

# Módulo 2. Diseño y desarrollo de software en el uso de API

## Introducción

Cada vez es más evidente que los ingenieros de redes deben aprender a desarrollar *software* para automatizar con las mejores prácticas muchas de las tareas de administración de red.

En este módulo revisaremos la actualidad de la ingeniería de desarrollo de *software* y cuáles son los métodos modernos para su implementación.

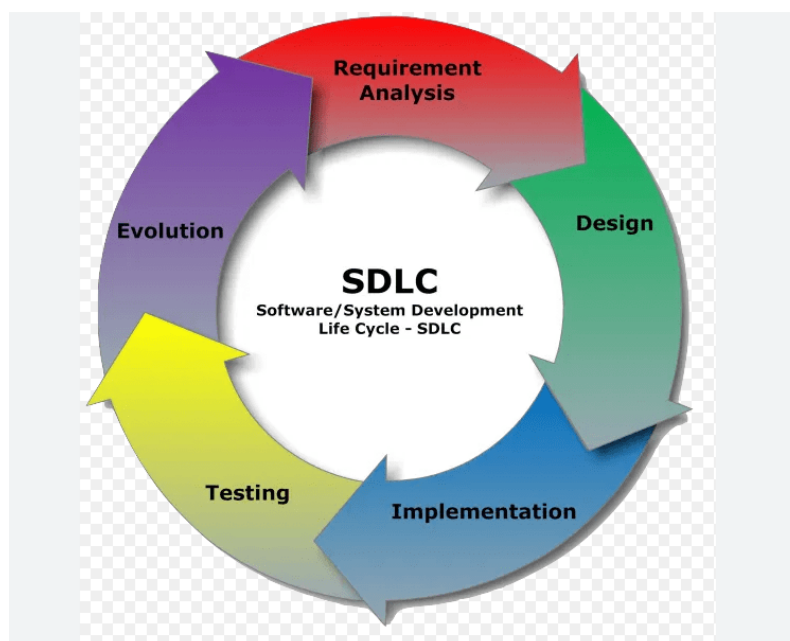
Un componente de *software* para la automatización muy importante son las API, en la actualidad existen muchos tipos de API, por lo tanto, es necesario comprender los fundamentos que las dirigen y cómo utilizarlas para facilitar la automatización.


## Video de inmersión

### Unidad 1. Diseño y desarrollo de *software*


#### Tema 1. Desarrollo y patrones de diseño de *software*

Figura 1: Desarrollo y patrones de diseño de *software*





El ciclo de vida del desarrollo de *software* (SDLC) comienza con un requerimiento y finaliza con la entrega. El proceso consta de cinco fases (algunos autores la dividen en seis fases). Cada fase es alimentada con la anterior. Las más comunes son:

- 
- 1 • Análisis de requerimientos
  - 2 • Diseño
  - 3 • Implementación
  - 4 • Pruebas
  - 5 • Evolución

En el desarrollo tradicional de *software* seguía estas fases una a una, en forma secuencial, es decir, no pasar a la siguiente sin terminar completamente la fase actual. Estos métodos están siendo remplazados, dándole más flexibilidad para permitir que las fases se repitan, se inviertan o desarrollen en paralelo.

### **Análisis de requerimientos**

El objeto de esta fase es responder una serie de preguntas, estado de la situación actual, las necesidades, expectativas, limitaciones, infraestructura actual, etc. Apuntando a qué problema(s) debe resolver el nuevo *software*.

Tengamos en cuenta que las respuestas no son por lo general correctas, esto debido a que el usuario ve el problema desde su perspectiva. Por ejemplo, un vendedor que debe consultar páginas de internet puede ver el problema culpando al Internet que es lento. El mismo problema lo ve el gerente comercial culpando a los vendedores que demoran mucho en realizar una cotización. En resumen, las respuestas son indicios que nos ayudarán a buscar la respuesta correcta.

En esta fase se generan muchas iteraciones con los usuarios para definir todos los requisitos del *front-end* y *back-end* del *software*.

Una vez obtenidos los requisitos se analizan los resultados para determinar:

- Si requisitos posibilitan la implementación del *software*.
- Si requisitos dan los tiempos presupuestados.
- Si es posible hacer pruebas.
- Qué indicador dará cuenta del cierre del proyecto.

Es de buenas prácticas terminar esta fase con un documento de especificación de requerimientos de software aprobado por todas las partes interesadas.

## **Diseño**

La fase de diseño recibe el documento de especificación de requerimientos para que los desarrolladores puedan diseñar el *software*. En esta fase se crean dos documentos:

- Diseño de alto nivel (HLD): describe el diseño general, los componentes y sus interacciones. Es un documento muy general del diseño propuesto.
- Diseño de bajo nivel (LLD): el LLD toma el HLD y detalla con rigurosidad todos los elementos que componen la solución. Protocolos para la comunicación, enumera los objetos y clases requeridas y otros aspectos del diseño.

## **Implementación**

La entrada de la fase de implementación es el HLD y LLD. En esta fase se toma la documentación de diseño y desarrollan el código de acuerdo a la documentación de diseño. Acá se construyen todos los componentes de software de la solución, haciendo que esta fase sea la más extensa del proyecto.

## **Pruebas**

En la fase de pruebas se toma todo el código generado en la implementación y se instala en entornos acotados de pruebas donde es posible recrear todas las situaciones que el *software* por diseño debe realizar. Existen distintos tipos de pruebas: funcionales, de integración, de rendimiento y de seguridad.

Los desarrolladores han aprendido cómo probar de manera más eficiente, cómo incorporar pruebas en flujos de trabajo automatizados y cómo probar *software* en muchos niveles diferentes de detalle y abstracción: desde las definiciones de funciones de bajo nivel más pequeñas hasta agregaciones de componentes a gran escala. También han aprendido que el *software* nunca está libre de errores y, en cambio, debe ser observable, probado en producción y resistente para que pueda permanecer disponible y funcionar adecuadamente, a pesar de los problemas.

## **Evolución**

En esta fase se toma como entrada el código de la fase de pruebas autorizado y se implementa

en el ambiente productivo. Luego de una marcha blanca, y si todo va como es esperado, el *software* está listo para su lanzamiento y entrega a los usuarios finales.

Además, en esta fase se brinda soporte, se corrigen errores encontrados en producción, se investigan mejoras en el *software* y pueden generarse nuevos requerimientos del cliente.

Esta fase termina con la entrega del documento de cierre de proyecto, y se comienza a trabajar en nuevas mejoras del *software* lo que nos lleva de vuelta a la fase de requerimientos y análisis.

## Tema 2. Control de versiones con GIT

Git es una aplicación de código abierto de un sistema de control de versiones distribuido muy popular y es una tendencia en el desarrollo de *software*.

- Fácil de aprender.
- Maneja todo tipo de proyectos.
- Rápido.
- Flexible.
- Reúne todas las características de un sistema de control distribuido.

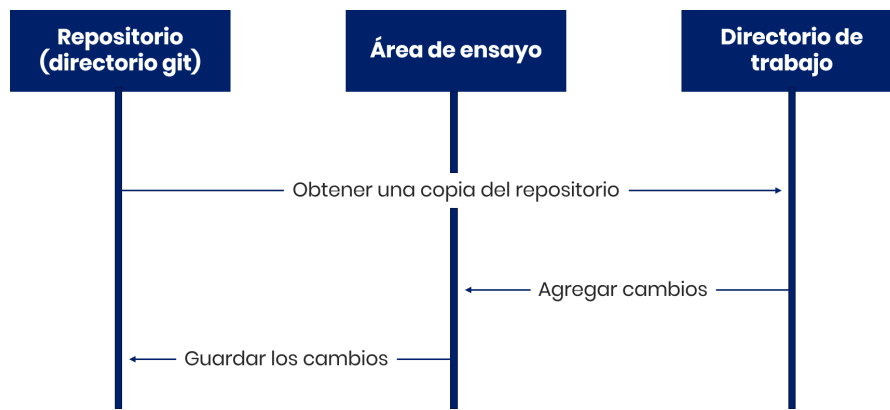
Git está disponible para Windows, Linux y MacOS. Tiene una interface GUI básica, pero todo su poder está en la línea de comandos. Almacena datos como instantáneas en lugar de las diferencias (el delta entre el archivo actual y la versión anterior). Cuando el archivo no cambia, Git hace una referencia a la última instantánea en lugar de crear una nueva.

Git tiene tres estados principales en los que se pueden encontrar tus archivos: confirmado (*committed*), modificado (*modified*), y preparado (*staged*).

- **Confirmado:** significa que los datos están almacenados de manera segura en tu base de datos local.
- **Modificado:** significa que has modificado el archivo, pero todavía no lo has confirmado a tu base de datos.
- **Preparado:** significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.

Esto nos lleva a las tres secciones principales de un proyecto de Git: El directorio de Git (*Git directory*), el directorio de trabajo (*working directory*), y el área de preparación (*staging area*). (Git, s. f., <https://bit.ly/3m3D3zu>)

**Figura 2: Tres secciones principales de un proyecto Git**



Fuente: elaboración propia.

**El directorio de Git es donde se almacenan los metadatos y la base de datos de objetos para tu proyecto. Es la parte más importante de Git, y es lo que se copia cuando clonas un repositorio desde otra computadora.**

El directorio de trabajo es una copia de una versión del proyecto. Estos archivos se sacan de la base de datos comprimida en el directorio de Git, y se colocan en disco para que los puedas usar o modificar.

El área de ensayo, generalmente contenido en tu directorio de Git, que almacena información acerca de lo que va a ir en tu próxima confirmación. A veces se le denomina índice ("index"), pero se está convirtiendo en estándar el referirse a ella como el área de preparación.

El flujo de trabajo básico en Git es algo así:

- Modificas una serie de archivos en tu directorio de trabajo.
- Preparas los archivos, añadiéndolos a tu área de preparación.
- Confirmas los cambios, lo que toma los archivos tal y como están en el área de preparación y almacena esa copia instantánea de manera permanente en tu directorio de Git. (Git, s. f., <https://bit.ly/3m3D3zu>)

## Comandos básicos de Git

Después de instalar Git en la máquina cliente, debemos configurarlo. Git proporciona un comando `git config` para obtener y establecer la configuración global de Git o las opciones de un repositorio.

Para configurar Git, debemos usar la opción `--global` para establecer la configuración global inicial.

Comando: `git config --valor de clave global`

El uso de la opción `--global` escribirá en el archivo global `~/.gitconfig`.

Para que cada usuario sea responsable de sus cambios de código, cada instalación de Git debe establecer el nombre y el correo electrónico del usuario. Para hacerlo, debemos usar los siguientes comandos:

```
$ git config --global user.name "<user's name>"  
$ git config --global user.email "<user's email>"
```

Donde `<user's name>` y `<user's email>` son el nombre y la dirección de correo electrónico del usuario, respectivamente.

## Crear un nuevo repositorio de Git

Cualquier proyecto (carpeta) en el sistema de archivos local de un cliente puede convertirse en un repositorio de Git. Git proporciona un `git init` comando para crear un repositorio de Git vacío o convertir una carpeta existente en un repositorio de Git. Cuando un proyecto nuevo o existente se convierte en un repositorio de Git, se crea un directorio `.git` oculto en esa carpeta de proyecto. Recordemos que el directorio `.git` es el repositorio que contiene los metadatos, como los archivos comprimidos, el historial de confirmaciones y el área de ensayo. Además de crear el directorio `.git`, Git también crea la rama maestra.

Comando: `git init`

Para convertir un proyecto nuevo o existente en un repositorio de Git, debemos usar el siguiente comando:

```
$ git init <project directory>
```

Donde `<project directory>` es la ruta absoluta o relativa al proyecto nuevo o existente. Para un nuevo repositorio de Git, primero se creará el directorio en la ruta proporcionada, seguido de la creación del directorio `.git`.

La creación de un repositorio de Git no realiza un seguimiento automático de los archivos del proyecto. Los archivos deben agregarse explícitamente al repositorio recién creado para poder realizar un seguimiento.

### Algunos comandos Git:

**\$ git init** Creará un nuevo repositorio local GIT. Usando git init [nombre del proyecto].

También podemos crear un repositorio dentro de un directorio especificando el nombre del proyecto.

**\$ git clone** Lo usamos para clonar un repositorio.

**\$ git add** Lo usamos para agregar archivos al área de preparación.

**\$ git commit** Lo usamos para crear un cambio que se guardará en el directorio git.

**\$ git config** Lo usamos para establecer una configuración específica de usuario, podría ser el email, usuario o tipo de formato.

**\$ git status** Lo usamos para que muestre la lista de archivos que cambiamos, junto con archivos que serán preparados y confirmados.

**\$ git push** Lo usamos para enviar confirmaciones a la rama maestra/principal del repositorio remoto.

**\$git remote** Nos permite ver todos los repositorios remotos.

**\$git checkout** Nos permite crear ramas y navegar entre ellas.

**\$git pull** Lo usamos para fusionar todos los cambios que hicimos en el repositorio local con el directorio de trabajo local.

### Tema 3. Conceptos básicos de codificación

Puede parecer fácil introducir código en un archivo y hacer que funcione. Pero, a medida que crece el tamaño y la complejidad del proyecto, y que otros desarrolladores (y partes interesadas) se involucran, se necesitan métodos disciplinados y mejores prácticas para ayudar a los desarrolladores a escribir mejor código y colaborar con él más fácilmente.

Cuando construimos el código tenemos que pensar que lo leerá otro desarrollador, quien debe entender rápidamente lo que tratamos de hacer. Las siguientes preguntas nos ayudarán a desarrollar un código estándar.

¿El formato del código es claro y sigue las prácticas de formato generalmente aceptadas para los lenguajes informáticos involucrados o cumple con los requisitos específicos de los estándares que usa el grupo de trabajo?

¿Se mantiene con **todas** las pestañas o **todos** los espacios?

¿Utiliza la misma cantidad de tabulaciones o espacios por nivel de sangría en todo el archivo?  
Algunos lenguajes, como Python, hacen de esto un requisito.

¿Tiene la muesca en los lugares correctos?

¿Utiliza un formato coherente para la sintaxis, como la ubicación de las llaves ({})?

¿Las variables, los objetos y otros nombres utilizados en el código son intuitivos?

¿El código está organizado de manera que tenga sentido? Por ejemplo, ¿se agrupan las declaraciones, con las funciones en la parte superior, el código principal en la parte inferior o de otra forma, según el idioma y el contexto?

¿El código está documentado internamente con los comentarios apropiados?

¿Cada línea de código tiene un propósito? ¿Eliminamos todo el código no utilizado?

¿Está escrito el código para que el código común pueda reutilizarse y para que todo el código pueda probarse fácilmente?

## **Métodos y funciones**

Los métodos y funciones comparten el mismo concepto, son bloques de código que realizan tareas cuando se ejecutan. Si el método o función no se ejecuta, esas tareas no se realizarán. Aunque no hay reglas absolutas, a continuación veremos algunas mejores prácticas estándar para determinar si una pieza de código debe encapsularse (en un método o función):

El código que realiza una tarea discreta, incluso si solo ocurre una vez, puede ser candidato para la encapsulación. Los ejemplos clásicos incluyen funciones de utilidad que evalúan entradas y devuelven un resultado lógico (por ejemplo, comparan la longitud de dos cadenas), realizan operaciones de E/S (por ejemplo, leen un archivo del disco) o traducen datos a otras formas (por ejemplo, analizar y procesar datos). En este caso, encapsula para mayor claridad y capacidad de prueba, así como para una posible reutilización o extensión en el futuro.

El código de tarea que se usa más de una vez probablemente debería encapsularse. Si copiamos varias líneas de código, probablemente deberíamos hacer de esto un método o una función.



Podemos acomodar ligeras variaciones en el uso utilizando la lógica para evaluar los parámetros pasados (ver más abajo).

Lo más poderoso de los métodos y funciones es que se pueden escribir una vez y ejecutar tantas veces como queramos. Si se usan correctamente, los métodos y funciones simplificarán el código y, por lo tanto, reducirán la posibilidad de errores.

Sintaxis de una función en Python:

```
# Define la función
def functionNombre:
...bloque con el código...

# llamar a la función
functionNombre()
```

## Argumentos y parámetros

Otra característica de los métodos y funciones es la capacidad de ejecutar el código en función de los valores de las variables que se pasan en la ejecución. Estos se llaman argumentos. Para usar argumentos al llamar a un método o función, el método o función debe escribirse para aceptar estas variables como parámetros.

Los parámetros pueden ser de cualquier tipo de datos y cada parámetro en un método o función puede tener un tipo de datos diferente. Los argumentos pasados al método o la función deben coincidir con los tipos de datos esperados por el método o la función.

Dependiendo del lenguaje de codificación, algunos lenguajes requieren que el tipo de datos se defina en el parámetro (los llamados lenguajes 'fuertemente tipados'), mientras que otros lo permiten opcionalmente.

Incluso cuando no se requiere la especificación del tipo de parámetro, suele ser una buena idea. Hace que el código sea más fácil de reutilizar porque puede ver más fácilmente qué tipo de parámetros espera un método o función. También aclara los mensajes de error. Los errores de discrepancia de tipo son fáciles de corregir, mientras que un tipo incorrecto pasado a una función puede causar errores que son difíciles de entender, más profundos en el código.

Los parámetros y argumentos añaden flexibilidad a los métodos y funciones. A veces, el parámetro es solo un indicador booleano que determina si ciertas líneas de código deben

ejecutarse en el método o función. Pensemos en los parámetros como la entrada al método o función.

Sintaxis de una función usando argumentos y parámetros en Python:

```
# Define la función
def functionNombre(parametro1,...,parametroN):
    # Los parámetros se usan igual que la variables locales
    ...bloque de código...
    # Llamar a la función
functionNombre("argumento1", 4, {"argumento3":"3"})
```

Los métodos y funciones realizan tareas y también pueden devolver un valor. En muchos idiomas, el valor devuelto se especifica mediante la palabra clave *return* seguida de una variable o expresión. Esto se llama declaración de devolución. Cuando se ejecuta una declaración de devolución, se devuelve el valor de la declaración de devolución y se omite cualquier código debajo. Es el trabajo de la línea de código que llama al método o función obtener el valor de la devolución, pero no es obligatorio.

Sintaxis de una función con declaración de retorno en Python:

```
# Define the function
def functionNombre(parametro1,...,parametroN):
    Variable = parametro1 * parametro2
    return Variable
    # Llamar a la función
myVariable = functionName("argumento1", 4, {"argumento3":"3"})
```

Ejemplo de métodos con parámetros

```
def circunferencia(radius):
```

```
    pi = 3.14
```

```
    circunferencia = pi * radius * 2
```

```
    return circunferencia
```

```
#
```

```
def printCircunferencia(radius):
```

```
    myCircunferencia = circunferencia(radius)
```

```
    print ("La circunferencia del circulo con un radio de " + str(radius) + " es " +
str(mycircunferencia))
```

```
#
```

```
radius1 = 2
```

```
radius2 = 5
```

```
radius3 = 7
```

```
#
```

```
printCircunferencia (radius1)
```

```
printCircunferencia (radius2)
```

```
printCircunferencia (radius3)
```

Si los métodos y las funciones comparten el mismo concepto, ¿por qué se nombran de manera diferente? La diferencia entre métodos y funciones es que las funciones son bloques de código independientes, mientras que los métodos son bloques de código asociados con un objeto, normalmente para la programación orientada a objetos.

## Módulos

Los módulos son una forma de crear fragmentos de código, independientes y autónomos, que se pueden reutilizar. Los desarrolladores suelen utilizar módulos para dividir un proyecto grande en partes más pequeñas. De esta forma, el código es más fácil de leer y comprender, y cada módulo se puede desarrollar en paralelo sin conflictos. Un módulo se empaqueta como un solo archivo. Además de estar disponible para la integración con otros módulos, debería funcionar de forma independiente.

Un módulo consta de un conjunto de funciones y, por lo general, contiene una interfaz para que se integren otros módulos. Es esencialmente una biblioteca y no se puede crear una instancia.

A continuación, veremos un módulo con un conjunto de funciones guardadas en un script llamado `circleModule.py`.

```
class Círculo:

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

    def circunferencia(self):
        pi = 3.14
        circunferenciaValor = pi * self.radius * 2
        return circunferenciaValor
```

```
def printcircunferencia(self):  
    mycircunferencia = self.circunferencia()  
    print ("circunferencia de un círculo con un radio de " + str(self.radius) + " es " +  
          str(mycircunferencia))
```

Una aplicación que existe en el mismo directorio que circleModule.py podría usar este módulo al importarlo, instanciar la clase y luego usar la notación de puntos para llamar a sus funciones, de la siguiente manera:

```
from CirculoModule import Círculo  
  
# Primero define la instancia de la clase Círculo  
circulo1 = Círculo(2)  
# LLama a la funcion printcircunferencia() para la clase instanciada clase Círculo1.  
Circulo1.printcircunferencia()  
  
# Dos instancias más y llamadas a los métodos de la clase círculo.  
  
círculo2 = Círculo(5)  
circulo2.printcircunferencia()  
  
Círculo3 = Círculo(7)  
Circulo3.printcircunferencia()
```

#### Tema 4. Formato de los datos

Para intercambiar información con servicios y equipos remotos podemos utilizar las API REST. También lo podemos lograr con las interfaces creadas en estas API, como las herramientas de interfaces de línea de comandos y los *kits* de desarrollo de *software* (SDK) utilizados para la integración con lenguajes de programación populares.

Las API permiten recibir y transmitir información en formularios estándares legibles por máquinas y humanos. Esto permite:

- Utilizar componentes de *software* ya creados o herramientas del lenguaje integradas para convertir mensajes en formularios fáciles de manipular y extraer datos, como parte del lenguaje de programación que esté utilizando.
- Escribir mensajes a través de código que las entidades remotas puedan consumir.
- Leer y comprender los mensajes recibidos confirmando que *software* está funcionando correctamente.
- Detectar fácilmente los errores en los mensajes.

Los formatos más utilizados hoy en día para intercambiar información con API remotas son XML, JSON y YAML. YAML es un superconjunto de JSON, por lo que es fácil convertir de un estándar a otro. XML es un estándar antiguo, más complejo de analizar y en algunos casos, difícil de convertir a otros formatos.

El análisis de XML, JSON o YAML es un requisito frecuente para interactuar con las interfaces de programación de aplicaciones (API). Un patrón frecuente en las implementaciones de API REST es el siguiente:

1. La autenticación mediante POSTing con una combinación de usuario/contraseña y recuperando un *token* que expira para su uso en la autenticación de solicitudes posteriores.
2. Ejecutar una solicitud GET a un extremo determinado (autenticando según sea necesario) para recuperar el estado de un recurso, solicitando XML, JSON o YAML como formato de salida.
3. Modificar el mensaje de vuelta porXML, JSON o YAML.
4. Ejecutar un POST (o PUT) en el mismo extremo (de nuevo, autenticando según sea necesario) para cambiar el estado del recurso, solicitando de nuevo XML, JSON o YAML como formato de salida e interpretándolo según sea necesario para determinar si la operación se realizó correctamente.

## XML

El lenguaje de marcado extensible (XML) es un derivado del lenguaje de marcado generalizado estructurado (SGML), y también el padre del lenguaje de marcado de hipertexto (HTML). XML es una metodología genérica para envolver datos textuales en etiquetas simétricas para indicar la semántica. Los nombres de archivo XML normalmente terminan en ".xml".

### Ejemplo de código XML

```
<?xml version="1.0"?>
<!--Listado de libros ☞
    <Catalog>
        <Book id="bk101">
            <Author>Garghentini, Davide</Author>
            <Title>XML Developer's Guide</Title>
```

```
<Genre>Computer</Genre>
<Price>44.95</Price>
<PublishDate>2000-10-01</PublishDate>
<Description>An in-depth look at creating applications
with XML.</Description>
</Book>
<Book id="bk102">
<Author>Garcia, Debra</Author>
<Title>Midnight Rain</Title>
<Genre>Fantasy</Genre>
<Price>5.95</Price>
<PublishDate>2000-12-16</PublishDate>
<Description>A former architect battles corporate zombies,
an evil sorceress, and her own childhood to become queen
of the world.</Description>
</Book>
</Catalog>
```

Este ejemplo simula la información que podría recibir de una API de administración de biblioteca, o podría ser información de servidores o equipos de red, etc.

## Cuerpo del documento XML

La primera línea es una parte especial conocida como prólogo, la segunda línea es un comentario. El resto del documento se denomina cuerpo.

Observemos cómo los elementos de datos individuales dentro del cuerpo (cadenas de caracteres legibles) están rodeados por pares simétricos de etiquetas, la etiqueta de apertura rodeada por símbolos < y >, que contiene un nombre, y la etiqueta de cierre, que es similar, pero con una "/" (barra oblicua) que precede a la de cierre.

La estructura del cuerpo del documento es como un árbol, con ramas que salen de la raíz, que contienen posibles ramas adicionales y, finalmente, nodos de hojas que contienen datos reales. Volviendo hacia arriba en el árbol, cada par de etiquetas en un documento XML tiene un par de etiquetas principal, y así sucesivamente, hasta llegar al par de etiquetas raíz.

## Nombre de etiquetas

Los nombres de etiquetas son definidos por el usuario, al construir una aplicación XML debemos utilizar nombres que le den significado a los elementos de datos, sus relaciones y jerarquía.

## El prólogo XML

El prólogo XML es la primera línea de un archivo XML. Tiene un formato especial, entre paréntesis `<?y ?>`. Contiene el nombre de la etiqueta xml y los atributos que indican la versión y una codificación de caracteres. Normalmente, la versión es "1.0", y la codificación de caracteres es "UTF-8" en la mayoría de los casos; de lo contrario, "UTF-16".

Incluir el prólogo y la codificación puede ser importante para que los analizadores, los editores y otro *software* puedan interpretar de forma fiable los documentos XML.

## Comentario XML

Los archivos XML pueden incluir comentarios, utilizando la misma convención de comentarios utilizada en los documentos HTML. Por ejemplo:

```
<!--Este es un comentario. Puede estar en cualquier parte -->
```

## Atributos XML

XML permite incrustar atributos dentro de las etiquetas para transmitir información adicional. En el siguiente ejemplo, el número de versión XML y la codificación de caracteres están dentro de la etiqueta xml. Sin embargo, los elementos vmid y type también podrían incluirse como atributos en la etiqueta xml:

```
<?xml version="1.0" encoding="UTF-8"?>
  <!-- Instance list -->
  <vms>
    <vm vmid="0101af9811012" type="t1.nano" />
    <vm vmid="0102bg8908023" type="t1.micro"/>
  </vms>
```

## Namespace XML

Un nombre de espacio de nombres es un identificador uniforme de recursos (URI). Normalmente, el URI elegido para el espacio de nombres de un vocabulario XML determinado describe un recurso bajo el control del autor o la organización que define el

vocabulario, como una URL para el servidor web del autor. Sin embargo, la especificación del espacio de nombres no requiere ni sugiere que se use el URI del espacio de nombres para recuperar información; simplemente es tratado por un analizador XML como una cadena.

Por ejemplo, el documento en <http://www.w3.org/1999/xhtml> en sí mismo no contiene ningún código. Simplemente, describe el espacio XHTML de nombres para los lectores humanos. El uso de un URI (como "<http://www.w3.org/1999/xhtml>") para identificar un espacio de nombres, en lugar de una cadena simple (como "xhtml"), reduce la probabilidad de que diferentes espacios de nombres usen identificadores duplicados. (Hmong, s. f., <https://bit.ly/3XUAJs5>)

## JSON

JSON, o Notación de objetos de JavaScript, es un formato de datos derivado de la forma en que se escriben los literales de objetos complejos en JavaScript (que a su vez es similar a cómo se escriben los literales de objetos en Python). Los nombres de archivo JSON normalmente terminan en ".json".

Aquí hay un archivo JSON de muestra que contiene algunos pares clave/valor. Observemos que dos valores son cadenas de texto, uno es un valor booleano y dos son matrices:

```
{
  "edit-config":
  {
    "default-operation": "merge",
    "test-operation": "set",
    "some-integers": [2,3,5,7,9],
    "a-boolean": true,
    "more-numbers": [2.25E+2,-1.0735],
  }
}
```

Cadenas = *default-operation*, *test-operation*.

Matrices = *some-integers*, *more-numbers*.

Booleano = *a-boolean*.

## Tipos de datos en JSON

Los tipos de datos básicos de JSON incluyen números (escritos como enteros positivos y



negativos, como flotantes con decimales o en notación científica), cadenas, booleanos ('verdadero' y 'falso') o nulos (valor dejado en blanco).

## Objetos en JSON

Al igual que en JavaScript, los objetos individuales en JSON comprenden pares clave/valor, que pueden estar rodeados por llaves, individualmente:

```
{"Nombre": "valor"}
```

Este ejemplo representa un objeto con un valor de cadena (por ejemplo, la palabra 'valor'). Un número o valor booleano no va entre comillas.

## Listas y mapas en JSON

Los objetos también pueden contener múltiples pares clave/valor, separados por comas, creando estructuras equivalentes a objetos complejos de JavaScript o diccionarios de Python. En este caso, cada par clave/valor individual no necesita su propio conjunto de corchetes, pero el objeto completo sí. En el ejemplo anterior, la clave "edit-config" identifica un objeto que contiene cinco pares clave/valor.

Los objetos compuestos JSON se pueden anidar profundamente, con una estructura compleja.

JSON también puede expresar matrices ordenadas de JavaScript (o 'listas') de datos u objetos. En el ejemplo anterior, las claves "algunos enteros" y "más números" identifican dichas matrices.

## YAML

YAML, un acrónimo de “YAML Ain't Markup Language”, es un superconjunto de JSON diseñado para una legibilidad humana aún más fácil. Se está volviendo más común como formato para archivos de configuración y, en particular, para escribir plantillas de automatización declarativas para herramientas como Ansible.

Como superconjunto de JSON, los analizadores YAML generalmente pueden analizar documentos JSON (pero no al revés). Debido a esto, YAML es mejor que JSON en algunas tareas, incluida la capacidad de incrustar JSON directamente (incluidas las comillas) en archivos YAML. JSON también se puede incrustar en archivos JSON, pero las comillas se deben escapar con barras invertidas \"o codificarse como entidades de caracteres HTML&quote;

Aquí hay una versión del archivo JSON de la subsección JSON, expresada en YAML. Usaremos esto como un ejemplo para entender cómo funciona YAML:

```
---
edit-config:
  a-boolean: true
  default-operation: merge
  more-numbers:
    - 225.0
    - -1.0735
  some-integers:
    - 2
    - 3
    - 5
    - 7
    - 9
  test-operation: set
...
```

## Estructura de archivos YAML

Como se muestra en el ejemplo, los archivos YAML normalmente se abren con tres guiones (--- solo en una línea) y terminan con tres puntos (... también solo en una línea. YAML también acomoda la noción de múltiples "documentos" dentro de un solo archivo físico, en este caso, separando cada documento con tres guiones en su propia línea.

## Tipos de datos YAML

Los tipos de datos básicos de YAML incluyen números (escritos como enteros positivos y negativos, como flotantes con un decimal o en notación científica), cadenas, booleanos (verdadero y falso) o nulos (valor dejado en blanco).

Los valores de cadena en YAML a menudo se dejan sin comillas. Las comillas solo son necesarias cuando las cadenas contienen caracteres que tienen significado en YAML. Por ejemplo, { ,una llave seguida de un espacio, indica el comienzo de un mapa. También se deben considerar las barras invertidas y otros caracteres especiales o cadenas. Si rodeamos el texto con comillas dobles, podemos escapar de los caracteres especiales en una cadena usando expresiones de barra invertida, como \n para nueva línea.

YAML también ofrece formas convenientes de codificar literales de cadenas de varias líneas.

## Objetos básicos

En YAML, los tipos de datos básicos (y complejos) se equiparan a claves. Las claves normalmente no están entrecomilladas, aunque pueden estar entrecomilladas si contienen dos puntos (:) u otros caracteres especiales. Las claves tampoco necesitan comenzar con una letra, aunque ambas características entran en conflicto con los requisitos de la mayoría de los lenguajes de programación, por lo que es mejor mantenerse alejado de ellas si es posible.

Dos puntos (:) separan la clave y el valor:

```
my_integer: 2
my_float: 2.1
my_exponent: 2e+5
'my_complex:key' : "my quoted string value\n"
0.2 : "can you believe that's a key?"
my_boolean: true
my_null: null # Se puede interpretar como una cadena vacía.
```

## Sangrado YAML y estructura de archivos

YAML no usa paréntesis ni contiene pares de etiquetas, sino que indica su jerarquía mediante sangría. Los elementos sangrados debajo de una etiqueta son “miembros” de ese elemento etiquetado.

La cantidad de sangría depende de nosotros. Se puede usar tan solo un espacio cuando se requiere sangría, aunque una mejor práctica es usar dos espacios por nivel de sangría. Lo importante es ser absolutamente coherentes y utilizar espacios en lugar de tabulaciones.

## Mapas y listas

YAML representa fácilmente tipos de datos más complejos, como mapas que contienen múltiples pares clave/valor (equivalentes a diccionarios en Python) y listas ordenadas.

Los mapas, generalmente, se expresan en varias líneas, comenzando con una clave de etiqueta y dos puntos, seguidos de miembros, con sangría en las líneas siguientes:

```
mymapa:  
  clave1: 5  
  clave2: 100
```

Las listas (matrices) se representan de manera similar, pero con miembros sangrados opcionalmente precedidos por un solo guion y un espacio:

```
mylist:  
  - 1  
  - 2  
  - 3
```

Los mapas y las listas también se pueden representar en la llamada “sintaxis de flujo”, que se parece mucho a JavaScript o Python:

```
mymap: { clave1: 5, clave2: 100}  
mylist: [1, 2, 3]
```

### Más características de YAML

YAML tiene muchas más funciones, que se encuentran con mayor frecuencia cuando se usa en el contexto de lenguajes específicos, como Python, o cuando se convierte a JSON u otros formatos. Por ejemplo, YAML 1.2 admite esquemas y etiquetas, que se pueden usar para desambiguar la interpretación de los valores. Por ejemplo, para forzar que un número se interprete como una cadena, puede usar la cadena `!!str`, que es parte del esquema YAML "Failsafe":

```
mynumericstring: !!str 0.1415
```

## Unidad 2. Uso de API

### Tema 1. Introducción a las API

#### ¿Qué es una API?

Una API permite que una pieza de *software* se comunique con otra. Una API es análoga a una toma de corriente. Sin una toma de corriente, ¿qué tendríamos que hacer para encender una computadora portátil?

- Abrir la pared.
- Desenvainar cables.
- Empalmar los cables juntos.
- Comprender todos los cables en la pared.

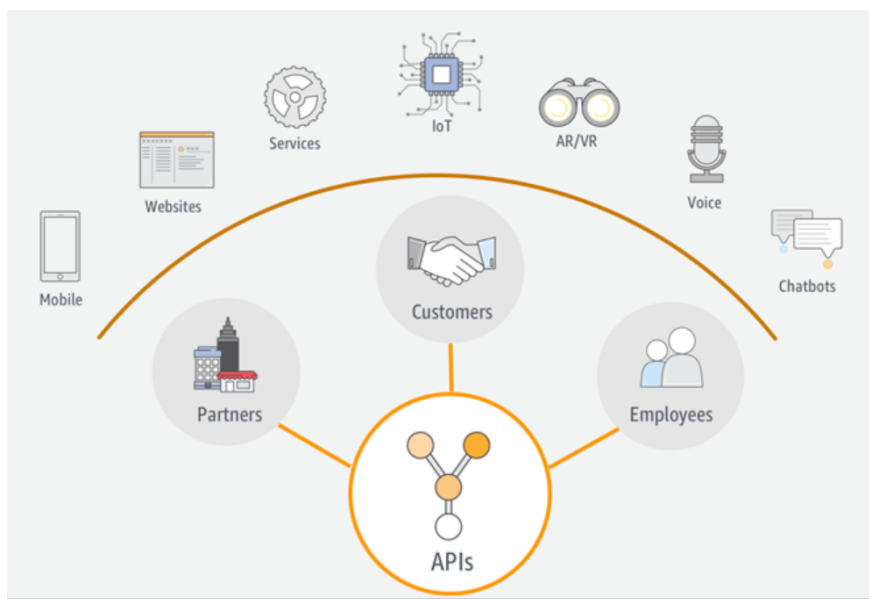
Una API define cómo un programador puede escribir una pieza de *software* para comunicarse con las funciones de una aplicación existente o incluso crear aplicaciones

completamente nuevas.

Una API puede usar interacciones o protocolos de comunicación comunes basados en la web, y también puede usar sus propios estándares patentados. Un buen ejemplo del poder de usar una API es una aplicación de recomendación de restaurantes que devuelve una lista de restaurantes relevantes en el área. En lugar de crear una función de mapeo desde cero, la aplicación integra una API de terceros para proporcionar la funcionalidad del mapa. El creador de la API especifica cómo y en qué circunstancias los programadores pueden acceder a la interfaz.

Como parte de este proceso, la API también determina qué tipo de datos, servicios y funciones expone la aplicación a terceros; si no está expuesto por la API, no está expuesto, punto. Es decir, suponiendo que la seguridad esté configurada correctamente, al proporcionar API, las aplicaciones pueden controlar lo que exponen de manera segura.

**Figura 3: Qué es una API**



Fuente: [imagen sin título sobre qué es una API], (2019). <https://bit.ly/3keTQiv>

Pensemos en esto como los botones en el tablero de un auto. Cuando giramos la llave o presionamos el botón *Engine ON* para arrancar el automóvil, lo que vemos es que el automóvil arranca. No vemos ni nos importa que el motor arranque el sistema de encendido electrónico, lo que hace que el gas ingrese al motor y los pistones comiencen a moverse. Todo lo que sabemos es que si deseamos que el automóvil arranque, tenemos que girar la llave o presionar el botón de encendido del motor. Si presionamos un botón diferente, como el botón de encendido de la radio, el automóvil no arrancará porque esa no era la definición que el automóvil (aplicación) precisó para arrancar el motor. El automóvil (aplicación) en sí puede hacer muchas más cosas además de arrancar el motor, pero esas cosas no están expuestas al conductor (usuario externo).

## ¿Por qué las API son tan populares?

Las API han existido durante décadas, pero la exposición y el consumo de API han crecido exponencialmente en los últimos 10 años más o menos. En el pasado, las aplicaciones estaban bloqueadas y la única integración entre ellas era a través de asociaciones predeterminadas. A medida que creció la industria del *software*, también lo hizo la demanda de integración. Como resultado, más y más aplicaciones comenzaron a exponer partes de sí mismas para que las usen aplicaciones de terceros o personas.

La mayoría de las API modernas están diseñadas en el producto en lugar de ser una ocurrencia tardía. Estas API, generalmente, se prueban exhaustivamente, al igual que cualquier otro componente del producto. Estas API son confiables y, a veces, incluso las utiliza el propio producto. Por ejemplo, a veces la interfaz de usuario de una aplicación se basa en las mismas API que se proporcionan a terceros.

La popularidad de los lenguajes de codificación más sencillos y simplificados, como Python, ha hecho posible que ingenieros que no son de software creen aplicaciones y consuman estas API. Obtienen los resultados que desean sin tener que contratar costosos talentos de desarrollo.

## API síncrona y asíncrona

Las API se pueden entregar de una de dos maneras: de forma síncrona o asíncrona. Debe saber la diferencia entre los dos, porque la aplicación que consume la API administra la respuesta de manera diferente según el diseño de la API. Cada diseño tiene su propio propósito, pero también su propio conjunto de complejidades en el lado del cliente o del servidor. El conjunto de API de un producto puede consistir en diseños tanto síncronos como asíncronos, donde el diseño de cada API es independiente de los demás. Sin embargo, para las mejores prácticas, la lógica detrás del diseño debe ser consistente.

## Figura 4: Venta de boletos



Fuente: [imagen sin título sobre venta de boletos], (s. f.). <https://shutr.bz/3xLRpaA>

Los boletos se venden por orden de llegada, este es un proceso síncrono.

Las API síncronas responden a una solicitud directamente y, por lo general, proporcionan datos (u otra respuesta adecuada) de inmediato.

### ¿Cuándo son síncronas las API?

Las API generalmente están diseñadas para ser síncronas cuando los datos de la solicitud están fácilmente disponibles, como cuando los datos se almacenan en una base de datos o en la memoria interna. El servidor puede obtener instantáneamente estos datos y responder de inmediato.

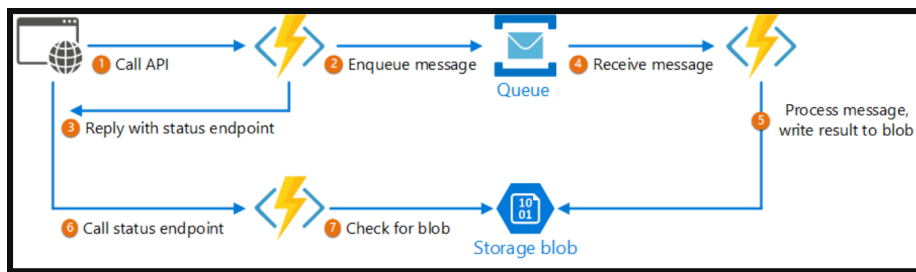
### Beneficios de un diseño API síncrono

Las API síncronas permiten que la aplicación reciba datos de inmediato. Si la API está diseñada correctamente, la aplicación tendrá un mejor rendimiento porque todo sucede rápidamente. Sin embargo, si no se diseña correctamente, la solicitud de API será un cuello de botella porque la aplicación tiene que esperar la respuesta.

### Procesamiento del lado del cliente

La aplicación que realiza la solicitud de API debe esperar la respuesta antes de realizar cualquier tarea de ejecución de código adicional.

### Figura 5: Comunicación síncrona



Fuente: [imagen sin título sobre comunicación síncrona], (s. f.). <https://bit.ly/3YZtPD9>

Las API asíncronas brindan una respuesta para indicar que se recibió la solicitud, pero esa respuesta no tiene ningún dato real. El servidor procesa la solicitud, lo que puede llevar tiempo, y envía una notificación (o activa una devolución de llamada) con los datos una vez que se ha procesado la solicitud. El cliente puede entonces actuar sobre los datos devueltos.

### ¿Cuándo son asíncronas las API?

Las API, generalmente, están diseñadas para ser asíncronas cuando la solicitud es una acción que toma algún tiempo para que el servidor la procese, o si los datos no están disponibles. Por ejemplo, si el servidor tiene que realizar una solicitud a un servicio remoto para obtener los datos, no puede garantizar que recibirá los datos inmediatamente para enviarlos al cliente. El hecho de que una API sea asíncrona no significa necesariamente que el cliente no obtendrá los datos de inmediato. Solo significa que no se garantiza una respuesta inmediata con datos.

### Beneficios de un diseño de API asíncrono

Las API asíncronas permiten que la aplicación continúe ejecutándose sin que se bloquee durante el tiempo que tarda el servidor en procesar la solicitud. Como resultado, la aplicación puede tener un mejor rendimiento porque puede realizar múltiples tareas y realizar otras solicitudes. Sin embargo, el uso innecesario o excesivo de llamadas asíncronas puede tener el efecto contrario en el rendimiento.

### Procesamiento del lado del cliente

Con el procesamiento asincrónico, el diseño de la API en el lado del servidor define lo que desea hacer en el lado del cliente. A veces, el cliente puede establecer un mecanismo de escucha o devolución de llamada para recibir estas notificaciones y procesarlas cuando se reciben. Dependiendo del diseño de la aplicación, el cliente también puede necesitar una cola para



almacenar las solicitudes para mantener el orden de procesamiento. Otros diseños de API necesitan que el cliente tenga un mecanismo de sondeo para conocer el estado y el progreso de una solicitud determinada.

## **Tema 2. Estilos de diseño y arquitectónico de API (RPC, SOAP, REST)**

La aplicación define cómo interactúan terceros con ella, lo que significa que no existe una forma “estándar” de crear una API. Sin embargo, aunque técnicamente una aplicación puede exponer una interfaz desordenada, la mejor práctica es seguir estándares, protocolos y estilos arquitectónicos específicos. Esto hace que sea mucho más fácil para los consumidores de la API aprender y comprenderla, porque los conceptos ya les resultarán familiares.

Los tres tipos más populares de estilos arquitectónicos de API son RPC, SOAP y REST.

### **RPC**

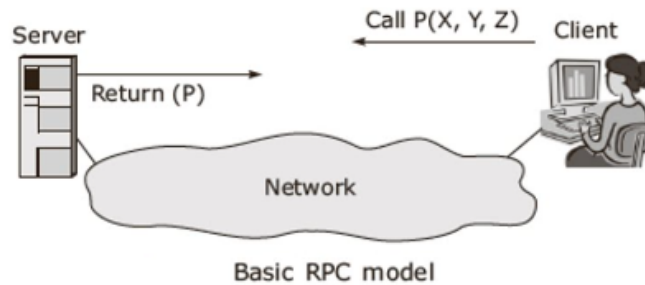
La llamada a procedimiento remoto (RPC) es un modelo de solicitud-respuesta que permite que una aplicación (que actúa como cliente) realice una llamada de procedimiento a otra aplicación (que actúa como servidor). La aplicación “servidor” normalmente se encuentra en otro sistema dentro de la red.

Con RPC, el cliente generalmente no sabe que la solicitud del procedimiento se está ejecutando de forma remota porque la solicitud se realiza en una capa que oculta esos detalles. En lo que respecta al cliente, estas llamadas a procedimientos son simplemente acciones que desea realizar. En otras palabras, para un cliente, una llamada a procedimiento remoto es solo un método con argumentos. Cuando se llama, el método se ejecuta y se devuelven los resultados.

### **Figura 6: Llamada a procedimiento remoto**

# Remote Procedure Call

- Basic RPC operation



Fuente: [imagen sin título sobre llamada de procedimiento remoto], (s. f.). <https://bit.ly/3KrH3Ep>

En el uso más común de RPC, el cliente realiza una solicitud síncrona al servidor y se bloquea mientras el servidor procesa la solicitud. Cuando el servidor termina con la solicitud, envía una respuesta al cliente, lo que desbloquea su proceso (esto no se aplica a las solicitudes asíncronas).

RPC es un estilo de API que se puede aplicar a diferentes protocolos de transporte. Las implementaciones de ejemplo incluyen:

- XML-RPC
- JSON-RPC
- NFS (Sistema de archivos de red)
- Protocolo simple de acceso a objetos (SOAP)

## SOAP

SOAP es un protocolo de mensajería. Se utiliza para la comunicación entre aplicaciones que pueden estar en diferentes plataformas o construidas con diferentes lenguajes de programación. Es un protocolo basado en XML que fue desarrollado por Microsoft. SOAP se usa comúnmente con el transporte del Protocolo de transferencia de hipertexto (HTTP), pero se puede aplicar a otros protocolos. SOAP es independiente, extensible y neutral.

## Independiente

SOAP fue diseñado para que todo tipo de aplicaciones puedan comunicarse entre sí. Las aplicaciones pueden construirse utilizando diferentes lenguajes de programación, pueden

ejecutarse en diferentes sistemas operativos y pueden ser tan diferentes como sea posible.

## Extensible

SOAP en sí mismo se considera una aplicación de XML, por lo que se pueden construir extensiones sobre él. Esta extensibilidad significa que puede agregar funciones como confiabilidad y seguridad.

## Neutral

SOAP se puede utilizar sobre cualquier protocolo, incluidos HTTP, SMTP, TCP, UDP o JMS.

## mensajes SOAP

Un mensaje SOAP es solo un documento XML que puede contener cuatro elementos:

- *Envelope*
- *Header*
- *Body*
- *Fault*

A continuación, un ejemplo de un mensaje SOAP.

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header/>
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Server</faultcode>
      <faultstring>Query request too large.</faultstring>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

### ***Envelope***

*Envelope* debe ser el elemento raíz del documento XML. En el *envelope*, el espacio de nombres proporcionado le indica que el documento XML es un mensaje SOAP.

### ***Header***

El *header* es un elemento opcional, pero si hay un *header*, debe ser el primer hijo del elemento *envelope*. Al igual que la mayoría de los demás *header*, contiene información específica de la

aplicación, como autorización, atributos específicos de SOAP o cualquier atributo definido por la aplicación.

### **Body**

El cuerpo contiene los datos a ser transportados al destinatario. Estos datos deben estar en formato XML y en su propio espacio de nombres.

### **Fault**

*Fault* es un elemento opcional, pero debe ser un elemento secundario del *body*. Solo puede haber un elemento de error en un mensaje SOAP. El *fault* proporciona información de error o estado.

## **REST**

REST es una interfaz para conectar varios sistemas basados en el protocolo HTTP (uno de los protocolos más antiguos) y nos sirve para obtener y generar datos y operaciones, devolviendo esos datos en formatos muy específicos, como XML y JSON.

El formato más usado en la actualidad es el formato JSON, ya que es más ligero y legible en comparación al formato XML. Elegir uno será cuestión de la lógica y necesidades de cada proyecto.

REST se apoya en HTTP, los verbos que utiliza son exactamente los mismos, con ellos se puede hacer *GET*, *POST*, *PUT* y *DELETE*. De aquí surge una alternativa a SOAP.

Cuando hablamos de SOAP hablamos de una arquitectura dividida por niveles que se utilizaba para hacer un servicio, es más complejo de montar como de gestionar y solo trabajaba con XML.

Ahora bien, REST llega a solucionar esa complejidad que añadía SOAP, haciendo mucho más fácil el desarrollo de una API REST, en este caso de un servicio en el cual nosotros vamos a almacenar nuestra lógica de negocio y vamos a servir los datos con una serie de recursos URL y una serie de datos que nosotros los limitaremos, es decir, será nuestro BACKEND nuestra lógica pura de negocios que nosotros vamos a utilizar.

REST no es solo una moda, y es por las siguientes razones que esta interfaz está teniendo tanto protagonismo en los últimos años:

Crea una petición HTTP que contiene toda la información necesaria, es decir, un *REQUEST* a un servidor tiene toda la información necesaria y solo espera una *RESPONSE*, ósea una respuesta en concreto.

Se apoya sobre un protocolo que es el que se utiliza para las páginas web, que es HTTP, es un protocolo que existe hace muchos años y que ya está consolidado, no se tiene que inventar ni realizar cosas nuevas.

Se apoya en los métodos básicos de HTTP, como son:

- *Post*: para crear recursos nuevos.
- *Get*: para obtener un listado o un recurso en concreto.
- *Put*: para modificar.
- *Patch*: para modificar un recurso que no es un recurso de un dato, por ejemplo.
- *Delete*: para borrar un recurso, un dato por ejemplo de nuestra base de datos.

Todos los objetos se manipulan mediante URI, por ejemplo, si tenemos un recurso usuario y queremos acceder a un usuario en concreto nuestra URI sería /user/identificadordelobjeto, con eso ya tendríamos un servicio USER preparado para obtener la información de un usuario, dado un ID.

### Ventajas de REST

Nos permite separar el cliente del servidor. Esto quiere decir que nuestro servidor se puede desarrollar en Node y Express, y nuestra API REST con Vue por ejemplo, no tiene por qué estar todos dentro de un mismo.

- En la actualidad tiene una gran comunidad como proyecto en Git Hub.
- Podemos crear un diseño de un microservicio orientado a un dominio (DDD)
- Es totalmente independiente de la plataforma, así que podemos hacer uso de REST tanto en Windows, Linux, Mac o el sistema operativo que nosotros queramos.
- Podemos hacer nuestra API pública, permitiendo darnos visibilidad si la hacemos pública.
- Nos da escalabilidad, porque tenemos la separación de conceptos de CLIENTE y SERVIDOR, por tanto, podemos dedicarnos exclusivamente a la parte del servidor. (Espanadero, s. f., <https://bit.ly/3IXNWD4>)

**Figura 7: Back-end vs. Front-end**



## Quién usa REST

Muchas empresas como Twitter, Facebook, Google, Netflix, LinkedIn y miles de startups y empresas usan REST. Todas estas empresas y servicios tienen su API REST, por un lado, con su lógica de negocio y, por otro lado, su parte *frontend*, con lo cual nos permite centrarnos también un poco más en lo que es nuestra lógica de negocio haciendo una API REST potente.

Estas API pueden ser públicas y lo pueden consumir otros usuarios, con lo cual tenemos una forma de dar visibilidad a nuestra API y de testearla, no podemos olvidar que la parte de TEST es una de las partes más importantes, pues ¿por qué no nos serviría como TEST que otros usuarios prueben nuestra API REST y nos den *feedback*?, pues aquí lo tenemos solo hay que securizarla y es lo que ocurre con estas empresas. (Espanadero, s. f., <https://bit.ly/3IXNWD4>)

### Tema 3. Introducción a las API REST y uso de *webhook*

Figura 8: Solicitudes GET en web



Fuente: [imagen sin título sobre solicitudes GET en web], (2020). <https://bit.ly/3xLCCwI>

Una API de servicio web REST (API REST) es una interfaz de programación que se comunica a través de HTTP mientras se adhiere a los principios del estilo arquitectónico REST.

Los seis principios del estilo arquitectónico REST son:

- Servidor de cliente
- *Stateless*
- Caché
- Interfaz uniforme
- Sistema en capas
- Código bajo demanda (opcional)

Debido a que las API REST se comunican a través de HTTP, utilizan los mismos conceptos que el protocolo HTTP:

- Solicitudes/respuestas HTTP
- Verbos HTTP
- Códigos de estado HTTP
- Encabezados/cuerpo HTTP

## **Solicitudes de API REST**

Las solicitudes de API REST son esencialmente solicitudes HTTP que siguen los principios REST. Estas solicitudes son una forma en que una aplicación (cliente) solicita al servidor que realice una función. Debido a que es una API, estas funciones están predefinidas por el servidor y deben seguir la especificación proporcionada.

Las solicitudes de API REST se componen de cuatro componentes principales:

- Identificador uniforme de recursos (URI)
- Método HTTP
- Encabezamiento
- Cuerpo

## **Identificador uniforme de recursos (URI)**

El identificador uniforme de recursos (URI) a veces se denomina localizador uniforme de recursos (URL). El URI identifica qué recurso quiere manipular el cliente. Una solicitud REST debe identificar el recurso solicitado; la identificación de recursos para una API REST suele ser parte de la URI.

Un URI es esencialmente el mismo formato que la URL que usa en un navegador para ir a una página web. La sintaxis consta de los siguientes componentes en este orden particular:

- *Scheme* (esquema)
- *Authority* (autoridad)
- *Patch* (ruta)
- *Query* (consulta)

Cuando se reúnen los componentes, un URI se verá así:

scheme:[//authority][path][?query]

**Figura 9: Componentes de un URI**

**http://localhost:8080/v1/libros/?q=DevNet**

**Scheme**                      **Authority**                      **Path**                      **Query**

Fuente: elaboración propia.

### ***Scheme* (esquema)**

El esquema especifica qué protocolo HTTP debe usarse. Para una API REST, las dos opciones son:

- http -- la conexión está abierta
- https -- la conexión es segura

### ***Authority* (autoridad)**

La autoridad, o destino, consta de dos partes que van precedidas de dos barras diagonales ( //):

- *host*,
- *puerto*.

El host es el nombre de host o la dirección IP del servidor que proporciona la API REST (servicio web). El puerto es el punto final de comunicación, o el número de puerto, que está asociado al host. El puerto siempre va precedido de dos puntos ( :). Tenga en cuenta que si el servidor utiliza el puerto predeterminado (80 para HTTP y 443 para HTTPS), el puerto puede omitirse del URI.

### ***Path* (ruta)**

Para una API REST, la ruta generalmente se conoce como la ruta del recurso y representa la ubicación del recurso, los datos o el objeto, que se manipulará en el servidor. La ruta está precedida por una barra inclinada ( /) y puede constar de varios segmentos separados por una barra inclinada ( /).

### ***Query* (Consulta)**

La consulta, que incluye los parámetros de consulta, es opcional. La consulta proporciona detalles adicionales sobre el alcance, el filtrado o la aclaración de una solicitud. Si la consulta está presente, va precedida de un signo de interrogación (?). No existe una sintaxis específica para los parámetros de consulta, pero normalmente se define como un conjunto de pares clave-valor que



están separados por un *ampersand* (&). Por ejemplo: <http://example.com/update/person?id=42&email=person%40example.com>

## Método HTTP

Las API REST usan los métodos HTTP estándar, también conocidos como verbos HTTP, como una forma de decirle al servicio web qué acción se solicita para el recurso dado. No existe un estándar que defina qué método HTTP se asigna a qué acción, pero la asignación sugerida se ve así:

**Tabla 1: Método HTTP**

Método HTTP	Acción	Descripción
POST	Crear	Cree un nuevo objeto
GET	Leer	Recuperar detalles de recursos del sistema
PUT	Actualizar	Reemplazar o actualizar un recurso existente
PATCH	Actualización parcial	Actualizar algunos detalles de un recurso existente
DELETE	Borrar	Eliminar un recurso del sistema

Fuente: elaboración propia.

## **Header (encabezado)**

Las API REST usan el formato de *header* HTTP estándar para comunicar información adicional entre el cliente y el servidor, pero esta información adicional es opcional. Los *header* HTTP tienen el formato de pares de nombre y valor separados por dos puntos (:), [nombre]:[valor]. Se definen algunos *header* HTTP estándar, pero el servicio web que acepta la solicitud de API REST puede definir *header* personalizados para aceptar.

Hay dos tipos de *header*: *header* de solicitud y *header* de entidad.

### **Header (encabezados) de solicitud**

Los encabezados de solicitud incluyen información adicional que no se relaciona con el contenido del mensaje.

Por ejemplo, a continuación vemos un encabezado de solicitud típico que puede encontrar para una solicitud de API REST:

**Tabla 2: Encabezado de solicitud de API REST**

Llave	Valor	Descripción
Autorización	DmFncmFudDp2YWdyYW50 básico	Proporcionar credenciales para autorizar la solicitud

Fuente: elaboración propia.

### **Header (encabezados) de entidad**

Los encabezados de entidad son información adicional que describe el contenido del cuerpo del mensaje.

Aquí hay un encabezado de entidad típico que puede encontrar para una solicitud de API REST:

**Tabla 3: Encabezado de entidad típico de API REST**

Llave	Valor	Descripción
Tipo de contenido	aplicación/json	Especificar el formato de los datos en el cuerpo

Fuente: elaboración propia.

### **Cuerpo**

El cuerpo de la solicitud de la API REST contiene los datos relacionados con el recurso que el cliente desea manipular. Las solicitudes de API REST que utilizan el método HTTP POST, PUT y PATCH normalmente incluyen un cuerpo. Según el método HTTP, el cuerpo es opcional, pero si se proporcionan datos en el cuerpo, el tipo de datos debe especificarse en el encabezado mediante la clave Content-Type. Algunas API están diseñadas para aceptar múltiples tipos de datos en la solicitud.

### **Respuestas de API REST**

Las respuestas de la API REST son esencialmente respuestas HTTP. Estas respuestas comunican los resultados de la solicitud HTTP de un cliente. La respuesta puede contener los datos que se solicitaron, significar que el servidor ha recibido su solicitud o incluso informar al cliente que hubo un problema con su solicitud.

Las respuestas de la API REST son similares a las solicitudes, pero se componen de tres componentes principales:

- Estado HTTP
- *Header* (encabezado)
- Cuerpo

## Estado HTTP

Las API REST usan los códigos de estado HTTP estándar en la respuesta para informar al cliente si la solicitud se realizó correctamente o no. El propio código de estado HTTP puede ayudar al cliente a determinar el motivo del error y, en ocasiones, puede proporcionar sugerencias para solucionar el problema.

Los códigos de estado HTTP son siempre de tres dígitos. El primer dígito es la categoría de la respuesta. Los otros dos dígitos no tienen significado, pero normalmente se asignan en orden numérico. Hay cinco categorías diferentes:

- 1xx - Informativo
- 2xx - Éxito
- 3xx - Redirección
- 4xx - Error del cliente
- 5xx - Error del servidor

### 1xx - informativo

Las respuestas con un código 1xx tienen fines informativos, lo que indica que el servidor recibió la solicitud, pero no terminó de procesarla. El cliente debe esperar una respuesta completa más tarde. Estas respuestas normalmente no contienen un cuerpo.

### 2xx - éxito

Las respuestas con un código 2xx significan que el servidor recibió y aceptó la solicitud. Para las API sincrónicas, estas respuestas contienen los datos solicitados en el cuerpo (si corresponde). Para las API asíncronas, las respuestas normalmente no contienen un cuerpo y el código de estado 2xx es una confirmación de que se recibió la solicitud, pero aún debe cumplirse.

### 3xx - redirección

Las respuestas con un código 3xx significan que el cliente tiene que realizar una acción adicional para que se complete la solicitud. La mayoría de las veces se necesita usar una URL diferente. Dependiendo de cómo se haya invocado la API REST, el usuario podría ser redirigido automáticamente sin ninguna acción manual.

### 4xx - error del cliente

Las respuestas con un código 4xx significan que la solicitud contiene un error, como una mala sintaxis o una entrada no válida, lo que impide que se complete la solicitud. El cliente debe tomar medidas para solucionar estos problemas antes de volver a enviar la solicitud.

### 5xx - error del servidor

Las respuestas con un código 5xx significan que el servidor no puede cumplir con la solicitud a pesar de que la solicitud en sí es válida. Dependiendo de qué código de estado 5xx en particular sea, es posible que el cliente desee volver a intentar la solicitud en otro momento.

**Nota:** podemos obtener detalles sobre cada código de estado HTTP del registro oficial de códigos de estado HTTP, que mantiene la Autoridad de números asignados de Internet (IANA).

### **Header encabezamiento**

Al igual que la solicitud, el *header* de la respuesta también usa el formato de encabezado HTTP estándar y también es opcional. El encabezado de la respuesta es para proporcionar información adicional entre el servidor y el cliente en un formato de par nombre-valor que está separado por dos puntos (:), [nombre]:[valor].

Hay dos tipos de *header*: *header* de respuesta y *header* de entidad.

### **Header (encabezados) de respuesta**

Los encabezados de respuesta contienen información adicional que no se relaciona con el contenido del mensaje.

Algunos encabezados de respuesta típicos que puede encontrar para una solicitud de API REST incluyen:

**Tabla 4: Encabezados de respuesta típicos que puede encontrar para una solicitud de API REST**

Llave	Valor	Descripción
Set-Cookie	JSESSIONID=30A9DN810FQ428P; Path=/ 	Se utiliza para enviar cookies desde el servidor
Cache-Control	Cache-Control: max-age=3600, public	Especificar directivas que <b>deben</b> ser obedecidas por todos los mecanismos de almacenamiento en caché

Fuente: elaboración propia.

## Encabezados de entidad

Los encabezados de entidad son información adicional que describe el contenido del cuerpo del mensaje. Un encabezado de entidad común especifica el tipo de datos que se devuelven.

**Tabla 5: Datos que se devuelven de una entidad común específica**

Llave	Valor	Descripción
Tipo de contenido	aplicación/json	Especificar el formato de los datos en el cuerpo

Fuente: elaboración propia.

## Cuerpo

El cuerpo de la respuesta de la API REST son los datos que el cliente solicitó en la API REST. El cuerpo es opcional, pero si se proporcionan datos en el cuerpo, el tipo de datos se especifica en el encabezado mediante la Content-Type clave. Si la solicitud de la API REST no tuvo éxito, el cuerpo puede proporcionar información adicional sobre el problema o una acción que debe tomar para que la solicitud sea exitosa.

## Paginación de respuesta

Algunas API, como una API de búsqueda, pueden necesitar enviar una gran cantidad de datos en la respuesta. Para reducir el uso de ancho de banda en la red, estas API paginarán los datos de respuesta.

La paginación de respuesta permite que los datos se dividan en fragmentos. La mayoría de las API que implementan la paginación permitirán que el solicitante especifique cuántos elementos desea en la respuesta. Debido a que hay varios fragmentos, la API también debe permitir que el solicitante especifique qué fragmento desea. No existe una forma estándar para que una API

implemente la paginación, pero la mayoría de las implementaciones usan el parámetro de consulta para especificar qué página devolver en la respuesta.

## Datos de respuesta comprimidos

Cuando el servidor necesita enviar grandes cantidades de datos que no se pueden paginar, los datos comprimidos son otra forma de reducir el ancho de banda.

Esta compresión de datos puede ser solicitada por el cliente a través de la propia solicitud de la API. Para solicitar una compresión de datos, la solicitud debe agregar el `Accept-Encoding` campo al encabezado de la solicitud. Los valores aceptados son:

- `gzip`
- `compress`
- `deflate`
- `br`
- `identity`
- `*`

Si el servidor no puede proporcionar ninguno de los tipos de compresión solicitados, enviará una respuesta con un código de estado de 406 -- Not acceptable.

Si el servidor cumple con la compresión, enviará la respuesta con los datos comprimidos y agregará el campo `Content-Encoding` al encabezado de la respuesta. El valor de `Content-Encoding` indica el tipo de compresión que se utilizó, lo que permite al cliente descomprimir los datos adecuadamente.

## ¿Qué es un *webhook*?

Un *webhook* es una devolución de llamada HTTP, o HTTP POST, a una URL específica que notifica a su aplicación cuando se produce una actividad o evento en particular en uno de sus recursos en la plataforma. El concepto es simple. Pensemos en pedirle a alguien “dinos de inmediato si sucede X”. Ese “alguien” es el proveedor del *webhook* y nosotros somos la aplicación.

Los *webhooks* permiten que las aplicaciones obtengan datos en tiempo real, ya que son activados por actividades o eventos particulares. Con los *webhooks*, las aplicaciones son más

eficientes porque ya no necesitan tener un mecanismo de sondeo. Un mecanismo de sondeo es una forma de solicitar información repetidamente del servicio, hasta que se cumpla una condición. Imaginemos tener que preguntarle a alguien, una y otra vez, “¿Ya sucedió X?” Molesto, ¿verdad? Eso es sondeo. El sondeo degrada el rendimiento tanto del cliente como del servidor debido al procesamiento repetido de solicitudes y respuestas. Además, el sondeo no es en tiempo real, porque los sondeos se realizan a intervalos fijos. Si ocurre un evento justo después de la última vez que encuestó, su aplicación no aprenderá sobre el cambio de estado hasta que expire el intervalo de sondeo y ocurra el siguiente sondeo.

Los *webhooks* también se conocen como API inversas, porque las aplicaciones se suscriben a un servidor de *webhook* registrándose con el proveedor de *webhook*. Durante este proceso de registro, la aplicación proporciona un URI para que el servidor lo llame cuando se produzca la actividad o el evento de destino. Este URI suele ser una API en el lado de la aplicación a la que llama el servidor cuando se activa el *webhook*.

Cuando se activa el *webhook*, el servidor envía una notificación al convertirse en la persona que llama y realiza una solicitud al URI proporcionado. Este URI representa la API para la aplicación, y la aplicación se convierte en el receptor de la llamada y consume la solicitud. Como resultado, para los webhooks, los roles se invierten; el servidor se convierte en cliente y el cliente se convierte en servidor. Múltiples aplicaciones pueden suscribirse a un solo servidor *webhook*.

### **Ejemplos:**

- La plataforma Cisco DNA Center proporciona *webhooks* que permiten que las aplicaciones de terceros reciban datos de la red cuando ocurren eventos específicos. Podemos registrar una aplicación con un URI de extremo REST particular que recibe un mensaje de Cisco DNAC cuando ocurre un evento en particular. Por ejemplo, si no se puede acceder a un dispositivo de red, el *webhook* de Cisco DNAC puede enviar un mensaje HTTP POST al URI que su aplicación ha registrado en Cisco DNAC. Luego, su aplicación recibe todos los detalles de la interrupción en un objeto JSON de ese HTTP POST para que pueda tomar las medidas adecuadas. En este caso, Cisco DNAC es el proveedor de *webhook*.
- Puede crear un *webhook* para que Cisco Webex Teams le notifique cada vez que se publiquen nuevos mensajes en una sala en particular. De esta forma, en lugar de que su aplicación realice llamadas repetidas a la API de Teams para determinar si se ha publicado un mensaje nuevo, el webhook le notifica automáticamente cada mensaje. En este caso, Cisco Webex Teams es el proveedor del webhook.

## Tema 4. Integración de Python y API RES

La interfaz del programador de aplicaciones (API) con Python brinda a los programadores de C y C++ acceso al intérprete de Python en una variedad de niveles. La API es igualmente utilizable desde C++, pero por brevedad generalmente se conoce como la API Python/C. Hay dos razones fundamentalmente diferentes para usar la API Python/C. La primera razón es escribir **módulos de extensión** para propósitos específicos; estos son módulos C que extienden el intérprete de Python. Este es probablemente el uso más común. La segunda razón es usar Python como componente en una aplicación más grande; esta técnica se conoce generalmente como integración (*embedding*) Python en una aplicación.

Escribir un módulo de extensión es un proceso relativamente bien entendido, donde un enfoque de «libro de cocina» (*cookbook*) funciona bien. Hay varias herramientas que automatizan el proceso hasta cierto punto. Si bien las personas han integrado Python en otras aplicaciones desde su existencia temprana, el proceso de integrar Python es menos sencillo que escribir una extensión.

Muchas funciones API son útiles independientemente de si está integrando o extendiendo Python; Además, la mayoría de las aplicaciones que integran Python también necesitarán proporcionar una extensión personalizada, por lo que probablemente sea una buena idea familiarizarse con la escritura de una extensión antes de intentar integrar Python en una aplicación real. (Python, s. f., <https://bit.ly/3xLndwf>)

Cuando se trata de desarrollo web en Python, hay dos *frameworks* muy utilizados: Django y Flask.

Django es más antiguo, más maduro y un poco más popular. En GitHub, este framework tiene alrededor de 28k estrellas, 1.5k colaboradores, ~170 releases, y más de 11k forks. En StackOverflow, aproximadamente el 1,2% de las preguntas realizadas en un mes determinado están relacionadas con Django.

Flask, aunque menos popular, no se queda atrás. En GitHub, Flask tiene casi 30.000 estrellas, ~445 colaboradores, ~21 versiones y casi 10k forks. En StackOverflow, hasta el 0,2% de las preguntas realizadas en un mes determinado están relacionadas con Flask.

Aunque Django es más antiguo y tiene una comunidad ligeramente mayor, Flask tiene sus puntos fuertes. Desde el principio, Flask fue construido con escalabilidad y simplicidad en mente. Las aplicaciones de Flask son conocidas por su ligereza, principalmente cuando se comparan con sus homólogas de Django. Los desarrolladores de Flask lo llaman un micro *framework*, donde micro (como se explica aquí) significa que el objetivo es mantener el núcleo simple pero extensible. Flask no tomará muchas decisiones por nosotros, como qué base de datos utilizar o qué motor de plantillas elegir. Por último, Flask también tiene una amplia documentación que aborda todo lo que los desarrolladores necesitan para



empezar.

Al ser ligero, fácil de adoptar, bien documentado y popular, Flask es una muy buena opción para desarrollar API RESTful. (Ibidem Group, s. f., <https://bit.ly/3IIQIPz>)

## Actividades

1. Cual de las siguientes **NO** corresponde a una fase del ciclo de vida del desarrollo de *software* (SDLC).

a) Análisis de requerimientos.

b) Diseño.

c) Implementación.

d) Pruebas.

e) Programación.

Justificación

2. ¿A qué estado de Git pertenece la siguiente afirmación: “significa que los datos están almacenados de manera segura en tu base de datos local”?

a) Confirmado

b) Modificado

c) Preparado

d) Establecido

Justificación

3. De acuerdo con el material, ¿cuáles de los siguientes formatos son los más utilizados para intercambiar información con API?

a) XML, JSON, YAML

b) XML, Python, YAML

c) XML, JSON, C++

d) C++, JSON, YAML

#### Justificación

4. Del siguiente código JSON, indicar qué variable es un valor booleano:

```
{  
  "edit-config":  
  {  
    "default-operation": "merge",  
    "test-operation": "set",  
    "some-integers": [2,3,5,7,9],  
    "a-boolean": true,  
    "more-numbers": [2.25E+2,-1.0735],  
  }  
}
```

a) true

b) a-boolean

c) more-numbers

d) set

#### Justificación

5. ¿A qué corresponde la siguiente definición? “Permite que una pieza de *software* se comuniquen con otra”.

a) Python

b) YAML

c) API

d) REST

#### Justificación

6. ¿A qué diseño arquitectónico de las API corresponde la siguiente definición?: “Es un modelo de solicitud-respuesta que permite que una aplicación (que actúa como cliente) realice una llamada de procedimiento a otra aplicación (que actúa como servidor). La aplicación servidor normalmente se encuentra en otro sistema dentro de la red”.

a) SOAP

b) RPC

c) REST

d) API

#### Justificación

7. ¿En qué protocolo se apoya REST para interconectar diferentes sistemas?

a) RPC

b) API

c) HTTP

d) DHCP

#### Justificación

8. Indicar cuál es el componente Authority de la siguiente URL: "https://intranet.aiep.cl/homepageprivate/notas/ingreso-notas"

a) https://

b) homepageprivate

c) /notas/ingreso-notas

d) //intranet.aiep.cl

#### Justificación

## Video de habilidades

## Glosario

## Referencias

[Imagen sin título sobre Back-end vs. Front-end], (s. f.).  
<https://www.ensalza.com/blog/diccionario/que-es-back-end/>

[Imagen sin título sobre comunicación síncrona], (s. f.). [https://learn.microsoft.com/es-es/azure/architecture/patterns/\\_images/async-request-fn.png](https://learn.microsoft.com/es-es/azure/architecture/patterns/_images/async-request-fn.png)

[Imagen sin título sobre llamada de procedimiento remoto], (s. f.).  
<https://www.youtube.com/watch?v=PtEkcbRO6dk>

**[Imagen sin título sobre qué es una API]**, (2019). <https://www.imagar.com/wp-content/uploads/2019/10/queesunapi.png>

**[Imagen sin título sobre solicitudes GET en web]**, (2020). <https://ccnadesdecero.es/wp-content/uploads/2020/09/Solicitudes-GET-en-Web.png>

**[Imagen sin título sobre venta de boletos]**, (s. f.). <https://www.shutterstock.com/image-vector/cinema-cashbox-sells-tickets-film-600w-1408314380.jpg>

**Espanadero, D.** (s. f.). REST SERVER. <https://github.com/DanielEspanadero/rest-server>

**Git** (s. f.). *1.3 Inicio - Sobre el Control de Versiones - Fundamentos de Git*. [https://git-scm.com/book/es/v2/Inicio---Sobre-el-Control-de-Versiones-Fundamentos-de-Git#:~:text=Esto%20nos%20lleva%20a%20las,de%20preparaci%C3%B3n%20\(staging%20area\)](https://git-scm.com/book/es/v2/Inicio---Sobre-el-Control-de-Versiones-Fundamentos-de-Git#:~:text=Esto%20nos%20lleva%20a%20las,de%20preparaci%C3%B3n%20(staging%20area)).

**Hmong** (s. f.). Espacio de nombres XML. [https://hmong.es/wiki/XML\\_Namespace](https://hmong.es/wiki/XML_Namespace)

**Ibidem Group** (s. f.). *Desarrollo de APIs RESTful con Python y Flask*. <https://www.ibidemgroup.com/edu/traduccion-apis-restful-python-flask/>

**Python** (s. f.). *Manual de referencia de la APU en C de Python*. <https://docs.python.org/es/3.9/c-api/intro.html>

**SSHteam** (s. f.). *SSDLC y desarrollo seguro [imagen]*. <https://sshteam.com/en/ssdlc-y-desarrollo-seguro/>