Async-Await

asynchronous functions and controlling the execution of functions using the await operator

The ability to write asynchronous functions is a major update in ES2017.

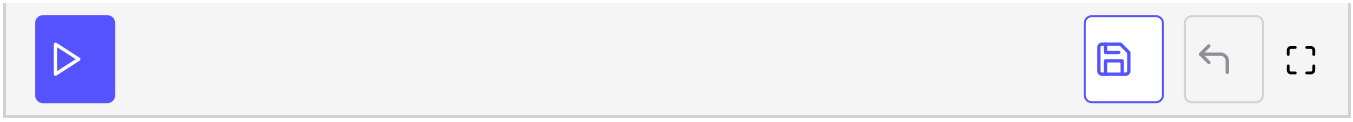
To understand this chapter, I suggest that you review the chapter on promises.

What are the asynchronous functions?

Asynchronous functions are functions that return a promise. We denote them by using the `async` keyword.

```
const loadData = async function( value ) {
    if ( value > 0 ) {
        return { data: value };
    } else {
        throw new Error( 'Value must be greater than 0' );
    }
}

loadData( 1 ).then( response => console.log( response ) );
loadData( 0 ).catch( error => console.log( error ) );
```



When `loadData` returns an object, the return value is wrapped into a promise. As this promise is resolved, the `then` callback is executed, console logging the response.

When `loadData` is called with the argument `0`, an error is thrown. This error is wrapped into a rejected promise, which is handled by the `catch` callback.

In general, return values of an `async` function are wrapped into a resolved promise, except if the return value is a promise itself. In the latter case, the promise is returned. Errors thrown in an `async` function are caught and wrapped into a rejected promise.

The `await` operator

The `await` operator

Await is a prefix operator standing in front of a promise.
As long as the promise behind the `await` operator is pending, `await` blocks execution. As soon as the promise is resolved, `await` returns the fulfillment value of the promise. As soon as the promise is rejected, `await` throws the value of rejection.

Let's see an example:

```
const delayedPromise = async () => {
    let p = new Promise( ( resolve, reject ) => {
        setTimeout( () => resolve( 'done' ), 1000 );
    } );
    const promiseValue = await p;
    console.log( 'Promise value: ', promiseValue );
}

delayedPromise();

// ... after 1 second
//> Promise value:  done
```

The `await` operator can only be used in asynchronous functions. If the `async` keyword is removed from the previous example, an error is thrown:

```
const delayedPromise2 = () => {
    let p = new Promise( ( resolve, reject ) => {
        setTimeout( () => resolve( 'done' ), 1000 );
    } );
    const promiseValue = await p;
    console.log( 'Promise value: ', promiseValue );
}
```

Combining async and await

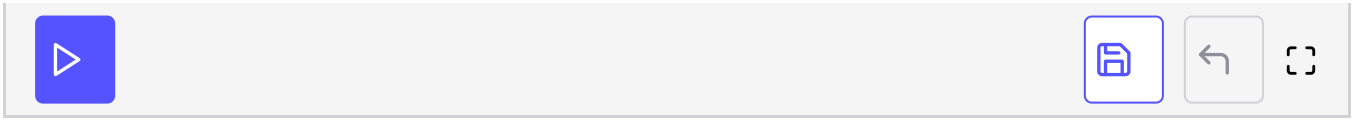
We already know that asynchronous functions return a promise.

We also know that the `await` keyword

- expects a promise as its operand,
- should be placed in asynchronous functions

As a result, we can wait for the response of asynchronous functions inside asynchronous functions.

```
const loadData = async () => {
    disableSave();
    const resultSet1 = await asyncQuery1();
    displayResultSet1( resultSet1 );
    const resultSet2 = await asyncQuery2();
    displayResultSet2( resultSet2 );
    enableSave();
}
```



This hypothetical `loadData` function loads two tables by accessing a server via an API.

First query 1 is executed. The execution of `loadData` is blocked until the promise returned by `asyncQuery1` is resolved.

Once `resultSet1` is available, the `displayResultSet1` function is executed.

Afterward, `asyncQuery2` is executed. Notice that this function is only called after the return value of `asyncQuery1` is resolved. In other words, `asyncQuery1` and `asyncQuery2` are *executed synchronously*.

Once `resultSet2` becomes available, the results are displayed.

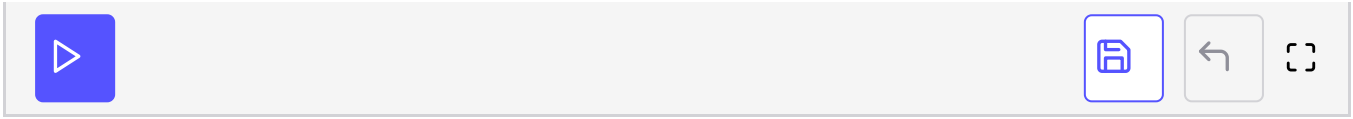
There is only one problem with this example. Imagine a web application accessing ten API endpoints. Assume that each server call takes one second in average. If our page can only be rendered after all ten asynchronous calls are executed, we will have to wait ten seconds until the user can browse our page. This is unacceptable.

This is why it makes sense to execute asynchronous queries in parallel. We can use `Promise.all` to create a promise that combines and executes its arguments in parallel.

```
const loadData = async () => {
    disableSave();
    const [resultSet1, resultSet2] = await Promise.all([
```

```
        asyncQuery1(),
        asyncQuery2()
    ] );

    displayResultSet1( resultSet1 );
    displayResultSet2( resultSet2 );
    enableSave();
}
```



In this example, all queries are executed asynchronously. If the array inside `Promise.all` contained ten queries, and each query took one second to execute, the execution time of the whole `Promise.all` expression would still be one second.

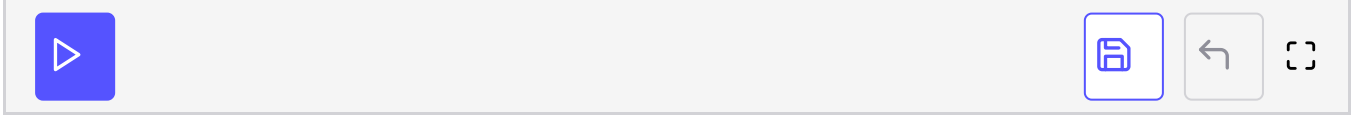
The two solutions are not equivalent though. Suppose that the average time taken to retrieve `resultSet1` is 0.1 seconds, while `resultSet2` can only be retrieved in one second.

In this case,

- the asynchronous version saves 0.1 seconds compared to the synchronous one,
- However, `displayResultSet1` is only executed after all queries are retrieved in the asynchronous version. This means that we can expect a 0.9 seconds delay compared to the synchronous version.

We can combine the advantages of the two versions by making use of the chainability of the `then` callback of promises.

```
const loadData = async () => {
    disableSave();
    const [resultSet1, resultSet2] = await Promise.all([
        asyncQuery1().then( displayResultSet1 ),
        asyncQuery2().then( displayResultSet2 )
    ] );
    enableSave();
}
```



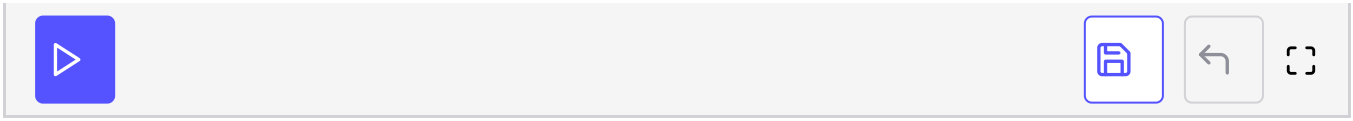
In this version of the code, the queries are retrieved asynchronously, and the corresponding `displayResultSet` handler function is executed as soon as the

corresponding promise is resolved. This means that the first query is rendered in 0.1 seconds, while the second query is rendered in one second.

Parallel execution without await

Let's remove the `disableSave` and `enableSave` functions from the previous example:

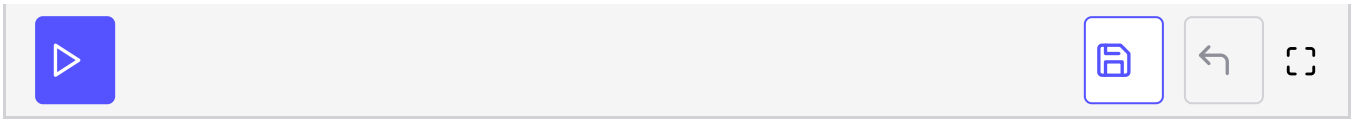
```
const loadData = async () => {
    const [resultSet1, resultSet2] = await Promise.all([
        asyncQuery1().then( displayResultSet1 ),
        asyncQuery2().then( displayResultSet2 )
    ] );
}
```



The function is still working as expected. However, the implementation is made complex for no reason.

We could simply execute the two asynchronous queries and their corresponding handlers one after the other without wrapping them in `Promise.all`:

```
const loadData = () => {
    asyncQuery1().then( displayResultSet1 );
    asyncQuery2().then( displayResultSet2 );
}
```



By not using `await`, we are not blocking execution of `asyncQuery2` before the promise of `asyncQuery1` is resolved. Therefore, the two queries are still executed in parallel.

Notice that this implementation of `loadData` is not even declared as `async`, as we don't need to return a promise in a vacuum, and we are not using the `await` keyword inside the function anymore.

Awaiting a rejected promise

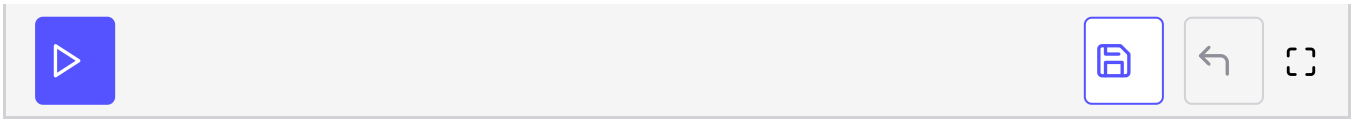
There are cases when the operand of `await` becomes a rejected promise. For instance,

- when reading a file that does not exist,
- encountering an I/O error, and
- encountering a session timeout in case of an API call,

our promise becomes rejected.

When promise `p` is rejected, `await p` throws an error. As a result, we have to handle all sources of errors by placing error-prone `await` expressions in `try-catch` blocks.

```
try {
    await p;
} catch( e ) {
    /* handle error */
}
```



Position of the async keyword

First, we can create named function expressions of asynchronous regular functions or arrow functions.

```
const name1 = async function() { ... }
const name2 = ( ...args ) => returnValue;
```

When creating function expressions, `async` is written in front of the function keyword.

```
async function name3() { ... }
```

Summary

Asynchronous functions are functions that return a promise. These functions can handle I/O operations, API calls, and other forms of delayed execution.

Awaiting for the resolution of a promise returns the resolved value of the promise or throws an error upon rejection. The `await` operator makes it possible to execute asynchronous functions sequentially or in parallel.

Async-await gives you an elegant way of handling asynchronous functions, and therefore, it is one of the most useful updates of ES2017.