**🔷 What Problems Did Promises Have?**

While **Promises** were a huge improvement over **callback hell**, they still had some **limitations** and **readability issues**, especially when dealing with multiple asynchronous tasks.

**⚠️ 1. Promise Chaining Becomes Hard to Read**

Even though you can chain .then() calls, deeply nested promises become hard to follow:

getUser()

.then(user => {

return getPosts(user.id);

})

.then(posts => {

return getComments(posts[0].id);

})

.then(comments => {

console.log(comments);

})

.catch(err => console.error(err));

* This chaining looks better than callbacks, but:
  + It’s still **less readable**
  + Harder to **debug** and use inside try/catch

**⚠️ 2. Error Handling with Promises is Clunky**

If you're working with multiple .then() calls, one mistake can crash the whole chain.

doSomething()

.then(() => doNextThing())

.then(() => doFinalThing())

.catch(error => console.error("Error!", error));

* If you forget a .catch() somewhere or nest incorrectly, errors may not be caught properly.

**⚠️ 3. Async Logic Still Looks Sync-Like but Isn't**

let data = fetchData(); // Doesn't wait!

console.log(data); // undefined

Even though fetchData() returns a promise, you **cannot use the result directly**, which confuses beginners and causes bugs.

**✅ Solution: async / await**

Introduced in ES2017, async/await is **syntactic sugar** over Promises to:

* Write **asynchronous code in a synchronous-looking way**
* Use **try/catch** easily for error handling
* Improve **readability**, **debuggability**, and **maintenance**

**✅ Example Rewritten with async/await**

async function loadData() {

try {

const user = await getUser();

const posts = await getPosts(user.id);

const comments = await getComments(posts[0].id);

console.log(comments);

} catch (err) {

console.error("Error:", err);

}

}

✅ Now it's:

* Easy to **read**
* Looks like normal code
* Uses **try/catch** like synchronous code
* Avoids **chaining hell**