Promises

What is a promise?

A **promise** is an object of the Promise class , created to resolve an asynchronous action, that is, an operation that may take a considerable amount of time and we don't want to block the main program thread while it's being processed. The constructor takes a function with two parameters, **resolve** and **reject** . These parameters are, in turn, functions.

resolve is called when the action completes successfully, optionally returning some data, while **reject** is used when an error occurs. A promise ends when one of these two functions is called.

To process the result of a promise, we need to subscribe to it by calling the **then** method , passing it a function that will handle the result. This function will execute when the promise resolves successfully (without errors).

# index.js

```
function getPromise() {
    return new Promise((resolve, reject) => {
        console.log("Promesa llamada...");
        setTimeout(function() {
            console.log("Resolviendo la promesa...");
            resolve(); // Promesa resuelta!.
        }, 3000); // Esperamos 3 segundos y acabamos la promesa
    });
}

// Imprimirá el mensaje pasados 3 segundos (la promesa termina)
getPromise().then(() => console.log("La promesa ha acabado!"));

console.log("El programa continúa. No espera que termine la promesa
(operación asíncrona)");
```

Calling the **then** method in turn returns another promise. This allows us to concatenate multiple calls to that method. The first receives the result of the original promise, while the remaining calls receive whatever the previous then method function returns. They can return any type of value, including another promise. The next **then method** would receive the result of that promise once it's resolved.

# index.js

```
function getPromise() {
    return new Promise((resolve, reject) => {
        setTimeout(function() {
            resolve(1);
        }, 3000); // Después de 3 segundos, termina la promesa
    });
}

getPromise().then(num => {
    console.log(num); // Imprime 1
    return num + 1;
}).then(num => {
    console.log(num); // Imprime 2
}).catch(error => {
    // Si se produce un error en cualquier parte de la cadena de
promesas... (o promesa original rechazada)
});
```

We'll go directly to the **catch** block if we throw an error inside a then section, or if the original promise is rejected with **reject** .

# index.js

```
getProducts().then(products => {
    if(products.length === 0) {
        throw 'No hay productos!'
    }
    return products.filter(p => p.stock > 0);
}).then(prodsStock => {
    // Imprime los productos con stock en la página
}).catch(error => {
    console.error(error);
});
```

Optionally, for readability or code reuse, we can separate the functions from the then (or catch) block instead of using anonymous functions or arrow functions.

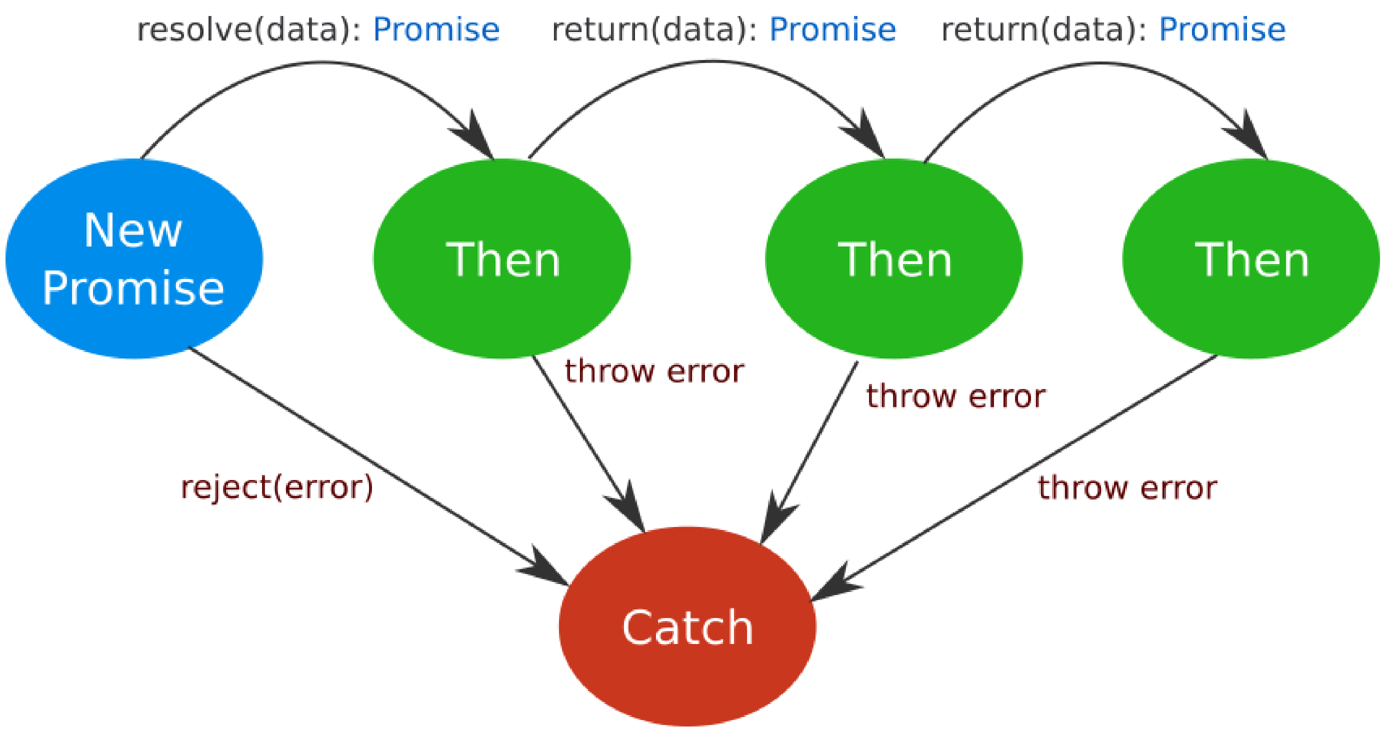
# index.js

```
function filterStock(products) {
    if(products.length === 0) {
        throw 'No hay productos!'
    }
    return products.filter(p => p.stock > 0);
}

function showProducts(prodsStock) {
    // Imprime el producto en la página
}

function showError(error) {
    console.error(error);
}

getProducts()
    .then(filterStock)
    .then(showProducts)
    .catch(showError);
```



Both the then method and the **catch** method can return a value directly or another promise. This means we can use the catch method to recover from an error, by fetching data from another source, for example. What a catch method returns would be collected in a then method that is then concatenated.

There's another method called **finally** , which can be added at the end and always runs, whether there's an error or not. You can use it, for example, to hide a data loading animation.

# index.js

```
promise.then(...).catch(...).finally(() => /*Ocultamos animación*/);
```

Promise.resolve y Promise.reject

Imaginemos por ejemplo. que tenemos una función de la cual se espera que devuelva los datos dentro de una promesa, llamando a un servicio web por ejemplo. Pero puede ser que tengamos los datos ya cargados previamente y se pueda ahorrar dicha llamada.

Como lo esperable es que devuelva siempre una promesa, para matener la consistencia de tipos, puede devolver inmediatamente dicho valor que ya tenemos almacenado encapsulado en una promesa con **Promise.resolve**, o una promesa con error usando **Promise.reject**. De esta forma nos ahorramos crear la promesa manualmente llamando al constructor (solo en el caso de promesas que resuelvan inmediatamente).

# index.js

```
function getInstantPromise() {
    return Promise.resolve(25); // Equivale a: return new
Promise((resolve, reject) => resolve(25));
}

getInstantPromise.then(val => console.log(val)); // Imprime 25
(inmediatamente)
function getRejectedPromise() {
    return Promise.reject('Error'); // Equivale a: new
Promise((resolve, reject) => reject('Error'));
}

getRejectedPromise.catch(error => console.log(error)); // Imprime
"Error" (inmediatamente)
```

Otros métodos de Promise

Promise.all

A veces tenemos que crear varias promesas en paralelo (varias llamadas a un servidor para obtener diferentes datos). Si necesitamos esperar para hacer algo una vez todas terminen en lugar de procesarlas por separado, podemos agruparlas en un array y utilizar el método Promise.all. Una vez haya terminado la última, se ejecutará el método then asociado que recibirá un array con los resultados de cada promesa.

# index.js

```
function make3AjaxCalls() {
    let p1 = Http.get(...); // Promesa 1
    let p2 = Http.get(...); // Promesa 2
    let p3 = Http.get(...); // Promesa 3
    return Promise.all([p1, p2, p3]);
}

make3AjaxCalls.then(results => /* results -> array con los 3
valores devueltos */ );
```

Si alguna de estas promesas es rechazada (error), se ignorará inmediatamente a las promesas pendientes, ejecutando directamente el método catch (si lo hubiera).

Promise.allSettled

Parecido al método anterior. Si queremos esperar a que todas las promesas acaben, aunque una o varias devuelvan un error, tenemos el método **Promise.allSettled**. En este caso, el método then recibirá un array de objetos (1 por promesa). Cada objeto tendrá 3 posibles propiedades:

- **status**: Contendrá los valores "fulfilled" (todo ha ido bien), o "rejected" (error).
- **value**: Cuando la promesa tenga el estado "fulfilled", contendrá el valor devuelto.
- **reason**: Cuando la promesa tenga el estado "rejected", contendrá el error.

# index.js

```
function make3AjaxCalls() {
    let p1 = Http.get(...);
    let p2 = Http.get(...);
    let p3 = Http.get(...);
    return Promise.allSettled([p1, p2, p3]);
}

make3AjaxCalls.then(results => results.forEach(result => {
    if(result.status === 'fulfilled') { // Si todo ha ido bien
        console.log(result.value);
    } else { // result.status === 'rejected' -> Ha habido un error
```

```
            console.log(result.reason);
        }
    }));
```

Promise.race

Otro método interesante es **Promise.race**. En este caso, devuelve el resultado de la primera promesa que termina, ignorando el resto. Útil, por ejemplo, si podemos obtener un dato de varias fuentes diferentes, y queremos la más rápida.

# index.js

```
function make3AjaxCalls() {
    let p1 = Http.get(...);
    let p2 = Http.get(...);
    let p3 = Http.get(...);
    return Promise.race([p1, p2, p3]);
}

make3AjaxCalls.then(result => /* Valor de la promesa más rápida */);
```

Async / Await

Las instrucciones **async** y **await** fueron introducidas en la versión ES2017. Se utilizan para que la sintaxis al trabajar con promesas sea más amigable.

await se utiliza para esperar que una promesa termine y nos devuelva su valor. El problema es que eso bloquea la ejecución del siguiente código hasta que la promesa acabe. Esto implicaría bloquear el programa durante un tiempo indeterminado. Por ello, sólo podemos usarlas dentro de funciones de tipo **async**. Estas funciones se ejecutan de forma asíncrona al resto del código, lo que impide que bloqueen el hilo principal.

An async function always returns a promise, which contains the value returned by the return statement. If there is no return statement, it also returns a promise, albeit an empty one (Promise<void>).

# index.js

```
async getProducts() { // Devuelve una promesa (async)
    const resp = await Http.get(`${SERVER}/products`); // Esperamos
la respuesta del servidor (Promesa)
    return resp.products;
}

// Lo mismo sin async/await
getProducts() {
  return Http.get(`${SERVER}/products`).then((response) => {
        return response.products;
  });
}
```

Dynamic import with promises

In addition to the import statement , a module can be imported with the **import() function** . This function returns a promise that resolves once the module has been downloaded and processed, returning an object containing everything the module exports. This allows us to perform dynamic imports, where the module name can be a variable, or simply not load a module until we actually need it for efficiency and performance reasons.

# index.js

```
const lang = /^es\b/.test(navigator.language) ? 'es' : 'en';
import(`./message-${lang}.js`).then(m => console.log(m.message));
```

# message-en.js

```
const lang = /^es\b/.test(navigator.language) ? 'es' : 'en';
import(`./message-${lang}.js`).then(m => console.log(m.message));
```

# message-es.js

```
const lang = /^es\b/.test(navigator.language) ? 'es' : 'en';
import(`./message-${lang}.js`).then(m => console.log(m.message));
```

Another very common use is not to load a module until it is needed, so that the initial loading of the program is faster if we avoid loading heavy modules (libraries) such as image processing, etc. until they are needed.

In short : While loading modules with import is static, and you must process the module and load it before continuing (if it hasn't been done previously), loading with the import() function is asynchronous and can be done at any point in the code.

top-level await

Since ES2020, we have included **top-level await** , which means we don't have to create async functions to be able to use await in the main code. However, we must remember that the execution of the rest of the code will be blocked until the **await** statement is resolved . We should only use this if the rest of the code in the current module is completely dependent on the outcome of the promise (in this case, the module we loaded). Otherwise, or if in doubt, don't use it.

Using it to load modules on which the rest of the current module's code depends, and which we cannot load with import (the name depends on a variable, or the loading of one or another module depends on a condition), is justified.

# index.js

```
const lang = /^es\b/.test(navigator.language) ? 'es' : 'en';
const m = await import(`./message-${lang}.js`)
console.log(m.message);
```

Fetch API (AJAX)

An AJAX call is a request to the web server made from JavaScript after the page has loaded. The most important feature of using these calls is that you don't need to reload the page (current or another) from scratch. It's therefore a good option for saving bandwidth, as the server sends only the data it needs (for example, to update information about a product on the website), and it also significantly improves performance.

AJAX is an acronym for **Asynchronous JavaScript And XML** , although today **JSON** is the most commonly used format for sending and receiving data from the server (JSON integrates natively with JavaScript). **Asynchronous** means that if we make a request, we won't receive the response immediately. We'll use **promises** to manage these requests.

Basic HTTP Methods

When we make an HTTP request, the browser sends to the server: headers (used to identify the client, language preferences, etc.), the HTTP request method, the URL and data or body of the request (if necessary).

The most commonly used request methods are:

- **GET** → This is generally a request to obtain data without modifying it. If the backend works with SQL databases, it would generate one (or several) **SELECT** queries . This type of request typically does not send associated data; all the information for the server is contained in the URL. The response will typically contain the data we requested from the server (as long as no errors occur).
- **POST** → Typically used for requests that involve creating a new resource in the backend (user registration, comment, product, etc.). This is equivalent to performing an **INSERT** query in SQL databases. The data to be recorded is included in the body of the request, not in the URL. There are requests, such as login requests, that, while they don't involve creating a resource, are made using POST to hide sensitive data like the password and not expose it in the URL.
- **PUT** → This operation updates an existing resource on the server. It's equivalent to **UPDATE** in SQL. The information identifying the object to be updated is usually sent in the URL (as in a GET request), and the data to be modified is sent in the call body (as in a POST request). Some servers differentiate between **PUT** (replacing a resource, forcing all fields to be sent even if some haven't changed) and **PATCH** (allowing only those fields whose value has changed to be included).
- **DELETE** → This operation deletes data from the server, which is equivalent to the SQL **DELETE** operation . Information identifying the object to be deleted will be sent in the URL (as in GET).

The Fetch API

This new JavaScript API makes our work much easier compared to traditional server calls using **XMLHttpRequest** . It's supported by all modern browsers.

To use this API, we have the global **fetch** function . This function returns a promise with a Response object. The promise can be rejected when there is an error communicating with the server, but not when the server returns an error code. We must manually control this and decide whether to throw an error.

GET request

By default, if we don't specify a method, the request is a GET request. Simply enter the URL you want to access and process the response.

If the response contains data (usually with these types of requests), we'll need to call the Response object's **json() method to deserialize the response as a JSON object. This method returns a promise containing the resulting JSON object. Other methods we can use are: text()** → Plain text response, **blob()** → The response is a file, **arrayBuffer()** → Byte array.

Example of GET request:

# index.js

```
function getProductos() {
    fetch(`${SERVER}/products`).then(resp => {
        if(!resp.ok) throw new Error(resp.statusText); // El
servidor devuelve un código de error
        return resp.json(); // Promesa
    }).then(respJSON => {
        respJSON.products.forEach(p => mostrarProducto(p));
    }).catch(error => console.error(error));
}
```

# answer

```
function getProductos() {
    fetch(`${SERVER}/products`).then(resp => {
        if(!resp.ok) throw new Error(resp.statusText); // El
servidor devuelve un código de error
        return resp.json(); // Promesa
    }).then(respJSON => {
        respJSON.products.forEach(p => mostrarProducto(p));
    }).catch(error => console.error(error));
}
```

Alternatively, if we use the **async/await** syntax :

# index.js

```
async function getProductos() {
    try{
        const resp = await fetch(`${SERVER}/products`)
        if(!resp.ok) throw new Error(resp.statusText); // El
servidor devuelve un código de error
        const respJSON = await resp.json(); // Promesa
        respJSON.products.forEach(p => mostrarProducto(p));
    } catch(error) {
       console.error(error);
    }
}
```

POST request

Normally, when we make a POST request, we send the contents of a form to the server. In this case, we have two options for submitting the form, which will depend on what the server we're sending the information to accepts. The first would be with the **'multipart/form-data'** encoding (Content-Type header) (default).

# index.html

```
<form id="formProducto">
    <p><input type="text" name="nombre" id="name"
placeholder="Nombre" required></p>
    <p><input type="text" name="descripcion" id="description"
placeholder="Descripcion" required></p>
    <p>Foto: <input type="file" name="foto" id="foto" required></p>
    <p><img id="imgPreview" src=""></p> <!-- Para previsualizar la
imagen seleccionada -->
    <button type="submit">Añadir</button>
</form>
```

# index.js

```
<form id="formProducto">
    <p><input type="text" name="nombre" id="name"
```

```
placeholder="Nombre" required></p>
    <p><input type="text" name="descripcion" id="description"
placeholder="Descripcion" required></p>
    <p>Foto: <input type="file" name="foto" id="foto" required></p>
    <p><img id="imgPreview" src=""></p> <!-- Para previsualizar la
imagen seleccionada -->
    <button type="submit">Añadir</button>
</form>
```

If the server requires us to send the data in JSON format, we must also serialize any files as images in Base64 to send them along with the rest of the data. We must also specify the ' **application/json** ' value in the **Content-Type** header , as well as serialize the data to be sent using the **JSON.stringify** method .

# index.html

```
<form id="formProducto">
    <p><input type="text" name="nombre" id="name"
placeholder="Nombre" required></p>
    <p><input type="text" name="descripcion" id="description"
placeholder="Descripcion" required></p>
    <p>Foto: <input type="file" name="foto" id="foto" required></p>
    <p><img id="imgPreview" src=""></p> <!-- Para previsualizar la
imagen seleccionada -->
    <button type="submit">Añadir</button>
</form>
```

# index.js

```
<form id="formProducto">
    <p><input type="text" name="nombre" id="name"
placeholder="Nombre" required></p>
    <p><input type="text" name="descripcion" id="description"
placeholder="Descripcion" required></p>
    <p>Foto: <input type="file" name="foto" id="foto" required></p>
    <p><img id="imgPreview" src=""></p> <!-- Para previsualizar la
imagen seleccionada -->
    <button type="submit">Añadir</button>
</form>
```

PUT request

The mechanism for making a PUT (or PATCH) request is the same as for a POST request. Often, we must specify the identifier of the resource we're going to update on the server in the URL. As with a POST request, the rest of the data will be included in the request body in 'multipart/form-data' format (by default) or in 'application/json' format.

# index.js

```
// ...
const resp = await fetch(`${SERVER}/products/${id}`, {
    method: 'PUT',
    body: JSON.stringify(producto),
    headers: {
        'Content-Type': 'application/json'
    }
});
// Resto del programa
```

DELETE request

This is the simplest type of request since it contains no data (everything is in the URL), and the server generally doesn't return any data either. Simply analyzing the response code will tell you whether the request was successfully deleted.

# index.js

```
async function borraProducto(id, prodHTML) {
    try {
        const resp = await fetch(`${SERVER}/products/${id}`, {
            method: 'DELETE'
        });
        if (!resp.ok) throw new Error(resp.statusText);
        prodHTML.remove(); // Eliminamos el producto del DOM si
todo ha ido bien
    } catch (error) {
        console.error("Error borrando producto: " + error);
    }
}
```

Organizing AJAX code into classes

In order to minimize the code needed to make requests with **fetch** and centralize the management of these requests (handling server responses with errors, etc.), we can create classes that help us have cleaner code.

For example, based on how certain libraries or frameworks like Angular work, we could have a class called **Http** with the necessary methods to make GET, POST, PUT and DELETE requests more directly.

# index.js

```
export class Http {
  async ajax(method, url, body = null) {
    const json = body && ! (body instanceof FormData);
    const headers = body && json ? { "Content-Type":
"application/json" } : {};
    const resp = await fetch(url, { method, headers, body : (json ?
JSON.stringify(body) : body) });

    if (!resp.ok) throw new Error(resp.statusText);

    if (resp.status != 204) {
      return resp.json();
    } else {
      return null; // 204 implica una respuesta sin datos
    }
  }

  get(url) {
    return this.ajax("GET", url);
  }

  post(url, body) {
    return this.ajax("POST", url, body);
  }

  put(url, body) {
    return this.ajax("PUT", url,  body);
  }

  delete(url) {
    return this.ajax("DELETE", url);
  }
}
```

Additionally, for each resource we manage in our application, we could create a class that handles requests related to that resource, with a method for each type of operation we perform. Since these are auxiliary classes that reside in different modules, it's not recommended that they perform operations with the DOM (only with data). This way, we leave DOM management to the main module (the one referenced from the HTML).

To stick to a coding philosophy, we're going to create classes similar to those called **services** in the Angular framework . The class that manages products, for example, will be called **ProductService** . This class's job is to manage how requests are made and the data we receive from the backend we're using.

# index.js

```
import { Http } from './http.class.js';
import { SERVER } from './constants.js';

export class ProductoService {
  #http;
  constructor() {
    this.#http = new Http();
  }

  async getProductos() {
    const resp = await this.#http.get(`${SERVER}/productos`);
    return resp.productos;
  }

  async add(producto) {
    const resp = await this.#http.post(`${SERVER}/productos`,
producto);
    return resp.producto;
  }

  async update(producto) {
    const resp = await
this.#http.put(`${SERVER}/productos/${producto.id}`, product);
    return resp.producto;
  }

  delete(id) {
    return this.#http.delete(`${SERVER}/productos/${id}`);
    }
  }
```

This way, the code for handling requests in the main module is greatly simplified. Here we can see an example of retrieving product data or inserting a new product.

# index.js

```
import { ProductoService } from './producto-service.class.js';

let productoService = new ProductoService();

async function getProductos() { // Obtener productos y añadirlos al DOM
    const productos = await productoService.getProductos();
    productos.forEach(p =>  mostrarProducto(p));
}

async function addProducto() { // Añadir un producto al servidor e insertarlo en el DOM
    let producto = {
        nombre: document.getElementById("nombre").value,
        descripcion: document.getElementById("descripcion").value,
        foto: imagePreview.src
    };

    const p = productoService.add(producto);
    mostrarProducto(p);
}

// Resto del código
```