Clase IV
PROGRAMAS DE CONSULTAS

IEEE ITBA
STUDENT BRANCH

Índice

Una ojeada a la Programación Funcional **1**

Métodos y Códigos de estado de HTTP **7**

- Métodos 8
- Códigos de estado 10
  - Respuestas informativas (1xx) 10
  - Respuestas satisfactorias (2xx): 10
  - Respuestas de redirección (3xx) 11
  - Respuestas de error del cliente (4xx) 11
  - Respuestas de error del servidor (5xx) 12

Desarrollo de una API básica **12**

- Introducción a las APIs y su importancia 12
- Ejemplo de uso de una API 13
- Librería de TypeScript para crear una API 14
- Consideraciones previas 14
- Modelado de los equipos en la API 14
- Modelado de la aplicación 18
- Revisión de endpoints 21

Una ojeada a la Programación Funcional

Cuando explicamos el for, mencionamos varias maneras de ciclar con este. Hoy, veremos otra más, el *forEach*. El *forEach* comparte la misma idea que el *for … of*. La diferencia es que el *forEach* se acerca a lo que se llama programación funcional.

Una gran diferencia entre estos dos ciclos, es que el *forEach* pertenece exclusivamente a la familia de arreglos (Arrays), mientras que el *for … of* pertenece a la familia de los objetos iterables (por ejemplo, una lista). Adicionalmente, el *for … of* es relativamente nuevo al mundo de JavaScript y tiene algunas ventajas. Se pueden hacer *for … of* de Arrays, Maps, Sets, Strings, HTMLCollection, etc. También, el *for … of* es más rápido que el *forEach*.

Sin embargo, el *forEach* nos servirá para introducir el tema Programación Funcional (FP sus siglas en inglés), la cual nos simplificará muchas de las tareas, que veremos a continuación. Con la programación funcional, las funciones son una variable más. También, pueden ser anónimas y declaradas como una variable.

Supongamos que tenemos un arreglo de Alumnos, del colegio donde trabajamos,con notas de cursadas de un curso de programación, que nos fue otorgado por la dirección del colegio donde trabajamos:

```
interface IAlumno {
    id: number;
    nombre: string;
    notaCursada: number;
}

const juan: IAlumno = {id: 1, nombre: 'Juan', notaCursada: 7},
      pedro: IAlumno = {id: 2, nombre: 'Pedro', notaCursada: 5},
      ariel: IAlumno = {id: 3, nombre: 'Ariel', notaCursada: 6},
      micaela: IAlumno = {id: 4, nombre: 'Micaela', notaCursada: 8},
      martina: IAlumno = {id: 5, nombre: 'Martina', notaCursada: 3},
      camila: IAlumno = {id: 6, nombre: 'Camila', notaCursada: 9},
      diana: IAlumno = {id: 7, nombre: 'Diana', notaCursada: 7};



let alumnos: readonly IAlumno[] = [juan, pedro, ariel, micaela, martina,
camila, diana];
```

De ese grupo de alumnos, nos pidieron obtener un arreglo con los que aprobaron el curso (nota > 6). Para eso, podemos hacerlo de múltiples maneras. Una de ellas es el *for ... of*:

```
import IAlumno from "../Interfaces/IAlumno";
import alumnos from "../Datos/alumnos";

let alumnosAprobados: IAlumno[] = [];
for (const alumno of alumnos) {
    if (alumno.notaCursada >= 6) {
        alumnosAprobados.push(alumno);
    }
}

console.log(alumnosAprobados);
```

El resultado tendría a los alumnos Juan, Ariel, Micaela, Camila y Diana:

```
[
  { id: 1, nombre: 'Juan', notaCursada: 7 },
  { id: 3, nombre: 'Ariel', notaCursada: 6 },
  { id: 4, nombre: 'Micaela', notaCursada: 8 },
  { id: 6, nombre: 'Camila', notaCursada: 9 },
  { id: 7, nombre: 'Diana', notaCursada: 7 }
]
```

Ahora, para transformar el *for ... of* a un *forEach*, lo único que hay que tener en cuenta es que el *forEach* toma como argumento una función de *callback*, que es una función que aplicará sobre cada elemento de la iteración. Esta función *callback*, a su vez, tomaría como argumento cada elemento *alumno*. Adicionando, el forEach se llama desde el arreglo como si fuera un método del mismo. Después, no cambia el resto del formato con respecto al *for ... of*:

```
import IAlumno from "../Interfaces/IAlumno";
import alumnos from "../Datos/alumnos";

let alumnosAprobados: IAlumno[] = [];
alumnos.forEach(alumno => {
    if (alumno.notaCursada >= 6) {
        alumnosAprobados.push(alumno);
    }
});
```

```
console.log(alumnosAprobados);
```

Nuevamente, obtendremos el mismo resultado.

Entonces, para terminar de resumir, lo que hacen los métodos funcionales que veremos hoy es aplicar una función *callback*, que recibe un argumento, o muchos, y realizan una acción por sobre cada elemento de un arreglo.

Por último, podemos mostrar el mismo ejemplo de recorrer un arreglo, pero con el *for* tradicional. Aquí notaremos que, no solo es otra forma de escribir lo mismo, sino que es más difícil cometer errores que de forma funcional. Esto es porque con la programación funcional, uno define el QUÉ queremos obtener, mientras que con programación imperativa, a donde pertenece este *for*, uno define el CÓMO obtener el resultado:

```
import IAlumno from "../Interfaces/IAlumno";
import alumnos from "../Datos/alumnos";

let alumnosAprobados: IAlumno[] = [];
for( let i: number = 0; i < alumnos.length; i++ ) {
    if (alumnos[i].notaCursada >= 6) {
        alumnosAprobados.push(alumnos[i]);
    }
}

console.log(alumnosAprobados);
```

Notamos que para poder iterar por sobre los alumnos, tenemos que crear una variable i, la cual tenemos que incrementar y tenemos que indicar cuándo queremos comenzar y dejar de iterar. También, nos encontramos que, para referirnos al alumno en cada iteración, tenemos que hacer *alumnos[i]*. Todo esto, en conjunto, puede hacer que uno mismo cometa errores más fácilmente.

Ahora, nos pidieron desde la dirección que ese grupo de alumnos, para facilitarle la lectura del mismo al rector, le asignemos un campo *aprobado*, que indica si el alumno fue o no aprobado en el curso. Para ello, tendríamos que modificar el conjunto de alumnos, que vimos que no está permitido (cada alumno es una constante y el arreglo es *readonly*):

1. Con un *for … of* la implementación es un tanto engorrosa. Tendríamos que crear un nuevo arreglo vacío de tipos AlumnoModificado (nueva interfaz), hacer un *for … of* donde, por cada alumno de alumnos, cree una nueva variable

AlumnoModificado, asignando el campo *aprobado*, cuyo valor sea *true* o *false* de acuerdo a la nota del alumno y; añada esta nueva variable al arreglo vacío de tipos Alumno Modificado. Además, de tener que hacer el nuevo arreglo de alumnos modificados, habría que crear otro más que le de el formato de *readonly* para que sea de solo lectura. Hasta ahí, tenemos que realizar muchas modificaciones:

```
import IAlumno from "../../Interfaces/IAlumno";
import alumnos from "../../Datos/alumnos";

interface AlumnoModificado extends IAlumno {
    aprobado: boolean;
}

let alumnosModificadosDificil: AlumnoModificado[] = [];
for( let alumno of alumnos ) {
    alumnosModificadosDificil.push({ ...alumno, ...{aprobado:
alumno.notaCursada > 6}});
}
let alumnosModificadosDificilSoloLectura: readonly  AlumnoModificado[] =
alumnosModificadosDificil;


console.log(alumnosModificadosDificilSoloLectura);
```

2. En cambio, con el nuevo método funcional que veremos, el *map*, podremos **mapear** los objetos de un array de un tipo de datos a otro, y retornar un nuevo arreglo con estos, todo en una sola línea. La función map toma, nuevamente, una función *callback*, la cual recibe un argumento (alumno), y dentro del *callback*, realiza el **mapeo** de un tipo de datos al otro. En este caso, sería agregando el campo *aprobado* al final:

```
import IAlumnoModificado from "../../Interfaces/IAlumnoModificado";
import alumnos from "../../Datos/alumnos";
import IAlumno from "../../Interfaces/IAlumno";

const alumnosModificados: readonly IAlumnoModificado[] = alumnos.map(
(alumno: IAlumno) =>
                                                                ({
...alumno, ... {aprobado: alumno.notaCursada > 6}})
);

console.log(alumnosModificados);
```

```
export default alumnosModificados;
```

En definitiva, el arreglo que nos queda es el siguiente:

```
[
  { id: 1, nombre: 'Juan', notaCursada: 7, aprobado: true },
  { id: 2, nombre: 'Pedro', notaCursada: 5, aprobado: false },
  { id: 3, nombre: 'Ariel', notaCursada: 6, aprobado: false },
  { id: 4, nombre: 'Micaela', notaCursada: 8, aprobado: true },
  { id: 5, nombre: 'Martina', notaCursada: 3, aprobado: false },
  { id: 6, nombre: 'Camila', notaCursada: 9, aprobado: true },
  { id: 7, nombre: 'Diana', notaCursada: 7, aprobado: true }
]
```

Tenemos que tener presente que el nuevo método *map* no permite más que pasar de un tipo de datos a otro. Si ahora queremos, además de eso, quedarnos únicamente con los alumnos aprobados, nos gustaría tener un método que haga todo esto solo, sin necesidad de un *for … of*. No es casualidad que este método exista, y sea muy utilizado. Nuevamente, el método recibe un *callback*, el cual es una función que tiene de argumento a un elemento del arreglo y la función en sí describe con qué elementos quedarse. Este método se llama *filter*:

```
import IAlumnoModificado from "../../Interfaces/IAlumnoModificado";
import alumnosModificados from "../map/conMap";

let alumnosModificadosAprobados: readonly IAlumnoModificado[] =
alumnosModificados.filter( (alumno: IAlumnoModificado) =>
alumno.aprobado);

console.log(alumnosModificadosAprobados);
```

Nos terminaría quedando el siguiente resultado:

```
[
  { id: 1, nombre: 'Juan', notaCursada: 7, aprobado: true },
  { id: 4, nombre: 'Micaela', notaCursada: 8, aprobado: true },
  { id: 6, nombre: 'Camila', notaCursada: 9, aprobado: true },
  { id: 7, nombre: 'Diana', notaCursada: 7, aprobado: true }
]
```

Volviendo al arreglo inicial de Alumnos, nos gustaría obtener el promedio de las notas. Para esto, podemos iterar mediante un *for … of*, dividir por la longitud del arreglo y obtener el resultado. Si bien es una tarea fácil, otro tipo de *reducciones* pueden llegar a ser tediosas mediante este método. Sin embargo, los métodos funcionales nos vienen a salvar de nuevo, con el método *reduce*. Este método recibe una función *callback*, que tiene varios argumentos, muchos de ellos opcionales.

El primer argumento es un acumulador que representa el resultado parcial que queremos obtener al final de la iteración. El segundo argumento es el valor actual que se incorporará a la iteración. El tercero es el índice del valor actual en el arreglo, y el cuarto es el arreglo que llamó al método (o solo sus propiedades).

Para obtener el promedio, primero debemos hacer un mapeo , o *map*, de los valores de notaCursada del arreglo de alumnos a un arreglo de números, *number*, para luego realizar la reducción, *reduce*, al promedio. Podemos hacer esto en dos partes:

```
import alumnos from "../../Datos/alumnos";

const notas: number[] = alumnos.map((alumno) => alumno.notaCursada);
const promedio: number = notas.reduce((promedio, nota) => promedio +
nota / notas.length, 0);

console.log(promedio);
```

En el primer paso, creamos un arreglo de números con las notas de los alumnos. En el segundo paso, usamos el método *reduce* para calcular el promedio, comenzando con un valor inicial de 0 y sumando cada nota dividida por la longitud del arreglo de notas.

Cabe destacar que en algunos lenguajes de programación existe una implementación nativa de estos dos métodos, llamada *mapReduce*. También es importante mencionar que el segundo argumento del método *reduce* es el valor inicial del acumulador, en nuestro caso la variable *promedio* comienza en 0.

Métodos y Códigos de estado de HTTP

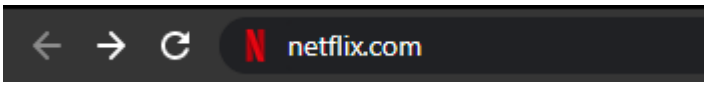
El objetivo principal de hoy es aprender acerca del uso de TypeScript en el mundo actual de Internet.

Aunque muchas personas lo conocen por su uso en la creación de sitios web, especialmente en lo que se conoce como "frontend", nosotros veremos cómo se puede utilizar en el "backend". Esta es la parte de la página web donde se procesa y transforma toda la información.

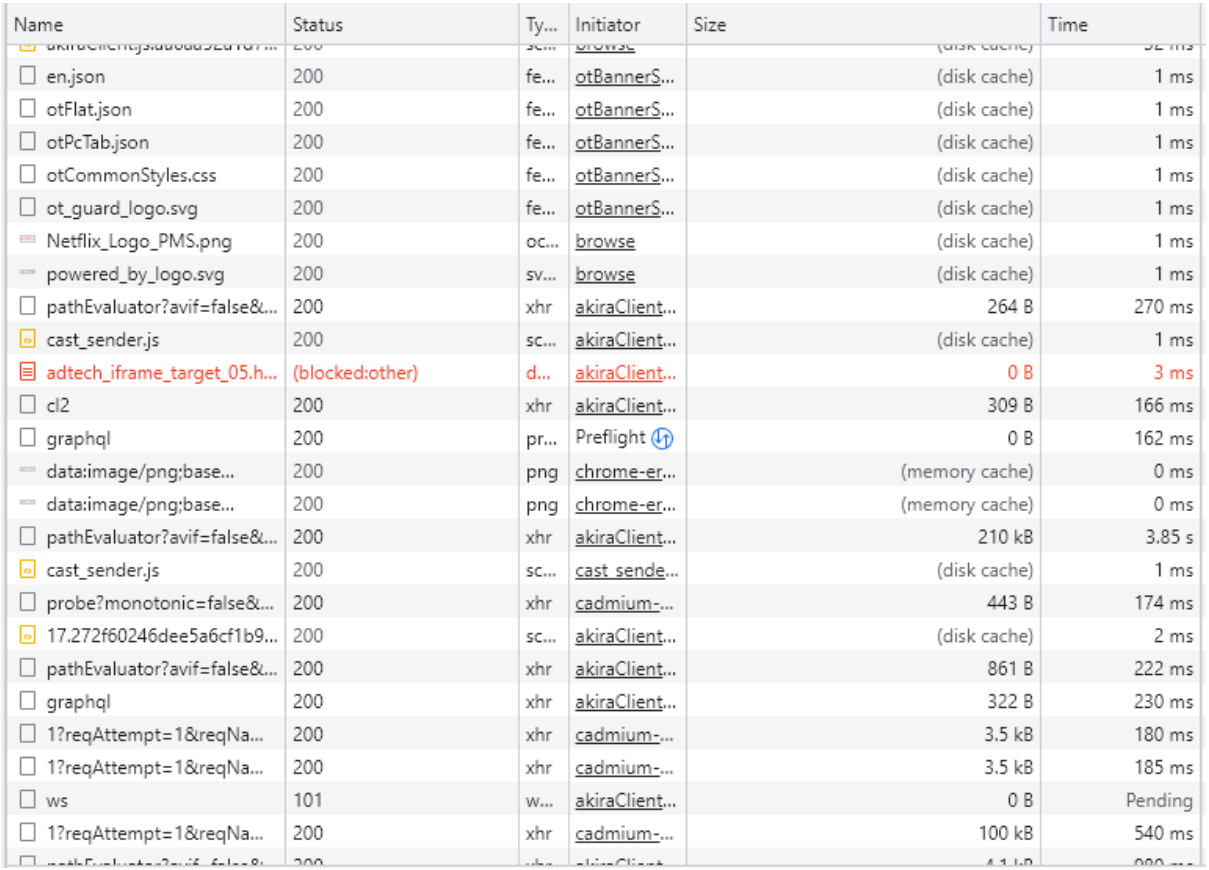
Para que todo funcione sin problemas, un grupo de ingenieros y expertos en Internet (IETF) estableció ciertas normas y acuerdos llamados RFC, que todos deben seguir para comunicarse de manera efectiva y evitar errores de interpretación. Puede encontrar todos los RFC en una página específica.

Cuando ingresamos una dirección web en el navegador, lo que sucede detrás de escena es que se envían solicitudes a los servidores de esa página web para obtener su contenido, como texto, imágenes, videos, entre otros. Todo esto se maneja a través del protocolo de comunicación HTTP o HTTPS. Existen más protocolos de comunicación, como FTP, pero no serán necesarios para este curso.

Un claro ejemplo de solicitudes HTTP/HTTPS es cuando queremos acceder a Netflix, por ejemplo:



Cuando uno apreta Enter, en lo que parece ser un instante, se le pide al servidor de Netflix que provea información, de varias películas, series, y de archivos que manejan la página:



Name	Status	Ty...	Initiator	Size	Time
en.json	200	fe...	otBannerS...	(disk cache)	1 ms
otFlat.json	200	fe...	otBannerS...	(disk cache)	1 ms
otPcTab.json	200	fe...	otBannerS...	(disk cache)	1 ms
otCommonStyles.css	200	fe...	otBannerS...	(disk cache)	1 ms
ot_guard_logo.svg	200	fe...	otBannerS...	(disk cache)	1 ms
Netflix_Logo_PMS.png	200	oc...	browse	(disk cache)	1 ms
powered_by_logo.svg	200	sv...	browse	(disk cache)	1 ms
pathEvaluator?avif=false&...	200	xhr	akiraClient...	264 B	270 ms
cast_sender.js	200	sc...	akiraClient...	(disk cache)	1 ms
adtech_iframe_target_05.h...	(blocked:other)	d...	akiraClient...	0 B	3 ms
cl2	200	xhr	akiraClient...	309 B	166 ms
graphql	200	pr...	Preflight	0 B	162 ms
data:image/png;base...	200	png	chrome-er...	(memory cache)	0 ms
data:image/png;base...	200	png	chrome-er...	(memory cache)	0 ms
pathEvaluator?avif=false&...	200	xhr	akiraClient...	210 kB	3.85 s
cast_sender.js	200	sc...	cast_sende...	(disk cache)	1 ms
probe?monotonic=false&...	200	xhr	cadmium-...	443 B	174 ms
17.272f60246dee5a6cf1b9...	200	sc...	akiraClient...	(disk cache)	2 ms
pathEvaluator?avif=false&...	200	xhr	akiraClient...	861 B	222 ms
graphql	200	xhr	akiraClient...	322 B	230 ms
1?reqAttempt=1&reqNa...	200	xhr	cadmium-...	3.5 kB	180 ms
1?reqAttempt=1&reqNa...	200	xhr	cadmium-...	3.5 kB	185 ms
ws	101	w...	akiraClient...	0 B	Pending
1?reqAttempt=1&reqNa...	200	xhr	cadmium-...	100 kB	540 ms

Métodos

Estos pedidos se pueden separar en dos partes: manejo de la página web y acceso a la base de datos. Los pedidos, tienen un método. Estos pueden variar, dependiendo de qué es lo que necesita uno del servidor sobre un servicio. Si queremos obtener una imagen, va a ser distinto el llamado a si queremos borrarla o modificarla. Algunos de estos métodos son:

- **GET**: Se utiliza para acceder a un recurso específico. Por ejemplo un archivo .html, o un archivo .jpg, .gif, .txt, etc.

- **POST**: También se usa para solicitar un recurso, al igual que GET, pero se usa cuando además se envía información y requiere alguna acción por parte del servidor web. Por ejemplo al llenar un formulario de inscripción, que en caso de haber sido llenado correctamente recibiremos la página de bienvenida y en caso de error una página indicando que hubo un error.

- **PUT**: Se suele utilizar para el reemplazo de un recurso por otro. Por ejemplo, pasa cuando editamos información de nuestro perfil de Facebook, Instagram, etc.

- **DELETE**: Por traducción directa, implica el borrado de un recurso especificado.

Nota: Los métodos PUT y DELETE no se suelen usar con HTTP/HTTPS, ya que para eso se usan otros protocolos más eficientes, como lo es FTP que, reiteramos, no cubre el curso.

También, pueden notar que hay un código de status o estado para cada pedido. Notamos que están el 200, el 101, y uno que parece estar bloqueado, por algún error. Los códigos de estado son vitales para la comunicación a través de internet, porque son indicadores de qué sucedió con un envío.

Códigos de estado

Tenemos cinco clases de mensajes, donde cada una tiene un rango de códigos distintos.

Respuestas informativas (1xx)

Estos estados nos indican que nuestro pedido fue recibido y entendido por el servidor. Se utilizan cuando el servidor tiene que hacer un procesamiento que tome más tiempo, de forma tal que el cliente no se impaciente, si lo explicamos en pocas palabras. Tenemos estados para:

- **continuación** (100): Se confirma que el pedido es válido. Se suele utilizar para cuando hay que enviar grandes cantidades de información, para hacerlo solo cuando el servidor confirma que la va a poder aceptar.

- **cambio de protocolo** (101): El cliente le pide al servidor un cambio de protocolo de comunicación (no se verá en este curso qué tipos de protocolo hay), y el servidor confirma que lo hará.

- **procesando** (102): hay algunos pedidos que pueden contener muchos subpedidos que tengan que ver con manejo de archivos, lo que hace que tome un tiempo largo para completar el pedido inicial. Por lo tanto, el servidor envía este código de estado para indicar que se encuentra procesando el pedido, pero

que aún no tiene una respuesta. De esta forma, el cliente no se “impacienta” y cancela el pedido.

- **early hints** (103): es un estado de respuesta informativa que permite al servidor enviar al cliente una pequeña cantidad de información antes de enviar la respuesta final, lo que puede mejorar el rendimiento y la eficiencia en la comunicación HTTP.

Respuestas satisfactorias (2xx):

La acción fue recibida, entendida y aceptada por el servidor. Algunos de los estados de éxito son:

- **OK** (200): Es la respuesta estándar de éxito en la comunicación HTTP. El tipo de respuesta depende exclusivamente del método HTTP utilizado, que veremos luego.
- **creado** (201): El pedido fue realizado, creando un nuevo recurso.
- **no hay contenido** (204): El servidor terminó de procesar el pedido, para el cual no hay respuesta. Esto puede pasar, por ejemplo, cuando uno envía un pedido para borrar cierto recurso. Al borrarse, el recurso ya no se encuentra más, por lo que el servidor envía 204.
- **contenido parcial** (206): El servidor solo envía parte del recurso o contenido, debido a que el cliente le pidió un cierto rango de este. Se utiliza mucho con la continuación de descargas, o descargas en simultáneo de un recurso dividido en múltiples subrecursos.

Respuestas de redirección (3xx)

El cliente debe tomar acciones para completar con el pedido. Se utilizan para la redirección URL (para generalizar, redirección del link). Algunos estados pueden ser:

- **múltiples posibilidades** (300): Indica que hay múltiples opciones de redirección para un mismo recurso. El cliente decide. Un ejemplo, es para presentar múltiples formatos de video.
- **movido permanentemente** (301): El recurso pedido se movió a otra URL. Desde ahora en adelante, usar esa.

Respuestas de error del cliente (4xx)

Estos códigos de estado se utilizan para indicar que ha ocurrido un error y fue causado por el cliente. Por lo general, también se incluye una descripción del error. Algunos estados:

- **mal pedido** (400): El servidor no quiere o no puede procesar el pedido, debido a un problema del cliente. Probablemente tenga que ver con algún error de sintaxis, tamaño de pedido muy grande, etc.
- **no autorizado** (401): El cliente no se encuentra autenticado, por lo que no tiene credenciales para realizar el pedido.
- **prohibido** (403): Similar al anterior, pero con la diferencia de que el cliente se encuentra autenticado, pero no tiene permisos suficientes para obtener el recurso.
- **no encontrado** (404): similar a **no hay contenido**, pero esta vez, el pedido fue solicitando un recurso en particular que no debería estar vacío. Se pidió un recurso inexistente en el servidor.

Respuestas de error del servidor (5xx)

- **error interno del servidor** (500): mensaje de error genérico, que se suele usar cuando ocurre algo inesperado y no hay un mensaje que sea acorde.
- **no implementado** (501): el servidor no conoce el pedido, o carece de la capacidad para poder llevarlo a cabo. Por ejemplo, cuando hay funcionalidades a implementar a futuro, se suele usar el código de error 501.
- **servicio no disponible** (503): el servidor no pudo llevar a cabo el pedido, porque está sobrecargado o en mantenimiento.
- **tiempo de espera de la puerta de enlace superado** (504): implica que nuestro intermediario, el servidor que actuaba como puerta de enlace (o proxy), no pudo obtener una respuesta a tiempo del servidor al que le hizo el pedido.

Desarrollo de una API básica

Introducción a las APIs y su importancia

Una API (Application Programming Interface) es un programa de consultas, que permite la comunicacion entre diferentes sistemas.

Las APIs son esenciales para el desarrollo de aplicaciones modernas, ya que permiten la integración de diferentes sistemas, permitiendo compartir datos y funcionalidades de manera segura y controlada.

Las API pueden ser de distintos tipos, desde una API de servicio web que ofrece acceso a una base de datos, hasta una API de plataforma que permite la integración de aplicaciones de terceros. También pueden ser públicas o privadas, dependiendo de su finalidad y seguridad.

Ejemplo de uso de una API

Para introducir el tema, tenemos un dataset de los Equipos de fútbol Rankeados en Junio del 2022:

Ranking	Team	Country	Points Scored	1 year change	previous point scored	symbol change
65	1. FC KÃ¶l	Germany	1668	160	1529	+
34	1. FC Unio	Germany	1748	32	1665	+
967	12 de Oct	Paraguay	1352	270	1390	-
265	1Âº de Ag	Angola	1512	11	1518	-
2530	1Âº de Ma	Angola	1218	24	1218	-
1189	2 de Mayc	Paraguay	1326	8	1326	-
1213	3 de Febr	Paraguay	1324	10	1324	-
727	AaB	Denmark	1388	12	1383	+
2141	Aalesund	Norway	1248	329	1225	+
1873	Aarau	Switzerlan	1266	7	1266	-
888	Aberdeen	Scotland	1361	419	1441	-
1638	Aberystw	Wales	1286	145	1298	-
1323	Abha Club	Saudi Arabi	1312	41	1306	+
945	Abia Warr	Nigeria	1354	151	1372	-
2020	ABM Gala	Vanuatu	1255	1	1255	-
1538	Aboomos	Iran	1292	32	1292	-
2014	AC Horser	Denmark	1255	0	1255	-
2139	AC Kuya S	Congo DR	1249	33	1250	-
2712	AC LALA F	Venezuela	1180	174	1220	-
2267	AC Mamal	Mali	1241	37	1241	-
6	AC Milan	Italy	1900	11	1831	+
1521	AC Oulu	Finland	1293	278	1270	+
2299	AC Range	Denmark	1238	90	1234	+
2259	Academia	Peru	1241	729	1295	-
1128	Academia	Ivory Coast	1332	256	1362	-
1811	Academia	Venezuela	1270	9	1270	-
2639	Academia	Puerto Ricc	1201	12	1201	-
1237	AcadÃ©m	Portugal	1322	6	1322	-
1774	AcadÃ©m	Angola	1272	2	1272	-
1963	AcadÃ©m	Angola	1259	421	1234	+
1591	Accra Gre	Ghana	1288	114	1279	+
2070	Accra Lion	Ghana	1252	37	1250	+
2641	ACS FC Ac	Romania	1200	1615	1344	-
1779	ACS Poli T	Romania	1272	4	1272	-
1139	AD Grecia	Costa Rica	1331	112	1319	+
1643	AD Guana	Costa Rica	1285	202	1300	-
2541	AD Munic	Costa Rica	1217	23	1217	-

Los datos en sí, sueltos como están, pueden no tener un significado interesante.

Para lo que sirve este dataset es para hacerle consultas como las siguientes:

- ¿Está un equipo dado dentro del dataset? Si es así, ¿Cuáles son sus datos?

- ¿Qué equipos se encuentran en el top 10?
- ¿Cuál es el equipo cuyo cambio positivo fue mayor?
- ¿Cuál es el top N de los equipos mejor rankeados?

Entre muchas otras.

Ahora que sabemos las normas de comunicación en Internet, vamos a conocer una librería de TypeScript para poder crear un programa de consultas a través de internet, que nos podrá responder estas preguntas que armamos, y cualquier persona que tenga acceso a esta aplicación también podrá.

Supongamos que también se podrían hacer llamados, no solo de consulta, sino que también para añadir datos, borrarlos o modificarlos. Si bien, en este dataset de ejemplo, no habría que agregar valores ficticios, lo haremos a modo de ejemplo.

Librería de TypeScript para crear una API

La librería se llama Express, y se puede bajar a través de *npm*, como hicimos con otras dependencias. Express es un framework minimalista, rápido y escalable, por lo que en poco tiempo y líneas de código vamos a tener una aplicación andando.

Consideraciones previas

Antes de comenzar, tenemos que pensar en dónde podría correr la aplicación (actualmente, de forma local), cómo hacerle consultas a este archivo de Excel (utilizaremos la librería de la clase pasada, *danfojs*), y qué métodos *http* vamos a utilizar para realizar estas consultas y modificaciones.

Para los métodos *http*, podríamos pensar lo siguiente:

- En caso de una consulta, el método es *GET*, ya que estaríamos accediendo a un recurso o varios.
- En caso de la creación de un nuevo equipo, el método sería *POST*.
- En caso de la modificación de un equipo ya existente, el método sería *PUT*.
- En caso de querer borrar un equipo, el método sería *DELETE*.

Modelado de los equipos en la API

En un principio, armaremos un modelado de los equipos:

```
export default interface IEquipo {
    club: string;
    ranking: number;
    pais: string;
    puntajeActual: number
    puntajeAnterior: number;
    cambioPuntaje: number;
    cambioPuntajeSigno: string;
    toJson: () => Partial<IEquipo>;
}
```

Recordando clases anteriores, la interfaz IEquipo es una representación abstracta de un equipo y define los métodos y propiedades que deben ser implementados por cualquier clase que la implemente. Esta interfaz permite que las clases que la implementan sean utilizadas de manera intercambiable y se pueda trabajar con ellas de manera genérica, lo que facilita la modularidad y escalabilidad del código.

Luego, implementaremos este modelo:

```
import IEquipo from "../Interfaces/IEquipo";

export default class Equipo implements IEquipo {
    private readonly _club: string;
    private readonly _pais: string;
    private readonly _puntajeActual: number;
    private readonly _puntajeAnterior: number;
    private readonly _ranking: number;

    constructor(club: string, pais: string, puntajeActual: number,
puntajeAnterior: number, ranking: number) {
        this._club = club;
        this._pais = pais;
        this._puntajeActual = puntajeActual;
        this._puntajeAnterior = puntajeAnterior;
        this._ranking = ranking;
    }

    get club(): string {
        return this._club;
    }

    get pais(): string {
```

```
        return  this._pais;
    }

    get puntajeActual(): number {
        return this._puntajeActual;
    }

    get ranking(): number {
        return this._ranking;
    }

    get puntajeAnterior(): number {
        return this._puntajeAnterior;
    }

    get cambioPuntaje(): number {
        return Math.abs(this.puntajeActual - this.puntajeAnterior);
    }

    get cambioPuntajeSigno(): string {
        const cambioPuntaje = this.cambioPuntaje;
        return cambioPuntaje > 0 ? '+': (cambioPuntaje < 0 ? '-' : '');
    }

    toJson(): Partial<IEquipo> {
        return {
            club: this._club,
            pais: this._pais,
            puntajeActual: this._puntajeActual,
            puntajeAnterior: this._puntajeAnterior,
            cambioPuntaje: this.cambioPuntaje,
            ranking: this._ranking,
            cambioPuntajeSigno: this.cambioPuntajeSigno,
        };
    }

}
```

A continuación, haremos el controlador de equipos, con funciones de lectura, obtener datos de un equipo, obtener los primeros N equipos más valorados, obtener la edad promedio de los primeros N equipos más valorados; utilizando la librería de *danfojs*:

```
import * as dfd from 'danfojs-node';
import Equipo from "../Clases/Equipo";
```

```
import * as dfd from "danfojs-node";
import Equipo from "../Clases/Equipo";

// Los campos (columnas) de la tabla de equipos de futbol. No
visualizaremos todos los campos en el ejemplo
enum IndiceDato {
    Ranking, Team, Country, PointsScored, YearChange,
    PPS, SymbolChange
}

export default class Equipos {
    // Danfo tiene problemas con el tipo DataFrame en readExcel
    private _equipos: any;

    public leerEquipos = async (nombreArchivo: string): Promise<void |
Error> => {
        try {
            this._equipos = await dfd.readExcel(nombreArchivo);
        } catch (e) {
            throw new Error("Ha fallado la lectura del archivo");
        }
    };

    private crearNuevoEquipo(equipoArreglo: any[]): Partial<Equipo> {
        return new Equipo(equipoArreglo[IndiceDato.Team],
equipoArreglo[IndiceDato.Country],
equipoArreglo[IndiceDato.PointsScored],
            equipoArreglo[IndiceDato.PPS],
equipoArreglo[IndiceDato.Ranking]);
    }

    public obtenerEquipo = (equipo: string): Partial<Equipo> | null => {
        if (this._equipos.Team.str.includes(equipo)) {
            const equipoObtenido =
this._equipos.query(this._equipos.Team.eq(equipo));
            const equipoObtenidoDatos = equipoObtenido.values[0];
            return this.crearNuevoEquipo(equipoObtenidoDatos).toJson();
        }
        return null;
    };

    public obtenerPrimerosN = (n: number): Partial<Equipo>[] => {
        const equiposOrdenados = this._equipos.sortValues("Ranking",
{ascending: true});
        const primerosNDatos = equiposOrdenados.values.slice(0, n);
        return primerosNDatos.map((equipoDatos: any[]) =>
```

```
this.crearNuevoEquipo(equipoDatos).toJson());
    };
}
```

Modelado de la aplicación

Finalmente, iniciaremos la aplicación de Express:

package.json:

```
{
  "name": "futbol",
  "version": "1.0.0",
  "description": "",
  "main": "index.ts",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "ts-node index.ts"
  },
  "author": "IEEE",
  "license": "ISC",
  "devDependencies": {
    "@types/express": "^4.17.15"
  },
  "dependencies": {
    "danfojs-node": "^1.1.2",
    "express": "^4.18.2",
    "http-status-codes": "^2.2.0",
    "nodemon": "^2.0.20"
  }
}
```

tsconfig.json:

```
{
  "compilerOptions": {
    "module": "commonjs",
    "esModuleInterop": true,
    "target": "es2022",
    "moduleResolution": "node",
    "sourceMap": true,
    "outDir": "dist",
    "baseUrl": ".",
```

```
    "paths": {
      "*": [
        "node_modules/*"
      ]
    }
  },
  "include": [
    "src/**/*"
  ]
}
```

EquiposHandler.ts:

```
import Equipos from "./Equipos";
import {Request, Response} from 'express';
import {ReasonPhrases, StatusCodes} from "http-status-codes";

export default class EquiposHandler {
    private _equipos = new Equipos();

    public leerEquipos = async (req: Request, res: Response):
Promise<void> => {
        try {
            await this._equipos.leerEquipos('./Rankings Football June
2022.csv');
            res.status(StatusCodes.CREATED).send();
        } catch (e) {
            res.status(StatusCodes.INTERNAL_SERVER_ERROR).send();
        }
    }

    public obtenerEquipo = (req: Request, res: Response): void => {
        const equipo = req.params.equipo.toString();
        try {
            const datosEquipo = this._equipos.obtenerEquipo(equipo);
            res.status(StatusCodes.OK).send(datosEquipo);
        } catch (e) {
            res.status(StatusCodes.INTERNAL_SERVER_ERROR).send();
        }
    }

    public obtenerPrimerosN = (req: Request, res: Response): void => {
        if (!!req.query.N) {
            try {
                const N = parseInt(<string>req.query.N);
```

```
                const datosEquipo = this._equipos.obtenerPrimerosN(N);
                if (datosEquipo == null) {

res.status(StatusCodes.NOT_FOUND).send(ReasonPhrases.NOT_FOUND);
                } else {
                    res.status(StatusCodes.OK).send(datosEquipo);
                }
            } catch (e) {
                res.status(StatusCodes.INTERNAL_SERVER_ERROR).send();
            }
        } else {
            res.status(StatusCodes.BAD_REQUEST).send('Cantidad de
equipos (N) faltante');
        }
    }
}
```

index.ts:

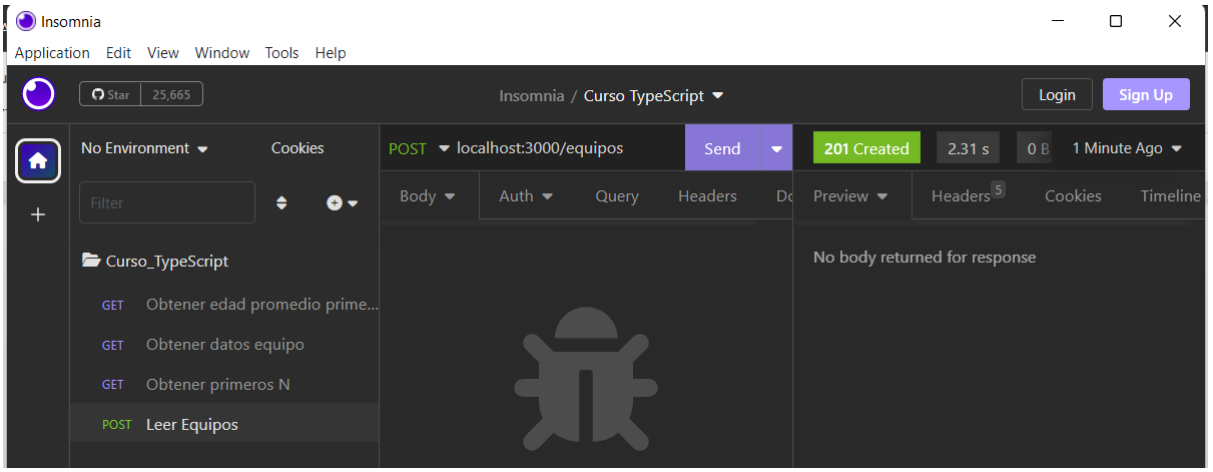
```
import express from "express";
import {Request, Response} from 'express';
import {StatusCodes} from "http-status-codes";
import EquiposHandler from "./Clases/EquiposHandler";

const PORT = 3000;

async function main() {
    const equiposHandler = new EquiposHandler();
    const app = express();

    app.listen(PORT, () => console.log("El servidor está corriendo en el
puerto " + PORT));

    // POST falso
    app.post('/equipos', async (req: Request, res: Response) => await
equiposHandler.leerEquipos(req,res));

    // GET
    app.get('/equipos', (req: Request, res: Response) =>
equiposHandler.obtenerPrimerosN(req, res));
    app.get('/equipos/:equipo', (req: Request, res: Response) =>
equiposHandler.obtenerEquipo(req, res));
    app.get('*', (req, res) =>
res.status(StatusCodes.NOT_FOUND).send('URL inválida'));
}
```

```
main();
```

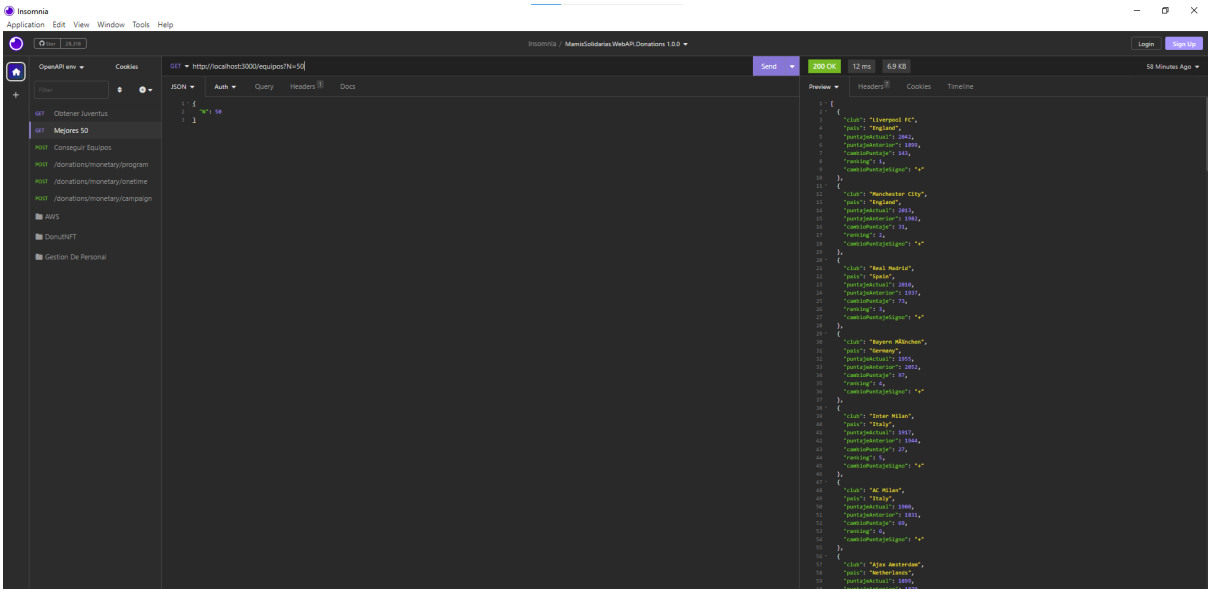
Revisión de endpoints

Ahora, para la parte de ver que funcionen lo que se llaman “endpoints” (el get de equipos y demás), utilizaremos el software de Insomnia, que lo hemos mencionado en la primera clase. Por suerte, es bastante simple para lo que requerimos utilizarlo. Aquí podremos hacer distintas llamadas a la API, con los distintos métodos http, y ver su resultado.

Por ejemplo, cuando uno clickea *Send* en el request *Leer Equipos* (**“Crearlos”**), con el método *POST*, tenemos lo siguiente:



Si hacemos una llamada “más compleja”, como ***Obtener primeros N***, con el método GET, podremos ver que en la parte derecha de la aplicación, nos aparecerán los resultados de la llamada a la API:


Insomnia
Application Edit View Window Tools Help
Insomnia / MamisSolidarias.WebAPI.Donations 1.0.0
Login
Sign Up
OpenAPI env
Cookies
GET http://localhost:3000/equipos?N=50
Send
200 OK
12 ms
6.9 KB
58 Minutes Ago
JSON
Auth
Query
Headers 1
Docs
Preview
Headers 7
Cookies
Timeline
GET Obtener Juventus
GET Mejores 50
POST Conseguir Equipos
POST /donations/monetary/program
POST /donations/monetary/onetime
POST /donations/monetary/campaign
AWS
DonutNFT
Gestion De Personal
{
"N": 50
}
[
{
"club": "Liverpool FC",
"pais": "England",
"puntajeActual": 2042,
"puntajeAnterior": 1959,
"cambioPuntaje": 843,
"ranking": 1,
"cambioPuntajeSigno": "+"
},
{
"club": "Manchester City",
"pais": "England",
"puntajeActual": 2013,
"puntajeAnterior": 1982,
"cambioPuntaje": 31,
"ranking": 2,
"cambioPuntajeSigno": "+"
},
{
"club": "Real Madrid",
"pais": "Spain",
"puntajeActual": 2010,
"puntajeAnterior": 1937,
"cambioPuntaje": 73,
"ranking": 3,
"cambioPuntajeSigno": "+"
},
{
"club": "Bayern München",
"pais": "Germany",
"puntajeActual": 1955,
"puntajeAnterior": 1852,
"cambioPuntaje": 97,
"ranking": 4,
"cambioPuntajeSigno": "+"
},
{
"club": "Inter Milan",
"pais": "Italy",
"puntajeActual": 1917,
"puntajeAnterior": 1944,
"cambioPuntaje": 27,
"ranking": 5,
"cambioPuntajeSigno": "+"
},
{
"club": "AC Milan",
"pais": "Italy",
"puntajeActual": 1900,
"puntajeAnterior": 1831,
"cambioPuntaje": 69,
"ranking": 6,
"cambioPuntajeSigno": "+"
},
{
"club": "Ajax Amsterdam",
"pais": "Netherlands",
"puntajeActual": 1899,