



# API REST

API REST

{desafío}  
latam\_



# ¿Qué aprenderemos en este módulo?

Implementar aplicaciones empresariales que disponibilizan servicios Rest utilizando el framework Express y el entorno Node.js para dar solución a los requerimientos de la organización.



***Describir las características principales de una arquitectura REST distinguiendo buenas prácticas para el diseño de una API REST para la interoperación de sistemas.***

***Implementar un servidor REST utilizando el framework Express para la disponibilización de recursos acorde a las buenas prácticas.***

- Unidad 1:  
API REST
- Unidad 2:  
Subida de archivos al servidor
- Unidad 3:  
JWT



Te encuentras  
aquí



## ¿Qué aprenderás en esta sesión?

- *Reconoce las características principales de una arquitectura REST para la interoperación de sistemas.*
- *Reconoce las buenas prácticas para la construcción de una API Rest.*

Imagina que estás desarrollando una aplicación web que necesita mostrar información en tiempo real, como pronósticos del clima. ¿Cómo podrías obtener estos datos de manera dinámica y mostrarlos en tu aplicación?



**`/* ¿Qué es una API REST? */`**

# ¿Qué es una API REST?

Una API REST (Representational State Transfer) es un servicio o conjunto de servicios web, que permiten la transferencia de datos en un entorno de red utilizando el protocolo HTTP.

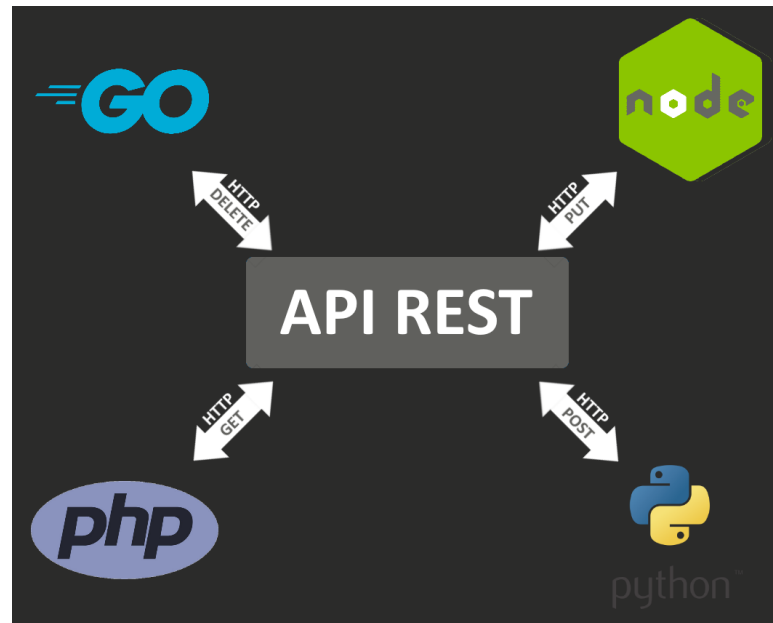
Ya sea a través de una red interna o pública como Internet, los servicios REST se encargan de interconectar aplicaciones que no necesariamente utilizan la misma tecnología, pero sí el mismo protocolo para el intercambio de datos.



# ¿Qué es una API REST?

Para intercambiar información, se realizan peticiones desde un cliente, como una aplicación o sitio web a un servidor, en donde esté alojado el servicio REST, el cual responde con la información solicitada.

Cada una de estas peticiones se realiza bajo las reglas del protocolo HTTP (como el uso de verbos: GET, POST, DELETE, PUT) y se reconocen bajo un identificador único llamado URI, sumado a un lenguaje en común como lo son XML o JSON.



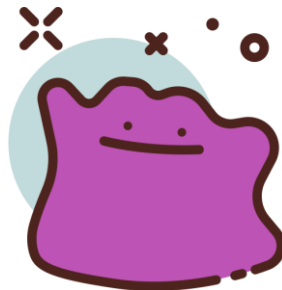


# ¿Qué es una API REST?

Un ejemplo de esta clase de servicios web, es la [PokeAPI](#), la cual disponibiliza una fuente de información y datos que pueden ser consultados con relación al mundo Pokémon.

Por ejemplo, si accedemos a la siguiente URL utilizando un navegador web:

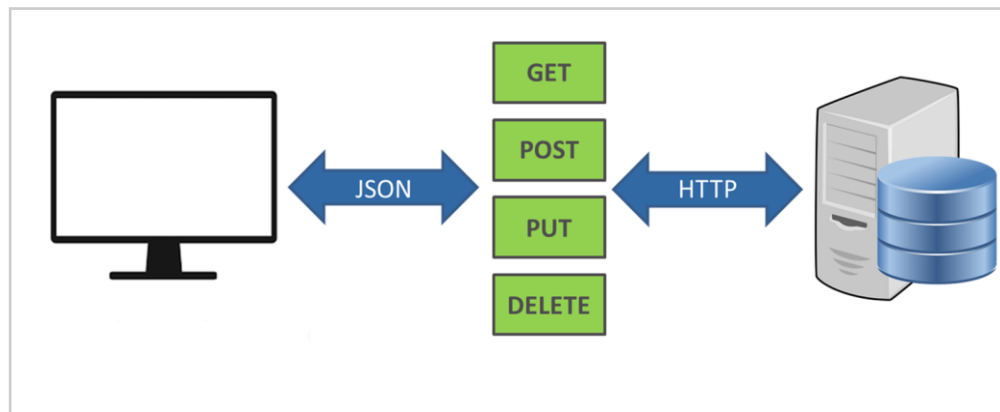
- <https://pokeapi.co/api/v2/pokemon/ditto>



# ¿Qué es una API REST?

Entonces las APIs REST funcionan mediante la definición de verbos HTTP. Cada uno de estos, tienen una función semántica, la cual nos advierte qué es lo que podemos esperar de cada petición:

- **GET:** Lectura y obtención de datos.
- **POST:** Inserción o almacenamiento de datos.
- **DELETE:** Eliminación de datos.
- **PUT:** Modificación de datos



# **/\* Diferencia entre API, REST, API REST y API RESTful \*/**

# Diferencia entre API, REST, API REST y API RESTful

La diferencia entre cada uno de estos conceptos radica en gran parte al servicio al que nos referimos. No me malinterpretes, cada una tiene su definición y esto no va a cambiar.

No obstante, en las documentaciones y material académico audiovisual, acostumbramos a referirnos a una API RESTful como simplemente la palabra o el acrónimo “API”, siendo esta por cuenta propia una representación de algo diferente.



# Diferencia entre API, REST, API REST y API RESTful

Para entenderlo un poco mejor, a continuación te muestro una lista con una definición breve de cada uno de estos conceptos:

- **API:** Por sus siglas Application Programming Interface (Interfaz de programación de aplicaciones), es una biblioteca de propiedades y/o métodos disponibilizada en la estructura de una instancia, mayormente representada en programación como un objeto.
- **REST:** Es una arquitectura de comunicación basada en el protocolo HTTP que sirve para comunicar dos sistemas diferentes, sin importar la base o la tecnología en la que fue creada.

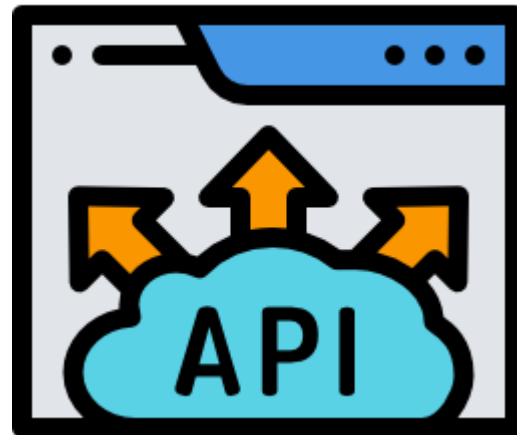
# Diferencia entre API, REST, API REST y API RESTful

- **API REST:** Es la mezcla del concepto de una API y la arquitectura REST, en pocas palabras sería disponibilizar en una ruta o endpoint funciones, propiedades o métodos ubicados dentro de una aplicación usando el protocolo HTTP. Está basado en verbos que ayudan a la construcción de sistemas CRUD, y por convención son escritas dentro de un servidor, teniendo como resultado un servicio WEB.
- **API RESTful:** Es lo mismo que una API REST, aunque algunos se refieren a esta como una API REST completa, es decir, que ofrece por lo menos los verbos GET, POST, PUT y DELETE en su programación.

**/\* Consumiendo API REST \*/**

# Consumiendo API REST

Podemos consultar APIs en cualquier parte de nuestra aplicación con las herramientas ya conocidas (fetch, AJAX, axios) y sabiendo esto podríamos incluir a nuestro servidor una ruta para el almacenamiento de datos provenientes de una API REST.





# Consumiendo API REST

## Ejemplo

Como ejemplo utilizemos la API de [randomuser](#) para guardar un usuario aleatorio y hacer una aplicación que consulte un servicio externo y persista la información obtenida en esa consulta.

Instala axios en tu proyecto y prosigue con los siguientes pasos para crear un ruta **"/random"** que consiga lo previamente dicho:

- **Paso 1:** Incluye la importación del paquete axios.

```
// Paso 1
const axios = require('axios')
```

# Consumiendo API REST

## Ejemplo

- Paso 2: Crea una ruta GET /random

```
// Paso 2
app.get("/random", async (req, res) => {
  // Paso 3
  const { data } = await axios.get("https://randomuser.me/api")
  // Paso 4
  console.log(data)
  res.send()
})
```

# Consumiendo API REST

## Ejemplo

- **Paso 3:** Utiliza axios para consultar la API en su end point:  
<https://randomuser.me/api/>
- **Paso 4:** Imprime por consola la data obtenida y concluye la consulta.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

[Symbol(kOutHeaders)]: [Object: null prototype] {
  accept: [Array],
  'user-agent': [Array],
  host: [Array]
}
},
data: {
  results: [ [Object] ],
  info: { seed: 'bb46bd515cbd8096', results: 1, page: 1, version: '1.3' }
}
}
```

**/\* Almacenando datos de una API REST  
\*/**

# Almacenando datos de una API REST

Bien, ahora que logramos consultar la API de randomuser a partir de la consulta a una ruta de nuestro servidor, lo que sigue es almacenar estos datos en nuestro Usuarios.json para persistir esta data consultada.

Para esto sigue los siguientes pasos:

- **Paso 1:** Guardar en una variable el objeto que estará en el índice 0 del arreglo results. Este objeto representa el usuario random que se está consultando y contiene en sus propiedades la información del mismo.
- **Paso 2:** Crea una variable usuario con los valores primer nombre, apellido y email del usuario random.

# Almacenando datos de una API REST

- **Paso 3:** Ingresa al arreglo usuarios, el usuario random.
- **Paso 4:** Sobrescribe el archivo "Usuarios.json" con la data nueva

```
app.get("/random", async (req, res) => {  
  const { data } = await axios.get("https://randomuser.me/api")  
  // Paso 1  
  const randomUser = data.results[0];  
  // Paso 2  
  const usuario = {  
    name: randomUser.name.first,  
    lastname: randomUser.name.last,  
    email: randomUser.email,  
  };  
  // Paso 3  
  const { usuarios } = JSON.parse(fs.readFileSync("Usuarios.json",  
    "utf8"));  
  usuarios.push(usuario);  
  // Paso 4  
  fs.writeFileSync("Usuarios.json", JSON.stringify({ usuarios }));  
  res.send();  
})
```

# Almacenando datos de una API REST

Ahora si consultas la siguiente dirección:

<http://localhost:3000/random> y revisas el documento

**Usuarios.json** podrás ver que se agregó un nuevo usuario random, tal como se muestra en la siguiente imagen:



The image shows a code editor with two tabs: 'index.js' and 'Usuarios.json'. The 'Usuarios.json' tab is active, displaying a JSON array of three user objects. The first object has the name 'Akira', lastname 'Toriyama', email 'akayama@example.com', and password '1234'. The second object has the name 'Gichin', lastname 'Funakoshi', email 'gfunakoshi@example.com', and password 'abcd'. The third object has the name 'Yann', lastname 'Blanc', and email 'yann.blanc@example.com'. The code is syntax-highlighted and includes line numbers from 1 to 21.

```
1  {  
2    "usuarios": [  
3      {  
4        "name": "Akira",  
5        "lastname": "Toriyama",  
6        "email": "akayama@example.com",  
7        "password": "1234"  
8      },  
9      {  
10       "name": "Gichin",  
11       "lastname": "Funakoshi",  
12       "email": "gfunakoshi@example.com",  
13       "password": "abcd"  
14     },  
15     {  
16       "name": "Yann",  
17       "lastname": "Blanc",  
18       "email": "yann.blanc@example.com"  
19     }  
20   ]  
21 }
```

# **/\* Características principales \*/**



# Características principales

La construcción de una API REST puede hacerse de diferentes maneras y se considera que su creación es un proceso subjetivo que dependerá de la perspectiva del desarrollador, junto con el modelo de negocio del servicio que se ofrezca.

- Están basadas en REST.
- Utiliza el protocolo HTTP.
- Sirve como puente de comunicación entre 2 aplicaciones.
- Son creadas bajo la arquitectura cliente-servidor.
- Ofrecen mayor estabilidad y escalabilidad en los sistemas.
- Permiten modularizar funcionalidades convirtiéndolas en microservicios.

**/\* Buenas prácticas: verbos HTTP,  
consejos iniciales \*/**

# Buenas prácticas

Las buenas prácticas, se pueden considerar como consejos que podemos tomar al momento de desarrollar aplicaciones. Estos consejos nacen de miles de debates entre expertos a lo largo de la historia, que buscan incansablemente “la mejor manera” de hacer las cosas.



# Buenas prácticas

- Creación de rutas para el consumo de subrecursos declarados en la URL.
- Creación de una nueva versión de la API para ofrecer un contrato o modelo de datos.
- Programación de endpoints dinámicos que permitan ordenar y seleccionar campos de un recurso o conjunto de recursos.
- Ofrecer paginación de nuestras colecciones para el consumo por lotes de los datos que se tienen a disposición.
- Devolver los posibles errores a través de un payload que exponga con una descripción personalizada la representación de lo sucedido.

# Buenas prácticas

## *Verbos HTTP*

Ahora que tienes una idea clara de las buenas prácticas de las API REST probablemente te preguntas ¿Cómo empezar a aplicarlas? Una buena forma de empezar, es asegurarnos de tener los siguientes verbos HTTP disponibles en nuestra API REST:

- **GET:** Consultas a recursos.
- **POST:** Crear recursos.
- **PUT:** Actualizar recursos.
- **DELETE:** Eliminar recursos.



# Buenas prácticas

## Consejos iniciales

- **Usar nombres y no verbos:** Con esto nos referimos a la nomenclatura de las rutas en las API REST, las cuales no deberían ser verbos sino nombres. A modo de ejemplo, en el siguiente código se muestra la forma errada de declarar una ruta para la creación de un curso:

```
app.post('/hacerUnNuevoCurso', (req,res) => {  
  ...  
})
```

# Buenas prácticas

## Consejos iniciales

- **Usar nombres en plural:** En el punto anterior se menciona que lo correcto sería cambiar el nombre de la ruta por “cursos”. El uso del plural no es casualidad, ya que las rutas normalmente están referenciadas a una colección o una tabla en una base de datos y como hablamos de varios “registros”, ocupamos los nombres en plural, logrando con esto una mejor comunicación con los clientes:

```
app.post('/cursos', (req,res) => {  
  ...  
})
```

# Buenas prácticas

## Consejos iniciales

- **Usar los métodos HTTP y no parámetros:** Es común ver en personas que están aprendiendo a crear API REST y que conocen el uso de los parámetros en las rutas, tienden a definir todo como parámetros, inclusive la funcionalidad de la consulta, como se muestra en el siguiente ejemplo:

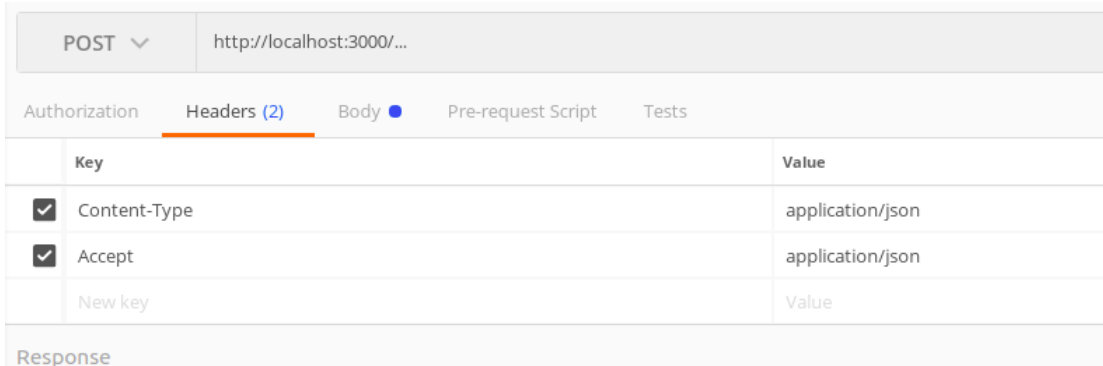
```
app.get('/cursos?nombre=javascript&nivel=10', (req,res) => {  
  ...  
})
```



# Buenas prácticas

## Consejos iniciales

- **Usar las cabeceras HTTP para la serialización de formatos:** Cuando hacemos consultas a una API normalmente ocupamos axios, el método fetch o en su defecto POSTMAN. En cualquiera de estas herramientas podemos especificar las cabeceras de la consulta y declarar el formato de los datos, aunque hoy en día se asume por defecto que serán JSON. En la siguiente imagen vemos un ejemplo de una consulta POST con POSTMAN:



The image shows the Postman interface for a POST request. The top bar indicates the method is POST and the URL is http://localhost:3000/... Below the top bar, there are tabs for Authorization, Headers (2), Body, Pre-request Script, and Tests. The Headers tab is selected and underlined. It contains a table with two headers: Content-Type and Accept, both checked with checkboxes and set to application/json. Below the table, there is a 'New key' row and a 'Response' section at the bottom.

Key	Value
<input checked="" type="checkbox"/> Content-Type	application/json
<input checked="" type="checkbox"/> Accept	application/json
New key	Value

Response

# Buenas prácticas

## *Consejos iniciales*

- **Permitir la sustitución del método HTTP:** Tal vez una de las buenas prácticas más olvidadas y omitidas, es permitir la sustitución del método HTTP usado en una consulta, esto es posible a través de middlewares definidos en el servidor que declaran que una petición puede incluir en sus cabeceras la propiedad “X-HTTP-Method-Override”, la cual sirve para cambiar por ejemplo una consulta POST por una PUT.

**`/* HATEOAS */`**

# HATEOAS

Por sus siglas en inglés Hypermedia as the Engine of Application State (Hipermedia como motor del estado de la aplicación), es un modelo de datos que podemos adoptar en nuestras API REST basado en recursos y subrecursos relacionados por enlaces.

Con HATEOAS logramos tener un mapa claro y simple de navegar para cualquier cliente que quiera consumir los recursos que ofrezca nuestra API REST.



# HATEOAS

La API pública <https://pokeapi.co/> aplica HATEOAS para la programación de sus rutas, podemos notarlo al consultar el siguiente endpoint <https://pokeapi.co/api/v2/pokemon/> y recibir lo que te muestro en la imagen.

```
4  {
5    "count": 1118,
6    "next": "https://pokeapi.co/api/v2/pokemon/?offset=20&limit=20",
7    "previous": null,
8    "results": [
9      {
10       "name": "bulbasaur",
11       "url": "https://pokeapi.co/api/v2/pokemon/1/"
12     },
13     {
14       "name": "ivysaur",
15       "url": "https://pokeapi.co/api/v2/pokemon/2/"
16     },
17     {
18       "name": "venusaur",
19       "url": "https://pokeapi.co/api/v2/pokemon/3/"
20     },
21     {
22       "name": "charmander",
23       "url": "https://pokeapi.co/api/v2/pokemon/4/"
```

# HATEOAS

Como podemos ver, la consulta a la url nos devuelve en la propiedad “results” un arreglo de objetos, donde cada objeto contiene solo dos propiedades: **name** y **url**.

La URL representa el subrecurso de cada pokémon, es decir, que si la consultamos obtendremos la información detallada y específica de cada uno de ellos.

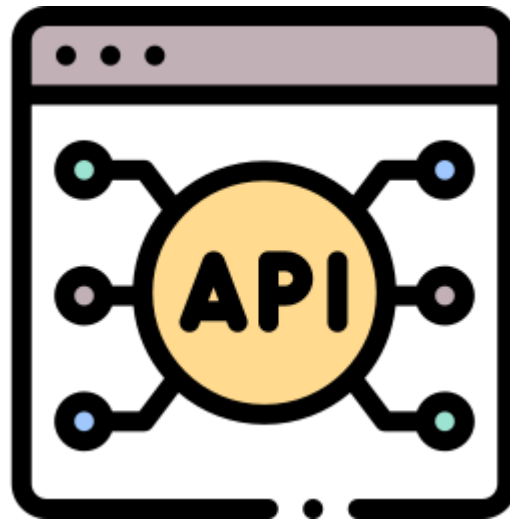
Para ver un ejemplo ingresa a la url de “bulbasaur” y deberás recibir la data que se observa en la siguiente imagen:

# HATEOAS

```
4   {
5   ▶   "abilities": [↔],
23   "base_experience": 64,
24   ▶   "forms": [↔],
30   ▶   "game_indices": [↔],
172  "height": 7,
173  ▶   "held_items": [↔],
176  "id": 1,
177  "is_default": true,
178  "location_area_encounters": "https://pokeapi.co/api/v2/pokemon/1/encounters",
179  ▶   "moves": [↔],
10243 "name": "bulbasaur",
10244 "order": 1,
10245 ▶   "species": {↔},
10249 ▶   "sprites": {↔},
10408 ▶   "stats": [↔],
10458 ▶   "types": [↔],
10474 "weight": 69
10475 }
```

# HATEOAS

La ventaja e importancia de considerar usar HATEOAS en nuestras API REST, se da por la necesidad de conocer la navegación de los recursos que ofrecemos en nuestro servicio, ello implica, que nuestros clientes al consultar un recurso puedan tener de primera mano el enlace directo a sus datos derivados





# Ejercicio guiado: Implementando HATEOAS



# Ejercicio guiado

## Implementando HATEOAS

La tienda All You Need Is Music Spa hará su apertura en un par de meses, por lo que necesita un sistema de consultas de sus productos y ha decidido contratar a un programador full stack developer para esto. Luego de unos días el programador ya tiene la aplicación en el lado del cliente y ha creado la base de datos. Para hacer las prueba registró 4 guitarras y ahora necesita proceder con la creación de la API REST aplicando HATEOAS.

Si no sabes de guitarras no tienes porqué preocuparte, pues no profundizaremos demasiado en la materia. Esto te servirá como simulación a un caso real, donde tendrás que crear o mantener APIs sobre rubros o áreas desconocidas e inevitablemente te tocará aprender un poco sobre el tema en pro de la lógica que debes aplicar en tu servicio.



# Ejercicio guiado

## Implementando HATEOAS

Para el ejercicio guiado se debe descargar el **Archivo Apoyo - Servidor base Parte I** de esta sesión en donde encontrarás un servidor base hecho con Express y una carpeta data con un archivo JavaScript que exporta un arreglo de objetos que contienen las 4 guitarras.

Para este ejercicio usarás el archivo **guitarras.js** para simular lo que te devolvería una consulta a una base de datos real.



# Ejercicio guiado

## Implementando HATEOAS

- **Paso 1:** Crear una función de nombre HATEOAS.
- **Paso 2:** Usar el método “map” para crear un arreglo de objetos, utilizando los datos de la guitarra. El objetivo es devolver solo 2 propiedades: name y href, en donde “href” deberá ser la siguiente ruta dinámica:  
<http://localhost:3000/guitarra/<id de la guitarra>>
  - La función debe retornar el arreglo creado.
- **Paso 3:** Crear una ruta **GET /guitarras** que devuelva un objeto con una propiedad “guitarras” cuyo valor sea la ejecución de la función HATEOAS creada en el paso 1.



# Ejercicio guiado

## Implementando HATEOAS

```
4  ▾ {
5  ▾   "guitarras": [
6  ▾     {
7         "name": "Dean 350f",
8         "href": "http://localhost:3000/guitarra/1"
9       },
10 ▾    {
11        "name": "Ibanez RG8570Z",
12        "href": "http://localhost:3000/guitarra/2"
13      },
14 ▾    {
15        "name": "Epiphone Les Paul LP-100",
16        "href": "http://localhost:3000/guitarra/3"
17      },
18 ▾    {
19        "name": "Fender American Performer",
20        "href": "http://localhost:3000/guitarra/4"
21      }
22   ]
23 }
```



# Ejercicio guiado

## *Implementando HATEOAS*

Ahora sigue los pasos para agregar una función y ruta que permita devolver el detalle de una guitarra consultando el siguiente endpoint:

<http://localhost:3000/guitarra/<id>>



# Ejercicio guiado

## Implementando HATEOAS

- **Paso 1:** Crear una función de nombre guitarra que reciba un parámetro “id” y utilice el método “find” para retornar el objeto de una guitarra según el id recibido.
- **Paso 2:** Crear una ruta **GET /guitarra/:id** que recibirá de forma dinámica el id de una guitarra que se desea consultar.



# Ejercicio guiado

## Implementando HATEOAS

- **Paso 3:** Almacenar en una constante el id recibido como parámetro en la ruta.
- **Paso 4:** Devolver la ejecución de la función “guitarra” pasando como argumento el id recibido en la ruta.





# Ejercicio guiado

## Implementando HATEOAS

```
4  {  
5    "id": 1,  
6    "name": "Dean 350f",  
7    "brand": "DEAN",  
8    "model": "350f",  
9    "body": "Stratocaster",  
10   "color": "Dark blue",  
11   "pickups": "Single Coil",  
12   "strings": 6,  
13   "value": 350,  
14   "stock": 2  
15 }
```

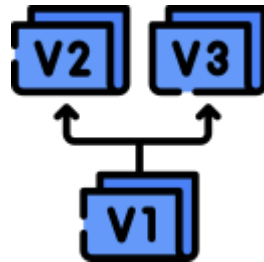


***/\* Versionamiento de APIs \*/***

# Versionamiento de APIs

Los versionamientos en las APIs existen para responder a nuevos requisitos que exigen los clientes de nuestro servicio, lo cual es muy normal de ver durante el funcionamiento de un sistema.

Por esto y muchas otras razones, se recomienda considerar toda la cantidad de casos de usos posible en el proceso inicial de creación.



# Versionamiento de APIs

Los versionamientos normalmente se representan por la “v” que podemos ver declarada como parámetro en la URL, seguida del número representativo de la versión

## Try it now!

`https://pokeapi.co/api/v2/` `pokemon/ditto`

Need a hint? Try [pokemon/ditto](https://pokeapi.co/api/v2/pokemon/ditto), [pokemon/1](https://pokeapi.co/api/v2/pokemon/1), [type/3](https://pokeapi.co/api/v2/type/3), [ability/4](https://pokeapi.co/api/v2/ability/4), or

# Versionamiento de APIs

Para aplicar este concepto en nuestro ejercicio de guitarras, realizaremos un pequeño cambio en la nomenclatura de las propiedades **"name"** y **"href"**.

Aunque pueda parecer un cambio insignificante, pero recordemos que por un carácter todo un software podría detenerse.

Cualquier cambio aunque sea minúsculo debe incluirse en una nueva versión e incluso alertar a los consumidores que en cierto periodo de tiempo la versión anterior dejará de estar vigente si fuese el caso y que deben migrar su consumo a la nueva versión lanzada a producción.

# Ejercicio guiado:

## Creando una versión de mi API REST



## Ejercicio guiado

### *Creando una nueva versión de mi API REST*

Cambiar el nombre de la ruta que ocupamos para devolver todas las guitarras por la siguiente **/api/v1/guitarras**.

Además, agregar una nueva función y una ruta para disponer la versión 2 de nuestra API, esta ruta deberá ser la siguiente **/api/v2/guitarras**.

Sigue los pasos para realizar este ejercicio:



# Ejercicio guiado

## Creando una nueva versión de mi API REST

- **Paso 1:** Cambiar el nombre de la función HATEOAS por **HATEOASV1**.

```
// Paso 1
const HATEOASV1 = () =>
  guitarras.map((g) => {
    return {
      name: g.name,
      href:
        `http://localhost:3000/guitarra/${g.id}`,
    };
  });
```





# Ejercicio guiado

## Creando una nueva versión de mi API REST

- **Paso 2:** Crear una nueva función llamada **HATEOASV2** que será una copia de la primera versión, siendo la única diferencia los nombres de las propiedades por “guitar” y “src”.

```
// Paso 2
const HATEOASV2 = () =>
  guitarras.map((g) => {
    return {
      guitar: g.name,
      src: `http://localhost:3000/guitarra/${g.id}`,
    };
  });
```



# Ejercicio guiado

## Creando una nueva versión de mi API REST

- **Paso 3:** Cambiar el nombre de la ruta **/guitarras** por **/api/v1/guitarras**, en donde utilizarás la función **HATEOASV1** para devolver el modelo de datos correspondiente a la primera versión.

```
// Paso 3
app.get("/api/v1/guitarras", (req, res) => {
  res.send({
    guitarras: HATEOASV1(),
  });
});
```



# Ejercicio guiado

## Creando una nueva versión de mi API REST

- **Paso 4:** Agregar una ruta **GET /api/v2/guitarras**, en donde utilizarás la función **HATEOASV2** para devolver el modelo de datos correspondiente a la segunda versión.

```
// Paso 4
app.get("/api/v2/guitarras", (req, res) => {
  res.send({
    guitarras: HATEOASV2(),
  });
});
```



# Ejercicio guiado

## Creando una nueva versión de mi API REST

Ahora intenta consultar la dirección

<http://localhost:3000/api/v2/guitarras> y deberás obtener lo que vemos en la siguiente imagen:

**{desafío}**  
latam\_

```
4  {  
5    "guitarras": [  
6      {  
7        "guitar": "Dean 350f",  
8        "src": "http://localhost:3000/guitarra/1"  
9      },  
10     {  
11       "guitar": "Ibanez RG8570Z",  
12       "src": "http://localhost:3000/guitarra/2"  
13     },  
14     {  
15       "guitar": "Epiphone Les Paul LP-100",  
16       "src": "http://localhost:3000/guitarra/3"  
17     },  
18     {  
19       "guitar": "Fender American Performer",  
20       "src": "http://localhost:3000/guitarra/4"  
21     }  
22   ]  
23 }
```



## Ejercicio guiado

### *Creando una nueva versión de mi API REST*

En el caso de no realizar una nueva versión al momento de modificar el contrato de datos, en un plano industrial, podría significar un impacto gigantesco considerando que las aplicaciones clientes están programadas para esperar por ejemplo “A”, y si reciben de repente “B” podría afectar de forma importante toda la lógica del sistema.



# Resumen

- Una **API REST** es un servicio o conjunto de servicios web, que permiten la transferencia de datos en un entorno de red utilizando el protocolo HTTP.
- Las APIs REST funcionan mediante la definición de verbos HTTP (GET - POST - DELETE - PUT)
- Con **HATEOAS** logramos tener un mapa claro y simple de navegar para cualquier cliente que quiera consumir los recursos que ofrezca nuestra API REST.

¿Cual es el verbo HTTP para  
obtener los datos de una  
API?



¿Cuáles son algunas de las  
buenas prácticas al momento  
de implementar una API  
REST?







## Próxima sesión...

- *Códigos de estado del protocolo HTTP*

**{desafío}**  
**latam\_**

*Academia de  
talentos digitales*

