Perfect — now I'll **rebuild async/await from the ground up**, exactly like we did for **Promises**, but **even more internally**.

This is **engine-level**, **event-loop aware**, and **interview-grade**.
I will **improve examples**, **fix misconceptions**, and **add what most tutorials never explain**.

🔬 JavaScript `async / await` — INTERNAL & EXECUTION-LEVEL MASTER CLASS

1️⃣ What `async / await` REALLY Is (Big Truth)

> ⚠️ **async/await is NOT a new async mechanism**

✅ It is **syntax sugar over Promises**
✅ It does **NOT create new threads**
✅ It does **NOT block JavaScript**

Mental Model

```
text

async/await
↓
Promise chains
↓
Microtask queue
↓
Event loop
```

Interview Trap ❗

❌ "await blocks JavaScript"
✅ **await only pauses the current async function**

2️⃣ What Happens When You Write `async`

```
js

async function foo() {
 return 10;
}
```

Engine Transformation (INTERNAL)

JavaScript internally rewrites this as:

```js
function foo() {
  return Promise.resolve(10);
}
```

Key Rules (MEMORIZE)

What you return	Actual return
value	Promise resolved with value
Promise	Same promise
throw error	Promise rejected
nothing	Promise resolved with `undefined`

Proof Example

```js
async function test() {
  throw "Error!";
}

test().catch(console.log);
```

➡️ Internally = `Promise.reject("Error!")`

3️⃣ `await` — What It ACTUALLY Does

```js
const result = await promise;
```

❌ Common Misunderstanding

> "await waits and blocks execution"

✅ Internal Reality

1. `await` **pauses the async function**
2. JavaScript engine:
   - Exits the function
   - Saves function state (execution context)

3. Registers a `.then()` internally
4. Continues executing other code
5. Resumes function later via **microtask**

Internal Rewrite (CRITICAL)

```
js

async function example() {
  const x = await p;
  console.log(x);
}
```

⬇️ Engine desugars to:

```
js

function example() {
  return p.then(x => {
    console.log(x);
  });
}
```

Interview Trap ❗

❌ `await` pauses JS thread

✅ `await` pauses **function execution only**

4️⃣ Async Function Execution Timeline

```
js

console.log("A");

async function f() {
  console.log("B");
  await Promise.resolve();
  console.log("C");
}

f();
console.log("D");
```

Output

```
css
```

```
A
B
D
C
```

Internal Explanation

1. `"A"` → sync
2. `"B"` → sync inside async
3. `await` → yields control
4. `"D"` → sync
5. Promise resolves → microtask
6. `"C"` → resumed

Key Rule

> **Code before `await` runs synchronously**

5️⃣ Async Function = State Machine

Internally, async functions are compiled into:

```
text

Function Object
└─ Generator-like State Machine
   ├─ Suspended
   ├─ Resumed
   └─ Completed
```

Each `await` creates a **pause point**.

Why This Matters

- Multiple awaits = multiple resumptions
- Debugging async functions becomes easier
- Stack traces remain readable

6️⃣ Error Handling with `try/catch` (Internals)

```
js

async function test() {
 try {
  await Promise.reject("Fail");
 } catch (e) {
```

```
    console.log(e);
  }
}
```

Internal Rewrite

```
js

Promise.reject("Fail")
  .then(...)
  .catch(e => console.log(e));
```

Important Rule

Error Source	Caught by
rejected promise	`catch`
thrown error	`catch`
outside async	❌ not caught

Interview Trap ❗

```
js

try {
  setTimeout(() => { throw "X" }, 0);
} catch {}
```

❌ Won't work

➡️ Different call stack

7️⃣ `await` Always Returns a Value (Even Errors)

```
js

const result = await promise;
```

- If fulfilled → value
- If rejected → throws

Equivalent Form

```
js
```

```
await p;
// is same as
p.then(v => v).catch(e => { throw e });
```

8️⃣ Sequential vs Parallel Await (VERY IMPORTANT)

❌ Slow (Sequential)

```
js

const a = await fetchA();
const b = await fetchB();
```

⏱️ Total time = A + B

✅ Fast (Parallel)

```
js

const [a, b] = await Promise.all([
  fetchA(),
  fetchB()
]);
```

⏱️ Total time = max(A, B)

Interview Question ❗

“How do you optimize multiple awaits?”

9️⃣ `await` Works on ANY Thenable

```
js

await 10;
```

Internally:

```
js

await Promise.resolve(10);
```

Why This Exists

- Interoperability
- Safe await chaining
- Consistent semantics

🔟 Async Functions & Call Stack

```js
async function a() {
  await b();
}

async function b() {
  await c();
}
```

Stack Behavior

- Stack unwinds at each await
- Context saved internally
- Resumes via microtask

➡️ **No stack overflow**

1️⃣1️⃣ Async Class Methods (Internal View)

```js
class User {
  async getName() {
    return "Anoop";
  }
}
```

Internally:

```js
User.prototype.getName = function () {
  return Promise.resolve("Anoop");
};
```

1️⃣2️⃣ Real-World Fetch Example (Improved)

```js
async function getUser() {
  try {
    const res = await fetch("https://api.github.com/users/1");
    if (!res.ok) throw new Error("HTTP Error");
```

```
  const data = await res.json();
  console.log(data.login);
 } catch (err) {
  console.error(err.message);
 }
}
```

Internal Flow

1. fetch → returns promise
2. await → suspend function
3. response arrives → microtask
4. function resumes
5. JSON parsed → another await
6. function completes

1️⃣3️⃣ Async vs Promise — REAL Comparison

Feature	Promise	async/await
Syntax	Chain-based	Sequential
Error handling	`.catch()`	`try/catch`
Debugging	Hard	Easy
Internals	Same	Same
Performance	Same	Same

🧠 FINAL INTERNAL SUMMARY (MEMORIZE)

```javascript
async function
  ↓
returns Promise
  ↓
await suspends function
  ↓
.then registered internally
  ↓
microtask resumes execution
```

🎯 INTERVIEW RAPID FIRE (WITH ANSWERS)

Q1: Does await block JS?

➡️ ❌ No, only pauses function

Q2: Is async faster than promises?

➡️ ❌ Same performance

Q3: Can await be used outside async?

➡️ ❌ (except top-level modules)

Q4: Does await create a new thread?

➡️ ❌ Never

Q5: Why async improves readability?

➡️ Linear control flow

⏭️ WHAT NEXT (SAME DEPTH)

Choose ONE:

1️⃣ **Event Loop + async/await timeline (visual)**
2️⃣ **Top-level await (ES2022)**
3️⃣ **Unhandled promise rejections**
4️⃣ **Fetch API internals**
5️⃣ **Async patterns & pitfalls**

Reply with the **number**.