Hash y APIs

Lo que necesitas saber antes de comenzar esta unidad

Para entender esta unidad es necesario que tengas dominio de los siguientes temas vistos al comienzo del módulo:

- Crear métodos o funciones.
- Alcance de Variables.
- Ciclos y estructuras de control.
- Bloques.
- Arreglos (Arrays) e iteradores.

Ciclos

Existen diversas formas de repetir una o más instrucciones varias veces.

- `while` y `until`
- `for`
- `.times`

Además, para iterar los arreglos (recorrer los elementos uno a la vez) existen métodos incluidos en los mismos arreglos para recorrerse a sí mismos.

- `.each`
- `.map` o `.collect`
- `.select`
- `.reject`
- `.inject`

Estos últimos métodos más el .times van acompañados de bloques para indicar que se debe hacer en cada iteración.

Creando bloques

Existen dos formas de declarar un bloque; entre `do` y `end`

```ruby
10.times do |i|
  puts i
end
```

Y de forma inline con las llaves.

```ruby
10.times { |i| puts i }
```

Alcance de las variables locales

También es importante que recordemos que las variables tienen alcance y si las definimos dentro del bloque solo existirán dentro del bloque.

```ruby
%w[1 2 3 4].each do |letter|
  money_symbol = '$'
  puts "#{money_symbol + letter}"
end

puts money_symbor # NameError: undefined local variable or method `money_symbol'
```

En el caso de los métodos, el alcance de las variables locales tiene límite entre el `def` y `end`:

```ruby
def foo
  bar = 5
end

foo
puts bar # NameError: undefined local variable or method `money_symbol'
```

No ocuparemos variables globales.

Índices de los arreglos

- Los elementos en el array tienen una posición, a esta posición se le llamada índice.
- Los índices van de cero hasta n - 1, donde n es la cantidad de elementos del arreglo.
- Si un array contiene 5 elementos, el primer elemento estará en la posición cero, y el último en la cuarta posición.

```ruby
a = [1, 2, 'hola', 'a', 'todos']
a[0] # => 1
a[1] # => 2
a[4] # => todos
```

Índices fuera de borde

En caso de que el índice sea mayor o igual a la cantidad de elementos obtendremos nil, sin ninguna otra consecuencia.

```
a[8] # => nil
```

Los índices también se pueden utilizar con números negativos y de esta forma referirse a los elementos desde el último al primero.

```
a = [1, 2, 3, 4, 5]
a[-1] # => 5
```

Membresía

Podemos saber si un elemento se encuentra dentro de un array utilizando el método `.include?`

```
a.include? 4 # => true
```

Remover un elemento

El método .delete entrega el elemento removido o nil si no lo encuentra.

```
a = [1, 2, 3, 4, 5, 6, 7]
a.delete(2) # => 2

a = ['Do', 'Re', 'Mi', 'Fa', 'Sol', 'La', 'Si']
a.delete(2) # => nil ¿Por qué?
```

Ejemplos de iteración con índice

El iterador each_with_index entrega además de cada elemento el índice (posición) que ocupa en el array partiendo desde el cero.

```
a = ['Do', 'Re', 'Mi', 'Fa', 'Sol', 'La', 'Si']
a.each_with_index do |value,index|
   puts "#{index + 1})#{value}"
end
```

Ejemplos de iteración con `.each`

```ruby
array = ['Ruby','Javascript', 'Git', 'CSS']
array.each do |technology|
   puts "En el BootCamp aprenderé #{technology}"
end
```

Transformación con `.each`

```ruby
array = [1, 2, 6, 1, 7, 2, 3]
new_array = []
array.each do |ele|
   new_array.push ele + 1
end
print new_array
```

Filtrando con `.each`

```ruby
array = [8, 2, 5.3, 7, 2, 9, 9, 6]
new_array = []
array.each do |ele|
  if ele > 5
    new_array.push(ele)
  end
end
print new_array
```

Creando un nuevo array a partir de otro con `.map`

El método .map entrega un nuevo array con el resultado del bloque aplicado a cada elemento. No modifica el array original.

```ruby
a = [1, 2, 3, 4, 5, 6, 7]

b = a.map do |e|
     e * 2
   end

# O con bloque inline
b = a.map { |e| e * 2 }
```

Filtrando con `.select`

```
a = [1, 2, 3, 4, 5, 6, 7]
b = a.select{ |x| x % 2 == 0} # seleccionamos todos los pares
# => [2,4,6]
```

Filtrando con `.reject`

```
a = [1, 2, 3, 4, 5, 6, 7]
b = a.reject{ |x| x.even? } # lo mismo que { |x| x % 2 == 0}, pero más legible (syntactic sugar)
```

Reduciendo con `.inject`

El método inject entrega un solo elemento concatenando sucesivamente el resultado de cada iteración.

```
a = [1, 2, 3, 4, 5, 6, 7]
b = a.inject(0){ |sum, x| sum += x } # => 28
```

Abrir y leer un archivo

Leer todo un archivo y guardarlo en un string

```
content = File.read 'file.txt' # Si el archivo no es muy grande
```

Transformar datos un archivo

```
data = File.open('data').read.chomp.split(',')
```

Leer un archivo de múltiples líneas

```
data = File.open('archivo2').readlines
```

Guardar los resultados

```
File.write('/path/to/file', 'datos')
```

Capítulo 1: Introducción a hash

Objetivos

- Conocer la utilidad de un hash.
- Crear un hash.
- Conocer los conceptos de llave y valor.
- Acceder a elementos dentro de un hash a través de la clave.
- Iterar hashes

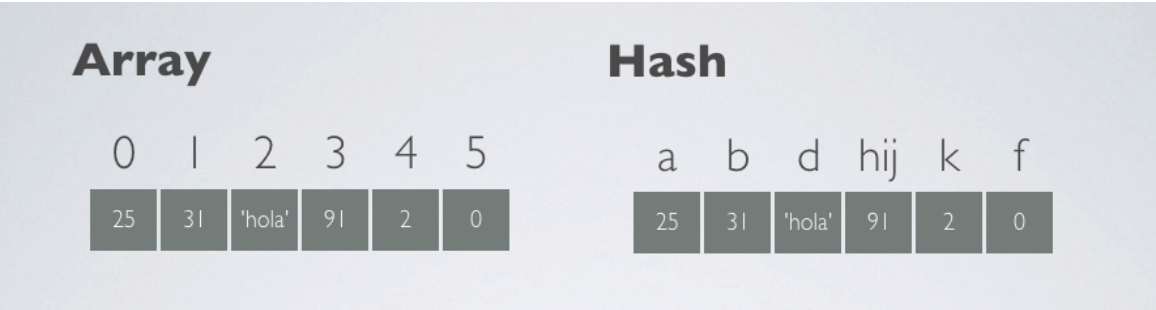
Hash

El hash es un tipo de contenedor similar al array pero nos permite acceder a los elementos a través de una llave, similar al concepto de diccionario, pero sin orden alfabético.

```
hash = {'a' => 30, 'b' => 31}
hash['b'] # => 31
hash['a'] # => 30
```

Array vs Hash

El array y el hash son estructuras similares pero tienen una gran diferencia.



Mientras que un array accedemos a los elementos por la posición que ocupan. En el hash accedemos a los elementos por la **llave** (o clave) que utilizamos al definirlos.

Las llaves no se generan automáticamente, las definimos al momento de agregar un elemento.

Hashes y diccionarios

El hashes también suele recibir el nombre de diccionario. Esto es porque se leen igual que un diccionario de la vida real.

En un diccionario buscamos una palabra y esta nos lleva a la definición. En un hash es la misma idea. Existe una **clave** que vendría siendo el equivalente a la palabra y un **valor** que vendría siendo el equivalente a la definición.

¿Para qué sirven los hashes?

Los hashes al igual que los arreglos son una forma de agrupar datos que nos permite guardar múltiples valores en una variable.

Existen varios tipos de situaciones donde los hashes son mejores que los arreglos para resolver algún determinado problema. En esta unidad resolveremos algunos problemas utilizando ambas estrategias para compararlas.

Crear un hash

Un hash se define con llaves `{}`, y cada par de clave y valor se asocia mediante una flecha, o sea juntar el igual con el mayor `=>`.

```
alumnos = {'Gonzalo' => 20, 'Juan Pablo' => 31, 'María José' => 29}
```

`=>`

Al operador `=>` se le conoce como hash rocket o simplemente como rocket. Sirve para especificar el valor asociado a una clave al momento de crear un hash.

```
{'Alumno1' => 80}
```

Alumno1 sería la clave, 80 el valor.

Buena práctica

Si el hash es grande lo escribiremos en múltiples líneas

```
alumnos = {'Gonzalo' => 20,
           'Juan Pablo' => 31,
           'María José' => 29,
           'David' => 32,
           'Cristian' => 33
          }
```

El último elemento del hash no lleva coma al final.

Cualquier objeto puede ser una clave o un valor

```
a = {nil => 1, 1001 => 2, 21.5 => 3, true => 4, [1,2,3] => 5}
```

Acceder a un elemento dentro de un hash

Podemos acceder a un elemento específico utilizando la clave. Tiene que ser exactamente la misma clave ingresada.

```
alumnos = {'Gonzalo' => 20}
alumnos['Gonzalo'] # 20
alumnos['gonzalo'] # nil
```

Si cambiamos algún detalle en la clave, Ruby lo entiende como una clave distinta y accederemos al valor correspondiente.

La clave tiene que ser única

En un diccionario solo puede haber un valor asociado a una clave.

```
alumnos = {'David' => 32, 'David' => 32}
#(irb):6: warning: key :David is duplicated and overwritten on line 6
```

Si hubiesen dos claves iguales Ruby no podría saber que valor estamos intentando rescatar.

Agregando un elemento a un hash

Si tenemos un hash definido como:

```
a = {'clave1' => 5}
```

Podemos agregar un elemento utilizando una clave nueva.

```
a["clave2"] = 9
puts a # {'clave1'=>5, "clave2"=>9}
```

Cuidado con la sintaxis, al momento de definir ocuparemos las llaves `{}` pero al acceder, o agregar elementos nuevos ocuparemos los corchetes `[]`

Cambiando un valor dentro de un hash

Podemos cambiar un elemento dentro de un hash utilizando la misma sintaxis pero utilizando una clave existente.

```
a = {'clave1'=>5, 'clave2'=>7}
a['clave2'] = 9
puts a # {'llave1'=>5, 'llave2'=>9}
```

Capítulo: Claves y Símbolos

Objetivos

- Conocer los símbolos en Ruby.
- Diferenciar una clave Símbolo con una clave String.
- Diferenciar la notación con : y con `=>.

Introducción

```
:hola_símbolo
```

Los símbolos son tipos de datos similares a los strings, pero están optimizados para ocupar menos memoria y son inmutables. Si en un programa tenemos declarado varias veces un string "Hola", cada vez será un objeto diferente (distinto espacio de memoria), pero si tenemos varias veces declarado el simbolo :hola, siempre se está referenciando al mismo objeto. En definitiva, son más eficientes en terminos de rendimiento computacional, pero son más limitados en sus métodos.

Los símbolos son muy utilizados como claves en los hashes.

Creando un símbolo

Los símbolos empiezan con ":"

```
a = :hola
puts a
```

¿Para qué sirven los símbolos?

Los símbolo son similares a los strings, pero son más simples. Soportan menos operaciones y sirven para representar estados o valores que no pueden cambiar como las llaves en los hashes.

```
a = {}
a[:foo] = 'foo'
a[:bar] = 'bar'
a[:foo] == 'foo' # => true
a[:bar] == 'bar' # => true
```

Los símbolos son más rápidos

Pero también tienen menos funcionalidades, o sea tienen menos métodos que los strings. Esto lo podemos probar ocupando el método `.methods`

```
print "prueba".methods - :prueba.methods
```

Los strings son mutables, los símbolos no

Los strings al ser mutables (poder cambiar de estado) son malos para representar cosas que no pueden cambiar. Para este tipo de situaciones son mejores los símbolos.

Por ejemplo supongamos que tenemos un semáforo que puede tener tres estados, y estos pueden ser `"rojo"`, `"amarillo"` o `"verde"`, en ese caso conviene utilizar símbolos y simplemente guardar el estado como símbolo, ejemplo `semáforo = :amarillo`

Arreglos y símbolos

Si el arreglo almacena posibles estados entonces también podemos aplicar la misma lógica.

```
estados_semaforo = [:verde, :amarillo, :rojo]
semaforo1 = estados_semaforo[0]
```

Hashes y símbolos

Los símbolos suelen ocuparse como clave de los hashes. Pero debemos ocupar un símbolo para acceder al elemento.

```
a = {:llave1 => 5, :llave2 => 10}
a[:llave1] # => 5
a['llave1'] # => nil
a[:'llave1'] # => 5
```

Hashes con símbolos y strings

Un hash puede tener simultaneamente claves que sean string y símbolos.

```
notas_alumnos = {'Camila' => 6, :David => 5}
```

Para acceder al respectivo valor tenemos que ocupar la clave correcto

```
notas_alumnos['Camila'] # => 6
notas_alumnos[:Camila] # => nil
notas_alumnos['David'] # => nil
notas_alumnos[:David] # => 5
```

Dos notaciones al momento de crear

`:` V.S. `=>`

Hasta al momento al crear un hash hemos agregado los pares de clave valor utilizando el operador rocket hash (o sea de la siguiente forma `clave => valor`). Existe otra forma de hacerlo, y es utilizando la notación `clave: valor`

Asociando elementos con `:`

La notación `'clave': valor` es muy interesante, convierte automáticamente un string en un símbolo. Creemos un diccionario utilizando ambas notaciones simultáneamente para compararlas.

```
a = {'forma1' => 5, 'forma2': 10}
```

Al utilizar esta forma vemos en el output ambos valores no se procesaron de la misma forma. Mientras que la clave `forma1` se mantuvo como string, la otra clave fue transformada en un símbolo de forma automática.

La notación con `:` es relativamente nueva

Esta forma de agregar elementos fue introducida en `Ruby 2.0`, Se aconseja utilizarla a menos que por algún motivo en especial necesitemos que la clave sea un string en lugar de un símbolo.

Otra ventaja de la notación de `:`

Las símbolos no tienen comillas, por lo mismo al ingresar claves lo podemos hacer sin ellas.

```
hash = { clave1: 5, clave2: 10, clave3: 15 }
```

Recordar esto será importante porque lo ocuparemos para muchos métodos que reciben hash.

Simple es mejor

Al momento de escoger alguna notación (si es que tienen el mismo resultado) debemos recordar que la forma más simple es mejor. Es es la filosofía que hace a Ruby un lenguaje fácil de leer.

Clave símbolo + valor símbolo

En la mayoría de los casos las claves serán símbolos, y en algunos casos los valores también serán símbolos.

```
hash = {clave1: :valor1, clave2: :valor2, clave3: :valor3}
```

Crearemos y veremos hashes de esta forma muy frecuentemente.

Haciendo memoria

Ya hemos ocupado esta notación, cuando abrimos un archivo CSV. Veamos el código:

```
CSV.open('data.csv', converters: :numeric)
```

Para Ruby esto es lo mismo que:

```
CSV.open('data.csv', {converters: :numeric})
```

Visitaremos nuevamente esta notación cuando veamos métodos que reciben hashes como argumentos

Reglas de sintaxis de un símbolo

Una de las reglas de símbolos es que no pueden ser un número o empezar con un número.

```
:hola2 # Válido
:2 # Inválido
:2hola # Inválido
```

Precaución con las claves que son números

Por la misma regla tenemos que tener cuidado cuando utilicemos la notación que convierte un string en un número.

```
colores = {'1' => 'verde', 2: 'amarillo' } #Error
```

El comportamiento también es raro cuando ocupamos un string que contiene solo un número

```
colores = {'1' => 'verde', '2': 'amarillo' }
=> #{"1"=>"verde", :"2"=>"amarillo"}
colores[:"2"] # => amarillo
```

Por lo mismo evitaremos ocupar números como clave.

Capítulo: Iterando hashes

Objetivos:

- Iterar un hash con una sola variable.
- Iterar un hash con dos variables.
- Transformar los valores dentro de un hash.
- Transformar los valores y generar un hash nuevo.

Iterando un hash

Los hash tienen clave y valor, si iteramos la variable esta será un arreglo con el par clave valor.

```
a = {clave1: '1', clave2: '2', clave3: '3'}

a.each do |i|
  puts i
  puts i.class
end
```

```
[:clave1, "1"]
Array
[:clave2, "2"]
Array
[:clave3, "3"]
Array
```

```
{:clave1=>"1", :clave2=>"2", :clave3=>"3"}
```

La otra forma de iterar un hash es ocupando dos variables, una para la clave y la otra para el valor

```
a = {clave1: '1', clave2: '2', clave3: '3'}

a.each do |k, v|
  puts "La clave es #{k} y el valor es #{v}"
end
```

Esta es la forma más comun de iterar un hash. Al iterar un diccionario se suele ocupar las variables k de **key** y v de **value**

Ejercicio de iteración de hash

A partir de un hash con un listado de países y la cantidad de usuarios por países.

```
paises = {
   Mexico: 70,
   Chile: 50,
   Argentina: 55
}
```

Se pide:

- Mostrar solo los países
- Mostrar solo los valores
- Mostrar solo los valores mayores a 55

```
# Mostrar solo los países

paises = {
   Mexico: 70,
   Chile: 50,
   Argentina: 55
}

paises.each do |k,v|
   puts k
end
```

```
# Mostrar solo los valores
paises.each do |k,v|
   puts v
end
```

```
# Mostrar solo los valores mayores a 55

paises.each do |k,v|
   puts v if v > 55
end
```

Ejercicio de transformación de los valores de un hash

Dado un hash con las ventas de los últimos 3 meses

```
ventas = {
   Octubre: 65000,
   Noviembre: 68000,
   Diciembre: 72000
}
```

Se pide:

- Modificar el hash para incrementar las ventas un 10%
- Generar un nuevo hash pero con las ventas disminuidas un 20%

```
ventas = {
   Octubre: 65000,
   Noviembre: 68000,
   Diciembre: 72000
}

ventas.each do |k,v|
   ventas[k] = v * 1.1
end
```

```
ventas = {
   Octubre: 65000,
   Noviembre: 68000,
   Diciembre: 72000
}

nuevas_ventas = {}
ventas.each do |k,v|
   nuevas_ventas[k] = v * 0.8
end

print nuevas_ventas
```

Desafío: Filtrando un hash

Dado un hash crear un método que devuelva otro hash pero filtrando todos aquellos que tienen valores inferior a 70000

Probar con el siguiente hash

```
ventas = {
   Octubre: 65000,
   Noviembre: 68000,
   Diciembre: 72000
}
```

Solución

```ruby
# Nos piden un método que reciba un hash
# y que devuelva un hash nuevo con los datos que cumplen una condición.

def filter(hash)
  filtered_hash = {}
end
```

```ruby
# Ahora iteramos y filtramos dentro del método

def filter(hash)
  filtered_hash = {}
  hash.each do |k,v|
    if v < 70000
      filtered_hash[k] = v
    end
  end
  filtered_hash # Recordemos que la última línea es el retorno
end

filter({
    Octubre: 65000,
    Noviembre: 68000,
    Diciembre: 72000
})
```

Así como existe un método para filtrar en los arreglos, existe uno para los hashes que lo estudiaremos mas adelante.

Capítulo: Operaciones típicas en un hash

Objetivos

- Eliminar un elemento dentro de un hash.
- Unir dos hashes en uno solo.
- Invertir un hash.
- Transformar las claves del hash en un arreglo.
- Transformar los valores del hash en un arreglo.

Introducción:

En este capítulo aprenderemos varias funcionalidades de los hashes que nos harán la vida más fácil. Todos los métodos que aprenderemos los podemos consultar desde la documentación.

Eliminar un elemento dentro de un hash.

Podemos eliminar una llave con su valor del hash ocupando el método `.delete`

```
a = {k1: 5, k2: 7}
a.delete(:k1)
print a
```

- Al igual que cuando accedemos al valor cuando borremos la clave debemos ser consientes de si es un símbolo o un string.
- El método delete además devuelve el valor borrado en caso de que necesitemos trabajar con él.

Uniendo hashes

Otra operación muy común es unir dos hashes, para eso ocuparemos el método merge.

```
a = {k1: 5, k2: 10}
b = {k3: 15, k4: 20}
a.merge(b)
```

Colisiones

Cuando dos hashes tienen la misma llave el segundo sobreescribe al primero

```
a = {k1: 5, k2: 10}
b = {k2: 15, k3: 20}
a.merge(b)
```

Las operaciones `a.merge(b)` y `b.merge(a)` entregan resultados distintos

Invirtiendo un hash

En algunas ocasiones puede ser útil invertir un hash para obtener una llave a partir de un valor

Por ejemplo supongamos que tenemos un hash con colores y sus valores hexadecimales Y queremos obtener el color a partir del código.

```
colors = {'red' => '#cc0000', 'green' => '#00cc000', 'blue' => '#0000cc' }
colors.invert['#cc0000'] #red
```

Obtener todas las claves de un hash

Lo podemos hacer de forma muy sencilla con `.keys`

```
colors = {'red' => '#cc0000', 'green' => '#00cc000', 'blue' => '#0000cc' }
colors.keys
```

Obtener todas los valores de un hash

Existe un método específico para esto llamado `.values`

```
colors = {'red' => '#cc0000', 'green' => '#00cc000', 'blue' => '#0000cc' }
colors.values
```

```
ventas = {
  zona_norte: [1000,500,300]
  zona_centro: [500, 500, 400]
}
```

Reconstruir un hash a partir de los keys y values

Existe un método llamado `.zip` que permite unir dos arreglos.

```
[1,2,3].zip([4,5,6])
```

Ejercicio práctico

Se tiene una base de datos de colores


```
{
  "aliceblue": "#f0f8ff",
  "antiquewhite": "#faebd7",
  "aqua": "#00ffff",
  "aquamarine": "#7fffd4",
  "azure": "#f0ffff",
  "darkorchid": "#9932cc",
  "darkred": "#8b0000",
  "darksalmon": "#e9967a",
  "navajowhite": "#ffdead",
  "navy": "#000080",
  "orchid": "#da70d6",
  "palegoldenrod": "#eee8aa",
  "peachpuff": "#ffdab9",
  "peru": "#cd853f",
  "pink": "#ffc0cb",
  "purple": "#800080",
  "rebeccapurple": "#663399",
  "red": "#ff0000",
  "saddlebrown": "#8b4513",
  "seashell": "#fff5ee",
  "sienna": "#a0522d",
  "silver": "#c0c0c0",
  "skyblue": "#87ceeb",
  "slateblue": "#6a5acd",
  "teal": "#008080",
  "thistle": "#d8bfd8",
  "tomato": "#ff6347",
  "turquoise": "#40e0d0",
  "violet": "#ee82ee",
  "wheat": "#f5deb3",
  "white": "#ffffff",
  "whitesmoke": "#f5f5f5",
  "yellow": "#ffff00",
  "yellowgreen": "#9acd32"
}
```


Se pide crear un programa llamado busqueda_colores.rb donde el usuario ingrese un color en hexadecimal y le muestra en pantalla el nombre del color, en caso de no encontrarlo aparecerá el texto no-no

Uso: busqueda_colores.rb #6a5acd #40e0d0

> slateblue

Solución iterando

Tenemos dos soluciones posibles, una sería iterar el arreglo hasta encontrar la clave, la otra sería invertir el arreglo como aprendimos y aprovechar las bondades del hash.

```
colors = {
  "aliceblue": "#f0f8ff",
  "antiquewhite": "#faebd7",
  "aqua": "#00ffff",
  "aquamarine": "#7fffd4",
  "azure": "#f0ffff",
  "darkorchid": "#9932cc",
  "darkred": "#8b0000",
  "darksalmon": "#e9967a",
  "navajowhite": "#ffdead",
  "navy": "#000080",
  "orchid": "#da70d6",
  "palegoldenrod": "#eee8aa",
  "peachpuff": "#ffdab9",
  "peru": "#cd853f",
  "pink": "#ffc0cb",
  "purple": "#800080",
  "rebeccapurple": "#663399",
  "red": "#ff0000",
  "saddlebrown": "#8b4513",
  "seashell": "#fff5ee",
  "sienna": "#a0522d",
  "silver": "#c0c0c0",
  "skyblue": "#87ceeb",
  "slateblue": "#6a5acd",
  "teal": "#008080",
  "thistle": "#d8bfd8",
  "tomato": "#ff6347",
  "turquoise": "#40e0d0",
  "violet": "#ee82ee",
  "wheat": "#f5deb3",
  "white": "#ffffff",
  "whitesmoke": "#f5f5f5",
  "yellow": "#ffff00",
  "yellowgreen": "#9acd32"
}

#search = ARGV[0]

search = '#6a5acd'
colors.each do |k,v|
  if v == search
    puts k
  end
end
```

Si bien esta solución muestra en pantalla el valor correcto no maneja la situación de cuando no lo encuentra. Podríamos estar tentados a poner esto:

```
#search = ARGV[0]
search = '#6a5acd'
colors.each do |k,v|
  if v == search
    puts k
  else
    puts "no-no"
  end
end
```

El problema de este acercamiento es que vamos a mostrar muchas veces el texto de no encontrado y solo debemos mostrarlo una vez. Para evitar esto ocuparemos una variable para guardar cuando encontremos el valor y preguntaremos por ella al final.

```
search = '#6a5acd'
founded = false
colors.each do |k,v|
  if v == search
    founded = true
    puts k
  end
end

puts "no-no" unless founded
```

Solución invirtiendo el arreglo

```
match = colors.invert[search]
if match
  puts match
else
  puts "no-no"
end
```

Solución elegante

```
match = colors.invert[search]
puts match ? match : "no-no"
```

Discutamos las soluciones

¿Por qué estudiamos varias soluciones a un problema si con una basta?

Esta pregunta es muy importante, un programador no busca memorizar la respuesta a los problemas, de hecho muy rara vez nos toca resolver exactamente el mismo problemas dos veces, pero el acercamiento a resolver un problema si puede ser reutilizado en situaciones similares.

Lo importante es conocer diversas estrategias que nos permitan después enfrentar problemas nuevos.

En esta ocasión aprendimos que invertir un hash puede ser mucho más fácil que iterarlo.

Ejercicio

Modificar el programa anterior para que el usuario pueda buscar múltiples colores.

Uso: busqueda_colores.rb #6a5acd #9acd32 #ffff00

> slateblue yellowgreen yellow

Tip: El usuario puede ingresar cuantos colores desee.

Discutamos primero la forma incorrecta de resolverlo

```
search1 = ARGV[0]
search2 = ARGV[1]
search3 = ARGV[2]

match = colors.invert[search1]
puts match ? match : "no-no"

match = colors.invert[search2]
puts match ? match : "no-no"

match = colors.invert[search3]
puts match ? match : "no-no"
```

Es una pésima solución porque si el usuario ingresa mas valores no funcionará. Otra forma de darse cuenta de que es mal código es porque estamos repitiendo el mismo código varias veces.

```
## Solución buena

# Simplemente tenemos que iterar el arreglo.
# ARGV = ['#6a5acd', '#6a5acd']

ARGV.each do |search|
  match = colors.invert[search]
  puts match ? match : "no-no"
end
```

Capítulo: Arreglos y hashes

Objetivos

- Convertir un hash en un arreglo.
- Convertir un arreglo en un hash.
- Transformar dos arreglos relacionados en un único arreglo.

Introducción

Los arreglos y los hashes tienen distintas ventajas, en algunos casos será más convenientes trabajar con hashes y en otras ocasiones será más sencillo trabajar con los arreglos.

Es importante aprender a diferenciar las situaciones ante las cuales nos enfrentamos para identificar la solución más simple.

Veamos un ejemplo

Supongamos que queremos almacenar alumnos y sus notas, perfectamente podríamos guardar la información en dos arrays, uno con los nombres y otro con las notas.

```
nombres = ['Alumno1', 'Alumno2', 'Alumno3']
notas = [10, 3, 8]
```

En este problema nos piden que nuestro programa haga dos cosas:

- Mostrar los alumnos con sus notas correspondientes.
- Mostrar la nota de un alumno dado su nombre.

Resolviendo el problema con arreglos

De esta forma si quisiéramos consultar la nota de un alumno podríamos recuperar la posición del alumno en el arreglo nombres y luego consultar el índice en notas.

```
index = nombres.index('Alumno1')
nota = notas[index]
```

Resolviendo el problema con hashes

Mientras que resolver el mismo problema con un hash puede ser mucho más sencillo.

```
products = {'product1' => 100,
            'product2' => 150,
            'product3' => 210}
puts products['product2']
```

Otro motivo importante

Algunos métodos reciben hashes y otros métodos reciben arreglos, tenemos que saber transformar los datos de un tipo a otro.

A lo largo del programa no siempre tendremos control sobre cuál es la estructura que estamos trabajando, por ejemplo puede ser que el resultado de un método devuelva un hash pero nosotros necesitemos un arreglo como entrada para otro método.

Convertir un hash en un arreglo.

Es muy simple convertir un hash en un arreglo, simplemente tenemos que llamar al método `.to_a`.

```
{k1: 5, k2: 7}.to_a # => [[:k1, 5], [:k2, 7]]
```

Convirtiendo un arreglo en un hash

Podemos invertir la transformación ocupando el método `.to_h`

```
[[:k1, 5], [:k2, 7]].to_h
```

```
{:k1=>5, :k2=>7}
```

Cruzando información de dos arreglos

```
nombres = ['Alumno1', 'Alumno2', 'Alumno3']
notas = [10, 3, 8]
```

Si tenemos la información en dos arreglos y queremos guardarla en un hash tenemos dos opciones.

- Iterar el arreglo con un índice
- Utilizar el método `.zip`

Iterando el arreglo con índice

```
nombres = ['Alumno1', 'Alumno2', 'Alumno3']
notas = [10, 3, 8]

hash = {}
nombres.count.times do |i|
   element = nombres[i]
   nota = notas[i]
   hash[element] = nota
end

print hash
```

Iterando el arreglo elemento a elemento

Iterar el arreglo elemento a elemento también es posible, pero un poco mas complejo.

```
nombres = ['Alumno1', 'Alumno2', 'Alumno3']
notas = [10, 3, 8]

hash = {}
nombres.each do |nombre|
   i = nombres.index(nombre) # obtenemos el índice
   nota = notas[i]
   hash[nombre] = nota
end

print hash
```

Transformando con `.zip`

```
nombres = ['Alumno1', 'Alumno2', 'Alumno3']
notas = [10, 3, 8]
nombres.zip(notas).to_h # {"Alumno1"=>10, "Alumno2"=>3, "Alumno3"=>8}
```

Group by

`group_by` es un método de los arreglos que nos permite agrupar un conjunto de elementos bajo cualquier criterio, veamos algunos ejemplos. Este método entrega un hash con el resultado. Cada key del hash devuelto corresponde a un grupo y los valores son arrays con los elementos pertenecientes a cada grupo.

```
[1, 2, 3, 4, 5, 6, 7, 8] .group_by{ |x| x.even? }
```

```ruby
[1, 2, 3, 4, 5, 6, 7, 8] .group_by{ |x| x > 4}
```

```ruby
["hola", "a", "todos"].group_by { |x| x.length  } # Agrupar por el largo
```

```ruby
[1, 2, 3, 4, 1, 2, 1, 5, 8].group_by { |x| x } # Agrupar por el mismo elemento
```

```ruby
['a', 'ab', 'abc', 'b', 'bc', 'bcd'].group_by { |x| x[0] }  # Por la primera letra
```

```ruby
['a', 1, 'b', 2, 'c'].group_by { |x| x.class } # Por tipo
```

`.group_by` es un método muy flexible y sirve para resolver diversos tipos de problemas.

Contando elementos dentro de un hash

Un uso relativamente frecuente de un hash es para contar elementos. Por ejemplo supongamos que tenemos un array con elementos y queremos contar la cantidad de veces que aparece cada uno de los elementos.

```ruby
[1, 2, 6, 7, 2, 5, 8, 9, 1, 3, 6, 7]
```

Resolveremos el problema con dos estrategias distintas:

- Iterar y contar
- Agrupar y contar

Estrategia 1: Iterar y contar

Para resolver el problema iteraremos el arreglo, cada vez que encontremos un elemento nuevo lo guardaremos en un hash con la cuenta en uno, y si encontramos un elemento que ya está dentro del hash lo incrementamos en uno.

```ruby
array = [1, 2, 6, 7, 2, 5, 8, 9, 1, 3, 6, 7]
hash = {}
array.each do |i|
  if hash[i]
    hash[i] += 1
  else
    hash[i] = 1
  end
end
puts hash
```

Estrategia 2 `.group_by`

```
# Los agrupamos
grouped = [1, 2, 6, 7, 2, 5, 8, 9, 1, 3, 6, 7] .group_by {|x| x}
# => {1=>[1, 1], 2=>[2, 2], 6=>[6, 6], 7=>[7, 7], 5=>[5], 8=>[8], 9=>[9], 3=>[3]}

#Luego remplazamos el valor por la cuenta
grouped.each do |k,v|
  grouped[k] = v.count
end
```

Capítulo: Introducción a APIs

Objetivos

- Conocer el concepto de **request** y **response**.
- Conocer la importancia de las API Rest para comunicar programas por internet.
- Utilizar Postman para realizar un request a una API.

Introducción a API

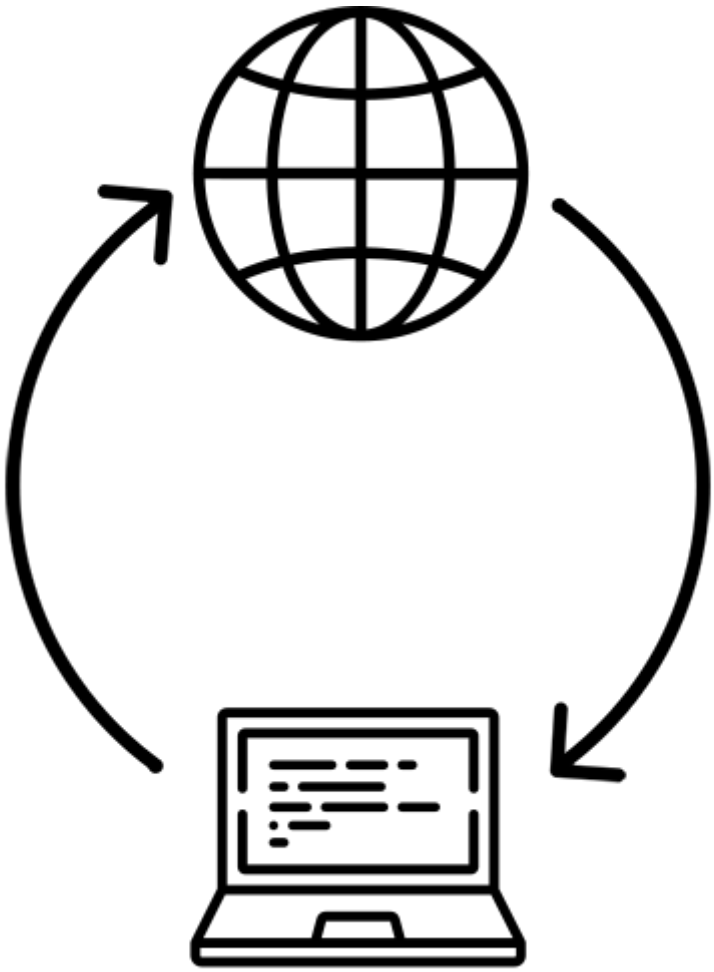
API, acrónimo de **A**pplication **P**rogram **I**nterface, es una interfaz de acceso que nos permite comunicar programas.

Existen diversos tipos de APIs, Particularmente hablaremos de una en especial llamada **API REST** y cuando utilicemos el concepto de API de ahora en adelante nos estamos refiriendo a una API REST.

API Rest

En términos muy simplificados una API REST es un programa para otro programa que sigue una serie de restricciones para estandarizar la iterconección (interfáz).

Es decir, es un programa fácil de ser consultado desde cualquier programa. Esto es muy útil para integrar sistemas y comunicar distintas aplicacaciones.



Al programa que consulta se le denomina **cliente**. Al programa que entrega la respuesta se le suele llamar **servidor**.

Ejemplos de APIs existentes

Existen miles de APIs para distintos propósitos.

- Clima y temperatura.
- Cambio de monedas.
- Indicadores económicos.
- Servicios para subir archivos.
- Compra y venta de criptomonedas.
- Servicios de geolocalización como googlemaps.
- Y muchas más

Cualquier persona puede construir una API y disponibilizarla a través de internet.

¿Cómo se usa una API?

Muy sencillo, a través de un cliente hacemos un **request** (solicitud) a una dirección y obtendremos como resultado un **response** (respuesta).

Un request (solicitud) es información que nosotros enviamos a un servidor. Para ser mas precisos desde ahora en adelante diremos un request a una **URL**(Uniform Resource Locator) y Response a la información que nos devuelve el servidor o servicio.

En este capítulo aprenderemos a hacer el pedido y analizar la respuesta.

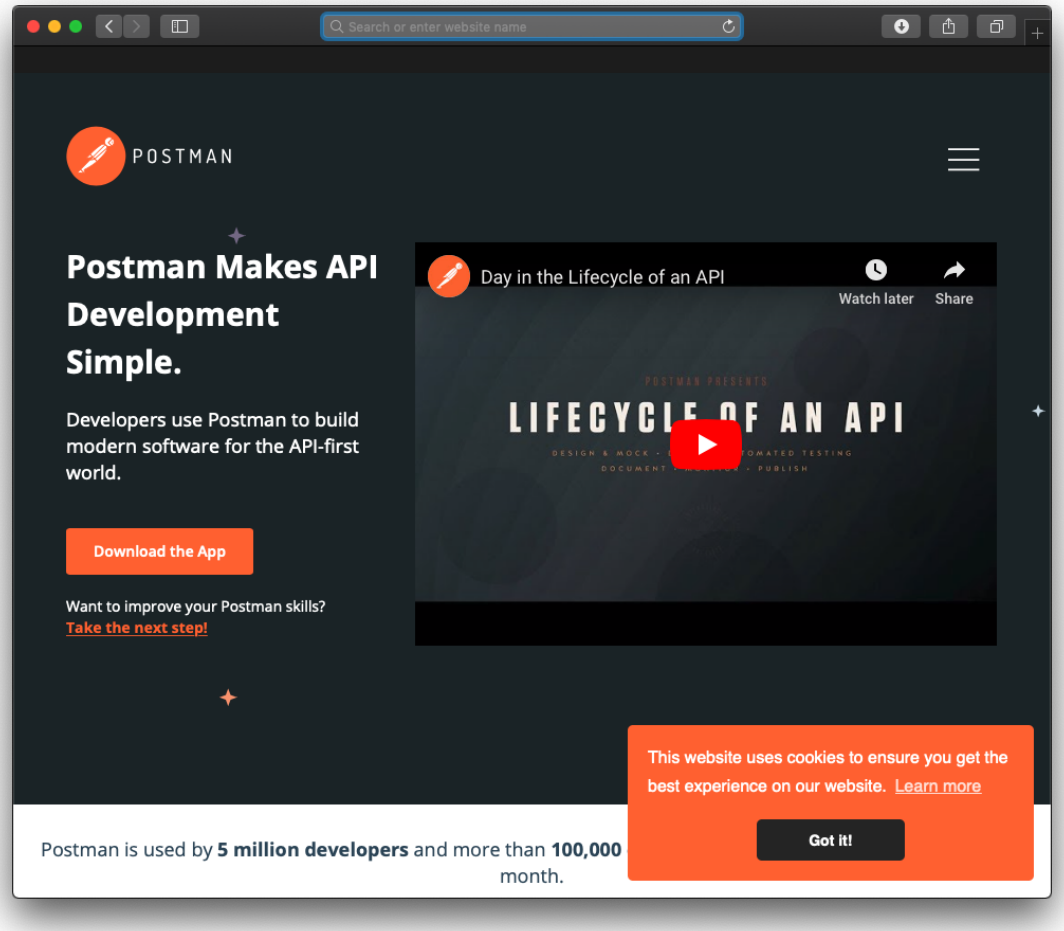
Otro concepto frecuente es el de Endpoint, que es lo mismo que una URL a un recurso. En otras palabras, son los terminales que expone una API.

Probando una API

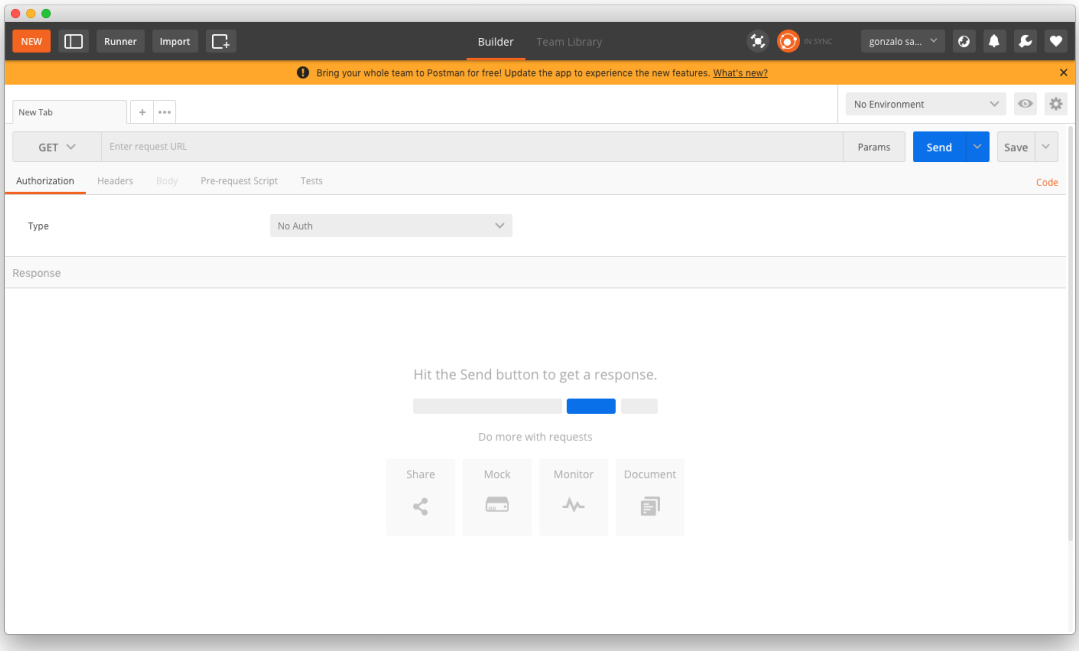
Existe una herramienta muy potente para probar APIs sin necesidad de programar y nos ayudará a comprender la idea. Esta Herramienta se llama **Postman**

Descargando Postman

Podemos descargar Postman desde la página oficial. https://www.getpostman.com



Instalaremos y abriremos la aplicación descargada, y deberíamos ver un panel como el siguiente.

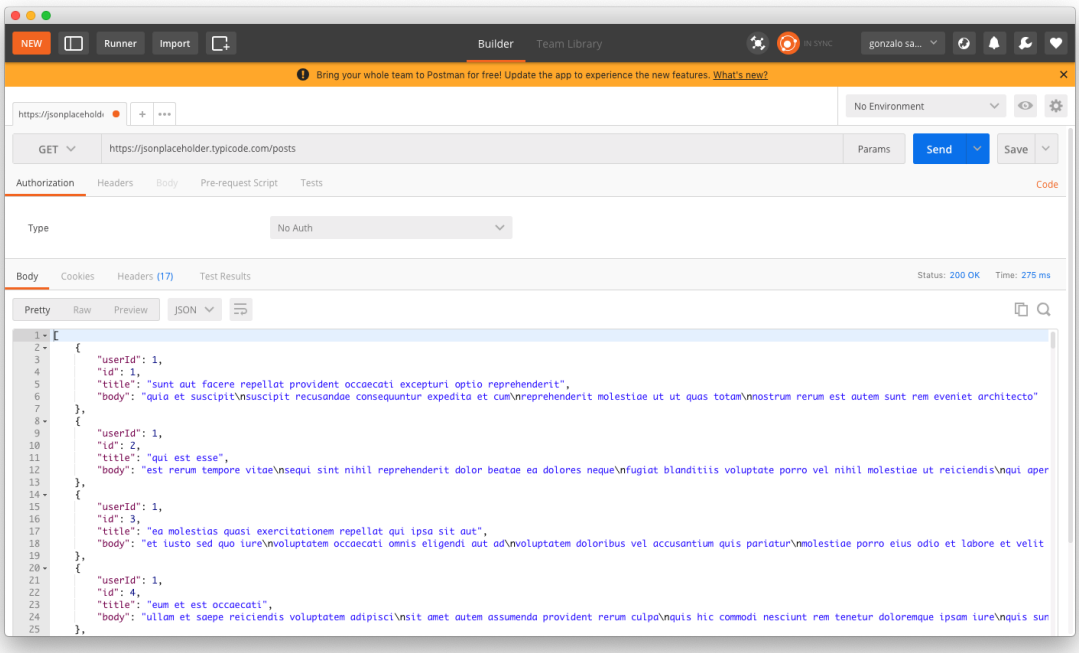


Nuestra primera consulta a una API

Nuestra primera consulta será a una API llamada jsonplaceholder `https://jsonplaceholder.typicode.com/`. La cual es una API falsa en el sentido de que no tiene información real, pero es una API en todo otro sentido.

Primer request

Donde dice Request URL pondremos la siguiente URL `https://jsonplaceholder.typicode.com/posts` Luego haremos clic donde dice **send** y obtendremos nuestra respuesta.



Luego de algunos milisegundos (o segundos en un mal caso) el servidor nos enviará una respuesta. Ese es el texto que aparece en la parte inferior de la imagen.

La respuesta se parece mucho a estructuras que ya conovemos. Primero vemos la apertura de un arreglo y luego varios diccionarios. La respuesta está en un formato llamado **JSON**.

JSON

JSON es acrónimo de **J**avascript **O**bject **No**tation. Es un formato para enviar información en texto plano, fácilmente legíble por humanos y fácilmente analizable por lenguajes de programación. Hoy en día es uno de los formato más utilizado para enviar información entre sistemas.

JSON no requiere de JavaScript

Que el acrónimo tenga la palabra Javascript no quiere decir que necesitemos saber javascript o utiliazar javascript. JSON es un formato plano que podemos utilizar desde cualquier lenguaje.

JSON a Ruby

En Ruby transformar un string con un JSON a un hash es tan sencillo como:

```
JSON.parse(string)
```

```
Lo estudiaremos con detalle en el siguiente capítulo
```

Capítulo: Consumiendo una API desde Ruby

Introducción

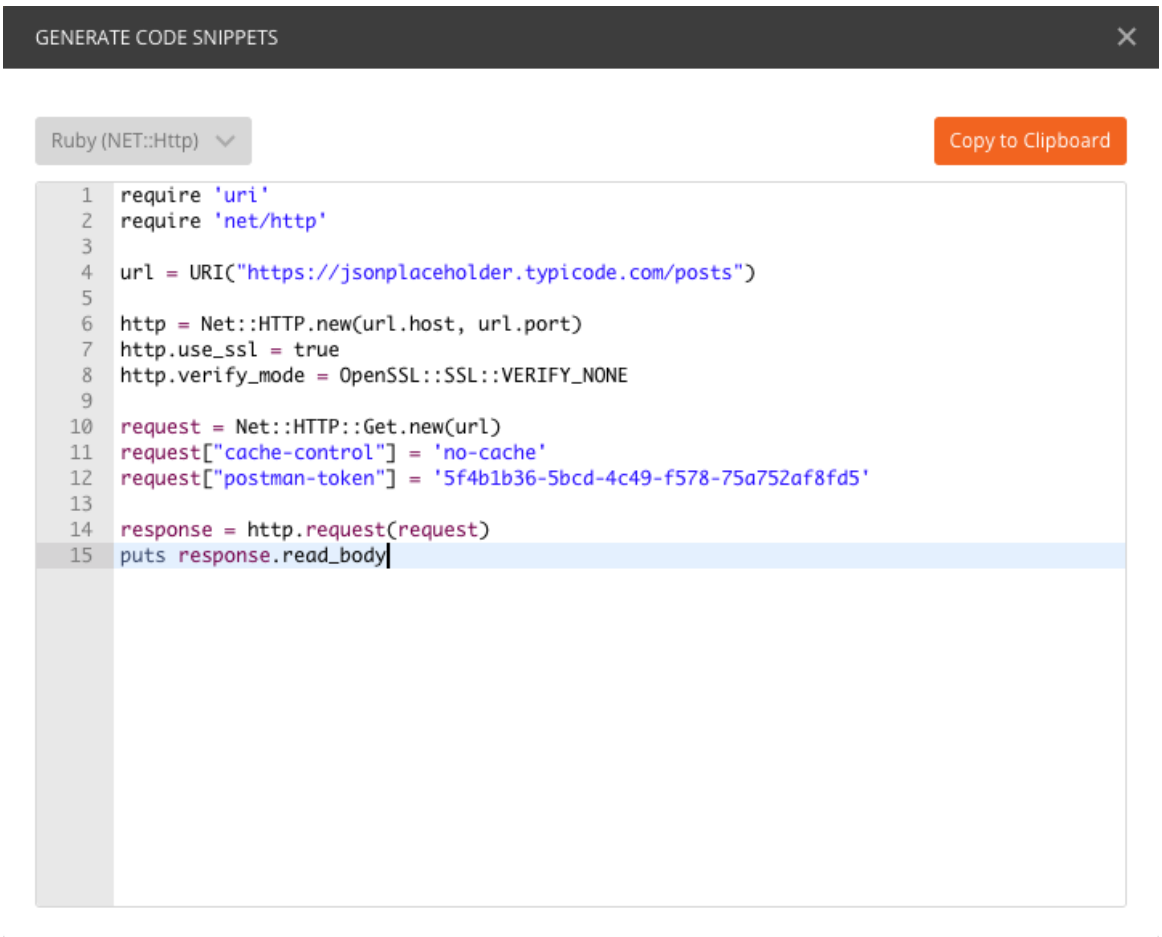
En el capítulo anterior aprendimos a realizar un request a una URL utilizando Postman, en este capítulo aprenderemos a hacer el request directamente desde Ruby.

Objetivos

- Utilizar Postman para obtener el código Ruby necesario para hacer un request.
- Transformar la respuesta de JSON a una estructura utilizable directamente en Ruby.

Obteniendo el código desde Postman

Postman nos entrega automáticamente el código para hacer un request a la API. Para esto haremos clic en el link donde dice **Code**



Este código podemos pegarlo directamente en un script y probarlo.

```ruby
require 'uri'
require 'net/http'

url = URI("https://jsonplaceholder.typicode.com/posts")

http = Net::HTTP.new(url.host, url.port)
http.use_ssl = true
http.verify_mode = OpenSSL::SSL::VERIFY_NONE

request = Net::HTTP::Get.new(url)
request["cache-control"] = 'no-cache'
request["postman-token"] = '5f4b1b36-5bcd-4c49-f578-75a752af8fd5'

response = http.request(request)
puts response.read_body[0..700] # Cortamos el texto para no imprimir las respuesta completa
```

Analizando la respuesta

El método `.request` devuelve un objeto que contiene dos elementos claves, el código de la respuesta y el cuerpo **body**.

El código de la respuesta

En caso de haber obtenido exitosamente la respuesta el código será 200. Existen varios códigos, no es necesario saberlos de memoria, podemos encontrarlos de forma muy rápida en internet.

Algunos de los códigos de respuesta mas relevantes son:

- 200: Ok
- 401: Unauthorized
- 403: Forbidden
- 404: Not Found
- 500: Server error

El contenido de la respuesta

La otra parte de la respuesta es el cuerpo, o **body**. El cual contiene la información que nos interesa. Esta información viene como un string, la cual hace muy complicado extraer información, por ejemplo si solo queremos saber un detalle my específico de la respuesta, o seleccionar elementos específicos de ellas.

"[\n {\n   \"userId\": 1,\n   \"id\": 1,\n   \"title\": \"sunt aut facere repellat provident occaecati excepturi optio reprehenderit\",\n   \"body\": \"quia et suscipit\\nsuscipit recusandae consequuntur expedita et cum\\nreprehenderit molestiae ut ut quas totam\\nnostrum rerum est autem sunt rem eveniet architecto\"\n },\n {\n   \"userId\": 1,\n   \"id\": 2,\n   \"title\": \"qui est esse\",\n   \"body\": \"est rerum tempore vitae\\nsequi sint nihil reprehenderit dolor beatae ea dolores neque\\nfugiat blanditiis voluptate porro vel nihil molestiae ut reiciendis\\nqui aperiam non debitis possimus qui neque nisi nulla\"\n },\n {\n   \"userId\": 1,\n   \"id\": 3,\n   \"title\": \"ea molestias quasi exercitationem repellat qui ipsa sit aut\",\n   \"body\": \"et iusto sed quo iure\\nvoluptatem occaecati omnis eligendi aut ad\\nvoluptatem doloribus vel accusantium quis pariatur\\nmolestiae porro eius odio et labore et velit aut\"\n },\n {\n   \"userId\": 1,\n   \"id\": 4,\n   \"title\": \"eum et est occaecati\",\n   \"body\": \"ullam et saepe reiciendis voluptatem adipisci\\nsit amet autem assumenda provident rerum culpa\\nquis hic commodi nesciunt rem tenetur doloremque ipsam iure\\nquis sunt voluptatem rerum illo velit\"\n },\n {\n   \"userId\": 1,\n   \"id\": 5,\n   \"title\": \"nesciunt quas odio\",\n

Pero el contenido dentro de este string tiene una estructura JSON, esto hace muy fácil transformar los datos en un hash o un array. A esto se le denomina **parsing**. Transformar un input (ya sea un archivo o datos introducidos por el usuario, o la respuesta de una API) en datos que pueden ser fácilmente manipulados.

Transformando la respuesta a JSON

El cuerpo de la respuesta es un string que contiene un JSON. Para transformarlo utilizaremos `JSON.parse`

```
require 'json'

results = JSON.parse(response.read_body)
puts results.class # Veremos que el resultado es un array
puts results[0] # Mostramos el primer elemento
```

Trabajando con la respuesta

El resto del trabajo dependerá de como venga estructurada la respuesta. Por ejemplo en este caso tenemos un arreglo, cada elemento se compone de un hash y cada clave de este hash es un string.

```
[
  {
    "userId": 1,
    "id": 1,
    "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
    "body": "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehenderit molestiae ut ut quas totam\nnostrum rerum est autem sunt rem eveniet architecto"
  },
  {
    "userId": 1,
    "id": 2,
    "title": "qui est esse",
    "body": "est rerum tempore vitae\nsequi sint nihil reprehenderit dolor beatae ea dolores neque\nfugiat blanditiis voluptate porro vel nihil molestiae ut reiciendis\nqui aperiam non debitis possimus qui neque nisi nulla"
  }, ...
```


Accediendo a un elemento de la respuesta.

Podemos acceder a un elemento en particular de las respuestas utilizando el índice del arreglo y la clave correspondiente.

puts results[0]['userId'] puts results[0]['title']

Iterando los resultados

Podríamos mostrar todos los títulos iterando el arreglo

```
results.each do |post|
  puts post['title']
end
```

Transformando nuestro request en un método

Realizaremos múltiples request donde solo cambiaremos la URL por lo que crearemos un método.

```
require 'uri'
require 'net/http'

def request(url_requested)
    url = URI(url_requested)

    http = Net::HTTP.new(url.host, url.port)

    request = Net::HTTP::Get.new(url)
    request["Cache-Control"] = 'no-cache'
    request["Postman-Token"] = 'fa0e5b4a-32fb-4ee0-afc1-4c2ff90eedde'
```

```
    response = http.request(request)
    JSON.parse(response.body)
end
```

Reutilizando el método

Luego para utilizar el método bastará con:

```
request('http://jsonplaceholder.typicode.com/posts')
```

Tenerlo como método nos permite realizar requests a otros direcciones con una sola línea de código.

Antes de empezar con una página nueva exploraremos la documentación de nuestra API. Saber leer este tipo de documentación es importante para desarrollar aplicaciones con API's.

Capítulo: API Rest

Objetivos:

- Entender la estructura tras una API Rest.
- Leer la documentación de una API Rest.
- Realizar requests de una API Rest.

Introducción

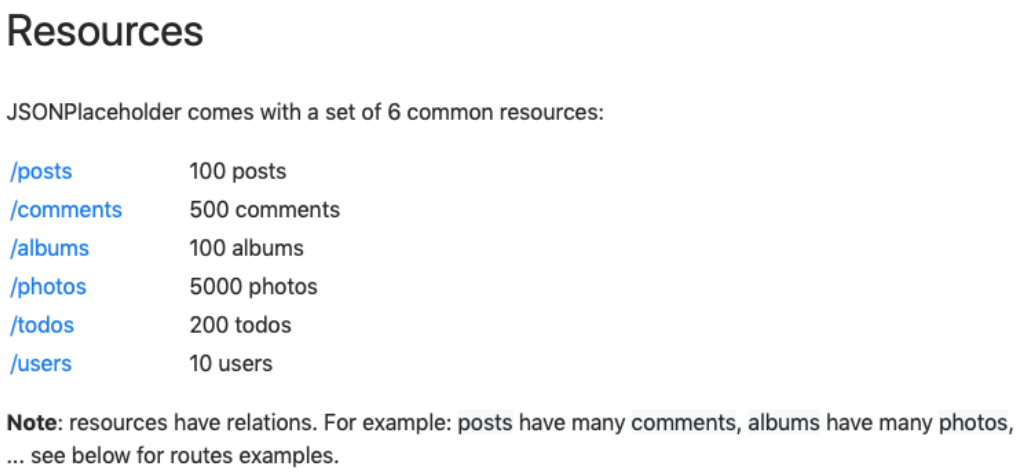
Las API Rest son un estándar sobre como crear y consumir APIs, en base a recursos y acciones. Cada acción se realiza sobre un recurso en específico y para esto tiene una dirección (o URL) asignada.

Para saber que recursos y direcciones tenemos disponibles tenemos que leer la documentación de la API.

Leyendo la documentación.

La documentación en cada página es distinta, pero particularmente lo que necesitamos en cada página es saber qué recursos tenemos disponibles, cuales son las rutas a esos recursos y con qué acción realizar la solicitud.

Los recursos



Resources

JSONPlaceholder comes with a set of 6 common resources:

/posts	100 posts
/comments	500 comments
/albums	100 albums
/photos	5000 photos
/todos	200 todos
/users	10 users

Note: resources have relations. For example: posts have many comments, albums have many photos, ... see below for routes examples.

¿Qué son los recursos?

Los recursos son elementos que podemos obtener y mostrar, y en algunos casos podemos enviar nuevos, actualizarlos y borrarlos.

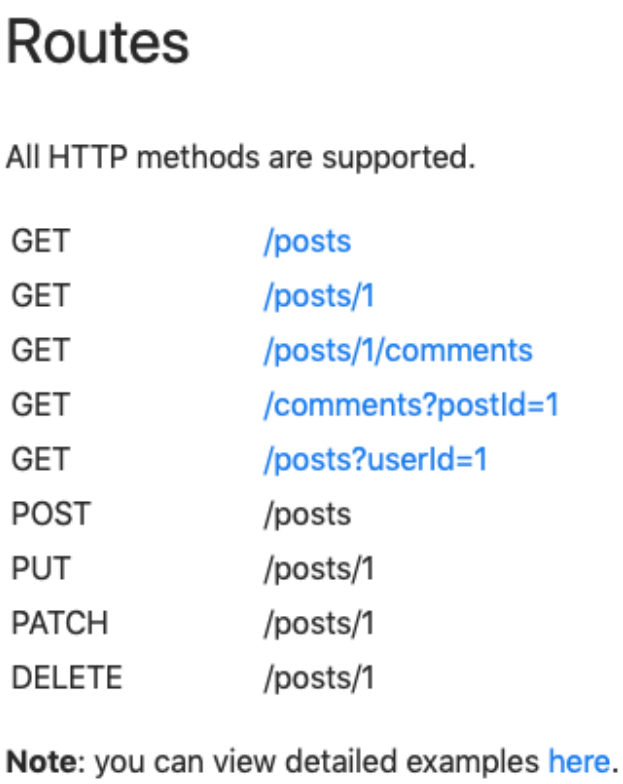
Por ejemplo en esta API tenemos posts, comentarios, álbumes, photos, lista de tareas (todos) y usuarios. Además hay relaciones entre estos recursos.

Para ver que acciones podemos hacer en específico con esto recursos necesitamos conocer las rutas (o direcciones)

Las rutas

Las rutas son las direcciones que nos llevan a los recursos, usualmente se presenta con una estructura relativa a la dirección de la API.

Una ruta tiene una dirección y un `http method` a estos métodos también les conoce como verbos.



Routes

All HTTP methods are supported.

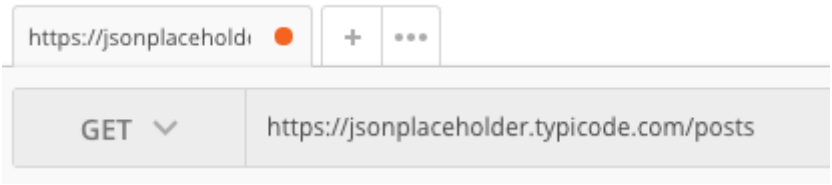
GET	/posts
GET	/posts/1
GET	/posts/1/comments
GET	/comments?postId=1
GET	/posts?userId=1
POST	/posts
PUT	/posts/1
PATCH	/posts/1
DELETE	/posts/1

Note: you can view detailed examples here.

Datos necesarios para un Request

Un request depende tanto de la dirección como del verbo.

En el request anterior no lo mencionamos pero sí utilizamos uno. El verbo GET. Lo podemos verificar en Postman:



También podemos observar que el verbo aparece específicamente en el código.

```
request = Net::HTTP::Get.new(url)
```

Por ahora seguiremos ocupando el verbo **GET**, pero haremos un request a otro recursos que nos permitirá obtener todas las fotos las fotos.

Utilizaremos nuestro método request que creamos con la url que apunta al recurso de fotos y el verbo se mantiene.

```
data = request('https://jsonplaceholder.typicode.com/photos')[0..10] # Tomemos solo 10
```

Obtendremos:

```
[{"albumId"=>1, "id"=>1, "title"=>"accusamus beatae ad facilis cum similique qui sunt",
"url"=>"https://via.placeholder.com/600/92c952",
"thumbnailUrl"=>"https://via.placeholder.com/150/92c952"}, {"albumId"=>1, "id"=>2,
"title"=>"reprehenderit est deserunt velit ipsam", "url"=>"https://via.placeholder.com/600/771796",
"thumbnailUrl"=>"https://via.placeholder.com/150/771796"}, {"albumId"=>1, "id"=>3, "title"=>"officia porro
iure quia iusto qui ipsa ut modi", "url"=>"https://via.placeholder.com/600/24f355",
"thumbnailUrl"=>"https://via.placeholder.com/150/24f355"}, {"albumId"=>1, "id"=>4, "title"=>"culpa odio
esse rerum omnis laboriosam voluptate repudiandae", "url"=>"https://via.placeholder.com/600/d32776",
"thumbnailUrl"=>"https://via.placeholder.com/150/d32776"}, ...
```

Analizando la respuesta

La estructura de la respuesta es la misma, un arreglo que contiene hashes donde cada hash representa una foto. y el proceso de obtener resultados también es el mismo, iteramos el arreglo y de cada hash obtenemos la información que necesitamos.

Como ejercicio de integración vamos a obtener todas las direcciones de las fotos y generar una página web con una galería.

Ejercicio de integración: Construyendo una página web a partir de las fotos

Primero idearemos el algoritmo, el secreto para resolver problemas complejos es descomponerlos en partes que ya sabemos hacer (divide y conquista).

1. Realizamos un request a `'https://jsonplaceholder.typicode.com/photos` y obtenemos el arreglo con hashes de fotos.
2. Generamos un arreglo a partir de las fotos del hash
3. Iteramos el arreglo con las fotos y guardamos sus resultados en un archivo
   3.1 Al momento de guardar agregaremos las etiquetas de HTML.

Solución

```
data = request('https://jsonplaceholder.typicode.com/photos')[0..10] # Limitamos los resultados a 10
photos = data.map{|x| x['url']}
html = ""
photos.each do |photo|
  html += "<img src=\"#{photo}\">\n"
end

File.write('output.html', html)
```

Si bien la página funciona se deja como ejercicio agregar el resto del html, puesto que el documento no tiene doctype, ni head, ni body. Se deja como tarea propuesta agregar la información para generar de forma automática una página web válida.

Capítulo: Más allá del GET

Objetivo

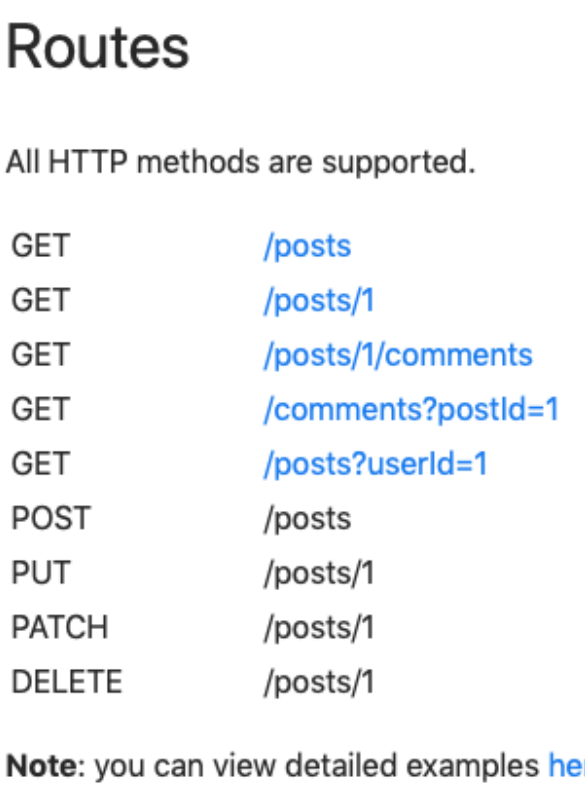
- Utilizar los verbos de POST, PUT y DELETE
- Conocer el concepto de negociación de contenido
- Utilizar headers para negociar contenido
- Utilizar el body de un request para enviar contenido
- Crear un nuevo objeto utilizando el método Post

Introducción

Las APIs no solo nos permiten obtener información, también podemos subir información, actualizar información y borrar información. En una API Rest estos pedidos suceden bajo otros verbos diferentes a **GET**

¿Cómo subir información?

En una API Rest hay un estándar definido. El método **Post** se utiliza para subir información.



Routes

All HTTP methods are supported.

GET	/posts
GET	/posts/1
GET	/posts/1/comments
GET	/comments?postId=1
GET	/posts?userId=1
POST	/posts
PUT	/posts/1
PATCH	/posts/1
DELETE	/posts/1

Note: you can view detailed examples here.

Pero esta parte de la documentación no nos dice mucho, pero hay un link a ejemplos, sigamos el link.

Creating a resource

```
// POST adds a random id to the object sent
fetch('https://jsonplaceholder.typicode.com/posts', {
    method: 'POST',
    body: JSON.stringify({
      title: 'foo',
      body: 'bar',
      userId: 1
    }),
    headers: {
      "Content-type": "application/json; charset=UTF-8"
    }
  })
  .then(response => response.json())
  .then(json => console.log(json))

/* will return
{
  id: 101,
  title: 'foo',
  body: 'bar',
  userId: 1
}
*/
```

El ejemplo de la documentación está en Javascript, pero de todas formas podemos leerlo poniendo atención en algunos elementos específicos.

Estudiemos los elementos clave:

- El request indica el método post.
- El request indica además headers donde hay un content-type
- El body está envuelo en un método .stringify que si buscamos en internet aprenderemos que es la operación inversa a .parse o sea que puede transformar un hash en un json.

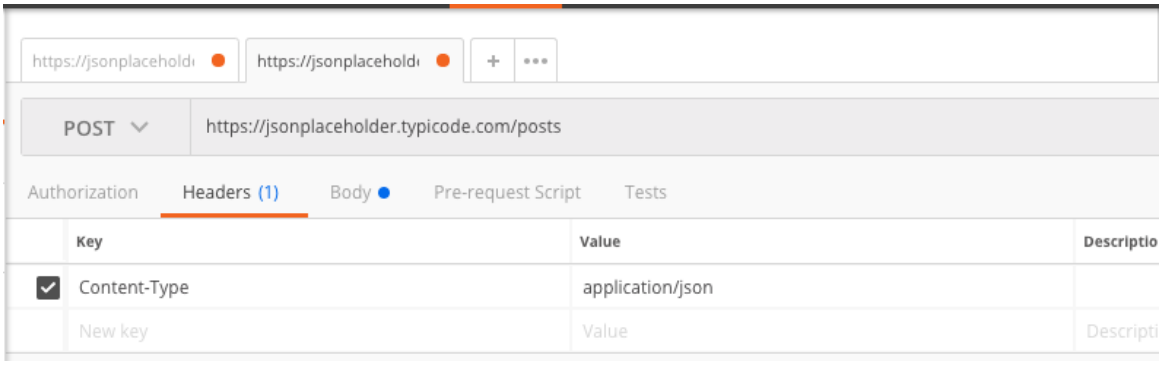
Utilicemos esto para agregar un artículo nuevo a la API

¿Qué son el header y el body?

Un request a una API Rest tiene un header y un body, en el header especificamos información general, por ejemplo codificación o tipo de contenido que vamos a enviar. En el body enviaremos información específica, por ejemplo si queremos subir un artículo aquí agregaremos el título y el contenido.

Subiendo un artículo desde Postman

Para subir un artículo nuevo necesitamos seleccionar el verbo post, e ingresar la URL de la documentación.



Negociación de contenido

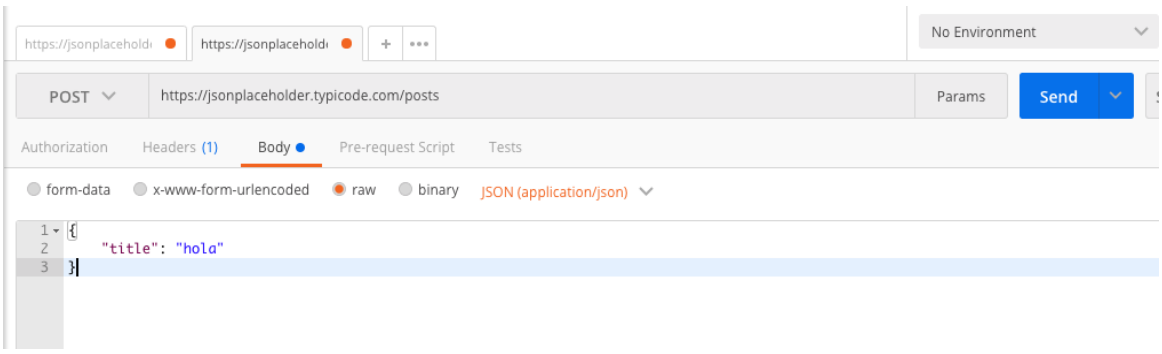
Además necesitaremos agregar el header `Content-type: application/json` Esto es porque algunas APIs pueden recibir distintos tipos de contenido.

Negociar el contenido permite que en un mismo endpoint se pueden resolver pedidos de distintos clientes en distintos formatos. Los tipos de contenido que puede manejar una API se especifican en la documentación, en este caso lo obtuvimos del ejemplo.

Agregando el contenido

Dentro de la API y del ejemplo vimos que un artículo se compone de un título **title**, de un cuerpo **body** y de un dueño **userId**

Como primera prueba subiremos un artículo. Para eso iremos al tab de body y seleccionar la opción **raw** y a la derecha marcaremos la opción JSON (application/json)



Aquí es muy importante indicar que los strings **deben utilizar doble comilla**, utilizar comillas simples alrededor de un string en un JSON dará como resultado en un error.

Tip: Validador de JSON en línea: https://jsonlint.com/

Observando el resultado

En la sección inferior bajo el tab body encontraremos el resultado entregado de la API indicando que se creó el artículo con id 101 y título "hola".

Intentemos guardar un nuevo artículo, ahora con `body` y `userId`

```
{
    "title": "Post 101",
    "body": "Este es nuestro primer post",
    "userId": 1
}
```

Obtendremos como respuesta:

```
{
    "title": "Post 101",
    "body": "Este es nuestro primer post",
    "userId": 1,
    "id": 101
}
```

Esta API en particular no es persistente

Para asegurar el funcionamiento con muchos usuarios distintos, esta API no guarda los cambios realizados, solo dice que los hizo, trabajar con una API que los guarda es exactamente igual.

Subiendo un artículo desde Ruby

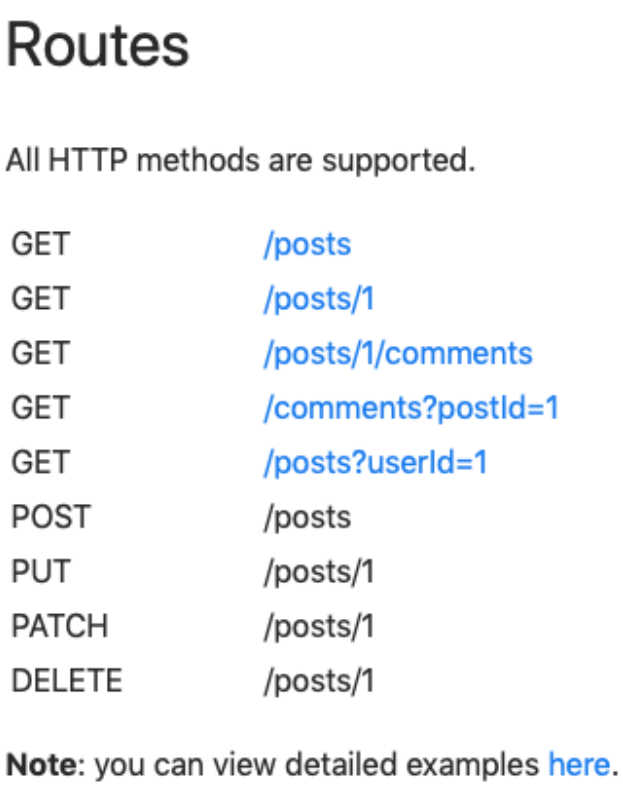
Para lograr esto copiaremos el código que genera Postman

```
require 'uri'
require 'net/http'

url = URI("https://jsonplaceholder.typicode.com/posts")

http = Net::HTTP.new(url.host, url.port)
http.use_ssl = true
http.verify_mode = OpenSSL::SSL::VERIFY_NONE

request = Net::HTTP::Post.new(url)
request["content-type"] = 'application/json'
request["cache-control"] = 'no-cache'
request["postman-token"] = 'f230cc3e-dcda-37b6-556d-fd1e9d334108'
request.body = "{\n\t\"title\": \"Post 101\",\n\t\"body\": \"Este es nuestro primer Post\",\n\t\"userId\": 1\n}\n"

response = http.request(request)
puts response.read_body
```

Actualizando un recurso

Los métodos para actualizar un recurso en una API rest son PUT o PATCH, si bien existe una diferencia entre estos la mayoría de las API no hace distinción.

Lo otro que tenemos que saber además del verbo es el endpoint (o dirección). Para esto veremos de nuevo la documentación.



Routes

All HTTP methods are supported.

GET	/posts
GET	/posts/1
GET	/posts/1/comments
GET	/comments?postId=1
GET	/posts?userId=1
POST	/posts
PUT	/posts/1
PATCH	/posts/1
DELETE	/posts/1

Note: you can view detailed examples here.

Actualizando el recurso desde Postman

Vemos en la documentación que a diferencia de subir un artículo que utiliza la ruta `/posts` la actualización utiliza la ruta `/posts/1` donde el número no tiene que ser necesariamente 1, si no que es identificador del recurso que queremos actualizar, para saber cuales son tenemos que listar los recursos (como lo hicimos al momento de aprender a ocupar las APIs)

En este caso los identificadores van desde 1 hasta 100, por lo tanto si queremos cambiar el artículo con id 20 tendríamos que hacer un request con método put a `https://jsonplaceholder.typicode.com/posts/20`

Ademas dentro de body tenemos que agregar los nuevos valores para el artículo.

Actualizando un artículo desde Ruby

Para estudiar como podemos modificar el artículo utilizando Ruby lo que haremos es copiar el código generado por Postman, y veremos que la idea es exactamente la misma.

```
require 'uri'
require 'net/http'

url = URI("https://jsonplaceholder.typicode.com/posts/20")

http = Net::HTTP.new(url.host, url.port)
http.use_ssl = true
http.verify_mode = OpenSSL::SSL::VERIFY_NONE

request = Net::HTTP::Put.new(url)
request["content-type"] = 'application/json'
request["accept-charset"] = 'UTF-8'
request["cache-control"] = 'no-cache'
request["postman-token"] = 'e12f50c0-9587-fbde-0960-61de47ba3517'
request.body = "{\n\t\"title\": \"Cambio de post\",\n\t\"body\": \"Actualizando el post\",\n\t\"userId\": 1\n}\n"

response = http.request(request)
puts response.read_body
```

Borrar un artículo

Para borrar un artículo seguiremos la misma idea, veremos de la documentación que el método es delete y que la dirección es `posts/id`

Borrando un artículo desde Ruby

Utilizaremos el código generado por Postman

```
require 'uri'
require 'net/http'

url = URI("https://jsonplaceholder.typicode.com/posts/20")

http = Net::HTTP.new(url.host, url.port)
http.use_ssl = true
http.verify_mode = OpenSSL::SSL::VERIFY_NONE

request = Net::HTTP::Delete.new(url)
request["content-type"] = 'application/json'
request["accept-charset"] = 'UTF-8'
request["cache-control"] = 'no-cache'
request["postman-token"] = '9a1b192d-aeb7-4509-688f-2fe0fbf5b4b0'
request.body = "{\n\t\"title\": \"Cambio de post\",\n\t\"body\": \"Actualizando el post\",\n\t\"userId\": 1\n}\n"

response = http.request(request)
puts response.read_body
```

Resumen del capítulo

En este capítulo aprendimos:

- Una API Rest es un estándar para ver, crear, actualizar y borrar recursos.
- Para ver recursos utilizaremos el verbo (o http method) **GET**
- Para agregar un recurso nuevo utilizaremos el verbo **POST**
- Para actualizar un recurso utilizaremos **PUT** o **PATCH**
- Para borrar un recurso utilizaremos **DELETE**

Capítulo: Ejercicio de integración con respuesta

Como ejercicio final vamos a consultar los últimos precios de bitcoin y generar un gráfico. Para esto ocuparemos los datos de cierre diario de la API de coindesk, estos se encuentran en la siguiente URL

`https://api.coindesk.com/v1/bpi/historical/close.json`

Se pide obtener los precios y fechas del último periodo y a partir de estos obtener un arreglo de todas las fechas donde el valor ha sido menor a 5000 USD.

Solución

Primero dividiremos el problema en 2 grandes partes.

- Hacer el request a la API de coindesk y obtener los resultados
- Procesar los resultados para obtener un arreglo con las fechas

```
# Obteniendo los precios de bitcoins

require 'uri'
require 'net/http'

def request(url_requested)
  url = URI(url_requested)

  http = Net::HTTP.new(url.host, url.port)
  http.use_ssl = true
  http.verify_mode = OpenSSL::SSL::VERIFY_PEER

  request = Net::HTTP::Get.new(url)
  response = http.request(request)
  return JSON.parse(response.body)
end

prices = request('https://api.coindesk.com/v1/bpi/historical/close.json')['bpi']
```

El segundo paso consiste en obtener el arreglo con las fechas, aquí hay distintas posibilidades. La primera pregunta que deberíamos hacernos es si existe un método en los hashes que nos permite seleccionar elementos en base a un valor, esto lo estudiamos previamente y sabemos que existe el método `.select`

```
# Solución 1

selected_data = prices.select {|k,v| v < 5000 }
selected_data.keys
```

Sin embargo hay muchas otras formas de resolver el problema, por ejemplo podríamos haberlo hecho utilizando `.each` y guardar en un nuevo hash todos los elementos que cumplan el criterio.

Otra opción bien distinta sería obtener un arreglo con únicamente los precios, filtrar el arreglo por los valores bajo 5000 y luego utilizar un diccionario invertido para buscarlos.

```
# Solución 2b

under_5000 = prices.values.select {|x| x < 5000 }
dates = under_5000.map {|x| prices.invert[x]}
```

Existen muchas soluciones a un problema siempre deberíamos preferir las mas sencillas.

Capítulo: SSL

Objetivos

- Conocer las implicaciones de SSL.
- Conocer la importancia de **no** enviar información sensible sin SSL.
- Entender la importancia de verificar el certificado.

Introducción

SSL es un acrónimo de Secure Sockets Layer, es una capa de seguridad que establece cifrado (encriptación) entre el cliente y el servidor y evita que la información que enviamos o recibimos sea leída por un tercero.

Esto sucede ya sea conectándonos a una página web por el navegador o una API.

Es importante que toda comunicación que maneje información sensible ocupe esta capa de cifrado debido a que es muy fácil incluso para inexpertos interceptar estas comunicaciones.

Vamos a revisar en términos generales como funciona el sistema, pero esto no es necesario para aprovechar las bondades de la seguridad.

Cómo funciona la encriptación por SSL

La clave para entender el funcionamiento de la encriptación es entender que hay dos claves, una para cifrar y otra descifrar. Para entender esto veámoslo con un ejemplo.

Supongamos por un momento que la comunicación es entre dos persona, Alice y Bob. Para cifrar el mensaje Alice tiene una clave y para descifrarlo tiene otra. Previo a comunicarse Alice se junta con Bob y le traspasa la clave para descifrar.

Esto permite que Alice cifre su mensaje con su clave, le envía el mensaje a bob y bob la descifra con la clave que la pasó previamente Alice. Esto tiene dos ventajas, primero el mensaje pasa cifrado por lo que nadie mas puede leerlo a menos que tenga la clave para descifrar, pero además tiene otra ventaja, la llave para descifrar solo sirve para descifrar mensajes de Alice, por lo que sabremos que el mensaje viene realmente de Alice y no de otra persona.

Clave pública, clave privada

Las dos claves son distintas por eso decimos que este sistema de cifrado es asimétrico. Una clave es para cifrar el mensaje la otra para descifrarlo.

Las claves tienen nombre, la clave que firma el mensaje pero jamás se comparte recibe el nombre de clave privada y la clave que se comparte recibe el nombre de clave pública.

Ventajas de SSL

El sistema de juego entonces tiene dos ventajas.

- Cifra el mensaje impidiendo que terceros puedan leerlo.
- Asegura que el emisor es quien dice ser, porque si alguien mas cifró el mensaje con una llave distinta, el mensaje no tendrá sentido al decifrarlo con la llave pública.

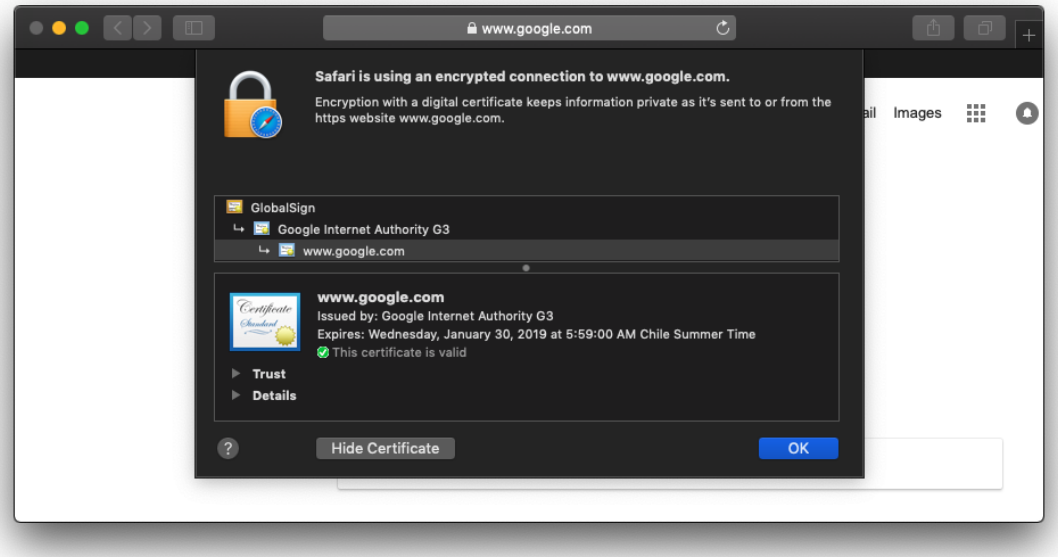
Todo el sistema funciona de forma automática

Todo lo que hemos hablado hasta ahora funciona de forma automática. Y al realizarlo vía código es casi automático, tendremos que agregar solo 2 líneas extra de código que explicitan que utilizaremos SSL.

La magia que sucede detrás

Al momento de conectarnos a un sitio web que utilice SSL con un navegador se establece un acuerdo, llamado en inglés **handshake** de forma automática. En este handshake el servidor envía un certificado que tiene el nombre del sitio y la clave pública al cliente.

Si nos conectamos con el navegador y hacemos click en el candado de la conexión segura podemos ver el certificado.



El cliente ocupa la clave pública que viene en el certificado para cifrar un mensaje y se la devuelve al servidor, si el servidor puede descifrar el nuevo mensaje significa que el servidor tiene la clave privada correcta.

Dentro del proceso ambos, cliente y servidor utilizando los números que se enviaron durante el handshakre crearon un clave secreta especial, el resto del proceso ocupa una sistema de cifrado simétrico utilizando esta única clave que nunca fue transmitida pero que ambos servidores tienen.

Conectándose con SSL

Postman identifica automáticamente si un request ocupa https y genera el código para conectarnos.

Lo vamos a copiar y dejar dentro del método

```
def request(url_requested)
  url = URI(url_requested)

  http = Net::HTTP.new(url.host, url.port)
  http.use_ssl = true # Se agrega esta línea
  http.verify_mode = OpenSSL::SSL::VERIFY_NONE # Se agrega esta otra línea

  request = Net::HTTP::Get.new(url)
  request["cache-control"] = 'no-cache'
  request["postman-token"] = '5f4b1b36-5bcd-4c49-f578-75a752af8fd5'

  response = http.request(request)
  return JSON.parse(response.body)
end
```

Veremos que la única diferencia con nuestro requests previos son estas líneas

```
http.use_ssl = true # Se agrega esta línea
http.verify_mode = OpenSSL::SSL::VERIFY_NONE # Se agrega esta otra línea
```

Esto es lo único que necesitamos para establecer una conexión segura, bueno, casi, todavía queda resolver el tema de la validez del certificado.

Hombre en el medio

Hombre en el medio (**Man in the middle**) es una estrategia de hacking que consiste en hacer de puente (**Proxy**) entre el cliente y el servidor.

El atacante se sitúa en el punto medio, captura la clave pública que envía el servidor y le envía una nueva clave pública al cliente. El cliente cree que se está conectando con el servidor directamente pero no es así y toda la comunicación queda comprometida.

Este ataque es muy fácil de lograr pero solo si el certificado no es validado (o auto firmado). Al momento de conectarse los certificados se revisan contra agentes certificadores. Un certificado validado (nombre de sitio + clave pública) complica la situación al atacante puesto que este difícilmente tendrá la clave privada asociada al certificado.

Protegiéndose del Hombre en el medio

El ataque de **Man in the middle** es fácil de lograr si el certificado no se verifica, por lo que esta línea es peligrosa.

```
http.verify_mode = OpenSSL::SSL::VERIFY_NONE
```

Es útil si estamos probando una API que estemos desarrollando, pero si estamos enviando información sensible (como por ejemplo un login, un password o un token) entonces es peligroso.

Podemos cambiar la línea por

```
http.verify_mode = OpenSSL::SSL::VERIFY_PEER
```

Corrigiendo nuestro código, finalmente nuestro método quedaría así:

```
require 'uri'
require 'net/http'

def request(url_requested)
  url = URI(url_requested)

  http = Net::HTTP.new(url.host, url.port)
  http.use_ssl = true
  http.verify_mode = OpenSSL::SSL::VERIFY_PEER

  request = Net::HTTP::Get.new(url)
  request["cache-control"] = 'no-cache'
  request["postman-token"] = '5f4b1b36-5bcd-4c49-f578-75a752af8fd5'
  response = http.request(request)
  return JSON.parse(response.body)
end
```

```
data = request('https://jsonplaceholder.typicode.com/photos')[0..5] # Tomemos solo 5
```

Veremos como resultado los primeros 5 elementos

```
[{"albumId"=>1,
  "id"=>1,
  "title"=>"accusamus beatae ad facilis cum similique qui sunt",
  "url"=>"https://via.placeholder.com/600/92c952",
  "thumbnailUrl"=>"https://via.placeholder.com/150/92c952"},
    ...
```

Capítulo: API con autenticación

Objetivos

- Obtener el id y la clave para autenticarse a una API
- Conectarse a una API que requiere autenticación
- Utilizar la API del diccionario de oxford
- Estudiar otros tipos de autenticación

Introducción

Muchas APIs requieren de autenticación para poder acceder a sus servicios.

Para autenticarse con una API nuestro primer paso consiste en conseguir una clave para conectarse, las mismas APIs disponen de servicio de registro y al completarlo te entregan un id.

Consiguiendo la clave de la API de Oxford Dictionary

Para conseguir esta clave entraremos a: https://developer.oxforddictionaries.com y llenaremos el formulario de registro, también necesitaremos confirmar nuestro email.

En algunos casos se utilizan dos claves, un app_id y un app_key, en otros casos un key y un secret, independiente de cuantas claves el sistema siempre es similar.

Una vez obtenido el key (o los keys) podremos acceder a la API

Autenticando desde Postman

Existen distintos tipos de autenticación, la mayoría de los modelos consiste en pasar un token en los **header** del request.

```
require 'uri'
require 'net/http'

def request(url_requested)
    url = URI(url_requested)

    http = Net::HTTP.new(url.host, url.port)
    http.use_ssl = true
    http.verify_mode = OpenSSL::SSL::VERIFY_PEER

    request = Net::HTTP::Get.new(url)
    request['app_id'] = 'Tú_ID'
```

```
    request['app_key'] = 'Tu_llave_secreta'
    response = http.request(request)
    return JSON.parse(response.body)
end

word = "test"
request("https://od-api.oxforddictionaries.com:443/api/v1/entries/en/#{word}")
```

Veremos que el resultado muestra mucha información, podemos mostrarla de forma ordenada con `pp

```
pp request("https://od-api.oxforddictionaries.com:443/api/v1/entries/en/#{word}")
```

La pregunta importante es que información es la que queremos, por ahora nos conformaremos con la definición **definitions**, así que iremos del final hasta el principio.

- Definitions es el key de un hash `['definitions']`
- El hash está dentro de un arrego, es el primer elemento `[0]['definitions']`
- Este arreglo parte de un hash bajo la clave **senses** `['senses'][0]['definitions']`
- Senses es un key de entries `['entries']['senses'][0]['definitions']`
- El hash está dentro de un arreglo y es el primer elemento `[0]['entries']['senses'][0]['definitions']`
- El arreglo está asociado al key lexicalEntries `['lexicalEntries'][0]['entries']['senses'][0]['definitions']`
- El diccionario es el primer elemento de un arreglo `[0]['lexicalEntries'][0]['entries']['senses'][0]['definitions']`
- Todo está bajo la clave de results `['results'][0]['lexicalEntries'][0]['entries']['senses'][0]['definitions']`

```
result = request("https://od-api.oxforddictionaries.com:443/api/v1/entries/en/test")
result['results'][0]['lexicalEntries'][0]['entries'][0]['senses'][0]['definitions']
```

No siempre es tan complicado, muchas veces los datos están con uno o dos niveles de profundidad, pero por lo mismo este es un buen caso de estudio.

Otros tipos de autenticación

En este caso nos tocó ingresar la autenticación en el header, esta es una de las opciones mas frecuentes pero algunas APIs tienen diferentes modelos de autenticación.

- En algunos casos nos tocará ingresar la clave en el body.
- En algunos caso el token de autenticación cambiará después de cada request y tendremos que usar el último valor para rescatar el siguiente.

También existen otros modelos mas complejos como el OAuth2, el cual delega la autenticación a otros proveedores de servicios web como Facebook, Google, Github, etc para probar la identidad de una solicitud.

De todas formas lo importante siempre es leer con calma la documentación y encontrar las pistas necesarias para lograr la conexión y en el peor de los casos siempre existe la opción de comunicarse con los desarrolladores de la API para pedir ayuda o buscar otra API que logre el mismo propósito.

Cuidado con los token

Lo último y mas importante es que muchos servicios tienen límites de consumo y otros tanto manejan información delicada, los tokens, passwords y otros datos delicados entregados no deben ser subidos a repositorios públicos o compartidos con terceros.

¿Cómo podemos manejar los tokens?

Con ARGVS (vector de argumentos)

Hay varias estrategias, podemos utilizar una muy simple que consiste en dejar los datos sensibles fuera del programa, al cargar el programa le pasaremos como parámetro los datos necesarios.

Con variables de entorno

En Linux y OSX podemos utilizar variables de entorno.

Esta opción consiste en utilizar variables que se definen en el terminal y las podemos llamar desde nuestro programa.

Veamos un ejemplo básico, en el terminal escribiremos `export a=5`. Luego entraremos a IRB y escribiremos ENV['a'] y observaremos el valor 5.

Persistiendo las variables de entorno

Pero este valor solo quedará en la sesión, al abrir un terminal nuevo no lo tendremos. Para hacerlo persistente podemos escribir esta línea en el archivo `.bashrc` o `.bash_profile`. Este archivo que se encuentra en la carpeta de usuario. Lo podemos abrir con un editor de texto y en la última línea agregar nuestra variable.

```
export API_1_SECRET_KEY=23123u2139183
```

Finalmente, después de guardar, para recargar los cambios realizados dentro del archivo ocuparemos la siguiente línea:

```
source ~/.bash_profile
```

Si queremos probar si funcionó podemos hacerlo desde IRB con `puts ENV['API_1_SECRET_KEY']` o directamente desde el terminal con `echo $API_1_SECRET_KEY`. El signo peso es necesario en el echo para indicar que es una variable.

Cierre

A lo largo de este módulo aprendimos sobre Hashes, API's, SSL y variables de entorno.

Revisemos algunos elementos claves que aprendimos.

- Los Hashes son estructuras de datos similares a un diccionario que guardan pares de información del tipo clave => valor.
- Los Hashes tienen métodos varios métodos muy útiles para filtrar, seleccionar, reducir, etc.
- JSON es un formato estándar para intercambiar información programáticamente. Es sencillo manipular y tranformar datos de JSON a Hash y viceversa.
- Las API's REST son una forma de arquitectura orientada a servicios que permite la interacción estandar entre diferentes sistemas no relacionados.
- Las API's pueden ser públicas o privadas (con autenticación).
- SSL es una capa de seguridad con encriptación para proteger la información enviada entre cliente y servidor.
- Las variables de entorno son una alternativa segura para manejar llaves de autenticación.

Palabras finales

Las API's están transformado los negocios y permitiendo la interconexión de muchos sistemas. Trabajar fluidamente con ellas es una habilidad muy útil para cualquier desarrollador en cualquier lenguaje o framework. Además, si complementamos con el manejo seguro de la información que estamos enviando mediante SSL y un adecuado manejo de credenciales estaremos creando aplicaciones robustas y preparadas para solucionar problemas en entornos productivos.

Temas interesantes que todavía nos falta aprender

Entender la arquitectura orientada a servicios siguiendo las restricciones REST nos permitirá hacer aplicaciones ordenadas y extensibles. Nos falta ahora aprender a desarrollar nuestras aplicaciones siguiendo estos principios y modelar nuestras soluciones con un enfoque en recursos y en acciones que haremos con ellos.