Curso de Back-End con Node.js (Avanzado)

Clase 02

Temario

Temario

- JSON.
- APIs.
- APIs REST.
- REST – Buenas prácticas.
- Ejercicios.

JSON

JavaScript Object Notation

JSON

- JSON = JavaScript Object Notation.
- Es un formato de texto, muy simple, ideado para intercambio de datos.
- En caso de ser almacenado como un archivo, es de extensión `.json`.
- Un JSON contiene información que se guarda en *pares clave-valor*.
- Las claves son *siempre* `String` y los valores pueden ser de tipo `Number`, `String`, `Boolean`, `Array`, `Object` y `null`.
- JSON deriva de JavaScript, pero es independiente del lenguaje de programación.

JSON – Ejemplo

```
{
    "number": 1,
    "unString": "Hack Academy!",
    "Boolean": true,
    "un array": ["node", "mongo", "express"],
    "objeto": {
        "clave": "valor"
    },
    "nada": null
}
```

APIs

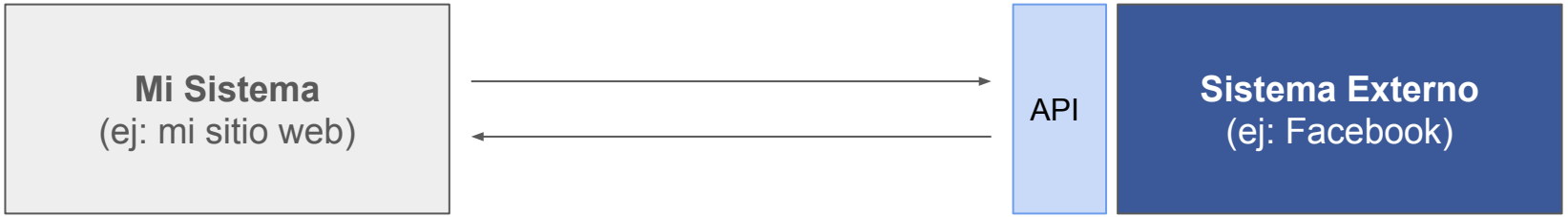
¿Qué es una API? (1/4)

Se podría decir que una API es una implementación del Patrón Fachada.

- API = *Application Programming Interface*.
- Es una interfaz que permite a 2 sistemas independientes comunicarse entre sí.
  - Uno de los sistemas **“provee”** la API (es dueño de la misma).
  - El otro **“consume”** la API.
- Quien provee la API debe especificar qué servicios se proveen y cómo se usan. La buena práctica es que toda API está acompañada de una documentación.
- 🤔 Notar que la definición anterior no habla de Internet y de hecho el concepto de API va más allá de sistemas conectados a una red. Por ejemplo, sistemas operativos como Windows o Android proveen una API que puede ser utilizada por los desarrolladores de aplicaciones para acceder a funcionalidades del sistema.

¿Qué es una API? (2/4)

Diagrama de ejemplo:



En este ejemplo, el sistema externo provee una API y nuestro sistema la consume.

El sistema externo debe especificar qué servicios se proveen y cómo se usan. En este ejemplo: https://developers.facebook.com/docs/graph-api.

¿Qué es una API? (3/4)

- Es común que la información intercambiada con una API tenga formato JSON.
- Las APIs pueden ser:
  - **Públicas**: cualquiera puede acceder a ellas. Ej: La API de Google Maps, Star Wars.
  - **Privadas**: sólo determinados actores pueden acceder a las mismas. Ej: una empresa crea un API para sus clientes o crea una API para sus sucursales.
  - **De acceso restringido**: para poder acceder a ellas es necesario autenticarse (es lo que ocurre en la mayoría de los casos). Ej: Open Exchange Rates (https://openexchangerates.org/).
  - **Gratis**: se pueden usar gratuitamente sin restricción de uso. Ej: Facebook (https://developers.facebook.com/docs/graph-api).
  - **Parcialmente gratis (freemium)**: hasta cierto uso se pueden usar gratis, luego es necesario pagar. Ej: Monkey Learn (http://www.monkeylearn.com/). Esta es una startup uruguaya!
  - **Pagas**: es necesario pagar para usarlas. Ej: Twilio (https://www.twilio.com/sms).

APIs REST

APIs REST (1/2)

Existe un tipo particular dentro de las Web APIs (también llamadas *web services*) que son las APIs REST (*REpresentational State Transfer*).

Estas APIs se caracterizan por:

- Interactuar con “recursos”, generalmente entidades del problema, identificados por una URL. Ej: `/api/users`.
- Utilizar verbos o métodos HTTP para interactuar con un recurso. Ej: GET, POST, PUT, PATCH y DELETE. → No hay que “inventar” URLs para cada acción que se quiera realizar sobre el recurso. Ej: `/api/users/borrar` vs. `/api/users/eliminar`.
- Ser *stateless*: no tener estado. Cada *request* tiene toda la información necesaria para que el servidor pueda procesar el llamado y no depende de *requests* anteriores.

APIs REST (2/2)

Gracias a lo anterior, las API REST brindan una interfaz uniforme (gran diferencia con las APIs SOAP).

En general, los datos intercambiados con una API REST están en formato JSON, pero no es un requisito obligatorio.

APIs REST – Terminología

- URL: *Uniform Resource Locator*. Como su nombre lo indica, su propósito es encontrar/dar acceso a un recurso.

- Recurso: Es una representación, ya sea en forma de objeto, de XML o JSON, de información en un sistema. Dicho recurso puede ser sujeto a lectura, escritura, actualización y/o borrado.

- Colección: Es un conjunto de recursos.

- Endpoint: Es una URL + un método HTTP.
  Ej: `[GET] /users` y `[POST] /users` son dos *endpoints* diferentes.

REST – Buenas prácticas

REST – Buenas prácticas (1/6)

1. Usar los verbos HTTP correspondientes a la acción deseada.

REST – Buenas prácticas (2/6)

2. En línea con el punto anterior, no crear URLs que indiquen la acción a realizar.

Por ejemplo, las siguientes URLs no son una buena práctica:

- `/getAllUsers`
- `/listar-users`
- `/deleteUser`
- `/crear-usuario`
- `/modificarUsuario`

La buena práctica es usar endpoints como `[GET] /users` y `[POST] /users`, respetando los verbos HTTP y usando el nombre en plural del recurso.

REST – Buenas prácticas (3/6)

3. Hacer un uso apropiado de los códigos de estado HTTP.

4. Ser consistentes con el *casing,* tanto a la hora de definir URLs como a la hora de definir nombres para variables, atributos, funciones, etc.

- Para URLs se suele usar `kebab-case`.
- Para lo demás, se suele usar `camelCase`.

REST – Buenas prácticas (4/6)

5. Usar *query params* para:

- Buscar (cuando no se puede hacer a través de la URL).
  Ej: `/users?search=Pablo`, para obtener todos los usuarios que contienen la palabra “Pablo”.
- Filtrar. Ej: `/restaurants?food=Pizza`, para obtener todos los restaurantes que venden Pizza.
- Ordenar. Ej: `/teams?sort=continent&order=1`.
- Paginar. Ej: `/teams?page=10` o `/teams?skip=30`. Este último es agnóstico al concepto de página, el cual está asociado a la presentación.

REST – Buenas prácticas (5/6)

6. Usar un prefijo de versión.

Las APIs son un “contrato” que se debe respetar para garantizar su usabilidad. Es decir, es importante que las APIs tengan un comportamiento predecible y consistente.

En caso de que un equipo de desarrollo necesite re-escribir o mejorar una API, es necesario romper dicho contrato, causando errores en quienes están consumiendo la API 😱.

REST – Buenas prácticas (6/6)

6. (continuación)

Para evitar este problema, se recomienda contar con un versionado de APIs y que puedan co-existir varias versiones de la API en simultáneo.

Cada versión de la API tendrá su propio prefijo en la URL y su propia documentación. Ejemplos:

- `/api/v1/users`
- `/api/v2/users`

Algunos usuarios consumirán la `v1` y otros consumirán la `v2`. A medida que sea posible, los usuarios de la `v1` deberían pasarse a la `v2`.

Ejercicio 1

Creación de una API de equipos de fútbol (países)

Ejercicio 1 (1/7)

- Crear un directorio y dentro de él inicializar un proyecto de Node con el comando: `npm init -y`.
- Instalar `express` como dependencia del proyecto: `npm i express`
- Crear un archivo `db.js` que simulará una base de datos a los efectos de probar los conceptos que se vieron en esta clase. En dicho archivo, colocar el contenido de este gist.
- Crear un archivo llamado `routes.js` que contendrá las rutas de la aplicación (mediante la creación de un `express.Router`).
- Crear un un archivo `teamController.js` que contendrá los *handlers* de las rutas creadas en el archivo anterior.

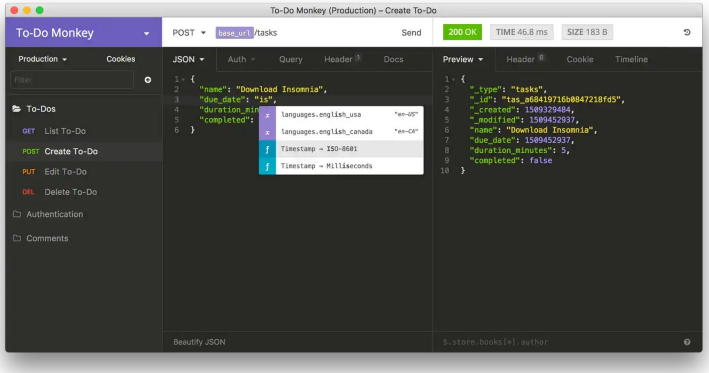
👉 Sugerencia: instalar nodemon (puede ser una instalación global) para no tener que “reiniciar” el servidor cada vez que se modifican los archivos JavaScript.

Ejercicio 1 (2/7)

Dado que a través de la barra de direcciones de un navegador sólo se pueden hacer *requests* de tipo GET, será necesario instalar un programa adicional para probar la API desarrollada.

Estos programas se llaman Cliente de APIs y permiten hacer diversos tipos de *requests* como PUT, PATCH, POST y DELETE.

Existen varias opciones como Postman o Paw, pero en este curso recomendamos usar Insomnia: https://insomnia.rest.



Ejercicio 1 (3/7)

Insomnia cuenta con una gran utilidad llamada *workspaces*.

Los *workspaces* son conjuntos de *requests* ya armados y configurados, los cuales se usan para testear *endpoints*. Además se pueden descargar y compartir en formato JSON.

Para la clase de hoy, se usará este JSON que contiene un *workspace* que se deberá importar en Insomnia: `My Workspace > Import/Export > Import Data > From URL` . Luego, seleccionar el *workspace* recién importado.

Si alguien utiliza Postman, podrá usar este otro JSON: `Import > Import From Link > Import` . Luego, seleccionar la *collection* recién importada.

Ejercicio 1 (4/7)

Una vez importado el *workspace* en Insomnia, se verán los 5 *requests* correspondientes a los 5 *endpoints* que se deberán crear en este ejercicio:

- `[GET] /teams` – Para obtener todos los *teams* en la “base de datos”.
- `[GET] /teams/:id` – Para obtener aquel *team* cuyo `id` *matchee* con el presente en la ruta del *request*.
- `[POST] /teams` – Para insertar en la “base de datos” el *team* que se pasará en el *body* del *request* (en formato JSON).
- `[DELETE] /teams/:id` – Para remover de la “base de datos” aquel *team* cuyo `id` *matchee* con el presente en la ruta del *request*.
- `[PATCH] /teams/:id` – Para alterar, con los datos presentes en el *body* del *request*, aquel *team* cuyo `id` *matchee* con el presente en la ruta del *request*.

Ejercicio 1 (5/7)

⚠️ Por defecto, Express ignorará los datos en formato JSON que lleguen al servidor dentro del *body* de un *request*.

Por lo tanto, si se quiere que el servidor reciba dichos datos, hay que especificarlo de forma explícita.

Además, es necesario *parsear* el JSON recibido para hacerlo programáticamente accesible, es decir, para crear un objeto JavaScript a partir de los datos recibidos.

Afortunadamente, esto es muy sencillo en Express. Se puede lograr usando un *middleware* llamado `express.json`. Más adelante veremos los *middlewares* con mayor detalle. Por ahora, simplemente considerar que son una función que se ejecuta automáticamente al recibir un *request*.

Ejercicio 1 (6/7)

Para usar el *middleware* `express.json` en todas las rutas de nuestra aplicación, debemos especificarlo usando la función `use`:

```
const express = require("express");
const app = express();

app.use(express.json());
```

Ahora, cuando lleguen datos en formato JSON, Express creará de forma automática un atributo `body` dentro del objeto *request* conteniendo dichos datos. 🥳

```
req.body
```

Ejercicio 1 (7/7)

⚠️ Notar que, en este ejercicio, todos los cambios realizados sobre los equipos (*teams*) residen en la memoria RAM del servidor.

Es decir, si se agrega un nuevo país al listado (ej: Uruguay) y luego se apaga el servidor, al volver a prenderlo, el listado vuelve a su estado original (el que está definido en `db.js`).

Para realizar modificaciones persistentes, habría que guardar los datos en una base de datos.

EXTRA

Ejercicio 2

Consumir una API de un tercero (Mailchimp)

EXTRA

Ejercicio 2 (1/2)

1. Crear una carpeta llamada `ejercicio_mailchimp`.
2. Inicializar un proyecto con el comando `npm init`.
   Pueden ignorar todas las preguntas haciendo “Enter” en cada una.
   Verificar que se haya creado el archivo `package.json`.
3. Instalar `express`.
4. Crear un archivo: `index.js`.
5. Crear el *endpoint* `[POST] /newsletter` que reciba dentro del `body` un JSON conteniendo un nombre, un apellido y un email.
   Luego, dichos datos deben ser enviados y guardados en Mailchimp, una plataforma para newsletter (envío masivo de emails). [Ver próxima diapositiva].
   A su vez, este *endpoint* debe responder con un mensaje de éxito o de error, dependiendo del resultado de la operación.

Ejercicio 2 (2/2)

EXTRA

6. Crearse una cuenta en Mailchimp y darle una vichada a la documentación para desarrolladores. Deberán conseguir una *API Key* (que es una especie de contraseña para poder acceder a la API de Mailchimp).

7. También deberán crear un *Audience* (Audiencia), la cual tendrá asociada una *Audience Id*. Este es un dato que necesitarán para el siguiente punto.

8. Desde Node.js se debe llamar a la API de Mailchimp y pasarle los datos del usuario que se quiere anotar en el *newsletter*. Esto lo podrán hacer de diversas maneras. Node.js ya trae un módulo llamado `https` que les puede servir (aunque es de "bajo nivel"). De lo contrario pueden instalar algún módulo de un tercero como Axios o, incluso mejor, una librería específica para usar Mailchimp.

9. Antes de enviar los datos a Mailchimp, realizar algún tipo de validación de los mismos. Por ejemplo: validar que ningún dato esté vacío y validar que el email tenga un formato válido.