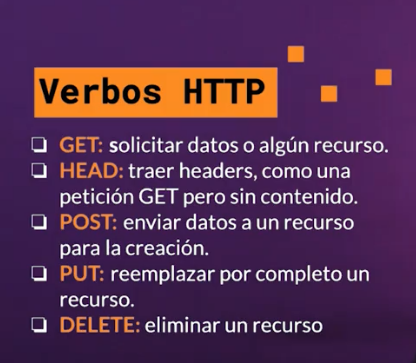
***Notas del Curso.***

**Estudiando el protocolo HTTP, verbos y status.**

Un protocolo especifica reglas en la comunicación entre dos entes, en este caso entre dos computadoras.

HTTP (Hyper Text Transfer Protocol) fue creado específicamente para la web.

Una de las cosas que especifica el protocolo HTTP son los verbos:



Otra de las cosas que especifica el protocolo HTTP son los código de estado (status codes). Sirven para describir el estado de la petición hecha al servidor.

**Códigos de estado HTTP**  
Sirven para describir el estado de la petición.

1xx: Esto indica que la petición fue recibida y está siendo procesada.  
2xx: La solicitud fue completada correctamente.  
3xx: Indica que hay acciones adicionales que se deben de hacer, por lo general se utilizan en las redirecciones.  
4xx: Errores del cliente. Indica que se realiza una solicitud errónea.  
5xx: Errores del servidor. Indica que el servidor presenta un error.

**Errores más comunes**

200: La petición fue correcta.  
201: Es el estado cuando la solicitud Post fue correcta.  
204: La solicitud fue procesada correctamente pero no devuelve información.  
400: Bad request, la solicitud no es correcta.  
401: Authorized, indica que se debe de realizar una autenticación antes de realizar una petición.  
403: Forbidden: No se tiene acceso al recurso aunque se esté autenticado.  
404: Not Found, el recurso no fue encontrado.  
500: Internal Server Error, el servidor indica que la solicitud no pudo ser procesada.

# Estructuras de las URLs

En un API es importante tener bien estructuradas las rutas por las cuales se usarán cada uno de los endpoints que contiene. Antes de entrar de lleno a explicar el API con el que trabajaremos en este curso veamos unos conceptos muy importantes a la hora de trabajar con APIs.

## **Recurso.**

Es la instancia o la representación de un objeto o la representación de algo, tiene datos y operaciones asociadas a él. Por ejemplo: course, material y video son recursos que tenemos disponibles en el API con la que trabajaremos y podemos realizar operaciones sobre ellos: crear, actualizar y eliminar.

**Colecciones.**

Es un conjunto de recursos, por ejemplo: courses es una colección de course.

**URL.**

(Uniform Resource Locator) es la ruta en la cual puede ser ubicado un recurso y ejecutar las operaciones sobre él.

**CRUD.**

Siglas que hacen referencia a las operaciones básicas que se pueden ejecutar sobre un recurso:

* C: Create (crear)
* R: Read (leer)
* U: Update (actualizar)
* D: Delete (eliminar)

# Endpoints

Es el punto final de la comunicación con un ente, en este caso, un endpoint está asociado a una URL y a las operaciones que podemos ejecutar. Este término es muy utilizado en las APIs.

Los endpoint definen operaciones que se quieren ejecutar sobre un recurso. Por ejemplo: si queremos un conjunto de operaciones CRUD sobre Cursos podríamos crear los siguientes endpoints:

* /get-all-courses : para obtener una colección de Cursos.
* /add-new-course: para crear un nuevo recurso de Cursos.
* /delete-all-courses: para eliminar todos los Cursos.

Y así sucesivamente. Pero, esto implicaría un problema. El API puede llenarse de endpoints que ejecutan una sola operación, esto no es escalable, significa que si por ejemplo el recurso Cursos pasa a llamarse Clases habría que actualizar absolutamente todas las URLs y asegurarse de ello puede convertirse en un dolor de cabeza.

Entonces, ¿cuál es la buena práctica en este caso?

Como lo vimos en la clase pasada, el protocolo HTTP especifica una serie de verbos que indican acciones. Lo ideal es que la operación que se ejecute sobre un recurso se obtenga a través del verbo HTTP de la petición y no que esté definido en el endpoint. Por ejemplo:

* /courses: Dependiendo del verbo HTTP se ejecutará una operación en particular. Quedaría así:
  + GET /courses: trae la colección de Cursos.
  + POST /courses: crea un nuevo recurso de Cursos.
  + DELETE /courses: elimina todos los cursos.

De esta manera queda mucho más organizado un API. Pero, qué pasa si queremos traer no una colección de cursos como en los casos anteriores, sino un recurso en específico.

Por lo general cada recurso tiene un identificador único, un ID, es con esto que llamaremos a un endpoint para que nos retorne la información del recurso. Por ejemplo:

* GET /courses/2/: vemos que acá se le está pasando algo que en los endpoints anteriores no veíamos, es el número 2, ese es el identificador único, de esta manera el API sabe que tiene que buscar el curso con ID 2 y retornarlo.

Así sucesivamente con cada uno de los verbos, por ejemplo: casi nunca se hace una eliminación en masa en un API, como el ejemplo que tenemos más arriba donde se eliminan todos los cursos. Lo ideal es que se elimine un recurso individualmente y no una colección, igualmente pasa con la actualización.

**Recursos Anidados.**

Hay veces en las que un recurso depende de otro recurso, por ejemplo, comentarios depende de materiales; usualmente en los APIs las anidaciones se hacen hasta dos niveles, es decir:

* materials/1/comments: estos son dos niveles
* materials/1/comments/2/answers/: son tres niveles, ahí lo ideal sería entonces separar el endpoint de comentarios y ejecutar comments/2/answers/

# Nuestro API

Ya he dado algunos spoilers sobre lo que nuestro API hace, es un clon de lo que Platzi es, una plataforma es donde tenemos Cursos, Materiales, Videos y Comentarios. El API es sencillo y es una prueba que utilizamos en este curso para explorar toda las capacidades que nos ofrece Postman para trabajar con ellos.

Una convención que se usa a la hora de documentar APIs es abstraer el endpoint de la URL del sitio al cual vamos a hacer la petición, puesto que esto al final es redundante de escribir, es decir, usualmente escribimos únicamente /api-token-auth/ en vez de [http://mistioweb.com/api-token-auth/](<http://mistioweb.com/api-token-auth/>).

**Los endpoint que tenemos:**

* /api-token-auth/
* /courses
* /courses/:id/
* /courses/:id/upload\_badge/
* /materials/
* /materials/:id/
* /materials/:id/comments/
* /comments/
* /comments/:id/
* /comments/:id/like/
* /comments/:id/dislike/

# Instalación de Postman.

Postman es multiplataforma, lo que implica que no importa el sistema operativo que uses Postman funcionará.

Es esta dirección <https://www.getpostman.com/downloads/> puedes encontrar la versión de Postman más reciente y descargarla, por defecto, el sitio sabe desde qué sistema operativo estás accediendo y te muestra como primera opción, si no, más abajo aparecen los demás sistemas operativos que soporta Postman.

# 

# En Windows

La instalación de Postman tiende a ser mucho más sencilla que la de una aplicación tradicional; no hay una serie de configuraciones que se deben aceptar o personalizar. Postman se instala en el sistema listo para utilizarse.

* Descargar Postman
* Abrir el archivo que se descargó, una vez abrá aparecerá una ventana así:



# Una vez instalado te pedirá que inicies sesión o si es el caso que crees una nueva cuenta.

# 

# Y ya está postman listo para usarse.

# 

# Llamados a un API con GET: parámetros en la URL

# 

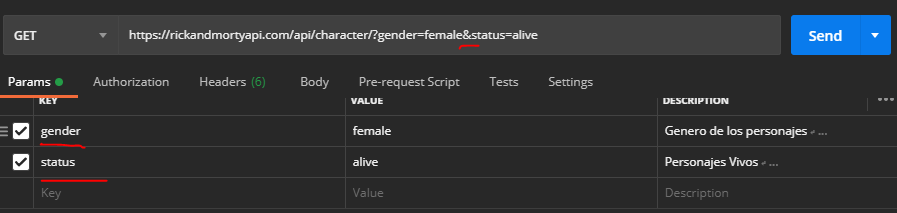
# Para enviar un parámetro por una URL colocamos un signo de pregunta mas el nombre del parámetro, y asignamos con un signo igual el valor del parámetro.

***?nombre\_parametro=valor***

En Postman podremos ir escribiendo los parámetros y añadirles descripciones.

También podremos marcarlos o desmarcarlos dependiendo del uso.

*Para separar entre parámetro y parámetro usamos* ***&***



# Llamados a un API con el método post utilizando JSON.

# Para enviar información por Postman por el método POST, debemos ubicarnos en la pestaña de Body en donde escribiremos nuestra información que va ser enviada en formato JSON. Esto también debemos configurarlo en el desplegable de los tipos de formatos.

# 

# Llamados a un API con el método PUT / PATCH.

# Las diferencias entre PUT y PATCH es que PUT *actualiza* por completo un recurso, y PATCH sólo de una parte.

# ¿A través de los métodos PUT y PATCH se puede enviar información con form-data? Respuesta: **Si**.

# Optimización de environment de postman y configuración de la colección.

# Podremos organizar un entorno en Postman, lo que nos permitirá organizar en un *Collection* y organizar por carpetas nuestro Endpoints (URL Request).

# También podremos almacenar *Variables* en nuestro entorno de desarrollo.

# 

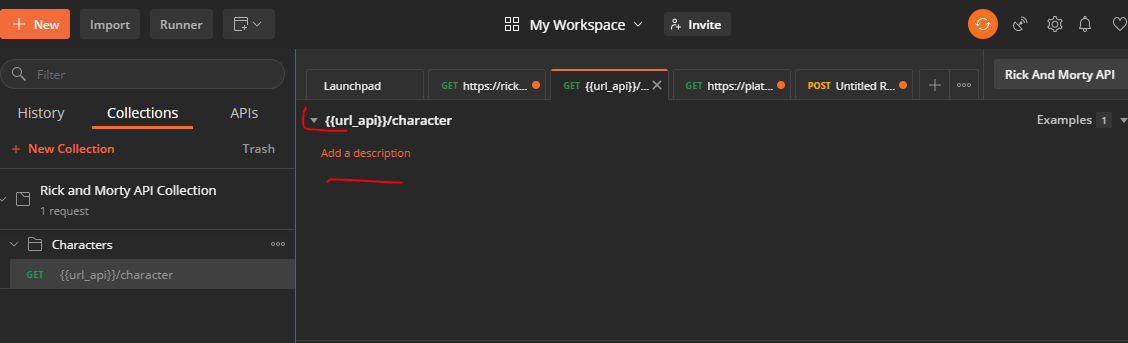
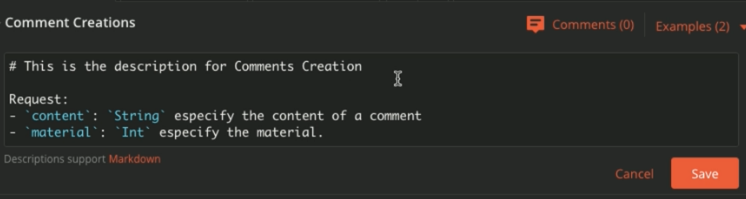
# Para llamar la variable almacenada en nuestro entorno solo usaremos{}

# 

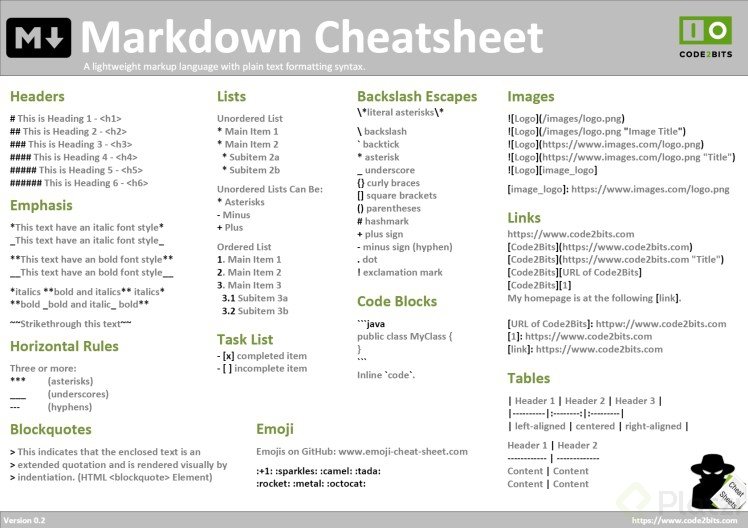
# Agregar ejemplos de responses.

# Podremos crear y guardar las respuestas que hemos solicitado.

**Descripción de endpoints.**

Para poder añadir descripciones como buena practica de ***documentar*** nuestros endpoints, haremos lo siguiente:

Para agregar la descripción podremos escribirlo en MarkDown



# Guardar información con una prueba automática en JS.

# Postman me permite establecer variables antes y después de un request, automatizando procesos. Para configurar un script antes del request debo escribirlo en la pestaña de Pre-request-Script:

# Para configurar un script después del request debo escribirlo en la pestaña de Tests.

# Las variables van a quedar almacenadas en el entorno:

# Creación de Pruebas para endpoints.

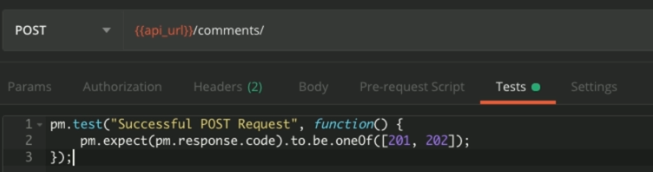
# Para crear validaciones de TEST (después del request) como por ejemplo si el request es un código de éxito 200 o si la respuesta es un JSON debemos agregar lo siguiente en la pestaña de Tests.

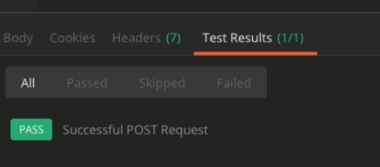
# 

# También podremos validar si dentro de nuestro response hay la información que queremos, en este caso queremos validar si contiene token. Como vimos anteriormente podremos crear variables, para limpiarlas usaremos el método unset()

# 

Para agregar un test de validación que me informe si el response contiene algún código HTTP (en este caso el código **201** o **202**), debo ingresar el siguiente script:





Importants Questions and Answers.

