p4-interazione-API

4. External API usage

> - **React for Dummies**, I.T.T. "Blaise Pascal" @ Cesena, IT
> - Written by: Nicholas Magi - nicholas.magi24[@]gmail.com

Table of contents

3.1 FlickrAPI

This part does not relate specifically to React, but it is a nice example to study how to interact to an external API in a React application.

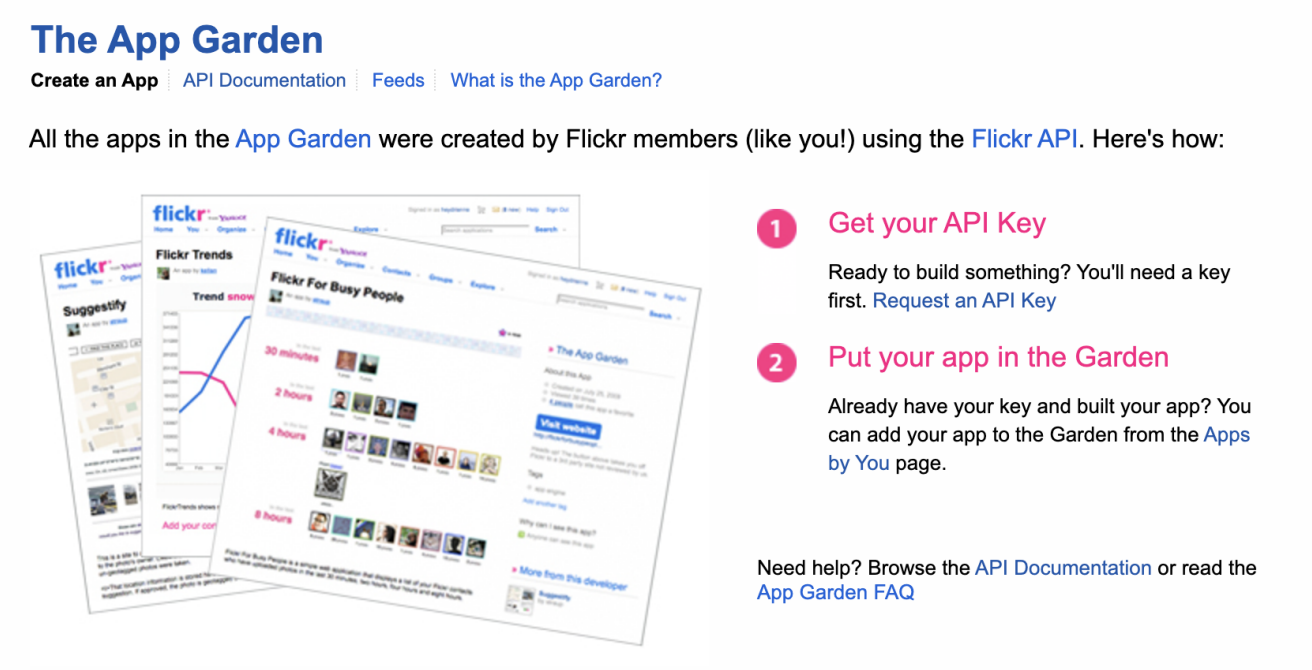
Consider the following website: https://compagnia-dei-pitagorici.web.app/. In the "Rassegne" section you can click one of the available cards to see the detail of the selected event (achieved via **routing**). In the specific page there's a gallery that automatically loads many different photos. There could be many ways to create the gallery, but the chosen one is via an **external API**. The API used is **FlickrAPI**, which relies on **Flickr** to store and organise photos.

3.1.1 Register to the service & upload some pictures

Go to https://flickr.com/ and create an account. Then choose the pictures you want to upload, and upload them. You may want to organise them into **Albums**.

3.1.2 Get your API key

Go to the API documentation and look for **App Garden** > **Create an app**. Click it and you should see the following page:



Click on **Request an API Key** > **APPLY FOR A NON-COMMERCIAL KEY** and complete the form you're given. Then your new key should appear! From now on, remember that you can see your key in the **App Garden** — which is just where you got the key from.

3.1.3 Study the documentation

You know what are your needs, so sieve through the documentation (API documentation) and look for what solves your problem. *No further help is given, but remember that we may*

*have organised our pictures in **albums**.*

3.2 Before fetching: Promise in JavaScript

3.2.1 Synchronous vs. Asynchronous operations

Asynchronous operations are, of course, different from synchronous ones. We typically develop a software thinking it synchronously, meaning we write instructions meant to be executed **sequentially** and **in the order we write them**. Think about a simple Console application in C#: you can define as many classes with many methods inside it as you want, but there will always be a **main** method that represents the principal control line from which the program **starts**, **calls the eventual methods** and **ends**.

```
static void Main(String[] args)
{
        // [...]
        int a = 42;
    int b = 119;
    int c = a + b;
    // Method call: when it carries out its task,
    // the program comes back to 'Main()'.
    Console.WriteLine(c);
    Console.ReadKey();
}
```

There are some operations, however, **whose synchronous implementation could extremely slow down your application**. What we want to develop represents the perfect example: getting some images from an external server. If the fetch operation was synchronous, the application would do nothing, render nothing and wait until all the images gets downloaded in the app. We don't want that to happen, because if the waiting time is approx. > 5 or 10 seconds the user would lose his focus and close the page.

In order to avoid that, JavaScript provides us the **Promise** object.

> **Definition**
>
> A Promise is a standard built-in object used to manage asynchronous operations (like fetching data) in JavaScript. The following explanation comes directly from the official documentation, so go check it for more information.

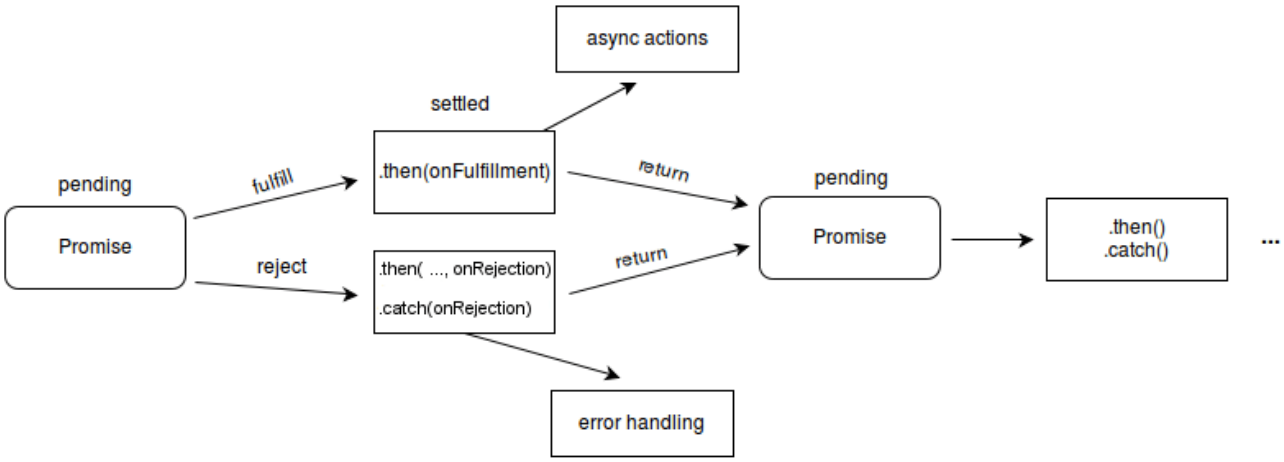
3.2.2 Promise: a proxy for the unknown

The idea behind a `Promise` is simple: given an asynchronous method, instead of immediately returning the final value, it returns a *promise* to supply the value at some point in the future.

A `Promise` is in one of these states:

- *pending*: initial state, neither fulfilled nor rejected.
- *fulfilled*: meaning that the operation was completed successfully.
- *rejected*: meaning that the operation failed.

Here follows a nice schema that represents how `Promise` is designed:



3.2 Let's fetch some data in React

Once you figured out what you need to use, let's use it inside our React application. Recall what you know about `hooks`: there's a particular one responsible for the *synchronization of the app with external services*. We're talking about `useEffect()` hook, but it is not the only one we implement here.

Think about it: we'll receive from an external server some data, where do we store it? We need something that can be changed and updated over time, and the changes must be reflected to the view. The `useState()` hook fits perfectly our needs.

The code we need to implement is the following:

```
import { useEffect, useState } from "react";
import { QUERY_URL, PICTURES_URL } from "./secrets";

function App() {

        const [photos, setPhotos] = useState([]);

        useEffect(() => {
                const fetchPhotos = async () => {
                try {
                        const res = await fetch(QUERY_URL, {
                                mode: "cors",
                                method: "GET"
                        })
```

```
                        const data = await res.json();
                        const fetchedResult = await Promise.all(
                                data.photoset.photo.map(async (pic) => {
                                        const url = PICTURES_URL(
                                                pic.server, pic.id,
pic.secret
                                        )
                                        return (
                                                <img
                                                        key={pic.id}
                                                        className="w-75"
                                                        src={url}
                                                        alt={`Photo
${pic.id}`}
                                                />
                                        )
                                })
                        )
                        // Update state after all images are processed
                        setPhotos(fetchedResult);
                } catch (err) {
                        console.error("Error fetching photos:", err);
                }
        };
        fetchPhotos();
        }, []);

        return (
        <>
                <h1 className="display-1 text-center pt-5">Galleria</h1>
                <Gallery pictures={photos} />
                <Footer/>
        </>
        );
}
```

It can look a bit scary at first glance, but let's try to break it down:

1. In the first part of you see:

```
const [photos, setPhotos] = useState([]);
```

which declares the array `photos` as state of the component `App()`.

2. Then there's the `useEffect()` call, which starts with an **asynchronous function declaration** (stored in `const fetchedData`). The first line of the function is:

```
const res = await fetch(QUERY_URL, { mode: "cors", method: "GET" })
```

which fetches the data from `QUERY_URL` (hidden to prevent you from directly copying it), specifying that it's a `GET` request that uses `CORS` (necessary to avoid some errors). In short, we are saying that we want only to **read data** from a server which does not run under the same domain of our application (**Cross-Origin Resource Sharing**).

Note the `await` keyword: it's used to perform asynchronous operations (declared so with the `async` keyword).

The `fetch` does not return directly the value we are looking for, but a `Promise` wrapping it. We need to "unpack" the data — hoping it gets retrieved correctly. That's why we have:

```
const data = await res.json();
```

The `json()` operation is **asynchronous** as well, since it's applied to a `Promise` object. The most important part, however, starts in the next line:

```
const fetchedResult = await Promise.all(
        data.photoset.photo.map(async (pic) => {
                const url = PICTURES_URL(
                        pic.server, pic.id, pic.secret
                )
        // [...]
```

We want to create the corresponding `img` for each photo contained in the photoset. Each `img` has a link as its `src`, which means it takes some time to effectively load in our application — and that this time is different for every picture.

So we can think of all the photos as different `Promise`s, and use the `Promise.all()` static method that takes `n` Promises and returns a single Promise that **fulfils when all the `n` do**. That's what happens in the core of the function. The result gets stored in a temporary array, which becomes the value of the `photos` state:

```
// Update state after all images are processed
setPhotos(fetchedResult);
```

That's it. Once done, just call the outer function (inside the `useEffect()`):

```
fetchPhotos();
```

You now have an array of `img` elements, you can choose whatever method you want to display them.