**Report about**

“What you learned about (Promises, async, and await)”

React Training – Task #7

The report includes What you learned about (Promises, async, and await) Functions.

* **Introduction**
* **Promise**

The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

* **Async**

An async function is a function declared with the async keyword, and the await keyword is permitted within it. The async and await keywords enable asynchronous, promise-based behavior to be written in a cleaner style, avoiding the need to explicitly configure promise chains.

* **Await**

The await operator is used to waiting for a Promise. It can only be used inside an async function within regular JavaScript code; however, it can be used on its own with JavaScript modules.

* **Subject** (Content)

1. **Promise**

* **Description**

A Promise is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method 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.

A pending promise can either be fulfilled with a value or rejected with a reason (error). When either of these options happens, the associated handlers queued up by a promise's then method are called. If the promise has already been fulfilled or rejected when a corresponding handler is attached, the handler will be called, so there is no race condition between an asynchronous operation completing and its handlers being attached.

* **Chained Promises**

The methods Promise.prototype.then(), Promise.prototype.catch(), and Promise.prototype.finally() are used to associate further action with a promise that becomes settled.

The .then() method takes up to two arguments; the first argument is a callback function for the resolved case of the promise, and the second argument is a callback function for the rejected case. Each .then() returns a newly generated promise object, which can optionally be used for chaining.

* **Example**

let myFirstPromise = new Promise((resolve, reject) => {

setTimeout( function() {

resolve("Success!")

}, 250)

})

myFirstPromise.then((successMessage) => {

console.log("Yay! " + successMessage)

});

1. **async**

* **Description**

Async functions can contain zero or more await expressions. Await expressions make promise-returning functions behave as though they're synchronous by suspending execution until the returned promise is fulfilled or rejected. The resolved value of the promise is treated as the return value of the await expression. The use of async and await enables the use of ordinary try / catch blocks around asynchronous code.

* **await and parallelism**

In the sequential start, execution suspends 2 seconds for the first await, and then another second for the second await. The second timer is not created until the first has already fired, so the code finishes after 3 seconds.

In concurrentStart, both timers are created and then awaited. The timers run concurrently, which means the code finishes in 2 rather than 3 seconds, i.e. the slowest timer. However, they await calls still run in series, which means the second await will wait for the first one to finish. In this case, the result of the fastest timer is processed after the slowest.

If you wish to safely perform two or more jobs in parallel, you must await a call to Promise. all or Promise.allSettled.

* **Example**

1. function resolveAfter2Seconds() {
2. console.log("starting slow promise")
3. return new Promise(resolve => {
4. setTimeout(function() {
5. resolve("slow")
6. console.log("slow promise is done")
7. }, 2000)
8. })
9. }
10. function resolveAfter1Second() {
11. console.log("starting fast promise")
12. return new Promise(resolve => {
13. setTimeout(function() {
14. resolve("fast")
15. console.log("fast promise is done")
16. }, 1000)
17. })
18. }
19. async function sequentialStart() {
20. console.log('==SEQUENTIAL START==')
21. const slow = await resolveAfter2Seconds()
22. console.log(slow) // 2. this runs 2 seconds after 1.
23. const fast = await resolveAfter1Second()
24. console.log(fast) // 3. this runs 3 seconds after 1.
25. }
26. async function concurrentStart() {
27. console.log('==CONCURRENT START with await==');
28. const slow = resolveAfter2Seconds() // starts timer immediately
29. const fast = resolveAfter1Second() // starts timer immediately
30. // 1. Execution gets here almost instantly
31. console.log(await slow) // 2. this runs 2 seconds after 1.
32. console.log(await fast) // 3. this runs 2 seconds after 1., immediately after 2., since fast is already resolved
33. }
34. function concurrentPromise() {
35. console.log('==CONCURRENT START with Promise.all==')
36. return Promise.all([resolveAfter2Seconds(), resolveAfter1Second()]).then((messages) => {
37. console.log(messages[0]) // slow
38. console.log(messages[1]) // fast
39. })
40. }
41. async function parallel() {
42. console.log('==PARALLEL with await Promise.all==')
43. // Start 2 "jobs" in parallel and wait for both of them to complete
44. await Promise.all([
45. (async()=>console.log(await resolveAfter2Seconds()))(),
46. (async()=>console.log(await resolveAfter1Second()))()
47. ])
48. }
49. sequentialStart() // after 2 seconds, logs "slow", then after 1 more second, "fast"
50. // wait above to finish
51. setTimeout(concurrentStart, 4000) // after 2 seconds, logs "slow" and then "fast"
52. // wait again
53. setTimeout(concurrentPromise, 7000) // same as concurrentStart
54. // wait again
55. setTimeout(parallel, 10000)
56. **Await**

* **Description**

The await expression causes async function execution to pause until a Promise is settled (that is, fulfilled or rejected), and to resume execution of the async function after fulfillment. When resumed, the value of the await expression is that of the fulfilled Promise.

If the Promise is rejected, the await expression throws the rejected value.

If the value of the expression following the await operator is not a Promise, it's converted to a resolved Promise.

An await splits execution flow, allowing the caller of the async function to resume execution. After the await defers the continuation of the async function, execution of subsequent statements ensues. If this await is the last expression executed by its function, execution continues by returning to the function's caller a pending Promise for completion of the awaits function and resuming execution of that caller.

* **Top-level await**

You can use the await keyword on its own (outside of an async function) within a JavaScript module. This means modules, with child modules that use await, wait for the child module to execute before they themselves run. All while not blocking other child modules from loading.

* **Example**

1. function resolveAfter2Seconds(x) {
2. return new Promise(resolve => {
3. setTimeout(() => {
4. resolve(x);
5. }, 2000);
6. });
7. }
8. async function f1() {
9. var x = await resolveAfter2Seconds(10);
10. console.log(x); // 10
11. }
12. f1();

* **Conclusions**

I learned a lot about the topic, how to write code and what the function and importance of each function are.