Técnicas de Programación

Carrera Programador full-stack

Async – Await / REST Services

Async & Await

•En **ES8** se incorpora dos palabras reservadas para facilitar la escritura de código con promesas

ASYNC

•Hace que una función devuelva una promesa
•El return se encapsulará en la promesa automaticamente

AWAIT

•Desencapsula el contenido de una promesa.
•Se reescribe como el THEN de la promesa
•Solo puede usarse dentro de funciones ASYNC

Async / Await

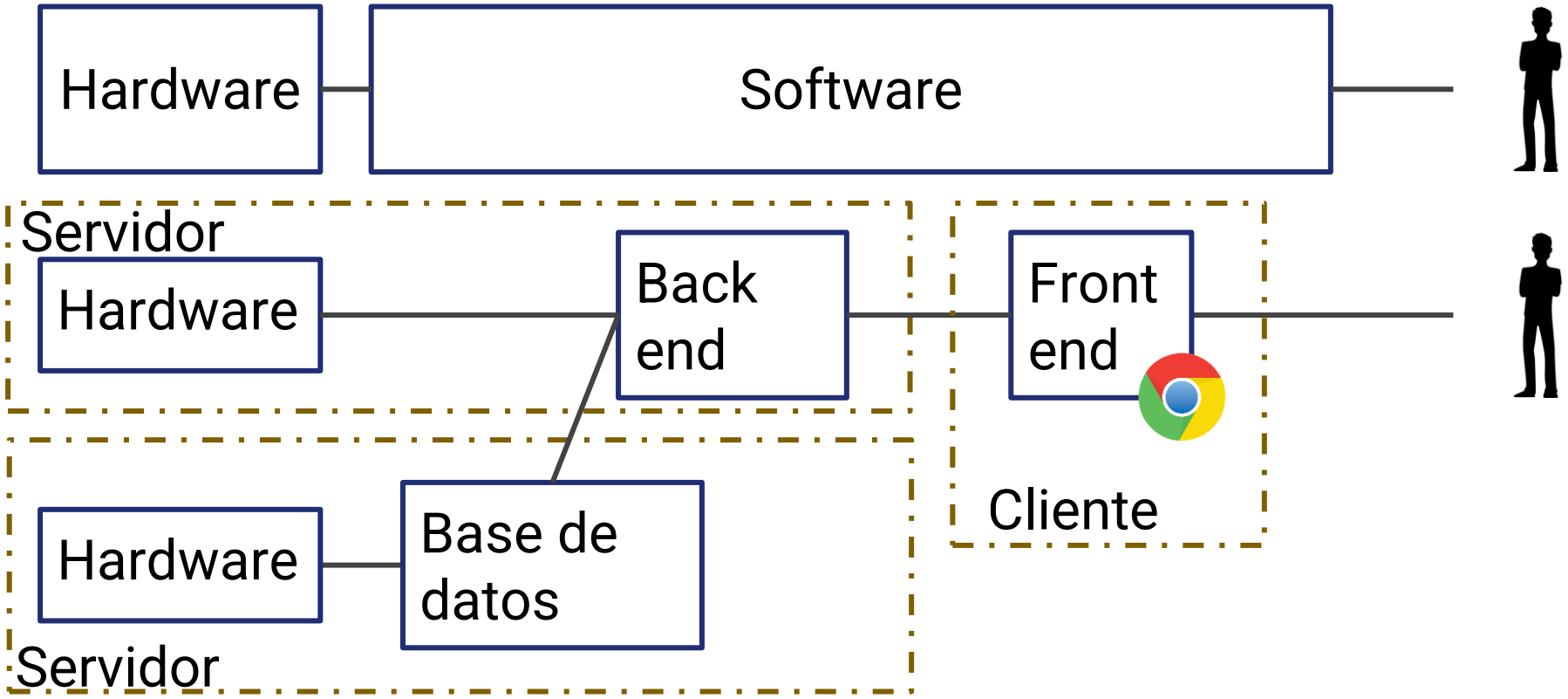
Se agrega la palabra async a la función que va a devolver una promesa, y se le agrega la palabra await a la función que es asíncrona y va a demorar un tiempo en devolver una respuesta.

```
async function fetchUser() {
  try {
    const auth = await checkAuth() // <- async operation
    const user = await getUser(auth) // <- async operation
    return user
  }
  catch (error) {
    return { name: 'Default', error: error }
  }
}
fetchUser()
    .then(user => console.log(user.name))
    .catch(err => console.log(err))
```

Carrera Programador full-stack

REST

Arquitectura de Sistemas Web
Repaso rápido



Cliente - Servidor

Cliente:

- Se encarga de manejar la interfaz gráfica del usuario

Servidor:

- Se encarga de:
  - Almacenar los datos
  - Dar seguridad (que cada uno vea solo sus datos, etc)

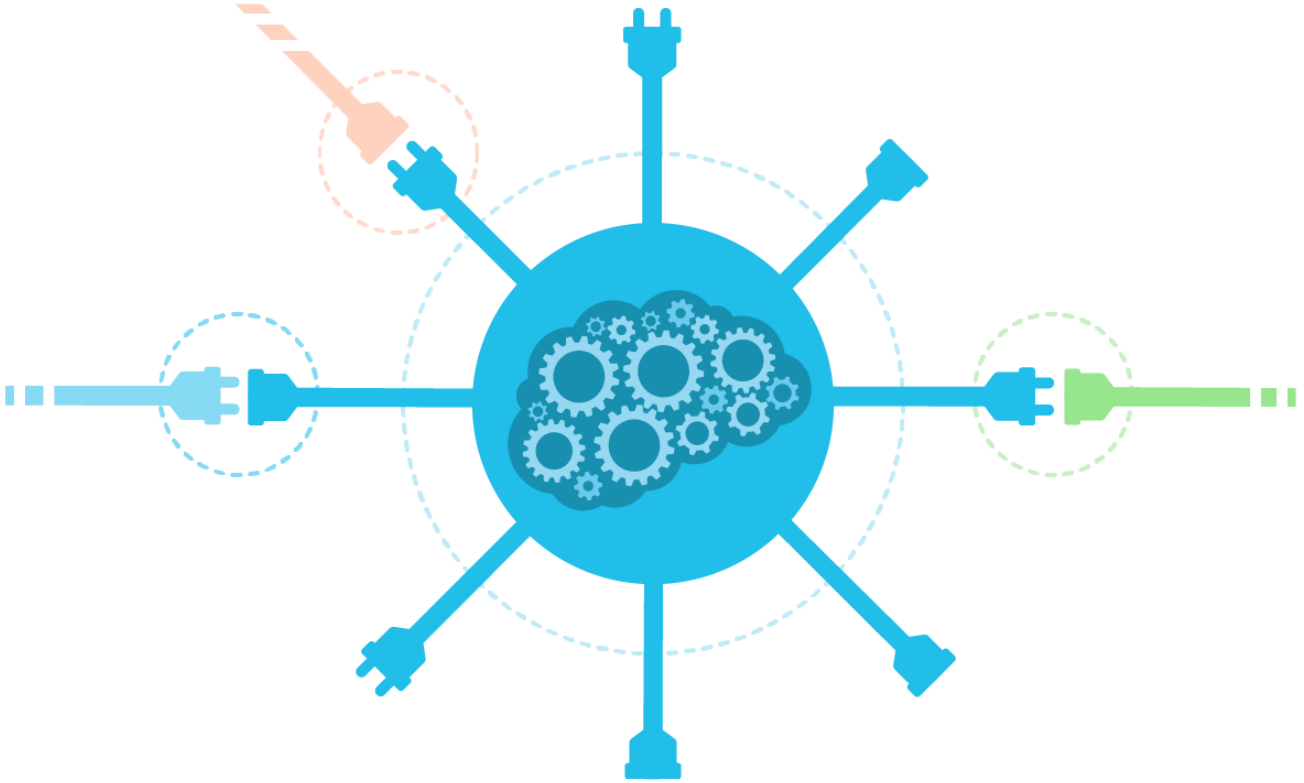
Acciones:

Estos son algunos de las acciones que vamos a ver al momento de consumir una API:

1. Traer todos los datos estáticos que están en el servidor
2. Traer un dato
3. Guardar nuevos datos
4. Borrar datos
5. Editar datos

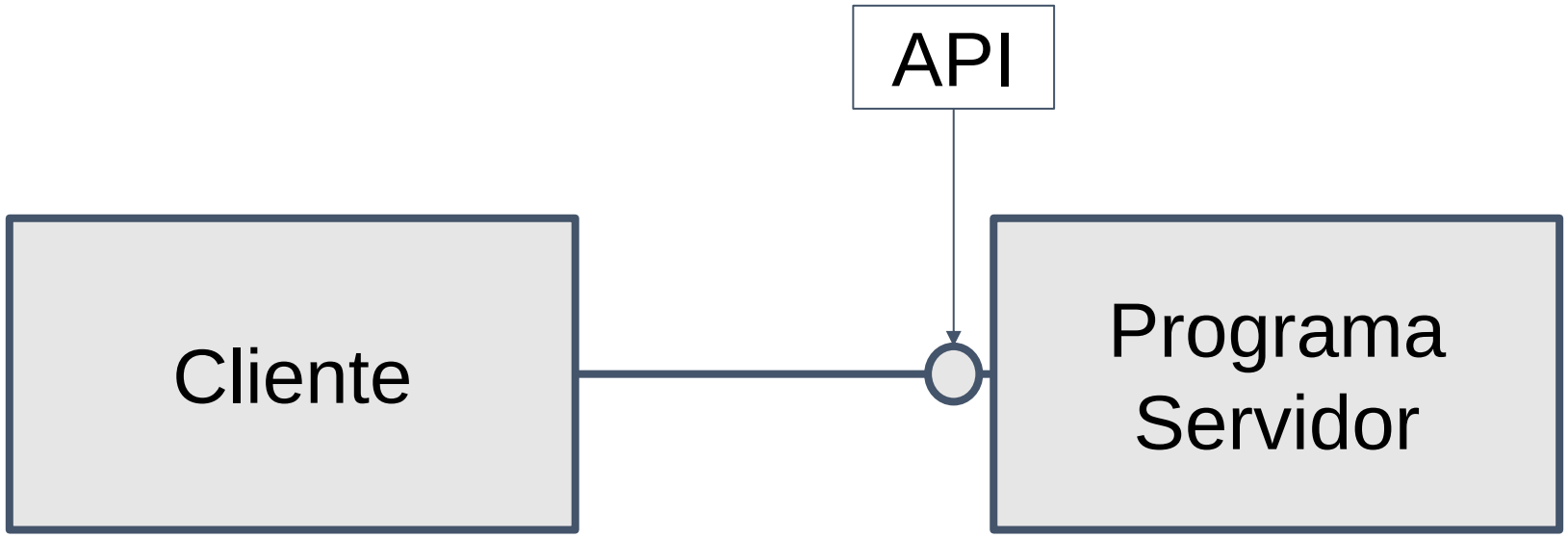
API

Que es una API y para qué sirve?



API

Una API es una interfaz que nos da una aplicación para comunicarnos con ella. es un contrato que le permite a dos o más softwares comunicarse entre sí y compartir información. Las API son responsables de casi todo lo que hacemos en la web. Son la manera cómo los datos se conectan de un lugar a otro, y luego a nuestros dispositivos. el mecanismo más útil para conectar dos softwares entre sí para el intercambio de mensajes o datos en formato estándar como JSON.



API

Como funciona?

Los datos son los que están en una BBDD o Servidor, etc.

La API es la puerta que conecta a los desarrolladores con los datos y también actúa como filtro.

Los desarrolladores son quienes manipulan o crean una aplicación para esos datos, con las API´s públicas se proporcionan formas para consumirlas.

El software se conectan a datos y servicios que brindan interesantes experiencias al usuario.

REST

- REST: Representational State Transfer, es un tipo de arquitectura de desarrollo web que se apoya totalmente en el estándar HTTP.
- Es el tipo de arquitectura más natural y estándar para crear APIs para servicios orientados a Internet.
- La mayoría de las APIs REST usan JSON para comunicarse.

REST

- Se asocian URLs a recursos.
- Al que se puede acceder o modificar mediante los métodos del protocolo HTTP.
- Se basa en acciones (llamadas verbos) que manipulan los datos.
  - POST: Crear un recurso
  - GET: Obtener uno o muchos recursos
  - PUT: Actualizar uno o muchos recursos
  - DELETE: Borrar un recurso
- Se utilizan los errores del protocolo HTTP.
  - 200 ok, 404 not found, etc.

API REST - EJEMPLO

- GET /users (en genérico /users)
  - Acceder al listado de users
- POST /users (en genérico /users)
  - Crear un user nuevo
- GET /users/123 (en genérico /users/:id_fact)
  - Acceder al detalle de **un** user
- PUT /users/123 (en genérico /users/:id_fact)
  - Editar el users, sustituyendo la **totalidad** de la información anterior por la nueva.
- DELETE /users/123 (en genérico /users/:id_fact)
  - Eliminar el user

Manejo de errores en REST

Se pueden utilizar los errores del protocolo HTTP:

- 200 OK Standard response for successful HTTP requests
- 201 Created
- 202 Accepted
- 301 Moved Permanently
- 400 Bad Request
- 401 Unauthorised
- 402 Payment Required
- 403 Forbidden
- 404 Not Found
- 405 Method Not Allowed
- 500 Internal Server Error
- 501 Not Implemented

https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

https://restfulapi.net/http-status-codes/

Consumir una API

A modo de ejemplo para poder utilizar REST vamos a utilizar algún servicio web que nos permita consumir una API lista, como es el caso de FakeStore API.

https://fakestoreapi.com/

Para qué usamos una API preparada?

Nos permite trabajar con el front end aunque nuestro back end no esté implementado. Es una muy buena forma de poder seguir trabajando y probar la aplicación con datos sin tener que esperar a que esté implementado el backend o que la base de datos tenga información.

Mientras tanto podemos seguir desarrollando la aplicación y prepararla para cuando esté lista.

Ejemplo archivo JSON

Vamos a tener que manipular y preparar archivos del tipo JSON para comunicarnos con la API:

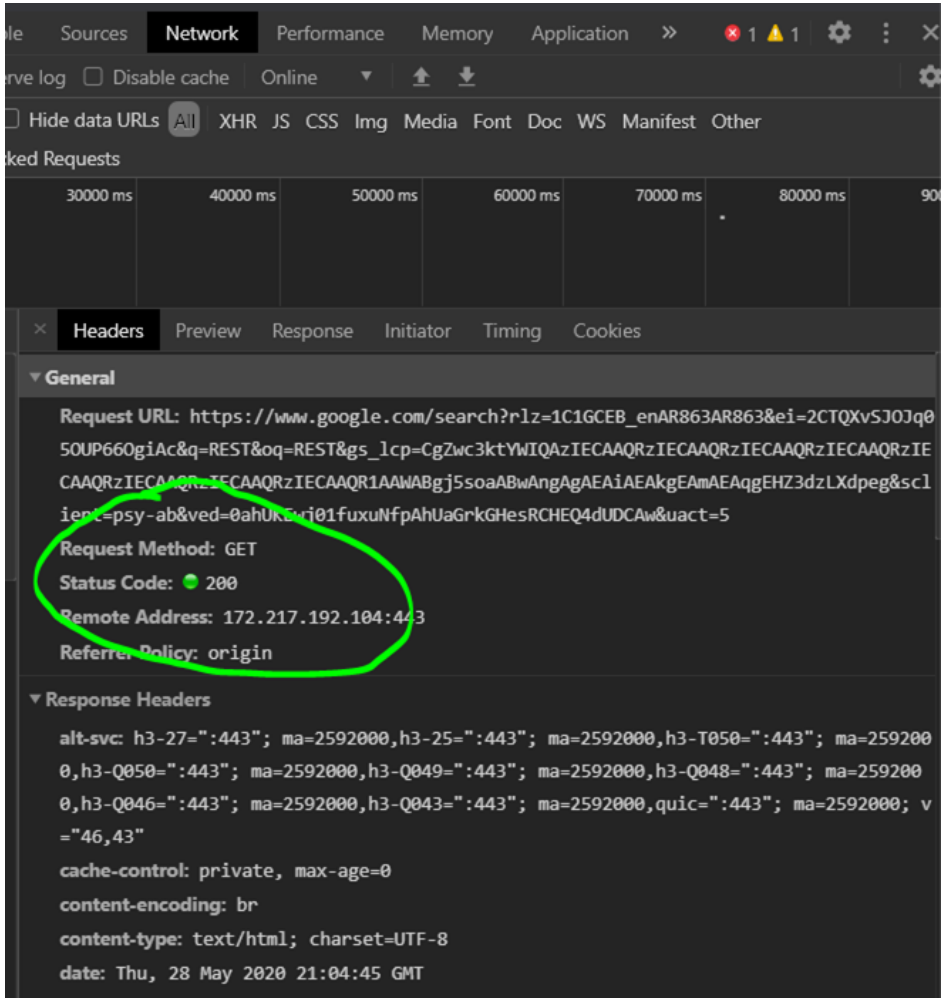
```
{
   "status": "OK",
   "compras": [
      {
         "producto": "Manzana",
         "precio": "20"
      }
   ]
}
```

Array de todos los registros de compras

Ver la respuesta JSON

- El navegador por defecto siempre hace GET para bajar las páginas.

- Si ponemos la URL en el navegador vemos directamente el JSON (aunque solo nos sirve para GET, no para otros métodos de HTTP).



JSON en el navegador


```
{
    "status": "OK",
    "nombres": [
        {
            "_id": "5b15575fdeb51c0400814076",
            "group": "ejemplos",
            "thingtype": "nombres",
            "thing": {
                "nombre": "Juan"
            },
            "__v": 0,
            "dateAdded": "2018-06-04T15:14:39.135Z"
        },
        {
            "_id": "5b155c47b9788a040091cdc9",
            "group": "ejemplos",
            "thingtype": "nombres",
            "thing": {
                "nombre": "Carlos"
            },
            "__v": 0,
            "dateAdded": "2018-06-04T15:35:35.067Z"
```


Extensión para Chrome:

JSON Formatter

Ejemplo de respuesta

Analizamos la estructura de la respuesta para poder leerla

```
{
    "status": "OK",
    "compras": [
        {
            "producto": "Manzana",
            "precio": "20"
        }
    ]
}
```

Consumir la API

Ahora nos falta consumir esa API desde el front-end.

Lo que vamos a poder realizar es hacer uso de los distintos verbos que mencionamos anteriormente, y poder trabajar y realizar operaciones con esa información.

Repaso de fetch() en ES7

ES7 incorpora la interfaz **fetch()**

```
let promise = fetch(url);
```

El uso más simple de fetch() toma un argumento (la ruta del recurso que se quiera traer) y **el resultado es una promesa** que contiene la respuesta (un objeto Response)

https://developer.mozilla.org/es/docs/Web/API/Fetch_API/Utilizando_Fetch

GET

`response.json()`

Al ejecutar res.json() se parsea (“compila”) a un objeto automáticamente.

Devuelve una promesa, dado que convertir a JSON puede demorar mucho.

```
fetch('https://rickandmortyapi.com/api/character/?page=19')
    .then(
        datos => { return (datos.json()); }
    )
    .then(
        datos => console.log(datos)
    )
    .catch(
        err => { console.log(err); }
    );
```

POST

En HTTP existen diferentes métodos.
Para la creación de un recurso se utiliza el método POST.
Vamos a configurar un POST para enviar datos a la API de MockAPI y poder crear datos.

```
fetch('https://5ecdadc47c528e00167cd6de.mockapi.io/users', {
    method: 'POST',
    body: JSON.stringify(data),
    headers: { 'Content-Type': 'application/json' }
})
    .then(datos => { return (datos.json()) })
    .then(datos => console.log(datos))
    .catch(err => { console.log(err) });
```

¿Cómo enviamos los datos?

Para trabajar con APIs REST, los datos en general se envían en formato JSON

Para la estructura del dato, usamos la misma de salida

```
{
  "producto": "Manzana",
   "precio": "20"
 }
```

PUT

- Similar al **POST.**
- En lugar de **crear** un item, vamos a **modificar** uno que ya existe.
- Vamos a necesitar el :**ID** del item a modificar.
- También los nuevos datos del item.

¿Cual es la URL que vamos a crear?

Combina **parámetros** y el acceso al **body** del request

PUT

Al igual que POST con PUT podemos modificar un recurso en nuestro servidor o base de datos, reemplazando por completo los datos anteriores.

```
fetch('https://5ecdadc47c528e00167cd6de.mockapi.io/users/22', {
    method: 'PUT',
    body: JSON.stringify(data),
    headers: { 'Content-Type': 'application/json' }
})
    .then(datos => { return (datos.json()) })
    .then(datos => console.log(datos))
    .catch(err => { console.log(err) });
```

DELETE

Para enviar el borrado desde el front-end:

- Se usa fetch, similar al POST pero cambiando la ruta
- Necesito poner muchos botones
  - Que cada boton sepa de que elemento es
  - Asignar el click (el mismo a todos)
  - Que el click pueda diferenciar qué botón fue

DELETE

Se hace el uso del verbo DELETE con el cual podemos eliminar un elemento en nuestro servidor o base de datos, para eso también se va a necesitar el id del item o recurso a eliminar.

```
fetch('https://5ecdadc47c528e00167cd6de.mockapi.io/users/21', {
    method: 'DELETE'
 })
    .then(datos => { return (datos.json()) })
    .then(datos => console.log(datos))
    .catch(err => { console.log(err) });
```

Ejercicios

Carrera Programador full-stack

Ejercicio

Consumir una API publica que les guste y consumirla con el método fetch.

Luego, una vez obtenidos los datos, vamos a mostrarlos en el navegador (pueden darle formato de card)