Perfect 👌 Now we're at **5. Asynchronous Programming** – one of the **most important Node.js topics for interviews**.
Since you're from Laravel/PHP, think of this as the **async equivalent of queues/events in Laravel**, but in Node.js it's **native & everywhere**.

---

5. Asynchronous Programming in Node.js

📌 Why Async?

- Node.js is **single-threaded** → blocking operations stop everything.
- Async programming lets Node.js **handle multiple I/O operations** concurrently without blocking the event loop.
- Common async tasks: DB queries, API calls, file reads, timers.

5.1 Callbacks

- **First way** of handling async tasks.
- A callback is a function passed as an argument to another function, executed after the task completes.

Example:

```
function getData(callback) {
  setTimeout(() => {
    callback("Data received");
  }, 1000);
}

getData((msg) => {
  console.log(msg); // Data received
});
```

👉 **Problem**: Leads to **Callback Hell** (nested pyramid code).

```
getUser(id, (user) => {
  getOrders(user.id, (orders) => {
    getItems(orders, (items) => {
      console.log(items);
    });
  });
});
```

5.2 Promises

- Introduced to solve **callback hell**.
- Represents a value that will be available **now, later, or never**.

Example:

```
function getData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve("Data received"), 1000);
  });
}

getData()
  .then((msg) => console.log(msg))  // Data received
  .catch((err) => console.error(err));
```

👉 Chaining is easier than nested callbacks.

5.3 Async/Await

- Introduced in ES2017.
- Makes async code look like **synchronous code**.
- Built on top of **Promises**.

Example:

```
function getData() {
  return new Promise((resolve) => {
    setTimeout(() => resolve("Data received"), 1000);
  });
}

async function fetchData() {
  try {
    const msg = await getData();
    console.log(msg); // Data received
  } catch (err) {
    console.error(err);
  }
}

fetchData();
```

👉 Cleaner syntax, better readability, easier error handling.

5.4 Error Handling

- **Callback style**: `if (err) return ...`
- **Promise style**: `.catch(err => ...)`
- **Async/Await style**: `try { ... } catch (err) { ... }`

Example:

```
async function fetchUser() {
  try {
    let user = await db.findUser(1);
    console.log(user);
  } catch (err) {
    console.error("Error fetching user:", err.message);
  }
}
```

5.5 Parallel & Sequential Async

- **Sequential** (one after another):

```
await task1();
await task2();
```

- **Parallel** (run together):

```
await Promise.all([task1(), task2()]);
```

👉 `Promise.all` is great for **multiple independent async calls**.

5.6 Event Loop & Async

- Async programming is possible because of **Event Loop** (from architecture).
- `setTimeout`, DB queries, FS reads → handled in background (libuv) → callback queue → event loop.

📌 Interview Q&A

☑ **Q: What is callback hell?**
👉 Nested callbacks that make code unreadable and hard to maintain.

☑ **Q: Difference between Promises and async/await?**
👉 Promises use `.then().catch()`, async/await provides cleaner syntax using `await` inside `try/catch`.

☑ **Q: What is the difference between process.nextTick() and setImmediate()?**
👉 `process.nextTick()` executes **before next event loop tick**, while `setImmediate()` runs **on next loop iteration**.

☑ **Q: How do you run multiple async tasks in parallel?**
👉 Use `Promise.all([task1(), task2()])`.

☑ **Q: How does async/await work under the hood?**
👉 It's syntactic sugar on top of **Promises**.

📌 Key Takeaways

- **Callbacks** → first async handling, but cause **callback hell**.
- **Promises** → solve callback hell with `.then().catch()`.
- **Async/Await** → modern, clean syntax, best practice.
- Use **try/catch** for errors in async/await.
- For parallel async → `Promise.all()`.

👉 Next, we can go into **6. Express.js Framework** – the most popular Node.js framework (like Laravel in PHP world).

Do you want me to explain Express.js **from scratch (routing, middleware, error handling)**, or should I focus more on **advanced Express concepts (scaling, performance, middleware patterns)** since you're already experienced?