Introducción a aplicaciones móviles

Aplicaciones móviles

Aplicaciones nativas:

Son aplicaciones desarrolladas específicamente para un sistema operativo móvil, como iOS (Apple) o Android (Google). Se utilizan lenguajes de programación específicos del sistema operativo, como Swift o Objective-C para iOS y Java o Kotlin para Android.

Tienen mejor rendimiento en contextos con requerimientos complejos, pero un alto costo de desarrollo.

Aplicaciones Híbridas:

Son aplicaciones que combinan elementos de aplicaciones nativas y web. Utilizan tecnologías web (HTML, CSS, JavaScript) y se ejecutan dentro de un contenedor nativo.

Poseen un tiempo de desarrollo más corto, reducen costos ya que permiten generar la aplicación para distintas plataformas. Pueden tener alguna limitación de alcance si la app debiera realizar acciones complejas que requieran uso de hardware muy específico.

Tipos de aplicaciones

Aplicaciones móviles de ejecución local

Son aplicaciones que ejecutan localmente en el dispositivo. No requieren de estar conectados a internet para realizar su función.

Ejemplos: Calculadora, app de notas, grabadora de video y sonido, etc.

Aplicaciones móviles de interacción con un servidor.

Son aplicaciones que requieren estar conectados a internet para poder desplegar su funcionalidad, ya que deben estar constantemente realizando consultas a un servidor. Generalmente son el tipo de aplicaciones multiusuario.

Ejemplos: redes sociales, sistemas de gestión, etc.

Arquitectura de despliegue

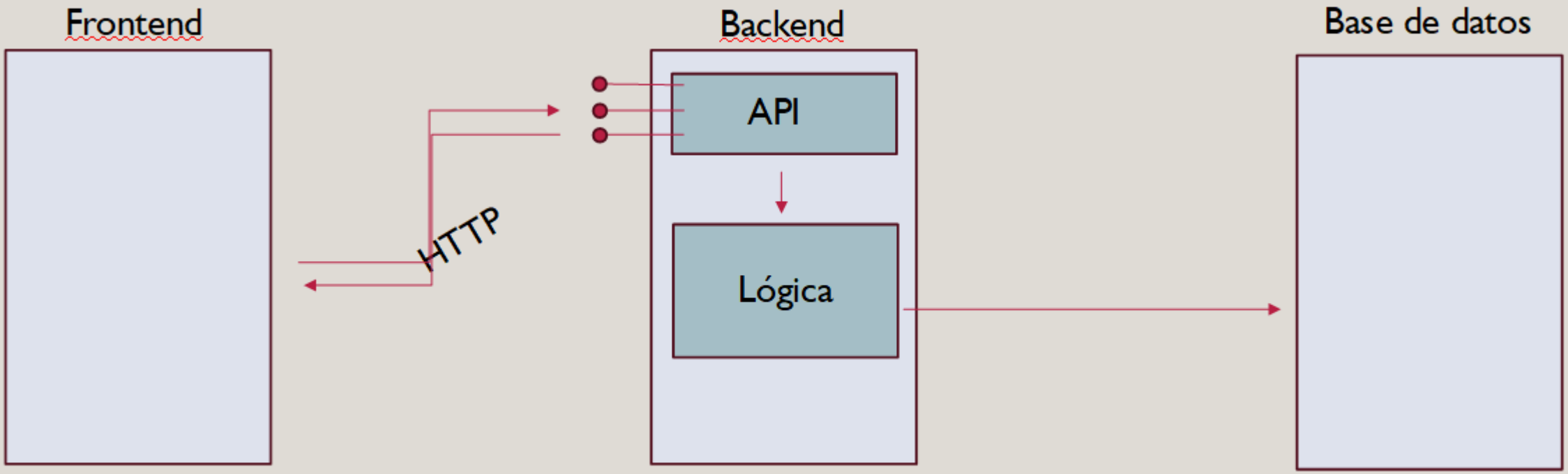
Para realizar una aplicación donde el usuario puede iniciar sesión y realizar acciones intervienen varios componentes.

Por primera vez tenemos claramente definidos los conceptos de FRONTEND y BACKEND.

FRONT END: Es la parte de una aplicación web o móvil que interactúa directamente con el usuario. Es la interfaz de usuario y todo lo que los usuarios ven y con lo que interactúan en su pantalla.

BACKEND: Es la parte del desarrollo que se encarga de la lógica del servidor, la base de datos, y la integración de la aplicación. Es la capa que no se ve pero que permite que el frontend funcione correctamente

Arquitectura de despliegue

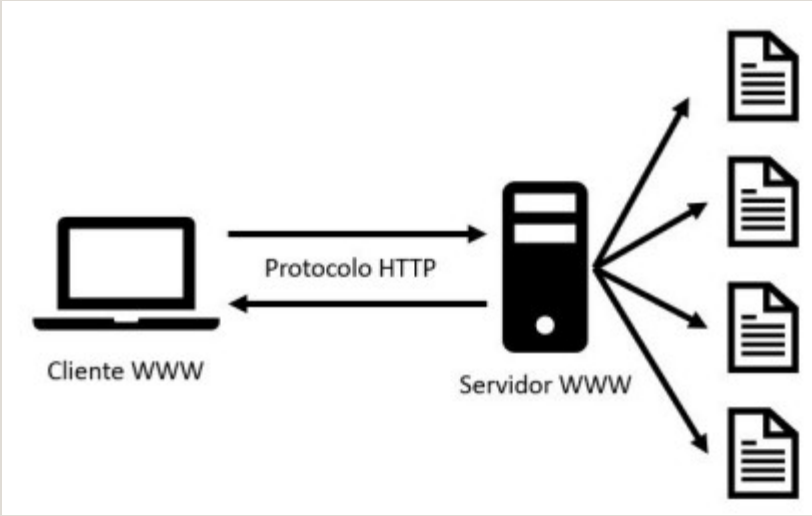


Protocolo HTTP

HTTP

El protocolo de transferencia de hipertexto (HTTP) es un protocolo de comunicación que permite la transferencia de información entre un cliente y un servidor.

Se utiliza para transmitir documentos hipermedia, como HTML. Se diseñó para la comunicación entre navegadores y servidores web, pero también se puede utilizar para otros propósitos. De esta forma se comunican las APIs.



Métodos HTTP

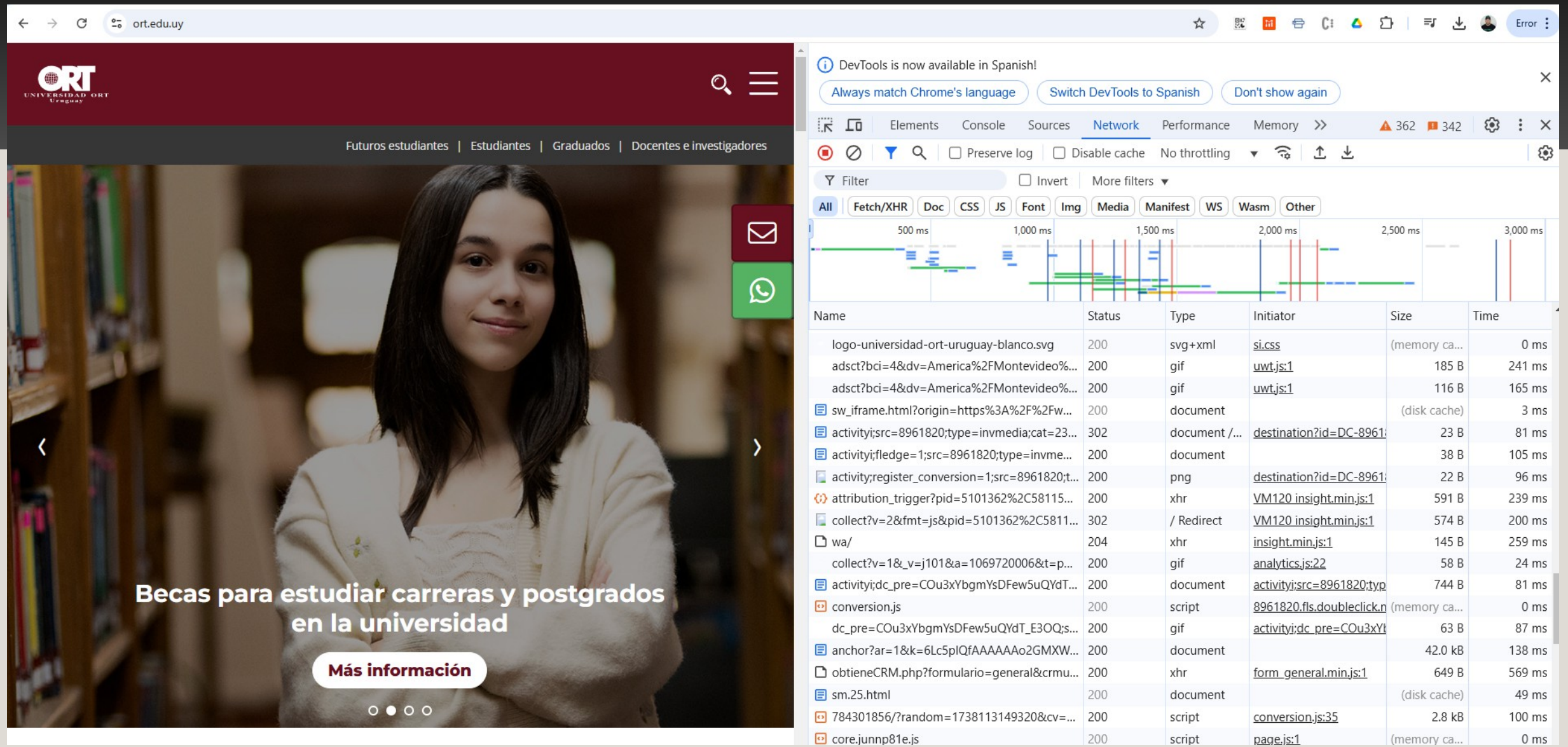
Método	Significado	Ejemplo
GET	Acceso a lectura de recurso	Acceso a listado de películas: GET /peliculas/ Acceso a detalle de película: GET /peliculas/1/
POST	Acceso a crear un nuevo recurso	Crear un una nueva película: POST /peliculas/
PUT	Acceso a actualizar un recurso	Actualizar película con identificador 1: PUT /peliculas/1/
DELETE	Acceso a eliminar un recurso	Eliminar película con identificador 1: DELETE /peliculas/1/

Status code

Para cada petición el servidor devuelve un código de respuesta con la información del recurso que haya sido solicitado.

El código de estado es un valor numérico que se envía junto con la respuesta a una solicitud http. Está basado en una convención que determina el código para cada situación.

- Informational responses (100 – 199)
- Successful responses (200 – 299)
- Redirection messages (300 – 399)
- Client error responses (400 – 499)
- Server error responses (500 – 599)


ort.edu.uy
Error
ORT
UNIVERSIDAD ORT Uruguay
Futuros estudiantes | Estudiantes | Graduados | Docentes e investigadores
Becas para estudiar carreras y postgrados en la universidad
Más información
DevTools is now available in Spanish!
Always match Chrome's language
Switch DevTools to Spanish
Don't show again
Elements
Console
Sources
Network
Performance
Memory
362
342
Preserve log
Disable cache
No throttling
Filter
Invert
More filters
All
Fetch/XHR
Doc
CSS
JS
Font
Img
Media
Manifest
WS
Wasm
Other
500 ms
1,000 ms
1,500 ms
2,000 ms
2,500 ms
3,000 ms
Name
Status
Type
Initiator
Size
Time
logo-universidad-ort-uruguay-blanco.svg 200 svg+xml si.css (memory ca... 0 ms
adsct?bci=4&dv=America%2FMontevideo%... 200 gif uwt.js:1 185 B 241 ms
adsct?bci=4&dv=America%2FMontevideo%... 200 gif uwt.js:1 116 B 165 ms
sw_iframe.html?origin=https%3A%2F%2Fw... 200 document (disk cache) 3 ms
activityi;src=8961820;type=invmedia;cat=23... 302 document /... destination?id=DC-8961 23 B 81 ms
activityi;fledge=1;src=8961820;type=invme... 200 document 38 B 105 ms
activity;register_conversion=1;src=8961820;t... 200 png destination?id=DC-8961 22 B 96 ms
attribution_trigger?pid=5101362%2C58115... 200 xhr VM120 insight.min.js:1 591 B 239 ms
collect?v=2&fmt=js&pid=5101362%2C5811... 302 / Redirect VM120 insight.min.js:1 574 B 200 ms
wa/ 204 xhr insight.min.js:1 145 B 259 ms
collect?v=1&_v=j101&a=1069720006&t=p... 200 gif analytics.js:22 58 B 24 ms
activityi;dc_pre=COu3xYbgmYsDFew5uQYdT... 200 document activityi;src=8961820;typ 744 B 81 ms
conversion.js 200 script 8961820.fls.doubleclick.n (memory ca... 0 ms
dc_pre=COu3xYbgmYsDFew5uQYdT_E3OQ;s... 200 gif activityi;dc_pre=COu3xYb 63 B 87 ms
anchor?ar=1&k=6Lc5plQfAAAAAAo2GMXW... 200 document 42.0 kB 138 ms
obtieneCRM.php?formulario=general&crmu... 200 xhr form_general.min.js:1 649 B 569 ms
sm.25.html 200 document (disk cache) 49 ms
784301856/?random=1738113149320&cv=... 200 script conversion.js:35 2.8 kB 100 ms
core.junnp81e.js 200 script page.js:1 (memory ca... 0 ms

API (application programming interface)

API (Interfaz de Programación de Aplicaciones)

Una API es una interfaz de comunicación que permite a diferentes aplicaciones comunicarse entre sí y compartir información y funcionalidades.

Es un **intermediario** entre dos sistemas, que permite que una aplicación se comunique con otra y pida datos o acciones específicas.

No posee interfaz de usuario, sino que ofrece **endpoints** (puntos de acceso). En ellos se solicitan las acciones. Un ejemplo de endponint puede ser https://miapp.com/login que espera recibir un usuario y contraseña, y responderá si pertenece a un usuario válido.

Una API permite agregar diversas funciones a sitios web y aplicaciones.

Ejemplo: Tesis Virbella

https://youtu.be/rxZc9oLqx28

Consumir APIs mediante endpoints

Un endpoint es una ubicación digital donde la API obtiene la solicitud de recursos. Básicamente es una URL que da información de recursos. El endpoint es un extremo del canal de comunicación

Para consumir una API es necesario

- Utilizar el nombre de la URL del endpoint
- Seleccionar el método HTTP
- Definir el cuerpo de la petición y sus parámetros
- Definir la estrategia de autenticación si es necesario

Consumir APIs mediante endpoints

Un endpoint es una ubicación digital donde la API obtiene la solicitud de recursos. Básicamente es una URL que da información de recursos. El endpoint es un extremo del canal de comunicación

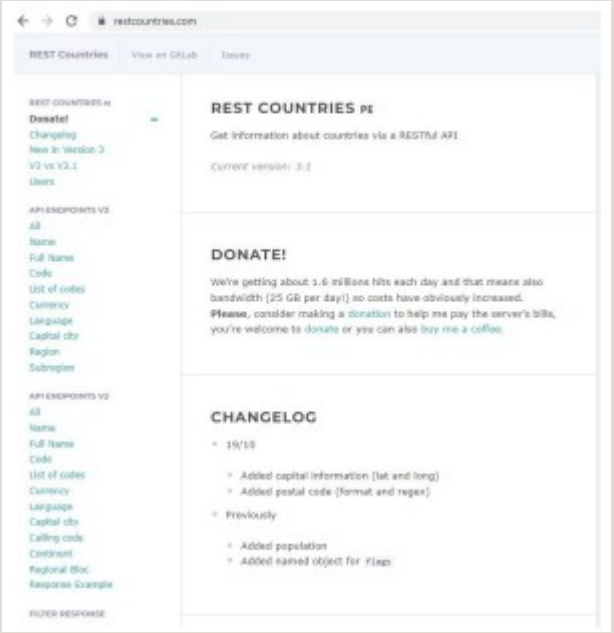
Para consumir una API es necesario

- Utilizar el nombre de la URL del endpoint
- Seleccionar el método HTTP
- Definir el cuerpo de la petición y sus parámetros
- Definir la estrategia de autenticación si es necesario

La documentación de la API

La documentación de API, abarca la redacción del contenido informativo sobre el uso de API.

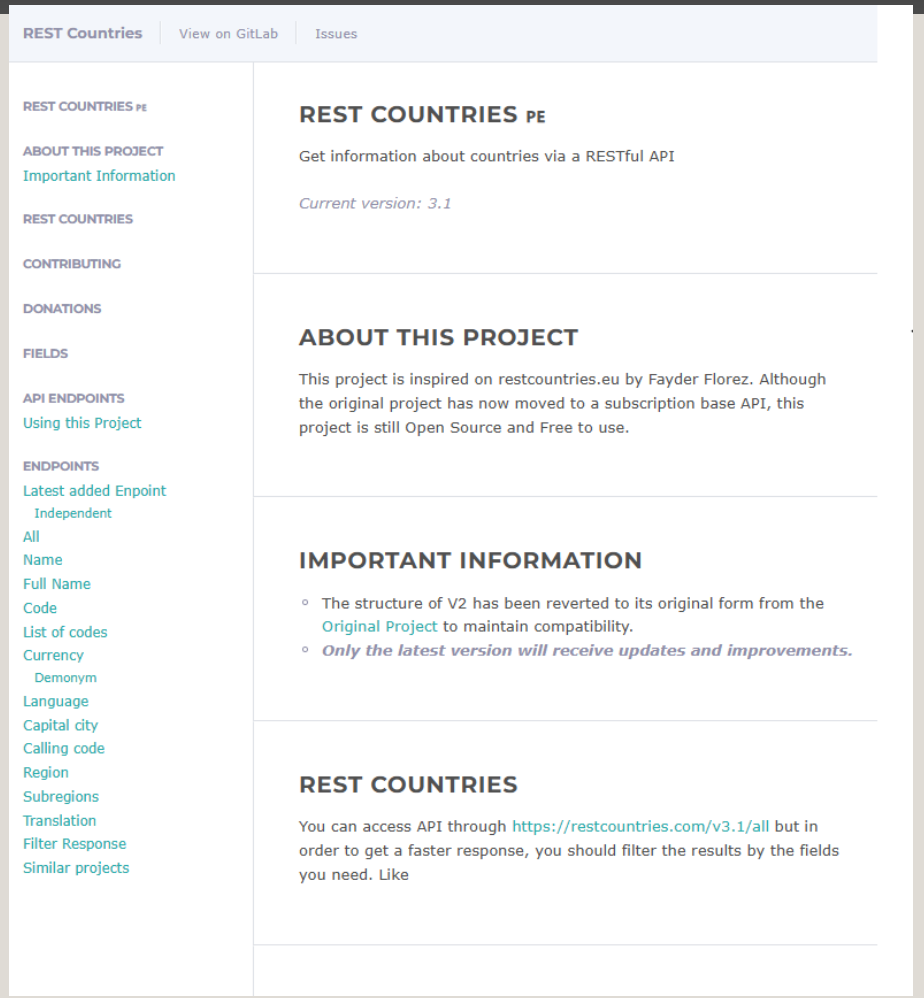
La guía se presenta, normalmente, bajo una serie de referencias y tutoriales acompañados de ejemplos que ayudan a los desarrolladores a entender la Interfaz de Programación de Aplicaciones API.



Ejemplos de una API

Para nuestro primer ejemplo vamos a utilizar https://restcountries.com

Los resultados que vamos a obtener van a estar en formato JSON



Archivos tipo JSON

JSON, que significa JavaScript Object Notation, es una notación usada para estructurar datos en forma de texto y transmitirlos de un sistema a otro, como en aplicaciones cliente-servidor o en aplicaciones móviles.

JSON no es un lenguaje de programación sino una notación específica por lo que puede emplearse en diferentes lenguajes.

La simplicidad con que los datos están estructurados en el formato JSON permite que sea utilizado en cualquier tipo de lenguaje de programación.

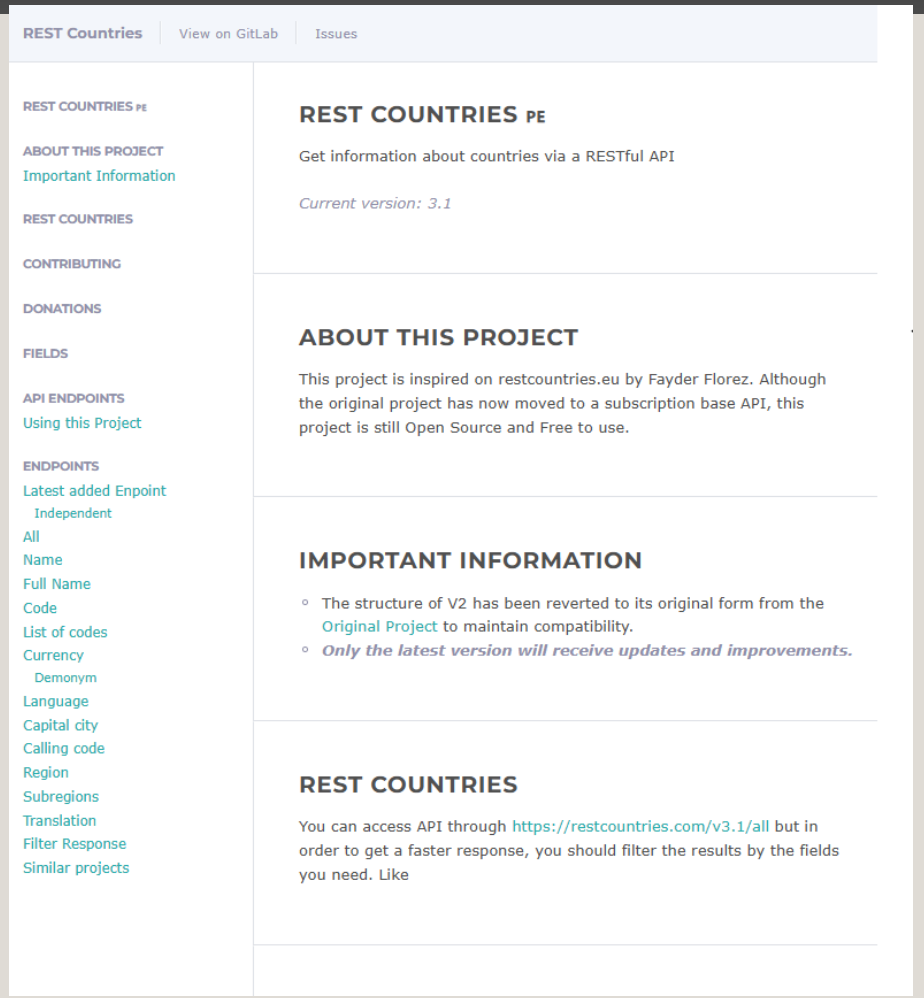
Diferencia con XML

- JSON es solo texto, no se puede colocar imágenes (pero si url con las mismas). JSON solo utiliza UTF-8, mientras que el XML ofrece esa y otras opciones.
- Los archivos JSON son fáciles de entender, pues su estructura y notación son bien sencillas.
- JSON maneja arrays.
- Los archivos son más livianos y el parseo es más sencillo.

Ejemplos de una API

Para nuestro primer ejemplo vamos a utilizar https://restcountries.com

Los resultados que vamos a obtener van a estar en formato JSON



Documentación y prueba de una API

Las APIs son programadas siguiendo estándares, pero no son iguales en su estructura de respuesta. Si bien los status code de respuesta deben seguir este estándar, su forma puede variar.

Cada API debe necesariamente estar acompañada por su documentación. Es vital para poder utilizarla.

Este proceso puede realizarse con Postman, Swagger entre otros.

A continuación, debemos abrir POSTMAN y comenzar a probar los endpoints.

Cada API está programada de una forma distinta. Debe explorar los retornos para tomar la mejor decisión.

Uso de Postman

Postman es una aplicación que nos permite testear APIs mediante una interfaz gráfica de usuario.

Es posible crear diferentes ambientes de prueba y ayuda a optimizar el tiempo de ejecución de las mismas.

Para descargar

https://www.postman.com/downloads/

Ejercicio en Postman

Utilizando https://restcountries.com realizar las siguientes consultas:

1) Todos los datos de los países
2) Todos los datos del Uruguay
3) Países de América
4) Países de SudAmérica
5) Países y capitales
6) Todos los nombres
7) Nombre del país y dominio de internet
8) Sorprender con una petición

Llamadas Asíncronas

Definición

Una petición asíncrona es una operación que, mientras esté siendo procesada, deja libre al dispositivo para que pueda hacer otras operaciones.

Cuando se llama a un endpoint de una API no tenemos control sobre el tiempo que tarda en responder. Generalmente este tiempo está determinado por la conexión propia del dispositivo que hace el request y la carga y espera a la que se somete por parte del servidor de la API.

Al desarrollar una aplicación con comunicación asíncrona debemos tener en cuenta este problema ya que al realizar una request a un método asíncrono este comienza a correr en un hilo aparte, y el código de nuestra aplicación sigue corriendo de forma convencional. Es decir que si estoy esperando saber un dato que viene de una llamada asíncrona, debo esperar a que vuelva para poder seguir. La ejecución del código no lo espera.

PROMESAS

Una promesa en JavaScript es un objeto que representa un valor que puede estar disponible ahora, en el futuro o nunca. Se utiliza para manejar operaciones asíncronas, como solicitudes a un servidor o temporizadores, sin bloquear el flujo de ejecución del código.
Las promesas tienen tres estados principales:

Pendiente (pending): La operación aún no se ha completado.

Cumplida (fulfilled): La operación se completó con éxito.

Rechazada (rejected): La operación falló.

Uso de Fetch

Javascript a través de su API Fetch nos proporciona un método llamado fetch() que permite realizar estas llamadas a los endpoits.

El método fetch() retorna una promesa. Al recibir el resultado de la operación, este puede ser Resolve, o Reject.

Las promesas no permiten invocar su retorno de la misma forma que hemos utilizado hasta el momento. Cuando invocamos un método fetch() debemos esperar su retorno y prepararnos para el mejor y peor escenario.

Uso de Fetch

El método fetch contiene la URL del endpoint y además toda la información que la API espera recibir. Esta información varía según el tipo de end point.

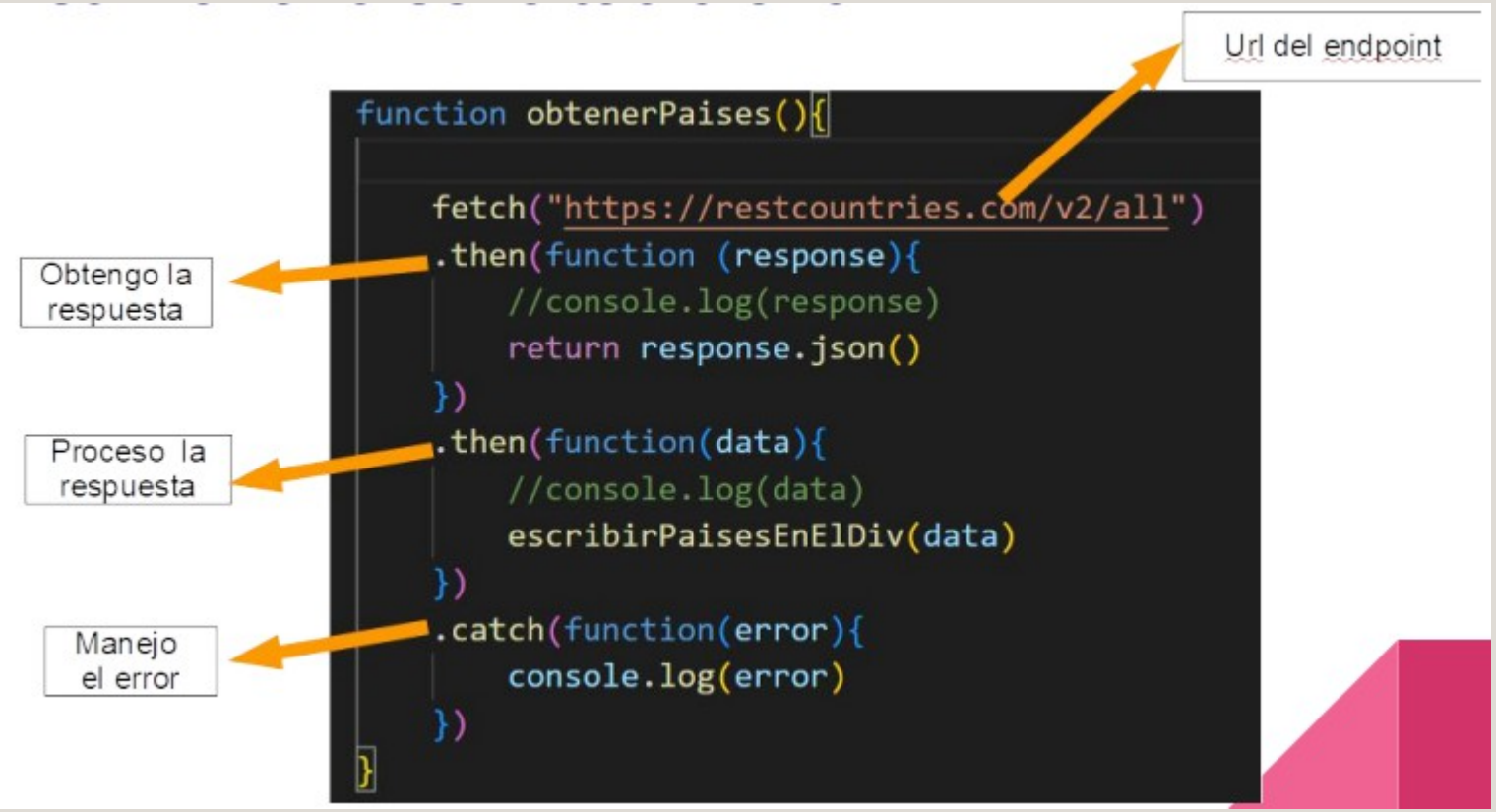
No siempre es necesario que el fetch contenga información adicional, pero la url SIEMPRE es un parámetro necesario.

En el siguiente código hacemos un fetch a un endpoint público que ofrece una lista de todos los países.

fetch('https://restcountries.com/v2/all’)

Como mencionamos anteriormente, el fetch retorna una promesa.

Ejemplo de Fetch



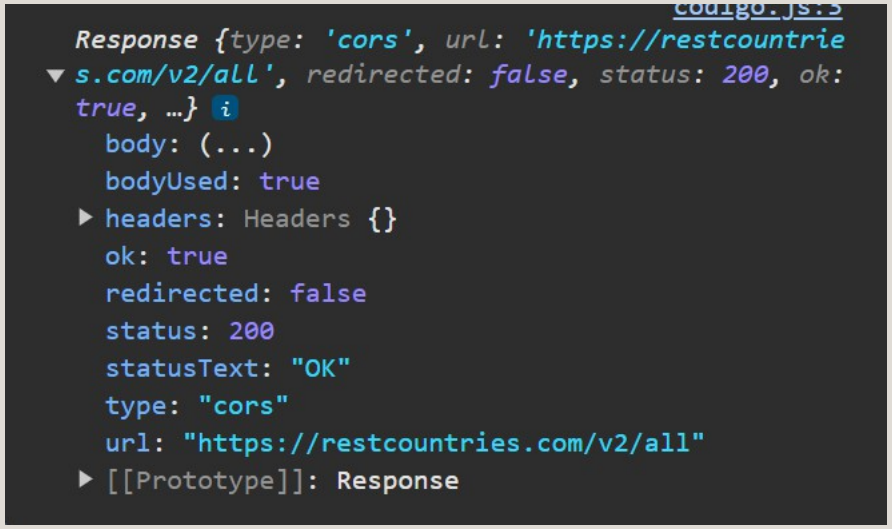
Uso de Fetch

Las promesas se resuelven a tiempos que no podemos manejar. Por eso cada promesa cuenta con un método then() que espera su retorno y nos permite trabajar con él. Dentro del método then() debemos ejecutar una función anónima que recibe una response por parámetros. Esta response es el Resolve de la promesa y llega como parámetro en la función anónima.

```
fetch('https://restcountries.com/v2/all')
    .then(function(response){
        console.log(response);
    })
```

Uso de Fetch

Esta response que nos llegó del request a la API, contiene lo siguiente:



Esta respuesta contiene información muy útil para el desarrollo del frontend. Podemos saber el status code y en base a él continuar con los requerimientos de la aplicación.

Uso de Fetch

La response es un objeto, que posee atributos. Como cualquier objeto podemos consultar por ellos y ver qué información tiene.

En este caso lo más relevante es el status code, que indica el resultado de la request.

Pero… ¿dónde están los países?

La respuesta depende del status. En este caso como respondió 200, sabemos que están en algún lugar.

Si hubiera sido 404, 401, 500, 501, etc, la lista de países no estaría accesible, pero podríamos tener un mensaje específico del error.

Cómo puedo acceder entonces a esa información.

Then().Then()

El contenido del body debe ser transformado en JSON, mediante un método json().

Sin embargo, este método es asíncrono. Por lo tanto, si se lo invoca retorna una promesa. Y como toda promesa, debe ser capturada en un then().

Entonces si queremos acceder al body, debemos retornar la response como json() en el primer then() y capturarlo como parámetro en una función anónima en un segundo then().

```
fetch('https://restcountries.com/v2/all')
    .then(function (response) {
        console.log(response);
        //Quiero acceder al body de la response
        //Entonces lo retorno como json()
        return response.json();
    }).then(function (data) {
        console.log(data);
    })
```

Then().Then()

Cuando se programa una API, se debe respetar el standard y eso implica que los códigos de status sean correctos. Si durante un proceso de registro de usuario, la API vuelve con un status code 400 (bad request) puede estar indicando que algo está mal con los datos del usuario. Pero ¿cómo sabemos qué dato falló? ¿El email ya existe en la base?, ¿Algún campo está vacío?, ¿las fechas no son válidas?.

Y si todo está bien.. ¿Me alcanza con el código 200?

Por este motivo, debemos conocer los datos que vuelven en el body.

TOKEN

Que una API esté publicada a Internet, no significa que sea una API pública. Las APIs cuentan con una protección compleja que permite autenticarse en el sistema, es decir hacer login.

Las APIs no conocen a los dispositivos que le solicitan acciones, no existe la variable de sesión o mecanismos de cliente servidor vistos anteriormente.

La forma que tiene una api para conocer a la persona que está solicitando un recurso privado, es mediante un TOKEN también llamado API KEY.

TOKEN

Cuando un usuario previamente registrado hace login, se le entrega un TOKEN o API KEY que es almacenado de forma local en el dispositivo.

Es una llave que deberá conservar y presentar SIEMPRE que desee obtener un recurso.

Esta llave se genera en el backend al momento del login y puede tener distintos ciclos de vida. Un token puede cambiar con cada login (dejando el anterior sin efecto) y puede tener un tiempo de vida.

BODY

Hay solicitudes de tipo POST, PUT (no aplica para GET) que necesitan información relevada en el front end. Por ejemplo, un registro de usuario deberá enviar a la API, todos los datos del usuario a dar de alta. Estos datos se envían en el BODY de la request.

HEADERS

Los headers o cabeceras son parte del proceso de establecimiento de una petición para obtener una respuesta a un servidor o plataforma web.

Son la parte central de los HTTP requests y responses, y transmiten información acerca del navegador del cliente, de la página solicitada y no puede ser vista a simple vista, si no que requiere de un analizador HTTP para poderlos ver.

En los headers se determina el type de la request (si es texto, archivo, JSON), y es aquí donde se incluye el TOKEN.

HTTP Request

Teniendo en cuenta todo lo anterior una request realizada en un fetch puede contener:

- URL del endpoint (siempre).
- Headers con datos de type y si corresponde un TOKEN o API KEY
- Method
- Body

Con toda esta información, cuando se resuelve la promesa regresa la response que esta compuesta por:

- Status code
- Header
- Body (para ver el body es necesario capturar la promesa del response.json()).

HTTP Request

Este es un ejemplo de un fetch completo. Los nombres del algunas variables y el formato del body dependen de la documentación, pero se presenta a modo informativo.

```
fetch(`http://miapp.com/productos`, {
    method: "POST",
    headers: {
        "Content-Type": "application/json",
        "apiKey": "0cc175b9c0f1b6a831c399e269772661",
    },
    body: {
        nombre: "Harina",
        tamanio: "1k",
        precio: 80
    }
})
    .then(function (response) {
        return response.json();
    })
    .then(function (data) {
        console.log(data);
    })
```

HTTP Request

El flujo normal de una response se cumple siempre que la promesa retorne **Resolve**.

Sin embargo la promesa puede retornar un **Reject.**

Por este motivo, el último bloque de la solicitud se encarga de gestionar el Reject mediante un catch.

```
fetch(`http://miapp.com/productos`, {
    method: "POST",
    headers: {
        "Content-Type": "application/json",
        "apiKey": "0cc175b9c0f1b6a831c399e269772661",
    },
    body: {
        nombre: "Harina",
        tamanio: "1k",
        precio: 80
    }
})
    .then(function (response) {
        return response.json();
    })
    .then(function (data) {
        console.log(data);
    }).catch(function (error) {
        console.log(error);
    })
```