# Async/Await in Lightning Web Components

When building Lightning  Components in Salesforce, you should certainly need to address asynchronous operations. Incase if you're calling Apex methods, working with promises, or managing UI state after data loads, asynchronous programming is a most fundamental part. Whereas promises have been the most used way, the async/await syntax provides a more readable and maintainable way while working with asynchronous code.

## Challenges with traditional promise handling

Even before async/await, developers primarily used promise chains using .then() and .catch() methods. This method while usage leads to nested loops in the code which will be difficult to understand. For example take a simple scenario where you need to retrieve account and related contact records.

```
import { LightningElement, track } from 'lwc';
import getAccount from '@salesforce/apex/AccountController.getAccount';
import getContacts from '@salesforce/apex/ContactController.getContacts';

export default class AccountViewer extends LightningElement {
    @track account;
    @track contacts;
    @track error;

    connectedCallback() {
        getAccount({ accountId: '001XXXXXXXXXXXXXXX' })
            .then(accountResult => {
                this.account = accountResult;
                return getContacts({ accountId: accountResult.Id });
            })
            .then(contactsResult => {
                this.contacts = contactsResult;
            })
            .catch(error => {
                this.error = error;
            });
    }
}
```

This works, but as you add more sequential operations, the promise chain grows longer and becomes harder to read. Error handling is separated from the logic, and the code structure doesn't clearly reflect the intended execution flow.

## Enter Async/Await

Async/await is simple syntax built on top of promises that makes asynchronous code look and behave more like synchronous code. The async keyword declares that a function will handle asynchronous operations, while await pauses the execution of the async function until a promise settles.

Lets see the same example as above but using async/await  where we retrieve account and contact records

```
import { LightningElement, track } from 'lwc';
import getAccount from '@salesforce/apex/AccountController.getAccount';
import getContacts from '@salesforce/apex/ContactController.getContacts';

export default class AccountViewer extends LightningElement {
    @track account;
    @track contacts;
    @track error;

    async connectedCallback() {
        try {
            const accountData = await getAccount({ accountId: '001XXXXXXXXXXXXXXX' });
            this.account = accountData;

            const contactsData = await getContacts({ accountId: accountData.Id });
            this.contacts = contactsData;
        } catch (error) {
            this.error = error;
        }
    }
}
```

As you con see how much cleaner this looks. The code by default reads from top to bottom, making it more clear what happens first, what is going to happen next, and how errors were handled. We get the same logic preserved as of traditional promise way and error handling is done at one place.

## How Async/Await Works in Practice

When you mark a function as async, it automatically returns a promise. If the function returns a value, that value becomes the resolved value of the promise. If the function throws an exception, the promise is rejected with that exception.

The await keyword can only be used inside async functions. When you await a promise, the function pauses execution until the promise settles. If the promise resolves, await returns the resolved value. If it rejects, await throws the rejection value, which you can catch with a try/catch block.

This pattern is particularly useful in LWC because most data operations are asynchronous. Whether you're calling Apex methods, using the Lightning Data Service, or working with external APIs, async/await helps you write cleaner code.

## Error Handling in Async/Await

Biggest advantage of using async/await is simplified version of error handling compared to normal promises. With normal promises, you need to use .catch() methods . but in async/await error handling can be done using try and catch with which most developers were comfortable with.

```
import { LightningElement, track } from 'lwc';
import getOpportunities from '@salesforce/apex/OpportunityController.getOpportunities';
import { ShowToastEvent } from 'lightning/platformShowToastEvent';

export default class OpportunityDashboard extends LightningElement {
    @track opportunities = [];
    @track isLoading = true;

    async connectedCallback() {
        try {
            this.opportunities = await getOpportunities({
                stageName: 'Closed Won'
            });
        } catch (error) {
            this.showToast('Error', 'Failed to load opportunities', 'error');
            console.error('Opportunity loading error:', error);
        } finally {
            this.isLoading = false;
        }
    }

    showToast(title, message, variant) {
        this.dispatchEvent(new ShowToastEvent({
            title,
            message,
            variant
        }));
    }
}
```

The finally block is especially useful for cleanup operations that should run regardless of whether the operation succeeded or failed, like hiding loading indicators.

## Running Operations in Parallel

A common misconception about async/await is that it forces sequential execution. While await does pause execution for a specific promise, you can still run multiple operations in parallel using Promise.all().

```
import { LightningElement, track } from 'lwc';
import getOpenCases from '@salesforce/apex/CaseController.getOpenCases';
import getRecentLeads from '@salesforce/apex/LeadController.getRecentLeads';
import getTaskCount from '@salesforce/apex/TaskController.getTaskCount';

export default class AgentDashboard extends LightningElement {
    @track cases = [];
    @track leads = [];
    @track taskCount = 0;
    @track isLoading = true;

    async connectedCallback() {
        try {
            // Start all requests simultaneously
            const casesPromise = getOpenCases();
            const leadsPromise = getRecentLeads();
            const tasksPromise = getTaskCount();

            // Wait for all promises to resolve
            const [casesData, leadsData, tasksData] = await Promise.all([
                casesPromise,
                leadsPromise,
                tasksPromise
            ]);

            this.cases = casesData;
            this.leads = leadsData;
            this.taskCount = tasksData;
        } catch (error) {
            console.error('Dashboard data error:', error);
        } finally {
            this.isLoading = false;
        }
    }
}
```

This pattern is crucial for performance. Instead of waiting for each operation to complete before starting the next one, all requests go out simultaneously. The await Promise.all() then waits for all of them to complete before continuing.

## Functional consideration for LWC Development

When using async/await in Lightning Web Components, there are several important considerations:

1. Lifecycle Methods: Using async you can make calls like connected callback,rendered call back or handler calls like handlesave. While the call are being in progress still the lightning components will still be rendered.
2. Wire Methods: The @wire decorator has its own way while dealing with asynchronous data. You cant use wire method with async/ await but you can use it by mixing it with imperative method. So async/await works well with human interactions using imperative method instead of automated calls using wire. Lets see a simple save operation using async/await.

```
import { LightningElement } from 'lwc';
import saveRecord from '@salesforce/apex/RecordController.saveRecord';
import { ShowToastEvent } from 'lightning/platformShowToastEvent';

export default class RecordEditor extends LightningElement {
    isSaving = false;
    async handleSave() {
        this.isSaving = true;

        try {
            const recordData = this.collectFormData();
            await saveRecord({ record: recordData });

            this.dispatchEvent(new ShowToastEvent({
                title: 'Success',
                message: 'Record saved successfully',
                variant: 'success'
            }));
        } catch (error) {
            this.dispatchEvent(new ShowToastEvent({
                title: 'Error',
                message: error.body?.message || 'Save failed',
                variant: 'error'
            }));
        } finally {
            this.isSaving = false;
        }
    }

    collectFormData() {
        // Implementation to gather form data
    }
}
```

4. Loading States: Always provide visual feedback during asynchronous operations. Use boolean flags to show/hide loading indicators, disable buttons, or display skeleton screens.

## Common Pitfalls to Avoid

While async/await simplifies asynchronous code, there are some common mistakes to watch out for:

- Unnecessary Sequential Execution: Don't write sequential awaits for independent operations. Use Promise.all() instead.
- You can use await only inside an await function without async function using await will throw an syntax error.
- Make await call inside try/catch block unless you want to send the errors to higher levels for communication through bubbling.
- While await pauses the execution it wont stop responsive nature of browser , browser still continues to run other codes parallely.

## Conclusion

Aync/await has made significant and structural change to javascript code in lightning components making it more readable ,understandable and provides better error handling. And developers feel it more familiar as it was similar to synchronous patterns , compared to normal promises.

You can start using async/await with simple apex calls and later using it for complex jobs, not just syntax it was also important understand when to use parallel or sequential operations.

The real value of async/await becomes apparent in larger components with multiple asynchronous dependencies. Code that would be difficult to follow with promise chains becomes clear and straightforward with async/await. Your future self and any other developers who work with your code will appreciate the clarity and maintainability that async/await brings to your Lightning Web Components.

In the ever-evolving landscape of Salesforce development, async/await represents a significant step forward in writing clean, efficient, and maintainable asynchronous code.