Exercise on ES6 Promises

In the following exercises, we will practice promises by handling success and failure in data retrieval.

Exercise 1:

List the names of the users retrieved via this endpoint using ajax.

Make sure you don't use jQuery, and you use promises to handle success and failure. Place the names in the `div` below, having a class `.js-names`, and separate them with a simple comma. Make sure you do not use jQuery to access or modify the content of the below div.

```
<div style="border:1px black solid;"
     class="js-names"></div>
```

Solution:

If you want to test the solution, you can simply prepend the test div to the beginning of the body:

```
document.body.innerHTML = `
<div style="border:1px black solid;"
class="js-names"></div>` +
document.body.innerHTML;
```

Notice the template literal that enables us to add new lines without closing the literal.

I assume that at least half of my readers came up with a solution that uses jQuery.

jQuery lowered the entry barriers to frontend development, and we can be grateful for it. However, jQuery has done its job, and we don't necessarily need it now that ES6 is expressive enough.

This solution will work without jQuery.

Feel free to study the documentation if you would like to learn more about the XMLHttpRequest or ready states:

- XMLHttpRequest.send14
- XMLHttpRequest.onreadystatechange15
- XMLHttpRequest.readyState16

Let's get the response first, and log it to the screen with an empty parse function.

```
let parse = function( response ) {
  console.log( response );
}
let errorHandler = function() {
  console.log( 'error' );
}
new Promise( function( resolve, reject ) {
  let request = new XMLHttpRequest();
  request.onreadystatechange = function() {
  if ( this.status === 200 && this.readyState === 4 ) {
    resolve( this.response );
  }
}
request.open('GET',
'http://jsonplaceholder.typicode.com/users'
);
request.send();
} ).then( parse ).catch( errorHandler );
```

Getting the response works as follows:

- we create a new promise
- inside the promise, we create an `XMLHttpRequest`
- we can parse the response once the `XMLHttpRequest` transitions to readyState `4`, and the `status` becomes `200`
- once the state change handler is registered, we open and send the request
- once all conditions are fulfilled, we resolve the promise

Note that there is no error handling in the code. In real life systems, you should create a `catch` after the `then`, and handle errors.

should create a `catch` after the `then`, and handle errors.

Also note that in real life code, it makes sense to abstract your API handlers. Let's continue with the proper `parse` method:

```
let parse = function( response ) {
  let element = document.querySelector( '.js-names' );
  element.innerHTML =
  JSON.parse( response )
  .map( element => element.name )
  .join(',');
}
```

Our last task is to handle errors in the promise. We will test this scenario with a wrong API URL:

```
new Promise( function( resolve, reject ) {
  let request = new XMLHttpRequest();
  request.onreadystatechange = function() {
    if ( this.status === 200 && this.readyState === 4 ) {
      resolve( this.response );
  }
}
request.onerror = function() {
  reject( new Error( this.statusText ) );
}
request.open(
'GET',
'http://erroneousurl.com/users'
);
request.send();
} ).then( parse ).catch( errorHandler );
```

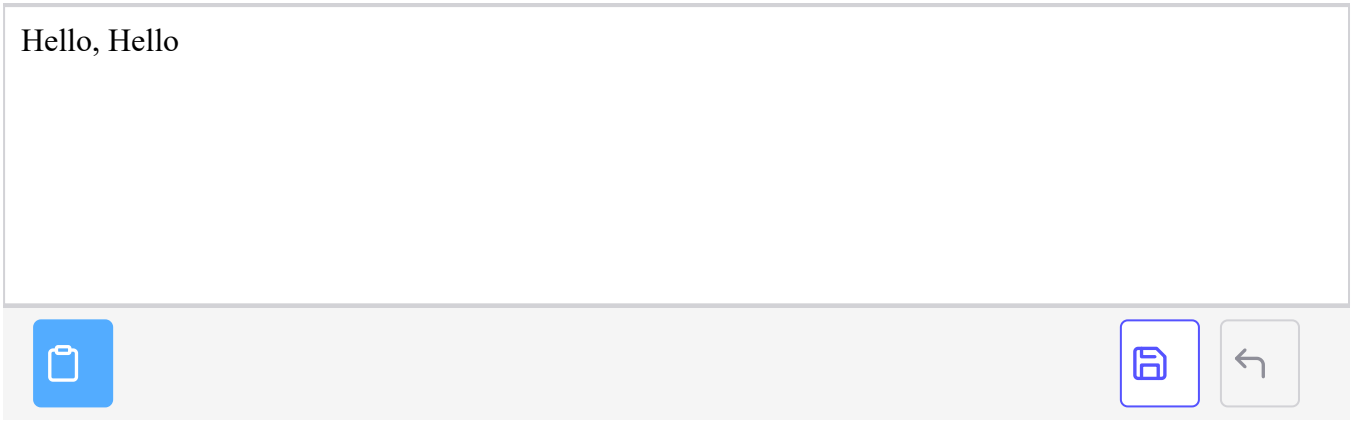
Defining an `onerror` handler does the task. Reject the promise to transition to the `catch` clause.

Output

JavaScript

HTML

CSS (SCSS)



Exercise 2:

Extend the exercise such that you disable the below text field and button while fetching takes place. Regardless of whether fetching is done or fetching fails, enable the text field and the button. The input field and the text field can be referenced using the `js-textfield` and `js-button` classes respectively.

```
<input type="text" class="js-textfield" />
<button class="js-button">
    Enable the Textfield
</button>
```

Solution:

Let's add the elements to the DOM first:

```
document.body.innerHTML = `
<input type="text" class="js-textfield" />
<button class="js-button">
  Enable the Textfield
</button>` +
document.body.innerHTML;
```

We will now define two functions: one to enable, and one to disable the fields.

```
let enableFields = function() {
  document.querySelector( '.js-textfield' )
    .removeAttribute( 'disabled' );
  document.querySelector( '.js-button' )
    .removeAttribute( 'disabled' );
}
let disableFields = function() {
```

```
  document.querySelector( '.js-textfield' )
    .setAttribute( 'disabled', true );

  document.querySelector( '.js-button' )
    .setAttribute( 'disabled', true );
}
```

Our first plan is to call the `disableFields` function in the then clause, and call the `enableFields` function at the end of the `then` callback and the `catch` callback.

```
let parse = function( response ) {
  let element = document.querySelector( '.js-names' );

  element.innerHTML =
    JSON.parse( response )
      .map( element => element.name )
        .join(',');
  enableFields();
}
let errorHandler = function() {
  console.log( 'error' );
  enableFields();
}
new Promise( function( resolve, reject ) {
  disableFields();
  let request = new XMLHttpRequest();
  request.onreadystatechange = function() {
    if ( this.status === 200 && this.readyState === 4 ) {
      resolve( this.response );
    }
  }
  request.onerror = function() {
    reject( new Error( this.statusText ) );
  }
  request.open(
    'GET',
      'http://erroneousurl.com/users'
  );
  request.send();
} ).then( parse )
  .catch( errorHandler );
```

Technically, it is possible to add a second `then` callback after the `catch` to call `enableFields` instead of repeating it in the `parse` and `errorHandler` methods.

If you chose to implement your code this way, make sure you catch all errors inside your cleanup method.