📘 Asynchronous Programming – Callback-Based API Call

https://chatgpt.com/share/69608e4c-8d6c-8004-9315-e78478d1b362

1️⃣ What is Asynchronous Programming?

Asynchronous programming allows JavaScript to:

- Start a slow task (API call, file read, timer)
- Continue executing other code
- Handle the result **later**, when ready

JavaScript is **single-threaded**, so async behavior is required to avoid blocking the main thread.

2️⃣ Real-World Use Case

Fetching user data from a server:

- Network request takes time
- Response comes later
- App must stay responsive

Hence, **callbacks** are used.

3️⃣ Code Under Discussion

```
function getUser(callback) {
  fetch("https://jsonplaceholder.typicode.com/users/1")
    .then(response ⇒ response.json())
    .then(data ⇒ callback(null, data))
    .catch(error ⇒ callback(error, null));
}

getUser((error, user) ⇒ {
  if (error) {
    console.log("Error:", error);
  } else {
    console.log("User:", user.name);
  }
});
```

4️⃣ Understanding the Code Step-by-Step

Step 1: Function Definition

```
function getUser(callback) {
```

- `getUser` accepts a **callback function**
- This callback will be called **after the async task finishes**
- Callback follows **error-first convention**

Step 2: `fetch()` – Asynchronous API Call

```
fetch("https://jsonplaceholder.typicode.com/users/1")
```

- Sends request to server
- Returns a **Promise**
- JavaScript does **not wait** here
- Request runs in background

Step 3: First `.then()` – Response Handling

```
.then(response => response.json())
```

- Executes after server responds
- Converts raw response to JSON
- Returns another Promise
- Arrow function is a **callback**

Step 4: Second `.then()` – Success Callback

```
.then(data => callback(null, data))
```

- Executes when JSON data is ready
- Calls the original callback
- `null` → no error
- `data` → successful result

Step 5: `.catch()` – Error Callback

```
.catch(error => callback(error, null));
```

- Executes if any error occurs
- Calls callback with:
  - `error` → error info
  - `null` → no data

5️⃣ Using the Callback Function

```
getUser((error, user) => {
```

- Callback receives two parameters:
  - `error`
  - `user`

Handling Result

```
if (error) {
  console.log("Error:", error);
} else {
  console.log("User:", user.name);
}
```

- Error handling first
- Success logic second
- This pattern is standard in JS & Node.js

6️⃣ Error-First Callback Pattern

```
callback(error, result);
```

Case	error	result
Success	null	data
Failure	error	null

This avoids confusion and is widely used.

7️⃣ Execution Flow (Timeline)

1. getUser() called
2. fetch() starts (async)
3. JS continues execution
4. Server responds
5. response.json() runs
6. data is ready
7. callback is executed

8️⃣ Why This is Called Callback-Based Async Code

- Functions are passed as arguments
- Executed later
- Multiple callbacks are chained
- Hard to scale & maintain

9️⃣ Drawbacks of Callback Approach

❌ Nested callbacks

❌ Difficult error handling

❌ Hard debugging

❌ Poor readability

This leads to **Callback Hell**

🔁 Callback Hell (Concept)

```
task1(() => {
  task2(() => {
    task3(() => {
      task4(() => {
        ...
      });
    });
  });
});
```

1️⃣0️⃣ Modern Alternative (Reference)

Promise-based

```
fetch(url)
  .then()
  .then()
  .catch();
```

async / await (Recommended)

```
const data = await fetch(url);
```

1️⃣1️⃣ Interview-Ready One-Liners

- **Callback:** A function passed as an argument to be executed later.
- **Async:** Non-blocking execution of long tasks.
- **Error-first callback:** First parameter is error, second is result.
- **Callback Hell:** Deeply nested callbacks making code unreadable.

1️⃣2️⃣ Key Takeaways (Revision)

✔️ `fetch()` is async

✔️ `.then()` and `.catch()` are callbacks

✔️ Callback executes after task completion

✔️ Error-first pattern is standard

✔️ Modern JS prefers `async/await`

1️⃣3️⃣ One-Line Summary

> This code demonstrates callback-based asynchronous programming where an API request is handled using callbacks for success and error handling.