Desarrollo Asincrónico.

Páginas dinámicas.

Ya sabemos armar páginas dinámicas. Esto por el momento nos trae nuevos beneficios en relación a armar páginas estáticas:

- Permite la inyección de datos
- Permite la carga dinámica de templates
- Permite el uso de partials

Aún nos está faltando algunas cosas

- Conectarnos con una DB
- Armar nuestros propios servicios

Cuando hablamos de servicios… ¿Qué son exactamente?

Servicios de terceros.

Un servicio de un tercero es un programa al cuál nos podemos comunicar y nos devuelve algo específico, por ejemplo:

- Pokemones (https://pokeapi.co/)
- Productos (https://fakestoreapi.com/)
- Usuarios (https://reqres.in/)

Más específicamente vamos a estar hablando de REST API

REST API

API: Application Programming Interface

Una **API** es un conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones. Suele considerarse como el contrato entre el proveedor de información y el usuario, donde se establece el contenido que se necesita del consumidor (la llamada) y el que requiere el productor (la respuesta).
https://www.redhat.com/es/topics/api/what-are-application-programming-interfaces

En otras palabras:

- Usuario hace una consulta a un servicio
- Servicio se encarga de darle una respuesta

REST: Representational State Transfer

Se conectan un cliente y un servidor y no se guarda ningún estado.

- Cada vez que me conecto con el servicio tengo que decirle quién soy.
- El cliente le pasa toda la información necesaria.
- Se trabajan las operaciones más importantes de tipo CRUD utilizando los verbos
  - POST
    - Agregar algo a una DB
  - GET
    - Traer algo de una DB
  - PUT
    - Modificar algo
  - DELETE
    - Eliminar algo

- Separación del cliente con el servidor (Wordpress - PHP)
- Escalabilidad

No me queda claro… REST API

Ya vamos a estar armando nuestras propias APIS, pero primero vamos a aprender a consumir algunas para que sea más fácil entender el armado.

Existe una persona en el mundo que armó una REST API que si el cliente le pide productos, esta se conecta con una base de datos y devuelve productos.

https://fakestoreapi.com/

Vamos a usarla en nuestra tienda de Ecommerce.

Utilizando FakeStore API

El objetivo que tenemos, es mostrar una lista de productos. Esos productos van a venir de la API de productos.

El primer desafío es hacer una consulta “al mundo exterior”… ¿Vieron que si agregamos la URL al navegador, nos devuelve un JSON con productos? Tenemos que lograr lo mismo desde nuestro BE y esto lo vamos a hacer mediante un paquete de terceros.

El paquete se llama request (entramos, vemos que está deprecado) y las personas de POSTMAN decidieron continuarlo, así que utilizaremos postman-request. Como siempre, para instalarlo en nuestro proyecto:

```
npm i postman-request
```

Lo que nos va a permitir este paquete es hacer una consulta al “exterior” o más técnicamente una consulta http.

```
const request = require('postman-request');
request('http://www.google.com', function (error, response, body) {
  console.log('error:', error); // Print the error if one occurred
  console.log('statusCode:', response && response.statusCode); // Print
  console.log('body:', body); // Print the HTML for the Google homepage.
});
```

1. Requerimos el paquete
2. Vemos que recibe dos parámetros
    1. Una URL a la cuál le vamos a hacer la petición
    2. Una función callback que se ejecuta una vez que esté listo el resultado.

Si observamos, vemos que la función callback recibe 3 parámetros.

1. Error (Si la llamada devuelve un error)
2. Response (Si la llamada devuelve una respuesta)
3. Body (El cuerpo de la respuesta)

Vamos a pasar esto a Arrow functions y vamos a hacer nuestra primer consulta.

```
request('https://fakestoreapi.com/products', (err, res, body) => {
  console.log(err);
  console.log(res);
  console.log(body);
});
```

Como vemos, esta función nos imprime algún tipo de respuesta. Dependiendo cada respuesta, tenemos que hacer algo en consecuencia.

Para no *contaminar* nuestro app.js, vamos a crear esta función en un custom module y vamos a hacer que devuelva o un error, o una lista de productos.

Request de products como custom module

1. Me creo una carpeta utils
2. Me creo un módulo products
3. Me creo una función getAllProducts
    1. Dentro de esta función hago el request.

```
const request = require('postman-request');

const getAllProducts = () => {
  request('https://fakestoreapi.com/products', (err, res, body) => {});
}

module.exports = {
  getAllProducts
}
```

Ahora tenemos que gestionar los errores

1. Si hay un error, muestro un mensaje de error de conexión con el servicio.
2. Si hay una respuesta, chequeo que exista body.
    1. Si existe body, devuelvo eso.

```
const getAllProducts = () => {
  request('https://fakestoreapi.com/products', (err, res, body) => {
    if(err) {
      return 'Error conectándose a la API';
    }
    if (res) {
      if(body) {
        return body;
      } else {
        return 'No se encontraron productos';
      }
    }
  });
}
```

2. Si no existe body, devuelvo que no encontré productos.

Este tipo de manejo de errores, va a cambiar. Se puede optimizar mucho más, pero lo vemos en breve.

Ahora que tenemos la función que trae productos, podemos integrarla en nuestro proyecto.

Integrando productos.

1. En el app.js requerimos el custom module

```
const products = require('./utils/products');
```

2. Ejecutamos la función que trae todos los productos y guardamos la respuesta en una variable de productos. Si queremos que esto pase cuando la gente entra a la página principal, lo debemos hacer sobre el get que devuelve la página principal.

```
app.get('', (req, res) => {

  const allProducts = products.getAllProducts();
  console.log(allProducts);

  res.render('index', {
    title: 'E-commerce',
    allProducts
  });
});
```

¿Funciona?

Solución 1 - Callback

Lo anterior no funciona porque, lógicamente tenemos un problema de asincronismo:

Estoy llamando a una función que tarda en devolverme algo, entonces ejecuto getAllProducts(); pero eso tarda, así que la variable allProducts es *undefined*. De alguna manera tengo que esperar esos resultados y una manera, es mediante callbacks.

Paso a paso

1. Yo se que getAllProducts va a recibir una función como parámetro, que se va a ejecutar cuando la información esté lista.

```
const getAllProducts = (callback) => {}
```

2. Cuando haya respuesta de la llamada a la API, ya sea por error o por éxito, vamos a ejecutar la función de callback. Recordemos.

   CUANDO TENGO INFO DISPONIBLE, LA MANDO EJECUTANDO LA FUNCIÓN

Pensemos un poco en la función callback… esta función se va a ejecutar cuando haya error o éxito… entonces tal vez pueda recibir dos parámetros, algo como:

```
const callback = (err, data) => {}
```

Vamos a decir:

1. Si la función devuelve error, le vamos a pasar:
   1. Un mensaje de cuál es el error
   2. El error que venga del servicio
   3. Data undefined
2. Si la función no devuelve error, le vamos a pasar:
   1. Undefined en el error
   2. Información del servicio en el data

Opción 1

```
callback({msg: 'Error en la conexión', err}, undefined);
```

Opción 2

```
callback(undefined, body);
```

Todo Junto:

```
const getAllProducts = (callback) => {
  request('https://fakestoreapi.com/products', (err, res, body) => {
    if(err) {
      return callback({msg: 'Error en la conexieon', err}, undefined);
    }
    if (res) {
      if(body) {
        return callback(undefined, body);
      } else {
        return callback({msg: 'No se encontrarons productos', err}, undefined);
      }
    }
  });
}
```

Finalmente nos queda definir esa función callback que estamos pasando. getAllProducts recibe una función callback, pero ¿Como es esta función? Por lo pronto sabemos, según lo que definimos arriba:

1. Es una función que recibe un parámetro de error
2. Es una función que recibe un parámetro de éxito

Entonces algo como

```
const callback = (err, data) => { }
```

Si esto lo hacemos anónimo y se la mandamos como parámetro a getAllProducts tendremos algo como:

```
products.getAllProducts((err, data) => { /* Recibo el error o la
respuesta */  });
```

Uniendo las partes

1. Creamos un módulo que trae productos
    1. Este módulo es asincrónico por lo cuál cuando busca la información de productos, tarda en devolvernos una respuesta. Para solucionar esto usamos un callback

Creo una función que cuando tenga información, ejecuta la función que recibió como parámetro (callback). Como esto puede tener errores, le mando dos parámetros: uno de error y otro de éxito.

2. Importamos el módulo en el app.js
3. Cuando alguien entra al índex, ejecutamos la función que trae productos y le pasamos la función que se va a ejecutar cuando estén listos.
4. Si hay error, le mando un mensaje de error al cliente
5. Si hay éxito le mando un mensaje de éxito al cliente

Todo junto:

```
app.get('', (req, res) => {
  products.getAllProducts((err, data) => {
    if (err) {
      return res.send(err.msg);
    }
    const products = JSON.parse(data);
    return res.render('index', {
      title: 'E-commerce',
      products
    });
  });
});
```

“Ejecuto la función que me trae todos los productos y le paso la función que se va a ejecutar una vez que los tenga.”

Cliente

Vimos que tenemos un servicio que armó un tercero que trae un montón de productos. Ese servicio lo consumimos desde el BE y armamos una página.

Tranquilamente podríamos crear ese servicio nosotros, sin necesidad de salir a buscar algo de un tercero, siempre y cuando tengamos nuestra propia DB.

La pregunta que surge es…

¿Se puede llamar a este servicio del lado del cliente **sin tener que pasar** por el BE?

Función fetch

Fetch es una función disponible en JavaScript **desde el lado del navegador,** (no es propia de JS, no está en NODE, por eso requerimos a un paquete externo)

Fetch nos va a dar como respuesta algo asincrónico, esto quiere decir que vamos a necesitar algo como un callback para esperar la respuesta.

La función fetch usa un método para esperar la información que se llama .then() y recibe como parámetro un método con una respuesta. Esto es parte de lo que se llaman **promises** y vienen a dar *más de facilidad* en cuando a la espera de resultados asincrónicos. Esto se va a explorar más adelante cuando integremos una DB en nuestro proyecto.

“Buscá la información de esta URL y cuando esté la información, traémela en la palabra response”

```
fetch('https://fakestoreapi.com/products').then((response) => {});
```

Una vez que tengas esa información, convertla a JSON y cuando esté, pasámela en la palabra data.

```
fetch('https://fakestoreapi.com/products').then((response) => {
  response.json().then((data) => {
    console.log(data);
  });
});
```

Una vez que tenemos el array de productos, podríamos crear con JavaScript del lado del navegador, las cards con los productos.

Fetch desde Front End - Request desde el BE

Lo que hicimos como primera opción se llama SSR, que es renderizar una página desde el servidor y darle la página resuelta. Lo que íbamos a hacer en la segunda opción es Client Side Rendering que es consumir el servicio desde el lado del cliente. Ambas tienen su pro y contra.

Como saber popular:

React trabaja con CSR

Next se inventó para usar React en SSR

Vue trabaja con CSR

Nuxt se inventó para usar Vue en SSR

Y como estas, hay varias más.

SSR	**CCR**
Optimizado para SEO	Carga de rutas dinámicas sin hacer un nuevo request al BE
Carga inicial más rápida (se construye en el servidor)	NO está optimizada para SEO
No está optimizada para la interacción constante con el usuario	Soporta múltiples interacciones (Twitter)

https://dev.to/alain2020/ssr-vs-csr-2617

¿Cuál elegir?

Depende del proyecto.