Promises and async-await

For all the exercises below I suggest you just create a single node-project to hold all solutions.

In this project, create a folder `exercises` and create a file per exercise and test by executing that file. For exercises that require a REST-API, export your solution as a module, and use it with a relevant route-handler.

1 Wrap a callback implementation in a promise based implementation

As you have seen, promises are much more convenient to use, compared to asynchronous callbacks. And with `async/await` you will think of asynchronous callbacks as “ludicrous”. In this exercise you must wrap a callback based design, in a promise, so you can use it with the `.then` notation and also with async/await

In this exercise you must create a design to produce an object with 6 *secure randoms* as sketched below:

```
{
  "title": "6 Secure Randoms",
  "randoms": [
    {"length": 48,"random": "A string with 48 random hex-characters"},
    {"length": 40,"random": "A string with 40 random hex-characters"},
    {"length": 32,"random": "A string with 32 random hex-characters"},
    {"length": 24,"random": "A string with 24 random hex-characters"},
    {"length": 16,"random": "A string with 16 random hex-characters"},
    {"length": 8,"random":  "A string with 8 random hex-characters"}
  ]
}
```

The 6 strings must be presented in the order given above.

You must use Nodes built in crypto module, as sketched below (the asynchronous version of the function):

```
require('crypto').randomBytes(SIZE, function(err, buffer) {
  let secureHex = buffer.toString('hex');
});
```

a) First implement the functionality without promises, using callbacks.

Hint: You don't have to complete this implementation, but implement it for the first 2-3 numbers so you have an example of the "pyramid of doom". Also consider the code if you were asked to produce 100 randoms ;-)

b) Use Promises to solve the problem.
Hints:

1) Create a function `makeSecureRandom(size)` that returns a promise, using the callback based design,provided by the `randomBytes(..)` method.
2) Since the result from one calculation does not influence the next (only order matters), use `Promise.all(..)` to execute the operations in parallel.

c) Refactor your solution into a module and export it
Extra: You could refine what you have created as sketched below, to make it more reusable:

```
const getSecureRandoms = require("./ex1-crypto-module");

getSecureRandoms([48,40,32,24,16,8]) //any size and values ok, as long as integers
.then(randoms => console.log(randoms))
```

d) Create a new file and test the module, like so:

- First, using plain promises
- after that, using async/await

e) Implement a simple REST-endpoint that returns a JSON-object as sketched above, given this URL: `api/securerandoms`

2 Chaining promises (fetch requests), and why GraphQL is cool ;-)

Enter this URL in your browser: https://swapi.dev/api/people/1/

- Use information from this link to find the first movie in which Luke Skywalker appeared
- Use information from this link to find the first species, which appeared in this movie
- Use information from this link to find the planet (homeworld) for this species

a) with plain promises

Now, Implement a method `getPlanetforFirstSpeciesInFirstMovieForPerson(id){}` which for id = 1 (Luke Skywalker) should log this info:

```
Name: Luke Skywalker
First film: The Empire Strikes Back
First species: Yoda's species
Homeworld for Specie: unknown
```

Hints:
1) This requires you to make a number of REST-requests (using fetch), read a value from the request, and use this value to perform a new request.
2) The lists in the responses are not sorted. For this exercise it's ok to just use the first URL in the list: Like films[0] will actually give you the second movie, see below:

```
"films": [ "https://swapi.dev/api/films/2/",
           "https://swapi.dev/api/films/6/",
           "https://swapi.dev/api/films/3/",
           "https://swapi.dev/api/films/1/",
           "https://swapi.dev/api/films/7/"],
```

***Info**: So why did I give this exercise the subtitle "and why GraphQL is cool"?*
*In order to get the information needed for the task above, you had to make **four** HTTP-requests. Let's do the same thing, using a GraphQL-endpoint.* Open a browser with this URL http://graphql.org/swapi-graphql and paste in the code below:

```
query {
   person(id: "cGVvcGxlOjE="){
    name
    filmConnection(first:1) {
      films{
        title,
        speciesConnection(first:1){
          species{
            name,id,
            homeworld{
              name
            }
          }
        }
      }
    }
  }
}
```

b) with async/await

Implement a new version of `getPlanetforFirstSpeciesInFirstMovieForPerson(id)`, that should use the much cleaner syntax of `async-await`:

```
getPlanetforFirstSpeciesInFirstMovieForPersonAsync(id)
```

Make sure to implement proper error handling when you test the method.

3 Async functions in serial and in parallel

Execution in serial

Use `fetch` and `async/await` to complete `fetchPerson(..)` below. When implemented, each line in `printNames()` must be executed "sequentially". Verify this with the debugger.

```
const URL = "https://swapi.dev/api/people/";

function fetchPerson(url){
  //Complete this function
}
async function printNames() {
  console.log("Before");
  const person1 = await fetchPerson(URL+'1');
  const person2 = await fetchPerson(URL+'2');
  console.log(person1.name);
  console.log(person2.name)
  console.log("After all");
}
```

With the previous design, HTTP requests were made sequentially and not in parallel. Since the second request does not require inputs from the first, this introduces an unnecessary performance overhead in the function (it blocks longer than it has to).

You will fix this problem in the next step. But first, let's measure *time consumption*, so we can see if it is a problem.

Use npm to install `performance-now`, and calculate the time spent in your sequential implementation using this example.

Execution in parallel

Fix the problem above, so that HTTP-requests are made in parallel.

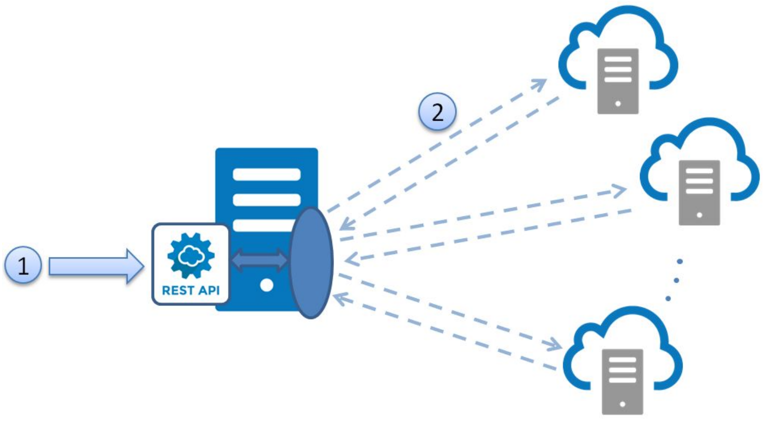
Measure the time spent the same way as above, to convince yourself that; *knowing how and when to perform request in serial or parallel* is important.

Extra, if you can't get enough

4 Handling many asynchronous web-requests

This exercise is a bit more challenging. You must implement a simple proof-of-concept solution for a site "similar" to Momondo.com, Hotels.com etc.

Your task is to implement a backend, that for an ingoing REST-request (1 in the figure below), spawns a number of outgoing requests (2, below), assembles and manipulates the result from all outgoing request into a response, returned back to (1).



The data we will use for this exercise will not make any business sense but should serve well to visualize the general idea in this kind of designs

You should use the five links below to simulate the requests to a number of "different" servers:

https://jsonplaceholder.typicode.com/photos?albumId=1
https://jsonplaceholder.typicode.com/photos?albumId=3
https://jsonplaceholder.typicode.com/photos?albumId=5
https://jsonplaceholder.typicode.com/photos?albumId=7
https://jsonplaceholder.typicode.com/photos?albumId=9

You could remove the query parameter from the request, and obtain all data from this dummy service with a single request, but that would not demonstrate what you are supposed to learn from this exercise. So use the links above, and perform five separate requests.

a) Test one of the URL's and inspect the JSON you receive (observe how the albumId is repeated for all albums, even when we requested items only with a given id).

- Now use es2005 Promises to fetch and combine data from all five URL's
- Make a response which includes the results for all the five requests, but manipulated as follows:
  - Use filter to include only out only those albums where the title contains exactly three words
  - Use map to include only the id and title of each album

This is how your final data should be built: http://promises.cooljavascript.dk/api/albumthreewords

b) Refactor your code (if necessary) into a node module, and export it

c) Implement a REST-endpoint that returns a JSON-object (as in the example above), given this URL:
`api/albumthreewords`

d) If you have time, change your module and add a new REST-endpoint which given this URL+parameter:
`api/albums/:words`

Should return only albums with the number of words in the title, given via the argument

5 If we were to implement our own "fetch-like" method

The example below illustrates how you often have use `fetch` (here for a simple GET):

```
fetch("https://swapi.dev/api/people/2/")
 .then(res=>{
   console.log(res.status);
   return res.json();
 })
 .then(person => console.log(person));
```

`fetch` returns a promise, which includes a number of properties/methods related to the request, see here for all details. In the example above, we log the status-property and call the `json()` method which "takes the response stream and reads it to completion, and then returns a promise that resolves with the result of parsing the body text as JSON".
Since, as we have seen, promises are chainable we can attach a new *then* to the response. In the example above we just log a single property from the object returned by the server.

3a)
Implement a function stringManipulator, which when called as sketched below will provide the output given below:

```
stringManipulator("JavaScript is cool, even when it sucks", 1000)
 .then(data => {
   console.log("From first promise: " + data.upperCased);
   console.log("From first promise: " + data.msgLength);
   return data.asJson()  //Opposite to when we do res.json() with fetch, since this
                         //should return a JSON-string
 })
 .then(res => {
   console.log("From second promise: " + res)
 });
```

Output from the function call above:

```
From first promise: JAVASCRIPT IS COOL, EVEN WHEN IT SUCKS
From first promise: 38
From second promise: {"words":["JavaScript","is","cool,","even","when","it","sucks"]}
```