relacionado.

En el caso de la instancia de **Blog**, si se llama `blog.autor`, se obtendrá la instancia completa del Autor al que este blog pertenece.

SQLModel, ASync y Alembic

SQLModel – Tipos de Relaciones en SQLModel:

Ejemplo de **Relación Uno a Muchos**: Un autor puede tener muchos artículos, pero un artículo solo puede tener un autor.

```
class Articulo(SQLModel, table=True):  
    __tablename__ = "articulos"  
    id: int = Field(default=None, primary_key=True)  
    titulo: str  
    contenido: str  
    autor_id: int = Field(foreign_key="autor.id")  
    autor: Autor = Relationship(back_populates="articulos")  
  
# En el modelo Autor, agregamos la relación:  
autor.articulos: List[Articulo] = Relationship(back_populates="autor")
```

SQLModel, ASync y Alembic

SQLModel – Tipos de Relaciones en SQLModel:

Ejemplo de **Relación Muchos a Muchos**: Un usuario puede seguir a muchos grupos, y un grupo puede tener muchos seguidores.

```
class Usuario(SQLModel, table=True):
    id: int = Field(default=None, primary_key=True)
    nombre: str
    grupos: List["Grupo"] = Relationship(back_populates="seguidores")

class Grupo(SQLModel, table=True):
    id: int = Field(default=None, primary_key=True)
    nombre: str
    seguidores: List[Usuario] = Relationship(back_populates="grupos")
```

Para manejar relaciones muchos a muchos, SQLModel generalmente crea una tabla de asociación detrás de escena. En este ejemplo, se crearía una tabla usuarios_grupos con dos columnas: usuario_id y grupo_id.

SQLModel, ASync y Alembic

Trabajo Asíncrono:

- Trabajar de manera asincrónica en FastAPI y SQLModel significa que las operaciones (como manejo de peticiones HTTP o interacción con la base de datos) se realizan de manera no bloqueante.
- Esto permite que una aplicación pueda manejar múltiples tareas al mismo tiempo, mejorando la eficiencia y el rendimiento, especialmente en aplicaciones con muchas operaciones de entrada/salida (I/O).

SQLModel, ASync y Alembic

Trabajo Asíncrono:

- Las versiones más recientes de Python tienen soporte para “código asíncrono” usando algo llamado “corutinas” con la sintaxis `async` y `await`.
- Veamos esto en tres partes:
 - Código asíncrono
 - `async` y `await`
 - Coroutines

SQLModel, ASync y Alembic

Trabajo Asíncrono:

- El código asíncrono permite que un programa informe al sistema que en algún punto deberá esperar a que otra operación termine
- Este enfoque es particularmente útil para operaciones I/O (entrada/salida), que suelen ser mucho más lentas comparadas con la velocidad del procesador o la memoria RAM
- La asincronía evita que el programa tenga que esperar secuencialmente a que cada tarea termine, lo que mejora la eficiencia al permitir que otras tareas se ejecuten mientras tanto.

SQLModel, ASYNC y Alembic

Trabajo Asíncrono:

- Las versiones modernas de Python permiten definir código asíncrono de manera intuitiva utilizando **async** y **await**.
- Estas herramientas hacen que el código se parezca al secuencial, pero permiten manejar operaciones que requieren pausas de forma no bloqueante, optimizando la eficiencia del flujo de trabajo.

SQLModel, ASync y Alembic

Trabajo Asíncrono:

- En Python, el uso de **await** está limitado a funciones definidas con **async def**, las cuales también deben ser llamadas utilizando **await**.
- Esto implica que las funciones asincrónicas solo pueden invocarse desde otras funciones también definidas con `async def`, creando una dependencia en la estructura del código que obliga a gestionar las llamadas de manera jerárquica.
- En el caso de FastAPI, no es necesario preocuparse por la invocación de la primera función asincrónica. FastAPI se encarga de gestionar automáticamente la ejecución inicial, ya que las funciones de operación de rutas (endpoints) definidas con **async def** son integradas directamente en el ciclo de eventos asincrónico del servidor.

SQLModel, ASync y Alembic

Trabajo Asíncrono:

- **Corutines:** Una coroutine es el término técnico que se utiliza para describir lo que retorna una función definida con **async def**.
- Python reconoce que esta estructura es similar a una función, con la diferencia de que puede iniciarse, detenerse internamente en los puntos donde haya un `await` y finalizar en algún momento.
- En general, el uso de código asíncrono mediante `async` y `await` suele resumirse como la implementación de "coroutines".

Ver ejemplo de uso

SQLModel, ASYNC y Alembic

Alembic:

- Alembic es una herramienta de migraciones para bases de datos, diseñada para trabajar con SQLAlchemy.
- Permite gestionar cambios en el esquema de una base de datos de manera estructurada y automatizada, asegurando que las modificaciones sean reproducibles y fáciles de mantener en entornos de desarrollo, pruebas y producción.

SQLModel, ASYNC y Alembic

Alembic:

- Alembic es una herramienta de migraciones para bases de datos, diseñada para trabajar con SQLAlchemy.
- Permite gestionar cambios en el esquema de una base de datos de manera estructurada y automatizada, asegurando que las modificaciones sean reproducibles y fáciles de mantener en entornos de desarrollo, pruebas y producción.

SQLModel, ASYnc y Alembic

Alembic:

- Por qué usar Alembic con SQLAlchemy
- **Automatización:**
 - Puede generar scripts de migración automáticamente detectando los cambios realizados en los modelos de SQLAlchemy.
- **Integración fluida:**
 - Alembic está diseñado para integrarse directamente con el motor de SQLAlchemy, lo que facilita el manejo de conexiones y operaciones en la base de datos.

SQLModel, ASYNC y Alembic

Alembic:

- Por qué usar Alembic con SQLAlchemy
- **Escalabilidad y control:**
- Proporciona comandos para gestionar migraciones individualmente, lo que es ideal en proyectos grandes donde se requieren cambios incrementales y controlados en el esquema.

SQLModel, ASync y Alembic

Alembic:

- Por qué usar Alembic con SQLAlchemy
- **Escalabilidad y control:**
- Proporciona comandos para gestionar migraciones individualmente, lo que es ideal en proyectos grandes donde se requieren cambios incrementales y controlados en el esquema.

SQLModel, ASYNC y Alembic

Alembic:

- Para comenzar a trabajar con Alembic, primero se necesita inicializar un entorno en el proyecto. Esto crea una estructura básica de carpetas y archivos necesarios para gestionar migraciones.
- Ejecutar el comando:

```
$ alembic init alembic
```
- Este comando genera una carpeta llamada **alembic** y un archivo de configuración **alembic.ini** en la raíz del proyecto.

SQLModel, ASYNC y Alembic

Alembic:

- Archivo alembic.ini:
 - Archivo de configuración principal de Alembic.
 - Contiene parámetros como:
 - Conexión a la base de datos (sqlalchemy.url).
 - Directorio de migraciones (script_location).
 - Configuraciones de logging.

Ver en el demo

SQLModel, ASYNC y Alembic

Alembic:

- Carpeta alembic:
 - Contiene todos los archivos relacionados con las migraciones. Su estructura inicial incluye:
 - Carpeta alembic/versions:
 - Almacena los archivos de migración generados.
 - Cada archivo de migración tiene un nombre único con un identificador (timestamp o UUID) y una descripción

Ver en el demo

SQLModel, ASYNC y Alembic

Alembic:

- archivo alembic/env.py:
 - Define el entorno de ejecución para las migraciones.
 - Contiene el enlace al motor de SQLAlchemy.
 - Permite personalizar cómo se ejecutan las migraciones, incluyendo el uso de un modelo SQLAlchemy para autogenerar migraciones.

Ver en el demo

SQLModel, ASYnc y Alembic

Alembic:

- archivo alembic/script.py.mako:
 - Plantilla para los archivos de migración.
 - Define el formato inicial de cada migración generada con Alembic.

Ver en el demo

SQLModel, ASync y Alembic

Alembic:

- archivo alembic/README:
 - Información breve sobre el propósito del directorio.

Ver en el demo

SQLModel, Async y Alembic

Alembic:

- Comandos básicos:

Acción	Comando	Descripción
Crear una migración	<code>alembic revision --autogenerate -m "Mensaje"</code>	Genera un archivo de migración basado en los cambios detectados en los modelos.
Aplicar todas las migraciones	<code>alembic upgrade head</code>	Aplica todas las migraciones pendientes hasta la última versión.
Revertir la última migración	<code>alembic downgrade -1</code>	Revierte solo la última migración aplicada.
Revertir a una versión exacta	<code>alembic downgrade <revision_id></code>	Deshace cambios hasta una versión específica del esquema.

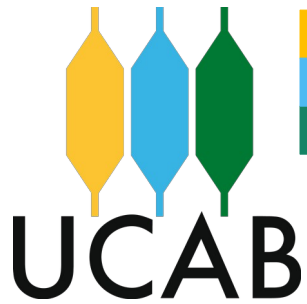
SQLModel, Async y Alembic

Referencias:

- <https://sqlmodel.tiangolo.com/learn/>
- <https://fastapi.tiangolo.com/async/>
- <https://alembic.sqlalchemy.org/en/latest/tutorial.html>
- <https://medium.com/@kasperjuunge/sqlmodel-10-ways-to-query-your-database-60a0722c1fd2>
-

FIN





Programación de APIs con Python



Seguridad con FastAPI

Profesor:
José Gregorio Castillo Pacheco

Security

Temario

- Introducción a la seguridad en APIs
- OAuth2 y JWT
- Seguridad con FastAPI

Security

Introducción

- El objetivo de la seguridad de la API es garantizar la **protección de los datos sensible**, evitar accesos no autorizados y mantener la integridad de las operaciones realizadas a través de la API.
- Impacto de una API no segura:
 - Exposición de datos confidenciales.
 - Riesgo de suplantación de identidad.
 - Vulnerabilidad frente a ataques como el man-in-the-middle o inyección de comandos.

Security

Importancia de la seguridad en APIs

- **Confidencialidad:** Proteger información como datos personales, financieros o de salud.
- **Autenticidad:** Garantizar que los usuarios o sistemas que interactúan con la API son quienes dicen ser.
- **Integridad:** Asegurarse de que los datos enviados y recibidos no sean alterados durante la transmisión.
- **Disponibilidad:** Prevenir ataques que puedan inutilizar los servicios.

Security

Escenarios comunes de autenticación y autorización

- **Autenticación (Authentication):** Es el proceso de verificar la identidad de un usuario o sistema.
- Ejemplos comunes:
 - Usuario y contraseña.
 - Autenticación basada en tokens como JWT.
 - Sistemas de autenticación federada (OAuth2, OpenID Connect).

Security

Escenarios comunes de autenticación y autorización

- **Autorización (Authorization):** Determina qué recursos o acciones están permitidos para un usuario autenticado.
- Ejemplos comunes:
 - Control de acceso basado en roles (RBAC): Un administrador puede realizar más acciones que un usuario estándar.
 - Control de acceso basado en atributos (ABAC): Permisos basados en características del usuario o contexto (e.g., hora del día, ubicación).

Security

¿Qué es OAuth2?

OAuth2 (Open Authorization 2.0) es un marco estándar para delegar acceso seguro a recursos de usuario sin exponer credenciales. Permite que un tercero acceda a recursos protegidos en nombre del usuario.

- **Características principales:**

- Basado en *tokens* de acceso en lugar de compartir contraseñas.
- Soporta diferentes flujos (*flows*) según el caso de uso:
 - *Authorization Code Flow*: Ideal para aplicaciones con backend seguro.
 - *Implicit Flow*: Usado en aplicaciones cliente (deprecated por seguridad).
 - *Password Flow*: Permite autenticación directa (aplicaciones de confianza).
 - *Client Credentials Flow*: Para interacciones entre servidores.

Security

¿Qué es OAuth2?

- **Ventajas:**
 - Flexibilidad en la autenticación y autorización.
 - Mejora la seguridad al no exponer credenciales.

Security

¿Qué es JWT (JSON Web Token)?

JWT es un estándar abierto para transmitir datos entre partes como un objeto JSON, asegurado con una firma digital que garantiza integridad y autenticidad.

- **Estructura de un JWT:**

- *Header* (Encabezado): Especifica el tipo de *token* y el algoritmo de firma (e.g., HMAC SHA256).
- *Payload* (Cuerpo): Contiene los datos (*claims*) a transmitir, como *sub*, *iat*, y *exp*.
- *Signature* (Firma): Asegura la integridad del *token* usando una clave secreta o un certificado.

Security

¿Qué es JWT (JSON Web Token)?

- **Ventajas:**
 - Compacto: Ideal para transmisión en cabeceras HTTP.
 - Verificable: No requiere una consulta al servidor si se usa una firma válida.

Security

Relación entre OAuth2 y JWT

- **Interconexión:**
 - OAuth2 no especifica el formato del token de acceso, pero JWT es una de las implementaciones más comunes por su simplicidad y compatibilidad.
 - JWT se utiliza como token de acceso para representar la identidad del usuario y sus permisos.
- **Ventajas del uso de JWT en OAuth2:**
 - *Portabilidad:* Los tokens JWT pueden ser usados directamente sin necesidad de consultar al servidor.
 - *Descentralización:* Permite delegar la validación a diferentes servicios.
 - *Firma digital:* Garantiza que los tokens no se hayan alterado.

Security

Seguridad en FastAPI

FastAPI incluye herramientas integradas para implementar esquemas de seguridad estándar, como OAuth2, autenticación basada en tokens y validación de credenciales.

1. OAuth2PasswordBearer:

- Permite implementar flujos OAuth2 utilizando tokens bearer.
- Útil para proteger rutas con autenticación basada en un token.
- Ejemplo:

```
from fastapi.security import OAuth2PasswordBearer

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")
```

Security

Seguridad en FastAPI

2. Depends:

- Define dependencias de seguridad para proteger rutas.
- Verifica automáticamente los tokens enviados en las solicitudes.
- Ejemplo:

```
from fastapi import Depends
```

```
async def get_current_user(token: str = Depends(oauth2_scheme)) :  
    # Lógica para validar el token
```


Security

Seguridad en FastAPI

3. Validación de Tokens JWT:

- FastAPI no gestiona directamente los tokens, pero facilita su integración con librerías como python-jose para generación y validación de JWT.
- Ejemplo:

```
from jose import jwt
SECRET_KEY = "your_secret_key"
ALGORITHM = "HS256"
```

```
def create_access_token(data: dict):
    return jwt.encode(data, SECRET_KEY, algorithm=ALGORITHM)
```

Security

Seguridad en FastAPI

4. Protección de rutas:

- Utiliza dependencias para asegurar que solo usuarios autenticados accedan a ciertas rutas.
- Ejemplo:

```
from fastapi import APIRouter, HTTPException
```

```
router = APIRouter()
```

```
@router.get("/users/me")
```

```
async def read_users_me(current_user: dict = Depends(get_current_user)):  
    return current_user
```

Security

Demostración practica:

-

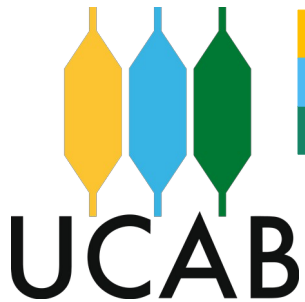
Security

Referencias:

- Hardt, D. (2012). The OAuth 2.0 Authorization Framework. RFC 6749
- <https://oauth.net/2/>
- Jones, M., Bradley, J., & Sakimura, N. (2015). JSON Web Token (JWT). RFC 7519
- <https://jwt.io/>
- <https://fastapi.tiangolo.com/tutorial/security/>

FIN





Programación de APIs con Python



Event Driven Architecture (EDA)

Profesor:
José Gregorio Castillo Pacheco

EDA

Temario

- Comprender los conceptos fundamentales de la Arquitectura Orientada a Eventos (EDA).
- Componentes Clave en EDA
- Conocer los patrones comunes en EDA, con ejemplos
- Integrar notificaciones vía WebSockets.

EDA

Introducción al EDA

- Event Driven Architecture (**EDA**) es un estilo de arquitectura de software donde el flujo de control está determinado por la producción y el consumo de eventos, lo que permite una mayor flexibilidad y escalabilidad en aplicaciones distribuidas y desacopladas.
- En EDA, un **evento** es cualquier cambio de estado que se genera como una respuesta a una acción, como la creación de un nuevo usuario, la actualización de un producto o una solicitud de pago.
- Este enfoque se basa en tres componentes clave: los **productores de eventos**, los **consumidores de eventos** y los **canales de comunicación** que facilitan el intercambio de estos eventos.

EDA

Introducción al EDA

- En una EDA, los eventos son generados por diversas fuentes, publicados en un bus de eventos o en un corredor de mensajes y consumidos por los componentes interesados de forma asíncrona.
- Este enfoque permite que los componentes del sistema no dependan directamente unos de otros, sino que reaccionen a los eventos que se producen de forma autónoma. Esto promueve una arquitectura más flexible, donde los sistemas pueden escalar sin complicaciones y manejar grandes volúmenes de datos de manera eficiente.
- Por ejemplo, en una tienda en línea, al realizar una compra se genera un evento que es procesado por otros servicios, como el de notificaciones o el de inventarios, de forma asíncrona para actualizar el stock o enviar una confirmación.

EDA

Introducción al EDA

- Algunas características de las EDA son
 - Comunicación asíncrona: El emisor y el receptor no tienen que esperarse mutuamente.
 - Enfoque dinámico: Rastrea los datos a medida que atraviesan una arquitectura.
 - Escalabilidad: Se pueden añadir instancias de servicio sin tener en cuenta lo que está ocurriendo aguas abajo.
 - Enrutamiento y filtrado de temas: Los servicios pueden dividirse rápida y fácilmente.

EDA

Introducción al EDA

- **Ventajas**
 - **Desacoplamiento de componentes**
 - Descripción: En una arquitectura orientada a eventos, los componentes del sistema están desacoplados, lo que significa que los productores de eventos no necesitan conocer a los consumidores de esos eventos y viceversa.
 - Beneficios:
 - Facilita el mantenimiento y la evolución del sistema, ya que los cambios en un componente no afectan a otros.
 - Permite la integración de nuevos componentes o la modificación de los existentes sin interrupciones significativas en el sistema.

EDA

Introducción al EDA

- **Ventajas**
 - **Escalabilidad y flexibilidad**
 - Descripción: La EDA permite que los componentes se escalen de forma independiente según la carga de trabajo.
 - Beneficios:
 - Mejora la capacidad de respuesta del sistema bajo carga alta, ya que se pueden añadir más consumidores para procesar los eventos.
 - Permite el uso de diferentes tecnologías y enfoques para distintos componentes, adaptándose a las necesidades específicas de cada uno.

Introducción al EDA

- **Ventajas**
 - **Reacción en tiempo real**
 - Descripción: Los eventos se procesan tan pronto como ocurren, lo que permite que el sistema reaccione en tiempo real.
 - Beneficios:
 - Es ideal para aplicaciones que requieren respuestas inmediatas, como monitoreo de sistemas, aplicaciones financieras, y notificaciones en tiempo real.
 - Mejora la experiencia del usuario final al proporcionar respuestas rápidas y actualizaciones en tiempo real.

EDA

Introducción al EDA

- **Desventajas**
 - **Complejidad adicional**
 - Descripción: La EDA introduce una capa adicional de complejidad en términos de diseño, implementación y mantenimiento del sistema.
 - Desafíos:
 - Requiere una buena comprensión de los patrones de diseño y de las herramientas utilizadas para manejar eventos.
 - Necesita un esfuerzo adicional para asegurar la coherencia y la integridad de los datos, especialmente en sistemas distribuidos.

EDA

Introducción al EDA

- **Desventajas**
 - **Dificultades en la depuración y monitoreo**
 - Descripción: La naturaleza asincrónica y distribuida de la EDA puede hacer que la depuración y el monitoreo del sistema sean más complicados.
 - Desafíos:
 - Identificar la causa de problemas puede ser más difícil, ya que los eventos pueden atravesar varios componentes y sistemas.
 - Requiere herramientas especializadas y prácticas para monitorear y rastrear eventos a lo largo del sistema, como sistemas de logging y tracing distribuidos.

EDA

Introducción al EDA

- **Desventajas**
 - **Latencia debido a la intermediación**
 - Descripción: El uso de un bus de mensajes o intermediarios para transmitir eventos puede introducir latencia en el sistema.
 - Desafíos:
 - La latencia adicional puede afectar el rendimiento del sistema, especialmente en aplicaciones de baja latencia.
 - Es importante optimizar la configuración y el despliegue del bus de mensajes para minimizar este impacto.

EDA

Componentes Clave en EDA

1. Manejo de Eventos
2. Bus de Mensajes
3. Persistencia y Recuperación
4. WebSockets para Notificaciones

Componentes Clave en EDA

Manejo de Eventos

1. Generación de Eventos:

- Los eventos se generan cuando ocurre un cambio de estado importante en el sistema. Estos cambios pueden ser causados por diversas acciones, como la creación de una nueva cuenta de usuario, la actualización del estado de un pedido, o la recepción de un pago.
- Ejemplo: Una transacción financiera exitosa genera un evento de "Transacción Completada".

EDA

Componentes Clave en EDA

Bus de Mensajes

2. Publicación de Eventos:

- Una vez que se genera un evento, debe ser publicado para que otros componentes del sistema puedan reaccionar a él. La publicación de eventos puede realizarse a través de un bus de mensajes, una cola de mensajes, o directamente a través de endpoints HTTP.
- Ejemplo: Un evento de "Transacción Completada" se publica en un bus de mensajes, como Redis o RabbitMQ.

EDA

Componentes Clave en EDA

Manejo de Eventos

3. Suscripción y Procesamiento de Eventos:

- Los componentes del sistema que necesitan reaccionar a ciertos eventos se suscriben a esos eventos. Estos suscriptores pueden ser servicios que realizan acciones específicas cuando reciben un evento.
- Ejemplo: Un servicio de notificaciones se suscribe al evento de "Transacción Completada" y envía un correo electrónico al usuario.

Componentes Clave en EDA

Persistencia y Recuperación de Eventos

4. Persistencia de Eventos:

- Los eventos deben ser almacenados de forma persistente para garantizar que el estado del sistema pueda ser reconstruido en cualquier momento. Esto es especialmente importante para garantizar la consistencia y la recuperación ante fallos.
- Ejemplo: Los eventos se almacenan en una base de datos relacional como PostgreSQL o en un almacén de eventos especializado.

EDA

Componentes Clave en EDA

Persistencia y Recuperación de Eventos

5. Reconstrucción del Estado a partir de Eventos:

- El estado actual del sistema puede ser reconstruido reproduciendo todos los eventos desde un punto inicial. Esto permite que el sistema mantenga un historial completo de todas las transacciones y cambios de estado.
- Ejemplo: Para conocer el estado actual de una cuenta de usuario, se reproducen todos los eventos relacionados con esa cuenta desde su creación.

Componentes Clave en EDA

WebSockets para Notificaciones

6. Distribución en Tiempo Real:

- En sistemas que requieren notificaciones en tiempo real, los eventos pueden ser transmitidos inmediatamente a los clientes a través de WebSockets o tecnologías similares.
- Ejemplo: Un evento de "Nuevo Mensaje" en una aplicación de chat se envía instantáneamente a los clientes conectados mediante WebSockets.

EDA

Patrones de Diseño en EDA

- **Publicación/Suscripción (Pub/Sub):** Un productor publica eventos en un bus, y los consumidores se suscriben para recibirlos de manera asíncrona.
- **Cola de Mensajes:** Los eventos se almacenan en una cola y se procesan en orden por los consumidores.
- **Event Sourcing:** Los eventos se almacenan y reproducen para reconstruir el estado del sistema.
- **CQRS (Command Query Responsibility Segregation):** Se separan los comandos (escrituras) y las consultas (lecturas) para optimizar la gestión de eventos.
- **Actor Model:** Los actores son componentes independientes que procesan mensajes y mantienen su propio estado.
- **State Machine:** El sistema cambia de estado en respuesta a eventos, permitiendo una transición controlada entre estados predefinidos.

EDA

Patrones de Diseño en EDA

- **Publicación/Suscripción (Pub/Sub):** En este patrón, los eventos son publicados por un productor de eventos y luego consumidos por múltiples suscriptores interesados en esos eventos.
- Los productores y consumidores están desacoplados, lo que significa que no necesitan conocerse ni depender directamente unos de otros.
- La comunicación se realiza a través de un bus de eventos o corredor de mensajes, facilitando la entrega de los eventos a los suscriptores.

EDA

Patrones de Diseño en EDA

- **Publicación/Suscripción (Pub/Sub):** Un ejemplo común de este patrón puede encontrarse en una aplicación de **notificaciones**.
- Cuando un usuario realiza una acción, como comprar un producto, se genera un evento. Este evento es publicado a través de un sistema de mensajes, y varios componentes del sistema, como el servicio de notificaciones por correo o el sistema de actualización de inventarios, consumen este evento de manera **asíncrona**.
- De esta forma, los sistemas pueden reaccionar al evento sin necesidad de conocer la implementación de los demás, lo que mejora la escalabilidad y flexibilidad del sistema.
- Ver ejemplo en VSCode

EDA

Patrones de Diseño en EDA

- **Cola de Mensajes(Message Queue):** es un patrón en el que los mensajes generados por los productores son colocados en una cola de mensajes, la cual es consumida por los consumidores en un orden secuencial o basado en prioridades.
- Este patrón permite desacoplar a los productores de los consumidores, lo que mejora la escalabilidad y robustez del sistema.
- La cola garantiza que los mensajes sean procesados en el orden en que fueron recibidos (FIFO - First In, First Out) o según políticas definidas.
- Este enfoque es especialmente útil cuando los consumidores no siempre están disponibles o cuando los productores generan mensajes a un ritmo más rápido que el que los consumidores pueden procesar.

EDA

Patrones de Diseño en EDA

- **Cola de Mensajes(Message Queue):**
- **Ejemplo:** Un sistema de procesamiento de pedidos donde los pedidos son enviados por el sistema de ventas y colocados en una cola de mensajes para ser procesados por un sistema de inventario o un sistema de envío.
- Los sistemas consumidores pueden leer los pedidos de la cola y procesarlos de manera independiente, permitiendo que los productores (ventas) no se vean bloqueados por los consumidores (inventario/envío).
- Ver ejemplo en VSCode

EDA

Patrones de Diseño en EDA

- **Event Sourcing:** El patrón de Event Sourcing se basa en la idea de almacenar todos los eventos que modifican el estado del sistema como una secuencia inmutable de registros de eventos, en lugar de almacenar solo el estado actual.
- En lugar de actualizar directamente una base de datos con el estado del sistema, se persisten los eventos que causan cambios en el estado. Esto permite reconstruir cualquier estado del sistema en cualquier momento simplemente reproduciendo los eventos.
- El principal beneficio de este patrón es que proporciona una historia completa de cambios y permite una auditoría completa del sistema.
- También facilita la recuperación ante fallos al poder reconstruir el estado a partir de los eventos previos. Además, el patrón es útil en sistemas que requieren alta disponibilidad y que no pueden permitir la pérdida de información debido a fallos.

EDA

Patrones de Diseño en EDA

- **Ventajas del Event Sourcing:**
 - **Auditoría Completa:** El patrón permite realizar un seguimiento completo de todos los cambios en el sistema.
 - **Recuperación ante fallos:** Si se pierde la base de datos o el estado actual, se pueden reconstruir los estados a partir de los eventos.
 - **Escalabilidad y rendimiento:** Permite manejar grandes volúmenes de datos, ya que el procesamiento de eventos puede ser distribuido y asíncrono.

EDA

Patrones de Diseño en EDA

- **Event Sourcing:**
- **Ejemplo:** En un sistema de gestión de cuentas bancarias donde cada transacción (depósito, retiro, etc.) se registra como un evento.
- En lugar de actualizar el saldo de la cuenta directamente, cada evento (transacción) se almacena, y el saldo actual se calcula sumando o restando los eventos aplicados a la cuenta.
- Este enfoque permite reconstruir el estado de la cuenta en cualquier punto en el tiempo.
- Ver ejemplo en VSCode

EDA

Patrones de Diseño en EDA

- **CQRS (Command Query Responsibility Segregation):** es un patrón arquitectónico que separa las operaciones de lectura (queries) de las operaciones de escritura (commands) en un sistema. En lugar de tener una única interfaz que maneje tanto las lecturas como las escrituras, CQRS divide el sistema en dos partes:
 - **Comandos (Commands):** Son operaciones que modifican el estado del sistema. Los comandos no devuelven datos, solo cambian el estado.
 - **Consultas (Queries):** Son operaciones que leen datos, sin modificar el estado.
- Este patrón es útil en sistemas donde las operaciones de lectura y escritura tienen diferentes requisitos de rendimiento, escalabilidad o consistencia.
- Al separar las responsabilidades, se pueden optimizar cada una de forma independiente, mejorando la eficiencia general del sistema.

EDA

Patrones de Diseño en EDA

- **Ventajas de CQRS:**
 - **Optimización por Separación:** Al separar las operaciones de lectura y escritura, se puede optimizar cada tipo de operación por separado, por ejemplo, escalando las consultas de manera independiente de las actualizaciones.
 - **Escalabilidad:** Permite escalar de manera más eficiente, especialmente cuando las consultas y los comandos tienen diferentes requisitos de rendimiento.
 - **Mejora en el Mantenimiento:** Facilita el mantenimiento del sistema al tener dos modelos separados para las operaciones de lectura y escritura.

Este patrón es útil en sistemas que necesitan **escalabilidad, alta disponibilidad** y donde las operaciones de lectura y escritura tienen diferentes características o requisitos de rendimiento.

EDA

Patrones de Diseño en EDA

- **CQRS (Command Query Responsibility Segregation):**
- **Ejemplo:** Un sistema de gestión de pedidos. Los usuarios pueden realizar pedidos (comandos), y los administradores pueden consultar el estado de esos pedidos (consultas).
- En lugar de manejar tanto las operaciones de lectura como de escritura con el mismo modelo, se pueden usar dos modelos diferentes: uno para manejar los comandos y otro para las consultas.
 - **Comando:** Crear un nuevo pedido.
 - **Consulta:** Obtener el estado de un pedido específico.
- Ver ejemplo en VSCode

EDA

Patrones de Diseño en EDA

- **Actor Model:** es un patrón de concurrencia basado en actores como unidades fundamentales de ejecución. Cada actor es un objeto que tiene su propio estado privado, que se comunica de manera asíncrona con otros actores mediante el envío de mensajes. Los actores pueden:
 - **Recibir mensajes** de otros actores.
 - **Modificar su propio** estado en respuesta a un mensaje.
 - **Enviar mensajes** a otros actores.
- Este modelo promueve una concurrencia sin bloqueos, ya que los actores no comparten estado directamente, evitando la necesidad de mecanismos tradicionales de sincronización.
- Es particularmente útil para aplicaciones distribuidas y de alta concurrencia, como sistemas de procesamiento de eventos o simulaciones en tiempo real.

EDA

Patrones de Diseño en EDA

- **Ventajas de Actor Model:**

- **Concurrencia sin bloqueos:** Los actores funcionan de manera independiente, lo que permite una alta concurrencia sin necesidad de bloqueos o sincronización.
- **Escalabilidad:** Al estar completamente aislados, los actores pueden distribuirse fácilmente en sistemas multi-core o sistemas distribuidos.
- **Simplicidad en la gestión de estado:** Los actores gestionan su propio estado, lo que elimina la necesidad de mantener un estado global o compartir datos entre ellos, reduciendo la complejidad del sistema.

Este patrón es especialmente útil para aplicaciones altamente concurrentes o distribuidas, donde la comunicación asíncrona y el aislamiento del estado de los actores son cruciales para el rendimiento y la escalabilidad.

EDA

Patrones de Diseño en EDA

- **Actor Model:**
- **Ejemplo:** un sistema de gestión de tickets de soporte técnico. Cada ticket es manejado por un "actor" que recibe mensajes relacionados con ese ticket, como resolver el ticket, asignar un técnico, o cambiar su estado. Cada actor está completamente aislado y maneja sus propios eventos de forma independiente.
 - **Actor:** Un ticket de soporte técnico.
 - **Mensaje:** Cambiar el estado del ticket, asignar un técnico, cerrar el ticket.
- Ver ejemplo en VSCode

EDA

Patrones de Diseño en EDA

- **State Machine (máquina de Estados):** se utiliza para modelar el comportamiento de un sistema que puede estar en uno o más estados a lo largo del tiempo, donde cada estado define cómo el sistema responde a ciertos eventos o entradas.
- En este patrón, las transiciones entre los estados se realizan en función de ciertos eventos que ocurren dentro del sistema. El sistema puede tener reglas específicas para cada estado, y las transiciones entre ellos se llevan a cabo de manera controlada.
- Este patrón es particularmente útil cuando un sistema tiene un comportamiento complejo, donde es necesario controlar y gestionar los diferentes estados y cómo el sistema reacciona ante eventos que pueden cambiar su estado.

EDA

Patrones de Diseño en EDA

- **Ventajas de State Machine:**

- **Claridad y organización:** La lógica de un sistema basado en máquinas de estado es más clara y estructurada, lo que facilita su mantenimiento y entendimiento.
- **Manejo de comportamiento complejo:** Permite manejar sistemas con varios estados posibles, garantizando que solo se realicen transiciones válidas.
- **Flexibilidad:** Facilita la adición o modificación de estados sin afectar el sistema en su totalidad.
- **Mejor trazabilidad:** Las transiciones entre los estados son explícitas, lo que facilita la trazabilidad de las acciones y decisiones tomadas.

EDA

Patrones de Diseño en EDA

- **State Machine:**
- **Ejemplo:** se puede utilizar en una aplicación de procesamiento de pedidos. Un pedido puede pasar por varios estados, como Pendiente, Procesado, Enviado y Entregado.
- El sistema puede manejar diferentes eventos (como la confirmación del pago, la preparación del pedido o el envío) que desencadenan transiciones entre estos estados.
- Cada estado tiene reglas específicas sobre qué eventos pueden suceder a continuación.
- Ver ejemplo en VSCode

EDA

Patrones de Diseño en EDA

WebSockets para Notificaciones

En este escenario, tenemos un sistema de **Pub/Sub** donde un publicador envía un mensaje o evento que será transmitido en tiempo real a los suscriptores mediante WebSockets.

En lugar de almacenar los mensajes en un bus de eventos, se transmiten directamente a los clientes conectados.

Ver ejemplo en VSCode

EDA

Referencias:

- <https://www.freecodecamp.org/news/implement-event-driven-architecture-with-react-and-fastapi/>
- <https://itnext.io/how-to-create-an-event-driven-architecture-eda-in-python-1c47666bc088>
- <https://www.cosmicpython.com/book/preface.html>
- [https://www.ibm.com/topics/redis#:~:text=Redis%20\(REmote%20Dictionary%20Server\)%20is%20an%20open,as%20an%20application%20cache%20or%20quick%2Dresponse%20database](https://www.ibm.com/topics/redis#:~:text=Redis%20(REmote%20Dictionary%20Server)%20is%20an%20open,as%20an%20application%20cache%20or%20quick%2Dresponse%20database)
- <https://velocity.tech/blog/build-an-event-driven-architecture-with-fastapi-and-redis-pub-sub-deploy-it-in-kubernetes>
- <https://www.eventstore.com/cqrs-pattern>
- <https://hackernoon.com/es/observador-vs-pub-sub-patron-50d3b27f838c>
- <https://itnext.io/how-to-tackle-scalability-with-cqrs-using-python-and-fastapi-b44506357c8b>

FIN

