Creado por:

Isabel Maniega

Trabajar con API RESTful

Algunas palabras sobre REST

REST no es en realidad una palabra, es un acrónimo. Proviene de tres palabras de igual importancia:

- **RE**presentational
- **S**tate
- **T**ransfer

Veámoslas por separado.

Representational

RE significa Representational. Significa que nuestra maquinaria almacena, transmite y recibe representaciones, mientras que el término representación refleja la forma en que los datos o estados se retienen dentro del sistema y se presentan a los usuarios (humanos o computadoras).

REST utiliza una forma muy curiosa de representar sus datos: siempre es texto. Texto puro y simple.

"Debe ser una broma", puede pensar ahora. "¿Cómo es posible enviar y recibir todo tipo de datos utilizando texto simple?"

Es una muy buena pregunta. Probablemente la mejor pregunta que se puede hacer ahora. REST se centra en un tipo de datos muy específico: los datos que reflejan estados.

En breve le contaremos más sobre esto. Ahora pasemos a la siguiente parte del acrónimo.

State

S significa State (estado). La palabra state (estado) es clave para entender qué es REST y para qué se puede utilizar.

Creemos que su conocimiento de clases y objetos puede ser muy útil en este caso. Queremos que lo utilice. Imagine un objeto cualquiera. El objeto contiene un conjunto (el conjunto más preferible es uno que no esté vacío) de propiedades. Podemos decir que los valores de todas las propiedades del objeto constituyen su estado. Si alguna de las propiedades cambia su valor, esto implica inevitablemente el efecto de cambiar el estado de todo el objeto. Este cambio se suele llamar transición.

Ahora imagine que el objeto está almacenado en otro lugar, no en su computadora, sino en un servidor ubicado al otro lado de la colina y muy lejos. Por supuesto, puede acceder a los recursos del servidor mediante la red, pero no puede simplemente obtener el objeto y transferirlo a su computadora. ¿Por qué no? Porque tiene que ser accesible para muchos (quizás unos pocos, quizás un millón) usuarios. Debe permanecer en el servidor.

Imagina que quieres (o debes) afectar el estado del objeto a través de la red. No, no puedes invocar ninguno de sus métodos. Lo siento, eso es imposible. No puedes hacerlo directamente. Pero puedes hacerlo usando REST. ¿Cómo? Te lo mostraremos muy pronto. Veamos ahora la última parte del acrónimo.

Transferencia

T significa Transferencia. La red (no solo Internet) puede actuar como un portador que le permite transmitir representaciones de estados hacia y desde el servidor.

Nota: no es el objeto, sino sus estados, o acciones capaces de cambiar los estados, los que están sujetos a la transferencia. Podemos decir (es una analogía muy pobre, pero funcionará aquí) que transferir los estados le permite lograr resultados similares a los causados por las invocaciones de métodos.

Sockets BSD

Los sockets de los que queremos hablarte no tienen nada que ver con la electricidad: no vamos a conectar nada a ellos ni vamos a extraer energía de ellos.

Un socket (en el sentido que nos interesa ahora) es una especie de punto final. Un punto final es un punto en el que se pueden obtener los datos y a donde se pueden enviar. Tu programa Python puede conectarse al punto final y usarlo para intercambiar mensajes entre él mismo y otro programa que trabaja en algún lugar lejano de Internet.

La historia de los sockets comenzó en 1983 en la Universidad de California en Berkeley, donde se formuló el concepto y se llevó a cabo la primera implementación exitosa.

La solución resultante fue un conjunto universal de funciones aptas para su implementación en casi todos los sistemas operativos y disponibles en todos los lenguajes de programación modernos. Se denominó sockets BSD, un nombre tomado de Berkeley Software Distribution, el nombre de un sistema operativo de clase Unix, donde se implementaron los sockets por primera vez.

Después de algunas modificaciones, el estándar fue adoptado por POSIX (un estándar de los sistemas operativos contemporáneos de tipo Unix) como sockets POSIX.

Podemos decir que todos los sistemas operativos modernos implementan sockets BSD de una manera más o menos precisa. A pesar de sus diferencias, la idea general sigue siendo la misma y esto es lo que vamos a contarte.

No queremos que nuestro curso sea una escuela de programación de redes, así que ten en cuenta que te presentaremos solo la información absolutamente esencial sobre cómo

se gestiona el tráfico de red. Nos centraremos, como siempre, en la programación en Python. Por cierto: los sockets BSD se implementaron originalmente en el lenguaje de programación "C", lo que es una buena razón para comenzar nuestro curso "C".

La idea principal detrás de los sockets BSD está estrechamente relacionada con la filosofía Unix contenida en las palabras "todo es un archivo". Un socket a menudo puede tratarse como un tipo muy específico de archivo. Escribir en un socket da como resultado el envío de datos a través de una red. Leer desde un socket te permite recibir los datos que vienen de la red.

Por cierto, MS Windows reimplementa los sockets BSD en forma de WinSock. Afortunadamente, no se nota la diferencia al programar en Python. Python los oculta por completo. Nos gusta Python por eso (y no solo por eso).

Prepárate para asimilar muchos términos y nociones nuevos. ¿Estás listo?

Dominios de sockets

Inicialmente, los sockets BSD se diseñaron para organizar la comunicación en dos dominios diferentes. Los dos dominios eran:

- **Unix domain** (Unix para abreviar): una parte de los sockets BSD que se utiliza para comunicar programas que funcionan dentro de un sistema operativo (es decir, presentes simultáneamente en el mismo sistema informático)
- **Internet domain** (INET para abreviar): una parte de la API de sockets BSD que se utiliza para comunicar programas que funcionan dentro de diferentes sistemas informáticos, conectados entre sí mediante una red TCP/IP (nota: esto no excluye el uso de sockets INET para comunicar procesos que funcionan en el mismo sistema)

En la siguiente parte, trataremos los sockets que funcionan en el dominio INET.

Dirección de socket

Los dos programas que desean intercambiar sus datos deben poder identificarse entre sí; para ser precisos, deben tener la capacidad de indicar claramente el socket a través del cual desean conectarse.

Los sockets de dominio INET se identifican (direccionan) mediante pares de valores:

- la dirección IP del sistema informático en el que se encuentra el socket;
- el número de puerto (más a menudo denominado número de servicio)

Dirección IP

Una dirección IP (más precisamente: dirección IP4) es un valor de 32 bits que se utiliza para identificar ordenadores conectados a cualquier red TCP/IP. El valor suele presentarse como cuatro números del rango 0..255 (es decir, ocho bits de longitud) unidos por puntos (p. ej., 87.98.239.87).

También existe un estándar IP más nuevo, llamado IP6, que utiliza 128 bits para el mismo propósito. Debido a su escasa prevalencia (según los datos publicados en

agosto de 2016, menos del 20 % de los ordenadores del mundo son accesibles mediante direcciones IP6), limitaremos nuestras consideraciones a IP4.

Número de socket/servicio

El número de socket/servicio es un número entero de 16 bits que identifica un socket dentro de un sistema en particular. Como ya habrás adivinado, hay 65.536 (2 ** 16) números de socket/servicio posibles.

El término número de servicio proviene del hecho de que muchos servicios de red estándar suelen utilizar los mismos números de socket constantes, por ejemplo, el protocolo HTTP, un portador de datos utilizado por REST, suele utilizar el puerto 80.

Protocolo

Un protocolo es un conjunto estandarizado de reglas que permiten que los procesos se comuniquen entre sí. Podemos decir que un protocolo es una especie de saber vivir de la red que especifica las reglas de comportamiento para todos los participantes.

Pila de protocolos (Protocol stack)

Una pila de protocolos es un conjunto multicapa (de ahí el nombre) de protocolos cooperativos que proporcionan un repertorio unificado de servicios. La pila de protocolos TCP/IP está diseñada para cooperar con redes basadas en el protocolo IP (las redes IP).

El modelo conceptual de los servicios de red describe la pila de protocolos de forma que los servicios más básicos y elementales se ubican en la parte inferior de la pila, mientras que los más avanzados y abstractos se ubican en la parte superior.

Se supone que cualquier capa superior implementa sus funcionalidades utilizando servicios proporcionados por la capa inferior adyacente (nota: es lo mismo que en las otras partes del sistema operativo, por ejemplo, su programa implementa su funcionalidad utilizando servicios del SO y los servicios del SO implementan sus funcionalidades utilizando instalaciones de hardware).

IP

El IP (Internetwork Protocol) es una de las partes más bajas de la pila de protocolos TCP/IP. Su funcionalidad es muy simple: puede enviar un paquete de datos (un datagrama) entre dos nodos de red.

IP es un protocolo muy poco confiable. No garantiza que:

- alguno de los datagramas enviados llegue al destino (además, si alguno de los datagramas se pierde, puede permanecer sin detectar)
- el datagrama llegará al destino intacto;
- un par de datagramas enviados llegarán al destino en el mismo orden en que fueron enviados.

Las capas superiores son capaces de compensar todas estas deficiencias del protocolo IP.

TCP

El TCP (Protocolo de Control de Transmisión) es la parte más alta de la pila de protocolos TCP/IP. Utiliza datagramas (proporcionados por las capas inferiores) y handshakes (un proceso automatizado de sincronización del flujo de datos) para construir un canal de comunicación fiable capaz de transmitir y recibir caracteres individuales.

Su funcionalidad es muy compleja, ya que garantiza que:

- un flujo de datos llegue al destino, o se informe al remitente de que la comunicación ha fallado;
- los datos lleguen al destino intactos.

UDP

El UDP (Protocolo de Datagramas de Usuario) se encuentra en la parte más alta de la pila de protocolos TCP/IP, pero por debajo del TCP. No utiliza protocolos de enlace, lo que tiene dos consecuencias graves:

- es más rápido que TCP (debido a que tiene menos gastos generales)
- es menos confiable que TCP.

Esto significa que:

- TCP es un protocolo de primera elección para aplicaciones donde la seguridad de los datos es más importante que la eficiencia (por ejemplo, WWW, REST, transferencia de correo, etc.)
- UDP es más adecuado para aplicaciones donde el tiempo de respuesta es crucial (DNS, DHCP, etc.)

Connection-oriented vs. connectionless communication

Una forma de comunicación que exige algunos pasos preliminares para establecer la conexión y otros pasos para finalizarla es la comunicación orientada a la conexión.

Por lo general, ambas partes involucradas en el proceso no son simétricas, es decir, sus roles y rutinas son diferentes. Ambos lados de la comunicación saben que la otra parte está conectada.

Una llamada telefónica es un ejemplo perfecto de comunicación orientada a la conexión.

Mire:

- los roles están estrictamente definidos: hay un llamante y hay un destinatario;
- el llamante debe marcar el número del destinatario y esperar hasta que la red enrute la conexión;

- el llamante debe esperar a que el destinatario responda la llamada (el destinatario puede rechazar la conexión o simplemente no responder la llamada)
- la comunicación real no comenzará hasta que se completen con éxito todos los pasos anteriores;
- la comunicación termina cuando cualquiera de las partes cuelga.

Las redes TCP/IP usan los siguientes nombres para ambos lados de la comunicación:

- el lado que inicia la conexión (llamante) se llama cliente;
- el lado que responde al cliente (llamado) se llama servidor.

Las comunicaciones orientadas a conexión se construyen generalmente sobre TCP.

Una comunicación que se puede establecer ad-hoc (snap - así de simple) es comunicación sin conexión. Ambas partes suelen tener los mismos derechos, pero ninguna de las partes es consciente del estado de la otra parte.

El uso de walkie-talkies es una muy buena analogía para la comunicación sin conexión, porque:

- cualquiera de las partes de la comunicación puede iniciar la comunicación en cualquier momento; solo es necesario pulsar el botón de hablar;
- hablar al micrófono no garantiza que alguien lo escuche (es necesario esperar a que llegue una respuesta para estar seguro)

Las comunicaciones sin conexión se construyen generalmente sobre UDP.

Bien. Tomar una dosis tan grande de teoría requiere algo de práctica lo antes posible. Hagámoslo.

Cómo obtener un documento de un servidor usando Python

Vamos a escribir nuestro primer programa haciendo uso de sockets de red. Por supuesto, utilizaremos Python para este propósito.

Estos son nuestros objetivos:

- queremos escribir un programa que lea la dirección de un sitio WWW (por ejemplo, pythoninstitute.org) utilizando la función estándar `input()` y obtenga el documento raíz (el documento HTML principal del sitio WWW) del sitio especificado;
- el programa muestra el documento en la pantalla;
- el programa utiliza TCP para conectarse al servidor HTTP.

Nuestro programa tiene que realizar los siguientes pasos:

- crear un nuevo socket capaz de manejar transmisiones orientadas a conexión basadas en TCP;
- conectar el socket al servidor HTTP de una dirección dada;
- enviar una solicitud al servidor (el servidor quiere saber qué queremos de él)
- recibir la respuesta del servidor (contendrá el documento raíz solicitado del sitio)

- cerrar el socket (finalizar la conexión)

Importar un socket

Necesitamos un socket. ¿Cómo obtenemos un socket? ¿Podemos pedirlo en una tienda de Internet? ¿Es gratis?

Sí, es gratis. Como probablemente sospechas, necesitamos un módulo especializado. Python ofrece precisamente ese módulo. No te sorprenderá si te decimos que el módulo se llama socket, ¿verdad?

Esto es lo que pondremos en la parte superior de nuestro código:

```
import socket
```

Obtención de la entrada del usuario

También necesitamos el nombre del servidor HTTP al que nos vamos a conectar. De hecho, no es nuestro problema. El usuario lo sabe mejor. Preguntémosle:

```
import socket

server_addr = input("What server do you want to connect
to? ")
```

La entrada del usuario puede tomar dos formas diferentes:

- puede ser el nombre de dominio del servidor (como www.pythoninstitute.org, pero sin el http:// inicial)
- puede ser la dirección IP del servidor (como 87.98.235.184), pero hay que decir firmemente que esta variante es potencialmente ambigua. ¿Por qué? Porque puede haber más de un servidor HTTP ubicado en la misma dirección IP - el servidor al que llegará puede no ser el servidor al que deseaba conectarse.

Puede sonar cínico - no es nuestro problema cuál de estas dos formas eligen nuestros usuarios. Ellos saben más. El cliente siempre tiene razón.

El módulo socket: creación de un socket

El módulo `socket` contiene todas las herramientas que necesitamos para trabajar con sockets. No vamos a presentar todas sus capacidades - como mencionamos antes, no nos estamos centrando ni nos centraremos en la programación de redes. Queremos mostrarte cómo funciona TCP/IP y cómo es capaz de actuar como un portador para REST.

Podemos decir que TCP/IP es interesante para nosotros sólo en la medida en que es capaz de transportar tráfico HTTP, y HTTP es interesante para nosotros sólo en la medida en que es capaz de actuar como un relé para REST.

El módulo socket proporciona una clase llamada `socket` (¡qué coincidencia!) que encapsula un conjunto de propiedades y actividades relacionadas con el

comportamiento real de los sockets. Esto significa que el primer paso es crear un objeto de la clase - así es como llevamos a cabo la creación:

```
import socket

server_addr = input("What server do you want to connect
to? ")
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Como puede ver, el constructor toma dos argumentos, ambos declarados dentro del módulo. Permítanos contarle sobre ellos:

- el primer argumento es un código de dominio (podemos usar el símbolo `AF_INET` aquí para especificar el dominio del socket de Internet, ¿lo recuerda?

Le dijimos sobre los dominios Unix e INET en la sección anterior); como los diferentes dominios requieren un socket completamente diferente, el dominio de destino debe conocerse en el momento;

- el último argumento es un código de tipo de socket (podemos usar el símbolo `SOCK_STREAM` aquí para especificar un socket de alto nivel capaz de actuar como un dispositivo de caracteres - un dispositivo que puede manejar caracteres individuales, ya que estamos interesados en transferir datos byte a byte, no como bloques de tamaño fijo (por ejemplo, una terminal es un dispositivo de caracteres, mientras que un disco no lo es)

Un socket de este tipo está preparado para funcionar sobre el protocolo TCP - es la configuración de socket predeterminada.

Si desea crear un socket para cooperar con otro protocolo, como UDP, deberá usar una sintaxis de constructor diferente.

Como puede ver, el objeto de socket recién creado será referenciado por una variable llamada `sock`. No, no se trata de la ropa. En serio.

Conexión a un servidor

Si utilizamos un socket en el lado del cliente, estamos listos para usarlo. Sin embargo, el servidor tiene que dar algunos pasos más. En general, los servidores suelen ser más complejos que los clientes (ya que un servidor atiende a muchos clientes simultáneamente); este es el momento en el que nuestras analogías telefónicas dejan de funcionar.

El socket configurado (al igual que el nuestro) puede conectarse a su contraparte en el lado del servidor. Observa el código en el editor: así es como realizamos la conexión.

El método `connect()` hace lo que promete: intenta conectar tu socket al servicio de la dirección y el número de puerto (servicio) especificados.

Nota: utilizamos la variante en la que los dos valores se pasan al método como elementos de una tupla. Es por eso que ves dos pares de paréntesis allí. Omitir uno de

ellos obviamente provocará un error.

Nota: la forma de la dirección del servicio de destino (un par que consiste en la dirección real y el número de puerto) es específica para el dominio INET. No esperes que se vea igual en otros dominios.

Puedes preguntar: ¿por qué 80? ¿Puedo poner algo más en lugar de esto? No, no puedes. 80 es un número de servicio conocido para HTTP. Cualquier navegador de Internet intentará conectarse al puerto número 80 de forma predeterminada, así que lo hacemos también.

¿Es posible que el intento de conexión falle? Por supuesto que sí. Hay muchas razones posibles: una dirección de servicio mal formada, un servidor inexistente, un error de conexión y más. ¿Cómo podemos descubrir eventos tan desagradables?

Si algo sale mal, el método `connect()` (y cualquier otro método cuyos resultados puedan ser infructuosos) lanza una excepción. Dejemos el problema a un lado por un momento. Por el momento podemos suponer que todo va bien.

La conexión está lista. El servidor ha aceptado nuestra conexión y tiene mucha curiosidad por saber qué escuchará de nosotros. No lo dejes esperar demasiado.

Pero... ¿qué es lo que realmente queremos decirle al servidor? ¿Cómo le hablamos al servidor HTTP para asegurarnos de que nos entiende? Tenemos que hablar en HTTP, por supuesto.

In [1]:
```
import socket

server_addr = input("What server do you want to connect to? ") # localhos
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((server_addr, 8050))
```

El método GET

El protocolo HTTP es uno de los protocolos de Internet más simples, pero aún es demasiado complejo para analizarlo en profundidad aquí. Por ahora, le diremos cómo obtener un documento raíz de un sitio WWW. Por supuesto, le contaremos más sobre esto más adelante.

Una conversación con el servidor HTTP consta de solicitudes (enviadas por el cliente) y respuestas (enviadas por el servidor).

HTTP define un conjunto de solicitudes aceptables: estos son los métodos de solicitud o palabras HTTP. El método que solicita al servidor que envíe un documento en particular con un nombre determinado se llama `GET` (se explica por sí solo, ¿no?).

Para obtener un documento raíz de un sitio llamado www.site.com, el cliente debe enviar la solicitud que contenga una descripción del método `GET` correctamente formada:

```
GET / HTTP/1.1\r\n
Host: www.site.com\r\n
```

```
Connection: close\r\n
\r\n
```

El método `GET` requiere:

- una línea que contenga el nombre del método (es decir, `GET`) seguido del nombre del recurso que el cliente desea recibir; el documento raíz se especifica como una sola barra (es decir, `/`); la línea también debe incluir la versión del protocolo HTTP (es decir, `HTTP/1.1`) y debe terminar con los caracteres `\r\n`; nota: todas las líneas deben terminar de la misma manera;
- una línea que contenga el nombre del sitio (por ejemplo, *www.site.com*) precedido por el nombre del parámetro (es decir, `Host:`)
- una línea que contenga un parámetro llamado `Connection:` junto con su valor `close`, que obliga al servidor a cerrar la conexión - después de que se atienda la primera solicitud; simplificará el código de nuestro cliente;
- una línea vacía es un finalizador de solicitud.

No parece muy claro, pero no excede nuestras capacidades, ¿verdad?

Vale, ya sabemos que HTTP no será nuestro lenguaje favorito, pero ¿cómo podemos enviar una petición de este tipo al servidor? Es sencillo. Tenemos que invocar un método desde dentro del objeto socket.

Su nombre es... ¿lo adivinas?

Solicitar un documento a un servidor

Sí, es `send`. Observa cómo lo combinamos con nuestro código de la lección anterior:

```
sock.send(b"GET / HTTP/1.1\r\nHost: " +
          bytes(server_addr, "utf8") +
          b"\r\nConnection: close\r\n\r\n")
```

El método `send()` no acepta cadenas de forma nativa; por eso tenemos que usar el prefijo `b` antes de las partes literales de la cadena de solicitud (traduce silenciosamente la cadena en bytes, un vector inmutable que consiste en valores del rango 0..255, que es el que más le gusta a `send()`) y también por eso deberíamos invocar `bytes()` para traducir la variable de cadena de la misma manera.

Nota: el segundo argumento de bytes especifica la codificación utilizada para almacenar el nombre del servidor. UTF8 parece ser la mejor opción para la mayoría de los sistemas operativos modernos.

La acción realizada por el método `send()` es extremadamente complicada: involucra no solo muchas capas del sistema operativo, sino también muchos equipos de red implementados en la ruta entre el cliente y el servidor, y obviamente el servidor mismo.

Afortunadamente, no tenemos que preocuparnos por eso.

Por supuesto, si algo dentro de este complejo mecanismo falla, `send` también fallará. Como es de esperar, se lanza una excepción.

De todos modos, la suerte está echada. La solicitud se ha enviado. ¿Qué podemos esperar del servidor?

Si el servidor está funcionando y hay un documento raíz listo para enviárnoslo, podemos recibirlo. Lo haremos ahora sin dudarlo.

Mire la versión final de nuestro código. La hemos proporcionado en el editor.

In [2]:
```
import socket

server_addr = input("What server do you want to connect to? ") # 127.0.0.
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((server_addr, 8050))
sock.send(b"GET / HTTP/1.1\r\nHost: " +
          bytes(server_addr, "utf8") +
          b"\r\nConnection: close\r\n\r\n")
```

Out[2]: 54

Solicitar un documento a un servidor: continuación

El método `recv()` (en nuestra humilde opinión, no es una abreviatura muy afortunada de recibir) espera la respuesta del servidor, la obtiene y la coloca dentro de un objeto recién creado de tipo bytes. Observa el código que hemos proporcionado en el editor.

El argumento especifica la longitud máxima aceptable de los datos que se recibirán. Si la respuesta del servidor es más larga que este límite, permanecerá sin recibir.

Deberás invocar `recv()` nuevamente (quizás más de una vez) para obtener la parte restante de los datos. Es una práctica general invocar `recv()` siempre que devuelva algunos datos.

Hay muchas cosas malas que pueden arruinar nuestro juego. Por ejemplo, es posible que el servidor no quiera hablar con nosotros.

La transmisión también puede causar algunos errores. Todas estas fatalidades generarán excepciones.

¿Qué sigue?

In [3]:
```
import socket

server_addr = input("What server do you want to connect to? ") # 127.0.0.
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((server_addr, 8050))
sock.send(b"GET / HTTP/1.1\r\nHost: " +
          bytes(server_addr, "utf8") +
          b"\r\nConnection: close\r\n\r\n")
reply = sock.recv(10000)
```

```
---------------------------------------------------------------------------
ConnectionRefusedError                    Traceback (most recent call last)
Cell In[3], line 5
      3 server_addr = input("What server do you want to connect to? ") # 127.0.0.1
      4 sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
----> 5 sock.connect((server_addr, 8050))
      6 sock.send(b"GET / HTTP/1.1\r\nHost: " +
      7           bytes(server_addr, "utf8") +
      8           b"\r\nConnection: close\r\n\r\n")
      9 reply = sock.recv(10000)

ConnectionRefusedError: [Errno 111] Connection refused
```

Cerrando la conexión

Como no queremos enviar ni recibir nada más, debemos comunicárselo al servidor. Lo haremos de una forma muy sencilla, como la siguiente:

```
import socket

server_addr = input("What server do you want to connect
to? ")
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((server_addr, 80))
sock.send(b"GET / HTTP/1.1\r\nHost: " +
          bytes(server_addr, "utf8") +
          b"\r\nConnection: close\r\n\r\n")
reply = sock.recv(10000)
sock.shutdown(socket.SHUT_RDWR)
```

Invocar `shutdown()` es como susurrarle un mensaje directamente al oído al servidor: "No tenemos nada más que decirte. Tampoco queremos saber nada de ti. El resto es silencio".

Gracias a eso, el servidor es consciente de nuestras intenciones.

Los siguientes argumentos de la función dicen más sobre nuestras visiones para el futuro:

- `socket.SHUT_RD` - no vamos a leer más los mensajes del servidor (nos declaramos sordos)
- `socket.SHUT_WR` - no diremos ni una palabra (en realidad, seremos mudos)
- `socket.SHUT_RDWR` - especifica la conjunción de las dos opciones anteriores.

¿Hay algo más que debamos hacer ahora?

Como nuestra solicitud `GET` exigió que el servidor cerrara la conexión tan pronto como se enviara la respuesta y el servidor fuera informado de nuestros próximos pasos (o más bien del hecho de que ya habíamos hecho lo que queríamos), podemos suponer que la conexión está completamente terminada en este momento.

Algunos dirían que cerrarla explícitamente es una diligencia exagerada. No compartimos esta opinión y preferimos cerrar la conexión expresándola literalmente.

El método `close()` sin parámetros lo hará por nosotros: vea nuestro código en el editor.

Bien. Hemos recibido algo. ¿Vale la pena verlo con nuestros propios ojos? No lo veremos hasta que lo veamos.

In [ ]:
```
import socket

server_addr = input("What server do you want to connect to? ") # 127.0.0.
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((server_addr, 8050))
sock.send(b"GET / HTTP/1.1\r\nHost: " +
          bytes(server_addr, "utf8") +
          b"\r\nConnection: close\r\n\r\n")
reply = sock.recv(10000)
sock.shutdown(socket.SHUT_RDWR)
sock.close()
```

¿Qué obtuvimos?

No esperes que nuestro código pueda mostrar el documento recibido de la misma manera que el navegador de Internet te lo muestra. Un código capaz de hacer algo como esto no cabría en tu pantalla.

Además, no queremos escribir un nuevo navegador. Solo queremos comprobar si los datos que recibimos parecen razonables.

Lo haremos de la forma más simple (pero muy elegante): simplemente lo imprimiremos utilizando la función incorporada `repr()`, que se encarga de la presentación clara (casi) textual de cualquier objeto. No necesitamos nada más.

Es por eso que la última línea de nuestro código se ve así: `print(repr(answ))`.

Nuestro código está completo: veámoslo en todo su esplendor en la ventana del editor.

In [4]:
```
#  poner en el input: www.site.com
import socket

server_addr = input("What server do you want to connect to? ") # 127.0.0.
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((server_addr, 8050))
sock.send(b"GET / HTTP/1.1\r\nHost: " +
          bytes(server_addr, "utf8") +
          b"\r\nConnection: close\r\n\r\n")
reply = sock.recv(10000)
sock.shutdown(socket.SHUT_RDWR)
sock.close()
print(repr(reply))
```

```
b'mensaje recibido'
```

¿Qué podemos esperar de la respuesta del servidor?

Si todo salió bien (el usuario ingresó una dirección válida, Internet funcionó como se esperaba, el servidor estuvo dispuesto a cooperar, etc.) es posible que vea algo como esto en la pantalla:

```
What server do you want to connect to?  www.site.com
b'HTTP/1.1 200 OK\r\nDate: Fri, 08 Mar 2019 08:24:41
GMT\r\nServer: UltraDNS Client Redirection Server\r\nLast-
Modified: Fri, 08 Mar 2019 08:24:41 GMT\r\nAccept-Ranges:
none\r\nConnection: close\r\nContent-Type:
text/html\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n'
```

¿Parece amigable? Por supuesto que no. Entonces... ¿qué vemos aquí en realidad?

De hecho, vemos dos partes separadas:

- la primera es el encabezado de respuesta. Te contaremos un pequeño secreto: la línea superior es la más importante, ya que indica si el servidor envió de vuelta el documento solicitado o no. Mira, hay un número de tres dígitos muy significativo: `200`.

Es tu número de la suerte, ya que es el código de estado. El número `200` es tu mejor amigo, ya que anuncia que la misión fue completamente exitosa y que tienes tu documento. Las siguientes líneas describen muchos detalles importantes, pero no los necesitamos ahora. Pasa a la primera línea vacía. Es muy importante ya que separa el encabezado de...

- el documento. Sí, aquí es donde comienza. Puede que esté muy inflado (normalmente lo está) y no queremos presentarlo completo.

Ahora no es importante. No vamos a estudiar HTML. Al menos, no aquí.

Vamos a profundizar en otra cosa: qué sucede si algo sale mal. Internet es un espacio hostil: muchas cosas pueden fallar. Por ejemplo...

Introducción de una dirección inexistente o mal formada

El usuario ha introducido una dirección inexistente o mal formada (no importa si se expresa como un nombre de dominio o una dirección IP). ¿Qué sucederá entonces? Puedes ver estos accidentes aquí:

```
What server do you want to connect to? a.non.existent.name
Traceback (most recent call last):
  File "cli.py", line 5, in <module>
    sock.connect((srvaddr, 80))
socket.gaierror: [Errno -2] Name or service not known

What server do you want to connect to? anonexistentname
Traceback (most recent call last):
  File "cli.py", line 5, in <module>
    sock.connect((srvaddr, 80))
socket.gaierror: [Errno -2] Name or service not known
```

La función connect lanza una excepción llamada `socket.gaierror` y su nombre proviene del nombre de una función de bajo nivel (generalmente no proporcionada por Python sino por el núcleo del sistema operativo) llamada `getaddrinfo()`. La función intenta, entre otras cosas, encontrar la información completa de la dirección con respecto al argumento recibido.

Como probablemente adivine, `connect()` usa `getaddrinfo()` para preparar una nueva conexión con el servidor. Si `getaddreinfo()` falla, se lanza una excepción y el viaje termina antes de comenzar. Triste pero cierto.

Nota: `socket.gaierror` cubre más de una posible razón para la falla. Nuestro ejemplo muestra dos de ellas: la primera cuando la dirección es sintácticamente correcta pero no corresponde a ningún servidor existente, y la segunda cuando la dirección está obviamente mal formada.

Como puede ver, la excepción es la misma, pero lleva información acompañada diferente sobre las razones.

También es posible que exista un servidor de una dirección específica y esté funcionando pero no proporcione el servicio deseado. Por ejemplo, un servidor de correo dedicado puede no responder a las conexiones dirigidas al puerto número 80.

Si desea provocar un evento de este tipo, reemplace 80 en la invocación `connect()` con cualquier número de cinco dígitos que no exceda 65535 (11111 parece ser una buena idea) y ejecute el código.

Apostamos a que verá algo como esto:

```
What server do you want to connect to? dedicated.server
Traceback (most recent call last):
  File "cli2.py", line 6, in <module>
    sock.connect((srvaddr, 11111))
ConnectionRefusedError: [Errno 111] Connection refused
```

Como puedes ver, la excepción es diferente a la anterior: su nombre anuncia que el servidor ha rechazado nuestra conexión. En otras palabras, el servidor no está destinado a proporcionar los servicios que queremos utilizar.

La excepción socket.timeout

La última excepción de la que queremos hablarte es `socket.timeout`. Esta excepción se genera cuando la reacción del servidor no se produce en un tiempo razonable; la duración de nuestra paciencia se puede configurar mediante un método llamado `settimeout()`, pero no queremos entrar en detalles. Esperamos que nos perdones.

Si realmente quieres inducir una excepción de este tipo, tendrás que hacer algo malo como interrumpir la conexión de red en medio de una transferencia o apagar el servidor en un momento elegido con precisión. No queremos que hagas esto. Queremos que recuerdes que una situación así puede ocurrir. Prepárate. Rust nunca duerme.

Armados con un conocimiento básico sobre TCP/IP, estamos listos para continuar nuestro viaje. Nos vemos en la próxima parada.

Haciendo la vida más fácil con el módulo de solicitudes

Hemos llegado al punto en el que podemos comenzar la etapa final de nuestro viaje: sabemos lo suficiente para comunicarnos con el servicio web utilizando JSON como portador de información. Desafortunadamente, nuestro conocimiento necesita ser complementado: necesitamos un servidor que brinde un servicio web y también necesitamos una herramienta más simple que el módulo socket para hablar con el servicio.

"Un momento", puede que se pregunte aquí. "¿El módulo socket ya no satisface nuestras necesidades?"

Sí, lo hace, pero es demasiado bueno. Es demasiado exigente y demasiado poderoso. Expone muchos detalles que no están necesariamente disponibles en los niveles superiores del diseño de software. El módulo `socket` es perfecto cuando desea comprender los problemas de red a nivel TCP y aprender qué desafíos enfrenta el sistema operativo cuando intenta establecer, mantener y cerrar conexiones a Internet. Por eso lo usamos antes cuando queríamos que entraras en el mundo de las comunicaciones en red, pero al mismo tiempo, el socket es demasiado pesado y está demasiado hinchado cuando solo quieres tener una pequeña charla con un servicio web.

¿Cuál de estas demandas (servidor o herramienta) debería satisfacerse antes? El servidor, por supuesto. Necesitamos nuestro propio servidor HTTP privado que funcione solo para nosotros y desempeñe con éxito el papel de una base de API RESTFul.

Hemos decidido usar un paquete gratuito y abierto llamado `FastAPI`. Actuará como una caja negra para nosotros y no queremos persuadirte para que mires dentro de ella.

De acuerdo. Vamos a empezar.

Realizamos un script de Python llamado main.py:

In [ ]:
```
from fastapi import FastAPI, status, Response
from models import Car
import sqlite3

# Para ejecutar: fastapi dev main.py

app = FastAPI()

# Creamos un archivo como extensión .db donde se almacenará la informació
conexion=sqlite3.connect("cars.db")


@app.get('/')
async def test():
    return "CARS DATABASE"
```

```
# Método GET a la url "/cars/"
# llamaremos a nuestra aplicación (<app name> + <método permitido>)
@app.get('/cars/')
async def cars(response:Response):
    try:
        cursor = conexion.execute("SELECT * FROM cars")
        response = []
        for fila in cursor.fetchall():
            data = {'id': fila[0], 'brand': fila[1],
                    'model': fila[2], 'production_year': fila[3],
                    'convertible': fila[4]}
            response.append(data)
        return response
    except Exception as e:
        print("Error al cargar el csv %s" % str(e))
        response.status_code = status.HTTP_404_NOT_FOUND
        return "404 NOT FOUND"


# POST de insertar un nuevo dato en el csv, última línea
# Método POST a la url "/insertData/"
@app.post("/cars/", status_code=201)
async def insert(item:Car):
    conexion.execute("INSERT INTO cars(brand, model, production_year, con
                     (item.brand, item.model,
                      item.production_year, item.convertible))
    conexion.commit()
    return item


# PUT actualizar la última línea del csv
# Método PUT a la url "/updateData/" + ID a modificar
@app.put("/cars/{item_id}")
async def updateData(item_id: int, item:Car):
    conexion.execute("UPDATE cars SET brand=?, model=?, production_year=?
                     (item.brand, item.model, item.production_year,
                     item.convertible, item_id))
    # Para que se realize el cambio añadimos .commit
    conexion.commit()
    return item

# DELETE eliminar la última línea del csv
# Método Delete a la url "/deleteData/" + id
@app.delete("/cars/{item_id}")
async def deleteData(item_id: int):
    conexion.execute("DELETE FROM cars WHERE id=?;", (item_id, ))
    #  Para que se realize el cambio añadimos .commit
    conexion.commit()
    return {"item_id": item_id, "msg": "Eliminado"}
```

Y un script llamado models.py:

In [ ]:

```
# Crearemos el modelo de los datos a aguardar
from pydantic import BaseModel

# Modelo de datos:
class Car(BaseModel):
    #id: int
```

```
    brand: str
    model: str
    production_year: int
    convertible: bool
```

Ahora abra su navegador de Internet favorito y escriba la siguiente URL en la línea de dirección:

```
http://localhost:8000
```

Esto significa que ordena al navegador que se conecte a la misma máquina en la que está trabajando actualmente (`localhost`) y desea que el cliente vaya al puerto número **8000** (el puerto predeterminado de `FastAPI`).

¡Felicitaciones! ¡Tiene su servidor funcionando y operativo!

Presione **Crtl-C** en la consola si desea finalizar el servidor, pero por ahora déjelo funcionando: ¡lo necesitaremos y los recursos que brinda!

El primer programa hace un uso muy básico del poder del módulo `requests` . Echa un vistazo:

```
import requests

reply = requests.get('http://localhost:8000')
print(reply.status_code)
```

El protocolo HTTP funciona mediante el uso de métodos. Podemos decir que el método HTTP es una interacción bidireccional entre el cliente y el servidor (nota: el cliente inicia la transmisión) dedicada a la ejecución de una determinada acción. `GET` es uno de ellos, y se utiliza para convencer al servidor de que transfiera algunos recursos solicitados por el cliente.

Una función `requests` llamada `get()` inicia la ejecución del método `HTTP GET` y recibe la respuesta del servidor. Como puede ver, el código es extremadamente simple y compacto: no necesitamos lidiar con una miríada de constantes, símbolos, funciones y nociones misteriosas.

Es como si dijéramos "Hola, servidor, envíame tu recurso predeterminado".

Los únicos detalles que necesitamos proporcionar son la dirección del servidor y el número de puerto del servicio, tal como lo hicimos al usar la línea de dirección del navegador. Nota: el número de puerto se puede omitir si es igual a 80, el puerto predeterminado de HTTP.

Por supuesto, es posible que el servidor se encuentre en algún lugar alejado de nuestro escritorio, por ejemplo, en el otro hemisferio. Lo único que cambiaremos entonces es la dirección del servidor: se formará como una dirección IP o un nombre de dominio completo (`FQDN`), pero no importa para `get()` : su comportamiento es siempre el

mismo: intentará conectarse al servidor, formará una solicitud `GET` y aceptará la respuesta.

Como puede ver, la función `get()` devuelve un resultado. Es un objeto que contiene toda la información que describe la ejecución del método `GET`.

Por supuesto, lo más importante que necesitamos saber es si el método `GET` ha tenido éxito. Es por eso que utilizamos la propiedad `status_code`: contiene un número estandarizado que describe la respuesta del servidor.

Si ejecutas el código y todo funciona como se espera, verás un resultado muy breve y simple:

```
200
```

Como lo define el protocolo HTTP, el código 200 significa “bien”.

Buenas noticias.

In [2]:
```
import requests

reply = requests.get('http://localhost:8000')
print(reply.status_code)
```

```
200
```

Todos los códigos de respuesta utilizados por HTTP se encuentran aquí: https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

El módulo de solicitudes ofrece muchas formas diferentes de especificar y reconocer códigos de estado.

Observa el código:

```
import requests

print(requests.codes.__dict__)
```

Vuelca el contenido de un diccionario de estado. El resultado es muy largo y desordenado; no lo incluiremos aquí, pero te animamos a que lo estudies con atención, comparando los valores que ves con los que se presentan en Wikipedia.

De todos modos, puedes usar la propiedad codes para probar los códigos de estado de una manera más detallada que comparándolos con valores enteros. ¿Qué opinas de este fragmento?

```
if reply.status_code == requests.codes.ok:
```

Se ve mucho mejor que sólo 200, ¿no?

In [3]:
```
import requests
```

```
print(requests.codes.__dict__)
```

{'name': 'status_codes', 'continue': 100, 'CONTINUE': 100, 'switching_protocols': 101, 'SWITCHING_PROTOCOLS': 101, 'processing': 102, 'PROCESSING': 102, 'early-hints': 102, 'EARLY-HINTS': 102, 'checkpoint': 103, 'CHECKPOINT': 103, 'uri_too_long': 414, 'URI_TOO_LONG': 414, 'request_uri_too_long': 122, 'REQUEST_URI_TOO_LONG': 122, 'ok': 200, 'OK': 200, 'okay': 200, 'OKAY': 200, 'all_ok': 200, 'ALL_OK': 200, 'all_okay': 200, 'ALL_OKAY': 200, 'all_good': 200, 'ALL_GOOD': 200, '\\o/': 200, '✓': 200, 'created': 201, 'CREATED': 201, 'accepted': 202, 'ACCEPTED': 202, 'non_authoritative_info': 203, 'NON_AUTHORITATIVE_INFO': 203, 'non_authoritative_information': 203, 'NON_AUTHORITATIVE_INFORMATION': 203, 'no_content': 204, 'NO_CONTENT': 204, 'reset_content': 205, 'RESET_CONTENT': 205, 'reset': 205, 'RESET': 205, 'partial_content': 206, 'PARTIAL_CONTENT': 206, 'partial': 206, 'PARTIAL': 206, 'multi_status': 207, 'MULTI_STATUS': 207, 'multiple_status': 207, 'MULTIPLE_STATUS': 207, 'multi_stati': 207, 'MULTI_STATI': 207, 'multiple_stati': 207, 'MULTIPLE_STATI': 207, 'already_reported': 208, 'ALREADY_REPORTED': 208, 'im_used': 226, 'IM_USED': 226, 'multiple_choices': 300, 'MULTIPLE_CHOICES': 300, 'moved_permanently': 301, 'MOVED_PERMANENTLY': 301, 'moved': 301, 'MOVED': 301, '\\o-': 301, 'found': 302, 'FOUND': 302, 'see_other': 303, 'SEE_OTHER': 303, 'other': 303, 'OTHER': 303, 'not_modified': 304, 'NOT_MODIFIED': 304, 'use_proxy': 305, 'USE_PROXY': 305, 'switch_proxy': 306, 'SWITCH_PROXY': 306, 'temporary_redirect': 307, 'TEMPORARY_REDIRECT': 307, 'temporary_moved': 307, 'TEMPORARY_MOVED': 307, 'temporary': 307, 'TEMPORARY': 307, 'permanent_redirect': 308, 'PERMANENT_REDIRECT': 308, 'resume_incomplete': 308, 'RESUME_INCOMPLETE': 308, 'resume': 308, 'RESUME': 308, 'bad_request': 400, 'BAD_REQUEST': 400, 'bad': 400, 'BAD': 400, 'unauthorized': 401, 'UNAUTHORIZED': 401, 'payment_required': 402, 'PAYMENT_REQUIRED': 402, 'payment': 402, 'PAYMENT': 402, 'forbidden': 403, 'FORBIDDEN': 403, 'not_found': 404, 'NOT_FOUND': 404, '-o-': 404, '-O-': 404, 'method_not_allowed': 405, 'METHOD_NOT_ALLOWED': 405, 'not_allowed': 405, 'NOT_ALLOWED': 405, 'not_acceptable': 406, 'NOT_ACCEPTABLE': 406, 'proxy_authentication_required': 407, 'PROXY_AUTHENTICATION_REQUIRED': 407, 'proxy_auth': 407, 'PROXY_AUTH': 407, 'proxy_authentication': 407, 'PROXY_AUTHENTICATION': 407, 'request_timeout': 408, 'REQUEST_TIMEOUT': 408, 'timeout': 408, 'TIMEOUT': 408, 'conflict': 409, 'CONFLICT': 409, 'gone': 410, 'GONE': 410, 'length_required': 411, 'LENGTH_REQUIRED': 411, 'precondition_failed': 412, 'PRECONDITION_FAILED': 412, 'precondition': 428, 'PRECONDITION': 428, 'request_entity_too_large': 413, 'REQUEST_ENTITY_TOO_LARGE': 413, 'content_too_large': 413, 'CONTENT_TOO_LARGE': 413, 'request_uri_too_large': 414, 'REQUEST_URI_TOO_LARGE': 414, 'unsupported_media_type': 415, 'UNSUPPORTED_MEDIA_TYPE': 415, 'unsupported_media': 415, 'UNSUPPORTED_MEDIA': 415, 'media_type': 415, 'MEDIA_TYPE': 415, 'requested_range_not_satisfiable': 416, 'REQUESTED_RANGE_NOT_SATISFIABLE': 416, 'requested_range': 416, 'REQUESTED_RANGE': 416, 'range_not_satisfiable': 416, 'RANGE_NOT_SATISFIABLE': 416, 'expectation_failed': 417, 'EXPECTATION_FAILED': 417, 'im_a_teapot': 418, 'IM_A_TEAPOT': 418, 'teapot': 418, 'TEAPOT': 418, 'i_am_a_teapot': 418, 'I_AM_A_TEAPOT': 418, 'misdirected_request': 421, 'MISDIRECTED_REQUEST': 421, 'unprocessable_entity': 422, 'UNPROCESSABLE_ENTITY': 422, 'unprocessable': 422, 'UNPROCESSABLE': 422, 'unprocessable_content': 422, 'UNPROCESSABLE_CONTENT': 422, 'locked': 423, 'LOCKED': 423, 'failed_dependency': 424, 'FAILED_DEPENDENCY': 424, 'dependency': 424, 'DEPENDENCY': 424, 'unordered_collection': 425, 'UNORDERED_COLLECTION': 425, 'unordered': 425, 'UNORDERED': 425, 'too_early': 425, 'TOO_EARLY': 425, 'upgrade_required': 426, 'UPGRADE_REQUIRED': 426, 'upgrade': 426, 'UPGRADE': 426, 'precondition_required': 428, 'PRECONDITION_REQUIRED': 428, 'too_many_requests': 429, 'TOO_MANY_REQUESTS': 429, 'too_many': 429, 'TOO_MANY': 429, 'header_fields_too_large': 431, 'HEADER_FIELDS_TOO_LARGE': 431, 'fields_too_large': 431, 'FIELDS_TOO_LARGE': 431, 'no_response': 444, 'NO_RESPONSE': 444, 'none': 444, 'NONE': 444, 'retry_with': 449, 'RETRY_WITH': 449, 'retry': 449, 'RETRY': 449, 'blocked_by_windows_parental_controls': 450, 'BLOCKED_BY_WINDOWS_PARENTAL_CONTROLS': 450, 'parental_controls': 450, 'PARENTAL_CONTROLS': 45

```
0, 'unavailable_for_legal_reasons': 451, 'UNAVAILABLE_FOR_LEGAL_REASONS':
451, 'legal_reasons': 451, 'LEGAL_REASONS': 451, 'client_closed_request':
499, 'CLIENT_CLOSED_REQUEST': 499, 'internal_server_error': 500, 'INTERNAL
_SERVER_ERROR': 500, 'server_error': 500, 'SERVER_ERROR': 500, '/o\\': 50
0, '✗': 500, 'not_implemented': 501, 'NOT_IMPLEMENTED': 501, 'bad_gatewa
y': 502, 'BAD_GATEWAY': 502, 'service_unavailable': 503, 'SERVICE_UNAVAILA
BLE': 503, 'unavailable': 503, 'UNAVAILABLE': 503, 'gateway_timeout': 504,
'GATEWAY_TIMEOUT': 504, 'http_version_not_supported': 505, 'HTTP_VERSION_N
OT_SUPPORTED': 505, 'http_version': 505, 'HTTP_VERSION': 505, 'variant_als
o_negotiates': 506, 'VARIANT_ALSO_NEGOTIATES': 506, 'insufficient_storag
e': 507, 'INSUFFICIENT_STORAGE': 507, 'bandwidth_limit_exceeded': 509, 'BA
NDWIDTH_LIMIT_EXCEEDED': 509, 'bandwidth': 509, 'BANDWIDTH': 509, 'not_ext
ended': 510, 'NOT_EXTENDED': 510, 'network_authentication_required': 511,
'NETWORK_AUTHENTICATION_REQUIRED': 511, 'network_auth': 511, 'NETWORK_AUT
H': 511, 'network_authentication': 511, 'NETWORK_AUTHENTICATION': 511}
```

Cuando sabes que la respuesta del servidor es correcta, te gustaría saber cuál es la respuesta en sí, ¿no?

La respuesta del servidor consta de dos partes: el encabezado y el contenido. Ambas partes tienen su representación en el objeto devuelto por la función `get()`.
Empecemos por el encabezado.

El encabezado de la respuesta se almacena dentro de la propiedad llamada headers (es un diccionario). Veámoslo:

```
import requests

reply = requests.get('http://localhost:8000')
print(reply.headers)
```

El programa produce la siguiente salida:

```
{'date': 'Sat, 05 Oct 2024 14:21:48 GMT', 'server':
'uvicorn', 'content-length': '22', 'content-type':
'application/json'}
```

Como puede ver, el encabezado de la respuesta consta de una serie de campos con valores asociados (en realidad, cada campo ocupa una línea de la respuesta). La mayoría de ellos no nos interesan, aunque algunos son cruciales, por ejemplo, Content-Type, que describe cuál es realmente el contenido de la respuesta del servidor.

Puede acceder a él directamente mediante una búsqueda rutinaria en el diccionario, como esta:

```
reply.headers['Content-Type']
```

In [4]:
```
import requests

reply = requests.get('http://localhost:8000')
print(reply.headers)
```

```
{'date': 'Tue, 10 Dec 2024 12:45:03 GMT', 'server': 'uvicorn', 'content-le
ngth': '15', 'content-type': 'application/json'}
```

In [5]:
```
import requests

reply = requests.get('http://localhost:8000')
print(reply.headers['Content-Type'])
```

```
application/json
```

El contenido de la respuesta sin procesar se almacena mediante la propiedad de texto:

```
import requests

reply = requests.get('http://localhost:3000')
print(reply.text)
```

La propiedad contiene texto simple tomado tal cual directamente del flujo de datos, por lo tanto, es solo una cadena. No se aplican conversiones.

El código produce el siguiente resultado:

```
CARS DATABASE
```

Nota: cuando tanto el cliente como el servidor saben que el contenido (sin importar qué parte lo envió) contiene un mensaje JSON, también es posible utilizar un método llamado `json()` que devuelve exactamente lo que podemos esperar: un diccionario o una lista de diccionarios. Te lo mostraremos en la siguiente sección.

En general, el protocolo HTTP define los siguientes métodos:

```
GET method
```

El método `GET` tiene como objetivo obtener una pieza de información (un recurso) del servidor; por supuesto, un servidor simple ofrece más de un recurso, por lo que el método `GET` tiene los medios para permitir que el cliente especifique con precisión sus demandas; si el cliente no tiene demandas e inicia `GET` sin identificación del recurso, la respuesta del servidor contendrá el documento raíz; esto es exactamente lo que vimos hace algún tiempo, cuando nuestro propio servidor nos envió este simple texto `CARS DATABASE`. en otras palabras, si desea que el servidor le brinde algo, `GET` es la forma de solicitarlo.

In [6]:
```
import requests

reply = requests.get('http://localhost:8000')
print(reply.text)
```

```
"CARS DATABASE"
```

In [10]:
```
import requests

reply = requests.get('http://localhost:8000/cars')
print(reply.text)
```

[{"id":1,"brand":"Ford","model":"Mustang","production_year":1972,"convertible":0},{"id":2,"brand":"Chevrolet","model":"Camaro","production_year":1988,"convertible":1},{"id":3,"brand":"Aston Martin","model":"Rapide","production_year":2010,"convertible":0},{"id":4,"brand":"Maserati","model":"Mexico","production_year":1970,"convertible":0},{"id":5,"brand":"Nissan","model":"Fairlady","production_year":1974,"convertible":0},{"id":6,"brand":"Mercedes Benz","model":"300SL","production_year":1957,"convertible":1}]

Método POST

- `POST`, al igual que `GET`, se utiliza para transferir un recurso, pero en la dirección opuesta: del cliente al servidor; al igual que en `GET`, se debe proporcionar la identificación del recurso (el servidor quiere saber cómo se debe nombrar la información que ha recibido); también se supone que el recurso que el cliente envía es nuevo para el servidor; no reemplaza ni sobrescribe ninguno de los datos recopilados anteriormente;
- para resumir, si desea darle al servidor algo nuevo, `POST` está listo para ser su entregador.

Método PUT

- `PUT`, de manera similar a `POST`, transfiere un recurso del cliente al servidor, pero la intención es diferente: el recurso que se envía tiene como objetivo reemplazar los datos almacenados anteriormente;
- En pocas palabras: si desea actualizar algo que el servidor está guardando actualmente, `PUT` sabrá cómo hacerlo.

Método DELETE

- `DELETE`: este nombre no deja lugar a dudas: se utiliza para ordenar al servidor que elimine un recurso de una identificación dada; el recurso no está disponible a partir de ese momento;
- Lo sentimos, pero no hay una forma más sencilla de explicarlo, creemos; no nos equivocamos, ¿verdad?

Métodos restantes: head, connect, options, trace

No los vamos a analizar aquí, pero siéntase libre de profundizar su conocimiento por su cuenta.

Como probablemente sospeche, todos los métodos HTTP enumerados tienen sus reflejos (o más bien hermanos) dentro del módulo `requests`.

Sí, tiene toda la razón.

El diagrama, aunque extremadamente simplificado y privado de detalles importantes, nos muestra las herramientas más importantes que usaremos pronto para jugar un juego con nuestro servidor. No se preocupe, le daremos a conocer todos los medios y argumentos adicionales.

Parece que nos hemos olvidado de un tema importante: ¿qué pasará si algo sale mal? El servidor puede fallar, los medios de transmisión pueden estar fuera de servicio, etc.,

etc., etc. ¿Cómo nos defendemos de todas estas desgracias?

Todas las funciones de `solicitudes` tienen la costumbre de lanzar una excepción cuando encuentran cualquier tipo de problema de comunicación, aunque algunos de los problemas parecen ser más comunes que otros.

Por ejemplo, echemos un vistazo a un problema crucial llamado `tiempo de espera`.

Es normal que el servidor no responda de inmediato: establecer conexiones, transmitir datos, buscar recursos; todos estos pasos llevan tiempo. Vale, lo sabemos, pero nuestra paciencia también es limitada. Por lo general, sabemos muy bien cuánto tiempo aceptamos esperar y no queremos esperar más. ¿Se pueden cumplir nuestras expectativas?

Por supuesto que sí, y una excepción será muy útil: mira:

```
import requests

try:
    reply = requests.get('http://localhost:8000',
timeout=1)
except requests.exceptions.Timeout:
    print('Sorry, Mr. Impatient, you didn\'t get your
data')
else:
    print('Here is your data, my Master!')
```

Como puedes ver, la función `get()` toma un argumento adicional llamado `timeout`: es el tiempo máximo (medido en segundos y expresado como un número real) que acordamos esperar por la respuesta de un servidor. Si se excede el tiempo, `get()` lanzará una excepción llamada `requests.exceptions.Timeout`.

Si el servidor está listo y no está muy ocupado, un segundo es más que suficiente para procesar una solicitud tan simple, por lo que debes esperar buenas noticias: el programa escribirá:

```
Here is your data, my Master!
```

Pero si cambias el tiempo de espera radicalmente a un valor inquietantemente pequeño como 0,00001, es muy probable que tengas que soportar las siguientes malas noticias:

```
Sorry, Mr. Impatient, you didn't get your data.
```

In [15]:
```
import requests

try:
    reply = requests.get('http://localhost:8000', timeout=1)
except requests.exceptions.Timeout:
    print('Sorry, Mr. Impatient, you didn\'t get your data')
else:
    print('Here is your data, my Master!')
```

```
Here is your data, my Master!
```

Por supuesto, los problemas pueden aparecer mucho antes, por ejemplo, al establecer la conexión:

```
import requests

try:
    reply = requests.get('http://localhost:8001',
timeout=1)
except requests.exceptions.ConnectionError:
    print('Nobody\'s home, sorry!')
else:
    print('Everything fine!')
```

Este código no tiene ninguna posibilidad de ejecutarse correctamente: está dirigiendo sus esfuerzos al puerto `8001`, mientras que nuestro servidor está escuchando en el puerto `8000`. Ninguna ayuda podrá solucionar este malentendido: el cliente y el servidor no se encontrarán y se generará una excepción. Su nombre es:

```
requests.exceptions.ConnectionError
```

In [18]:
```
import requests

try:
    reply = requests.get('http://localhost:8001', timeout=1)
except requests.exceptions.ConnectionError:
    print('Nobody\'s home, sorry!')
else:
    print('Everything fine!')
```

```
Nobody's home, sorry!
```

Errar es humano, por lo que también es posible que usted u otro desarrollador dejen la URL del recurso en un estado algo deformado. Observe el código en la ventana del editor.

Los desastres de este tipo se producen mediante una excepción denominada:

```
requests.exceptions.InvalidURL
```

Hemos reunido todas las excepciones de solicitudes en un solo lugar y las presentamos como un árbol: así es como se ve:

```
RequestException
|___HTTPError
|___ConnectionError
|   |___ProxyError
|   |___SSLError
|___Timeout
|   |___ConnectTimeout
|   |___ReadTimeout
|___URLRequired
|___TooManyRedirects
```

```
      |___MissingSchema
      |___InvalidSchema
      |___InvalidURL
      |   |___InvalidProxyURL
      |___InvalidHeader
      |___ChunkedEncodingError
      |___ContentDecodingError
      |___StreamConsumedError
      |___RetryError
      |___UnrewindableBodyError
```

Algo bueno para todos.

Ahora que ya conocemos bien el mundo de las solicitudes, podemos dar el siguiente paso y entrar en el mundo de los servicios web. ¿Estás listo?

In [19]:
```
import requests

try:
    reply = requests.get('http://////////////')
except requests.exceptions.InvalidURL:
    print('Recipient unknown!')
else:
    print('Everything fine!')
```

```
Recipient unknown!
```

Introducción de una dirección inexistente o mal formada

Hemos llegado al punto en el que estamos listos para reunir todos los nuevos datos y herramientas y unir todas estas piezas en un bloque funcional. Ya sabes cómo funciona HTTP, cómo se monta sobre la pila TCP y cómo el servidor HTTP puede hacer mucho más por nosotros que simplemente almacenar y publicar imágenes elegantes y videos divertidos.

De hecho, un servidor web correctamente entrenado puede ser una puerta de entrada muy efectiva y conveniente a bases de datos muy complicadas y pesadas u otros servicios diseñados para almacenar y procesar información. Además, la estructura de la base de datos (o del servicio) puede variar, por ejemplo, puede ser una base de datos relacional simple que reside en un solo archivo o, por el contrario, una enorme nube distribuida de servidores cooperativos; pero la interfaz proporcionada al usuario (tú) siempre será la misma.

Podemos decir que para eso se inventó REST. Gracias a él, programas muy diferentes escritos en tecnologías muy diferentes pueden utilizar datos compartidos a través de una interfaz universal.

La interfaz en sí permite al usuario realizar un conjunto básico de operaciones: son elementales, pero lo suficientemente complejas como para crear servicios complejos. Un conjunto de cuatro operaciones se esconde bajo el siguiente acrónimo misterioso:

El acrónimo CRUD: **C**reate **R**ead **U**pdate **D**elete

Nota: este es un acrónimo de cuatro letras, lo que lo hace realmente especial. Vamos a arrojar algo de luz sobre él.

C means Create

Si tiene la capacidad de "crear", puede agregar nuevos elementos a la recopilación de datos, por ejemplo, escribir una nueva publicación en el blog, agregar una nueva imagen a la galería, almacenar los datos de un nuevo cliente en una base de datos de clientes, etc.

En el nivel "REST", la creación de nuevos elementos se implementa mediante el método "POST HTTP".

R means Read

La lectura y recuperación es la capacidad básica de explorar datos almacenados en una colección, por ejemplo, leer publicaciones en el blog de alguien, ver imágenes en una galería, estudiar los registros de clientes en una base de datos, etc.

En el nivel `REST`, la recuperación de elementos se implementa mediante el método `GET HTTP`.

U means Update

Actualizas los datos dentro de una colección cuando modificas el contenido del elemento seleccionado sin eliminarlo, por ejemplo, editas la publicación de tu blog, cambias el tamaño de una imagen en la galería, ingresas la información de ventas del cliente actual, etc.

En el nivel `REST`, la actualización de los datos existentes se implementa mediante el método `PUT HTTP`.

Para completar nuestro conjunto, necesitamos la D de Delete (eliminar).

La eliminación se produce cuando eliminas tu publicación del blog, purgas una imagen de la galería o cancelas la cuenta de un cliente.

En el nivel `REST`, la eliminación de datos existentes se implementa mediante el método `DELETE HTTP`.

Ahora estamos listos para realizar algunos experimentos simples pero instructivos con `JSON`. Lo usaremos como un lenguaje intermedio para comunicarnos con el servidor HTTP, implementando CRUD y almacenando una colección de muestra de datos.

Estas son nuestras suposiciones:

- Haremos uso del `FastAPI` presentado anteriormente; intentaremos que funcione a pleno rendimiento con las cuatro letras que componen CRUD;
- Nuestra base de datos inicial, procesada según nuestras demandas por el `FastAPI`, será una colección de autos retro escritos en la base de datos

`cars.db`; el `FastAPI` leerá la base de datos y manejará su contenido de acuerdo con nuestras acciones;
- cada automóvil se describe por:
- `id` – un número de artículo único; nota – cada artículo en la colección debe tener la propiedad de este nombre – así es como el servidor identifica cada artículo y diferencia los artículos entre sí;
- `brand` – una cadena;
- `model` – una cadena;
- `production_year` – un número entero;
- `convertible` – un valor booleano;
- la base de datos inicial contiene datos de seis automóviles – no se sorprenda si el servidor modifica su contenido; si desea restablecer la colección al estado inicial, detenga el servidor (use Ctrl-C para este propósito), reemplace el archivo con su versión original (siempre puede descargarlo) y vuelva a iniciar el servidor.

Comenzaremos nuestro viaje con la letra R (leer). Intentaremos convencer al servidor de que nos muestre todos los coches que ofrece.

Nota: `FastAPI` asume que la colección de datos hereda su nombre del nombre del archivo de datos de origen. Como hemos nombrado el archivo coches, el servidor también publicará los datos como coches. Tienes que usar el nombre en la URI a menos que quieras obtener el documento predeterminado (raíz), lo que no nos sirve para nada.

Mira el código en el editor. Es muy básico hasta ahora, pero pronto crecerá; te lo prometemos:

- línea 1: importamos el módulo `requests`;
- línea 3: nos vamos a conectar al servidor dentro del bloque try; esto nos permitirá protegernos contra los posibles efectos de los problemas de conexión;
- línea 4: formamos la solicitud `GET` y la dirigimos al recurso llamado `cars` ubicado en el servidor que trabaja en la dirección especificada como localhost, escuchando en el puerto número `8000`;
- línea 5: ¿hemos tenido éxito?
- línea 6: desafortunadamente, hemos fallado; parece que el servidor no está funcionando o es inaccesible;
- línea 8: buenas noticias: ¡el servidor respondió! Veamos el código de estado;
- línea 9: imprimimos los datos que nos ha enviado el servidor (bastante aburrido); el contenido de la respuesta se almacena como una propiedad de texto del objeto de respuesta;
- línea 11: malas noticias: al servidor no le agradamos ni nosotros ni nuestra solicitud.

Si todo va bien, deberíamos ver el contenido completo del archivo `cars.db` impreso en la pantalla. Por supuesto, este no es un logro muy especial, pero ahora estamos seguros de que el servidor está listo para servirnos y estamos listos para darle órdenes.

In [20]:
```
import requests

try:
```

```
    reply = requests.get("http://localhost:8000/cars")
except requests.RequestException:
    print("Communication error")
else:
    if reply.status_code == requests.codes.ok:
        print(reply.text)
    else:
        print("Server error")
```

```
[{"id":1,"brand":"Ford","model":"Mustang","production_year":1972,"converti
ble":0},{"id":2,"brand":"Chevrolet","model":"Camaro","production_year":198
8,"convertible":1},{"id":3,"brand":"Aston Martin","model":"Rapide","produc
tion_year":2010,"convertible":0},{"id":4,"brand":"Maserati","model":"Mexic
o","production_year":1970,"convertible":0},{"id":5,"brand":"Nissan","mode
l":"Fairlady","production_year":1974,"convertible":0},{"id":6,"brand":"Mer
cedes Benz","model":"300SL","production_year":1957,"convertible":1}]
```

El servidor HTTP es capaz de transferir prácticamente cualquier tipo de datos: texto, imagen, vídeo, sonido y muchos otros. La pregunta que tenemos que afrontar y responder es: ¿cómo reconocemos que realmente hemos recibido el mensaje JSON?

Sí, por supuesto, es obvio que hemos recibido lo que esperábamos, pero es bastante imposible instalarlo "a simple vista" en cada fragmento de código del cliente. Afortunadamente, hay una forma más sencilla de resolver este problema. El encabezado de respuesta del servidor contiene un campo llamado `Content-Type`. El valor del campo es analizado por el módulo `requests`, y si su valor anuncia JSON, un método llamado `json()` devuelve la cadena que contiene el mensaje recibido.

Hemos modificado un poco el código: mire en el editor y analice las líneas:

- Línea 9: imprimimos el valor del campo `Content-Type`;
- Línea 10: imprimimos el texto devuelto por el método `json()`.

Esto es lo que tenemos:

```
application/json
[{'id': 1, 'brand': 'Ford', 'model': 'Mustang',
'production_year': 1972, 'convertible': False}, {'id': 2,
'brand': 'Chevrolet', 'model': 'Camaro',
'production_year': 1988, 'convertible': True}, {'id': 3,
'brand': 'Aston Martin', 'model': 'Rapide',
'production_year': 2010, 'convertible': False}, {'id': 4,
'brand': 'Maserati', 'model': 'Mexico', 'production_year':
1970, 'convertible': False}, {'id': 5, 'brand': 'Nissan',
'model': 'Fairlady', 'production_year': 1974,
'convertible': False}, {'id': 6, 'brand': 'Mercedes Benz',
'model': '300SL', 'production_year': 1957, 'convertible':
True}]
```

Tenga en cuenta la línea que comienza con `application/json`: esta es una premisa utilizada por el módulo de solicitudes para diagnosticar el contenido de la respuesta.

In [21]:
```
import requests

try:
    reply = requests.get("http://localhost:8000/cars")
except:
    print("Communication error")
else:
    if reply.status_code == requests.codes.ok:
        print(reply.headers['Content-Type'])
        print(reply.json())
    else:
        print("Server error")
```

```
application/json
[{'id': 1, 'brand': 'Ford', 'model': 'Mustang', 'production_year': 1972,
'convertible': 0}, {'id': 2, 'brand': 'Chevrolet', 'model': 'Camaro', 'pro
duction_year': 1988, 'convertible': 1}, {'id': 3, 'brand': 'Aston Martin',
'model': 'Rapide', 'production_year': 2010, 'convertible': 0}, {'id': 4,
'brand': 'Maserati', 'model': 'Mexico', 'production_year': 1970, 'converti
ble': 0}, {'id': 5, 'brand': 'Nissan', 'model': 'Fairlady', 'production_ye
ar': 1974, 'convertible': 0}, {'id': 6, 'brand': 'Mercedes Benz', 'model':
'300SL', 'production_year': 1957, 'convertible': 1}]
```

Leer mensajes JSON sin procesar no es muy divertido. Para ser honesto, no es nada divertido. Hagamos las cosas un poco más divertidas y escribamos un código sencillo para presentar las respuestas del servidor de una manera elegante y clara.

Observa el código en el editor. Este es nuestro intento de afrontar este ambicioso desafío.

Analicémoslo:

- Línea 3: hemos recopilado todos los nombres de las propiedades en un solo lugar; los usaremos para realizar búsquedas en los datos JSON e imprimir una hermosa línea de encabezado sobre la tabla;
- Línea 4: estos son los anchos ocupados por las propiedades;
- Línea 7: usaremos esta función para imprimir el encabezado de la tabla;
- Línea 8: iteramos a través de `key_names` y `key_widths` acoplados entre sí mediante la función `zip()`;
- Línea 9: imprimimos el nombre de cada propiedad expandido a la longitud deseada y colocamos una barra al final;
- Línea 10: es hora de completar la línea de encabezado;
- Línea 13: usaremos esta función para imprimir una línea llena con los datos de cada automóvil;
- Línea 14: la iteración es exactamente la misma que en `showhead()`, pero...
- Línea 15: ...imprimimos el valor de la propiedad seleccionada en lugar del título de la columna;
- Línea 19: usaremos esta función para imprimir el contenido del mensaje JSON como una lista de elementos;
- Línea 20: presentaremos al usuario una tabla encantadora con un encabezado...
- Líneas 21 y 19: ...y un conjunto de datos de todos los automóviles de la lista, un automóvil por línea;

- Línea 31: hacemos uso de nuestro nuevo código aquí.

El resultado ahora se ve de la siguiente manera:

```
id        | brand          | model     | production_year
| convertible    |
1         | Ford           | Mustang   | 1972
| False          |
2         | Chevrolet      | Camaro    | 1988
| True           |
3         | Aston Martin   | Rapide    | 2010
| False          |
4         | Maserati       | Mexico    | 1970
| False          |
5         | Nissan         | Fairlady  | 1974
| False          |
6         | Mercedes Benz  | 300SL     | 1957
| True           |
```

In [22]:
```python
import requests

key_names = ["id", "brand", "model", "production_year", "convertible"]
key_widths = [10, 15, 10, 20, 15]


def show_head():
    for (n, w) in zip(key_names, key_widths):
        print(n.ljust(w), end='| ')
    print()


def show_car(car):
    for (n, w) in zip(key_names, key_widths):
        print(str(car[n]).ljust(w), end='| ')
    print()


def show(json):
    show_head()
    for car in json:
        show_car(car)


try:
    reply = requests.get('http://localhost:8000/cars')
except requests.RequestException:
    print('Communication error')
else:
    if reply.status_code == requests.codes.ok:
        show(reply.json())
    else:
        print('Server error')
```

```
id        | brand          | model     | production_year     | convertible
|
1         | Ford           | Mustang   | 1972                | 0
|
2         | Chevrolet      | Camaro    | 1988                | 1
|
3         | Aston Martin   | Rapide    | 2010                | 0
|
4         | Maserati       | Mexico    | 1970                | 0
|
5         | Nissan         | Fairlady  | 1974                | 0
|
6         | Mercedes Benz  | 300SL     | 1957                | 1
|
```

Por defecto, un servidor que implementa la versión 1.1 de HTTP funciona de la siguiente manera:

- espera la conexión del cliente;
- lee la solicitud del cliente;
- envía su respuesta;
- mantiene activa la conexión, esperando la próxima solicitud del cliente;
- si el cliente está inactivo durante algún tiempo, el servidor cierra silenciosamente la conexión; esto significa que el cliente está obligado a establecer una nueva conexión nuevamente si desea enviar otra solicitud.

Agreguemos un nuevo auto a nuestra oferta. Esto significa que ahora invitaremos a la letra C al escenario. Agregar un nuevo elemento significa que tendremos que enviar el elemento al servidor. Tenga cuidado, ya que esto requerirá algunos pasos adicionales.

Los nuevos pasos están codificados en la línea 1 y en las líneas 21 a 53:

- Línea 1: agregamos json a la lista de importación; lo necesitaremos para hacer una representación textual del nuevo elemento/automóvil;
- Línea 39: si vamos a enviar algo al servidor, el servidor debe saber qué es realmente; como ya sabe, el servidor nos informa sobre el tipo de contenido mediante el campo `Content-Type`; podemos usar la misma técnica para advertir al servidor que estamos enviando algo más que una simple solicitud. Es por eso que preparamos nuestro campo `Content-Type` con el valor apropiado;
- Línea 40: ¡Mire! ¡Este es nuestro nuevo auto! Preparamos todos los datos necesarios y los empaquetamos dentro de un diccionario Python; por supuesto, los convertiremos a JSON antes de enviarlo al mundo;
- Línea 45: queremos comprobar cómo se ve el mensaje JSON resultante; es mejor prevenir que curar;

- Línea 47: aquí es donde suceden las cosas más importantes; invocamos la función `post()` (observe la URI; solo apunta al recurso, no al elemento en particular) y establecemos dos parámetros adicionales: uno (encabezados) para complementar el encabezado de la solicitud con el campo `Content-Type`, y el segundo (datos) para pasar el mensaje JSON a la solicitud.

Crucemos los dedos porque lo ejecutaremos ahora.

Este es el resultado:

```
{"id": 7, "brand": "Chevrolet", "model": "Camaro",
"production_year": 1988, "convertible": true}
reply=201
id         | brand           | model     | production_year
| convertible    |
1          | Ford            | Mustang   | 1972
| 0              |
2          | Chevrolet       | Camaro    | 1988
| 1              |
3          | Aston Martin    | Rapide    | 2010
| 0              |
4          | Maserati        | Mexico    | 1970
| 0              |
5          | Nissan          | Fairlady  | 1974
| 0              |
6          | Mercedes Benz   | 300SL     | 1957
| 1              |
7          | Chevrolet       | Camaro    | 1988
| 1              |
```

¡Guau! ¡Funcionó! ¡Qué alegría! Observa el código de estado del servidor: es `201` ("creado").

In [23]:
```
import json
import requests

key_names = ["id", "brand", "model", "production_year", "convertible"]
key_widths = [10, 15, 10, 20, 15]


def show_head():
    for (n, w) in zip(key_names, key_widths):
        print(n.ljust(w), end='| ')
    print()


def show_empty():
    for w in key_widths:
        print(' '.ljust(w), end='| ')
    print()


def show_car(car):
    for (n, w) in zip(key_names, key_widths):
        print(str(car[n]).ljust(w), end='| ')
```

```
    print()


def show(json):
    show_head()
    if type(json) is list:
        for car in json:
            show_car(car)
    elif type(json) is dict:
        if json:
            show_car(json)
        else:
            show_empty()


h_close = {'Connection': 'Close'}
h_content = {'Content-Type': 'application/json'}
new_car = {'id': 7,
           'brand': 'Chevrolet',
           'model': 'Camaro',
           'production_year': 1988,
           'convertible': True}
print(json.dumps(new_car))
try:
    reply = requests.post('http://localhost:8000/cars', headers=h_content
    print("reply=" + str(reply.status_code))
    reply = requests.get('http://localhost:8000/cars/', headers=h_close)
except requests.RequestException:
    print('Communication error')
else:
    if reply.status_code == requests.codes.ok:
        show(reply.json())
    elif reply.status_code == requests.codes.not_found:
        print("Resource not found")
    else:
        print('Server error')
```

```
{"id": 7, "brand": "Chevrolet", "model": "Camaro", "production_year": 198
8, "convertible": true}
reply=201
id        | brand          | model     | production_year     | convertible
|
1         | Ford           | Mustang   | 1972                | 0
|
2         | Chevrolet      | Camaro    | 1988                | 1
|
3         | Aston Martin   | Rapide    | 2010                | 0
|
4         | Maserati       | Mexico    | 1970                | 0
|
5         | Nissan         | Fairlady  | 1974                | 0
|
6         | Mercedes Benz  | 300SL     | 1957                | 1
|
8         | Chevrolet      | Camaro    | 1988                | 1
|
```

La letra `U` de Update, actualiza uno de los elementos existentes. Actualizar un elemento es en realidad similar a agregar uno.

Eche un vistazo al código en el editor. Vamos a analizarlo.

- Línea 33: el encabezado es el mismo que antes, ya que enviaremos json al servidor;
- Líneas 34 a 38: estos son los nuevos datos para el elemento con id igual a 6; nota: hemos actualizado el año de producción (debería ser 1967 en lugar de 1957)
- Línea 40: ahora invocamos la función `put()`; nota: tenemos que crear una URI que indique claramente el elemento que se está modificando; además, debemos enviar el elemento completo, no solo la propiedad modificada.

Y esto es lo que vemos en la pantalla:

```
res=200
id         | brand          | model     | production_year
| convertible    |
1          | Ford           | Mustang   | 1972
| 0              |
2          | Chevrolet      | Camaro    | 1988
| 1              |
3          | Aston Martin   | Rapide    | 2010
| 0              |
4          | Maserati       | Mexico    | 1970
| 0              |
5          | Nissan         | Fairlady  | 1974
| 0              |
6          | Mercedes Benz  | 300SL     | 1957
| 1              |
7          | Mercedes Benz  | 300SL     | 1967
| 1              |
```

In [24]:
```
import requests, json

key_names = ["id", "brand", "model", "production_year", "convertible"]
key_widths = [10, 15, 10, 20, 15]


def show_head():
    for (n, w) in zip(key_names, key_widths):
        print(n.ljust(w), end='| ')
    print()


def show_empty():
    for w in key_widths:
        print(' '.ljust(w), end='| ')
    print()


def show_car(car):
    for (n, w) in zip(key_names, key_widths):
        print(str(car[n]).ljust(w), end='| ')
    print()
```

```
def show(json):
    show_head()
    if type(json) is list:
        for car in json:
            show_car(car)
    elif type(json) is dict:
        if json:
            show_car(json)
        else:
            show_empty()


h_close = {'Connection': 'Close'}
h_content = {'Content-Type': 'application/json'}
car = {'id': 8,
       'brand': 'Mercedes Benz',
       'model': '300SL',
       'production_year': 1967,
       'convertible': True}
try:
    reply = requests.put('http://localhost:8000/cars/8', headers=h_conten
    print("res=" + str(reply.status_code))
    reply = requests.get('http://localhost:8000/cars/', headers=h_close)
except requests.RequestException:
    print('Communication error')
else:
    if reply.status_code == requests.codes.ok:
        show(reply.json())
    elif reply.status_code == requests.codes.not_found:
        print("Resource not found")
    else:
        print('Server error')
```

```
res=200
id        | brand          | model     | production_year     | convertible
|
1         | Ford           | Mustang   | 1972                | 0
|
2         | Chevrolet      | Camaro    | 1988                | 1
|
3         | Aston Martin   | Rapide    | 2010                | 0
|
4         | Maserati       | Mexico    | 1970                | 0
|
5         | Nissan         | Fairlady  | 1974                | 0
|
6         | Mercedes Benz  | 300SL     | 1957                | 1
|
8         | Mercedes Benz  | 300SL     | 1967                | 1
|
```

Ahora vamos a cambiar las letras y convertir a R en nuestro héroe actual. Te mostraremos cómo eliminar un automóvil de nuestra oferta.

Como ya sabes, esto se hace usando el método `DELETE`, que está cubierto por la función `delete()`. Además, haremos algo más: dividiremos nuestra acción en dos etapas:

1. le pediremos al servidor que elimine un automóvil de una identificación específica sabiendo que el servidor mantendrá activa la conexión;

2. le pediremos al servidor que presente el contenido actual de la oferta y sugiera que la conexión se cierre inmediatamente después de la transmisión.

Analiza el código en el editor:

- Línea 37: preparamos nuestro propio encabezado de solicitud que complementará el predeterminado que se envía silenciosamente junto con cada solicitud; es un diccionario con la clave Connection (este es el mismo nombre que el enviado por el servidor) y el valor establecido en `Close`; lo enviaremos al servidor pronto;

- Línea 39: utilizamos `delete()` – anote la URI que describe el elemento a eliminar;

- Línea 40: imprimimos el código de estado del servidor;

- Línea 41: le pedimos al servidor que nos muestre la lista completa de autos, pero también enviamos nuestra solicitud para cerrar la conexión – esto se hace configurando un parámetro llamado headers;

- Línea 46: nos gustaría verificar si el servidor ha respetado nuestra recomendación.

Y este es nuestro resultado:

```
res=200
id          | brand           | model      | production_year
| convertible    |
1           | Ford            | Mustang    | 1972
| 0              |
2           | Chevrolet       | Camaro     | 1988
| 1              |
3           | Aston Martin    | Rapide     | 2010
| 0              |
4           | Maserati        | Mexico     | 1970
| 0              |
5           | Nissan          | Fairlady   | 1974
| 0              |
6           | Mercedes Benz   | 300SL      | 1957
| 1              |
```

Como puede ver, la eliminación fue exitosa (no hay ningún automóvil con `id 8` en la lista), el servidor respondió con `200` (“ok”) y cumplió con nuestra solicitud.

In [25]:
```
import requests

key_names = ["id", "brand", "model", "production_year", "convertible"]
key_widths = [10, 15, 10, 20, 15]


def show_head():
    for (n, w) in zip(key_names, key_widths):
```

```
        print(n.ljust(w), end='| ')
    print()


def show_empty():
    for w in key_widths:
        print(' '.ljust(w), end='| ')
    print()


def show_car(car):
    for (n, w) in zip(key_names, key_widths):
        print(str(car[n]).ljust(w), end='| ')
    print()


def show(json):
    show_head()
    if type(json) is list:
        for car in json:
            show_car(car)
    elif type(json) is dict:
        if json:
            show_car(json)
        else:
            show_empty()


headers = {'Connection': 'Close'}
try:
    reply = requests.delete('http://localhost:8000/cars/8')
    print("res=" + str(reply.status_code))
    reply = requests.get('http://localhost:8000/cars/', headers=headers)
except requests.RequestException:
    print('Communication error')
else:
    if reply.status_code == requests.codes.ok:
        show(reply.json())
    elif reply.status_code == requests.codes.not_found:
        print("Resource not found")
    else:
        print('Server error')
```

```
res=200
id        | brand          | model     | production_year     | convertible
|
1         | Ford           | Mustang   | 1972                | 0
|
2         | Chevrolet      | Camaro    | 1988                | 1
|
3         | Aston Martin   | Rapide    | 2010                | 0
|
4         | Maserati       | Mexico    | 1970                | 0
|
5         | Nissan         | Fairlady  | 1974                | 0
|
6         | Mercedes Benz  | 300SL     | 1957                | 1
|
```

¡Felicitaciones! ¡La actualización fue un éxito total!

Además, hemos completado nuestro viaje por las tierras de CRUD y REST. Fue un viaje largo pero muy satisfactorio.

Creado por:

Isabel Maniega