Node.js Event Loop Explained

1. What is the Event Loop?

The **Event Loop** is the heart of Node.js. It allows Node.js to handle **non-blocking I/O operations** (file reading, network requests, database queries, etc.) on a **single thread** efficiently.

- Node.js is **single-threaded** but can handle thousands of requests **asynchronously**.
- The **event loop** continuously checks for pending tasks and executes them in different **phases**.

---

2. How the Event Loop Works?

The event loop runs in **phases**, executing callbacks in each phase. The major phases are:

- **Timers Phase** – Executes setTimeout() and setInterval() callbacks.
- **I/O Callbacks Phase** – Executes I/O-related callbacks (e.g., network, filesystem).
- **Idle/Prepare Phase** – Used internally by Node.js.
- **Poll Phase** – Retrieves new I/O events and executes ready I/O callbacks.
- **Check Phase** – Executes setImmediate() callbacks.
- **Close Callbacks Phase** – Executes callbacks for closed connections (e.g., socket.on('close')).

This cycle repeats indefinitely while Node.js is running!

---

3. Event Loop in Action – Example

```
console.log('Start');
setTimeout(() => {
   console.log('Timeout callback');
}, 0);
setImmediate(() => {
   console.log('Immediate callback');
});
Promise.resolve().then(() => console.log('Promise resolved'));
console.log('End');
//  Output Order:
// Start
// End
// Promise resolved
// Immediate callback
// Timeout callback
```

Why This Happens?

- **Synchronous code** runs first (console.log('Start'), console.log('End')).
- **Promises** (Promise.resolve().then(...)) are part of the **microtask queue**, which runs **before** the next event loop cycle.
- **setImmediate()** runs in the **Check Phase**, before setTimeout(0), which runs in the **Timers Phase**.

---

4. Key Differences Between setTimeout() and setImmediate()

setTimeout(() => console.log('setTimeout'), 0);
setImmediate(() => console.log('setImmediate'));

□ **Output:**

- If no I/O operation is happening: setTimeout might run **before** setImmediate.
- If inside an I/O callback, setImmediate **always** runs before setTimeout.

Example inside an I/O callback:

```
const fs = require('fs');
fs.readFile(__filename, () => {
  setTimeout(() => console.log('setTimeout inside I/O'), 0);
  setImmediate(() => console.log('setImmediate inside I/O'));
});
// □ Output:
// setImmediate inside I/O
// setTimeout inside I/O
```

✔ setImmediate executes **before** setTimeout when inside an I/O operation.

---

5. Microtasks: process.nextTick() vs. Promise.then()

- process.nextTick() runs **immediately after the current operation**, before the event loop continues.
- Promise.then() runs at the **end of the current microtask queue**, before the next event loop cycle.

Example:

```
console.log('Start');
process.nextTick(() => console.log('nextTick callback'));
Promise.resolve().then(() => console.log('Promise callback'));
console.log('End');
//  Output:
// Start
// End
// nextTick callback
// Promise callback
```

✔ process.nextTick() runs **before** Promise.then().

---

6. Summary & Best Practices

- **The event loop enables non-blocking I/O** by running tasks asynchronously.
- **Microtasks (process.nextTick() & Promise.then()) run before the next loop cycle.**
- **Use setImmediate() for post-I/O operations**, as it runs before setTimeout(0).
- **Avoid process.nextTick() overuse**, as it can starve the event loop.

Want to Learn More?

Would you like a deeper dive into:

- **How the Event Loop Handles File I/O?**
- **Performance Optimizations in Asynchronous Code?**
- **Debugging Event Loop Behavior?**

Let me know what interests you!