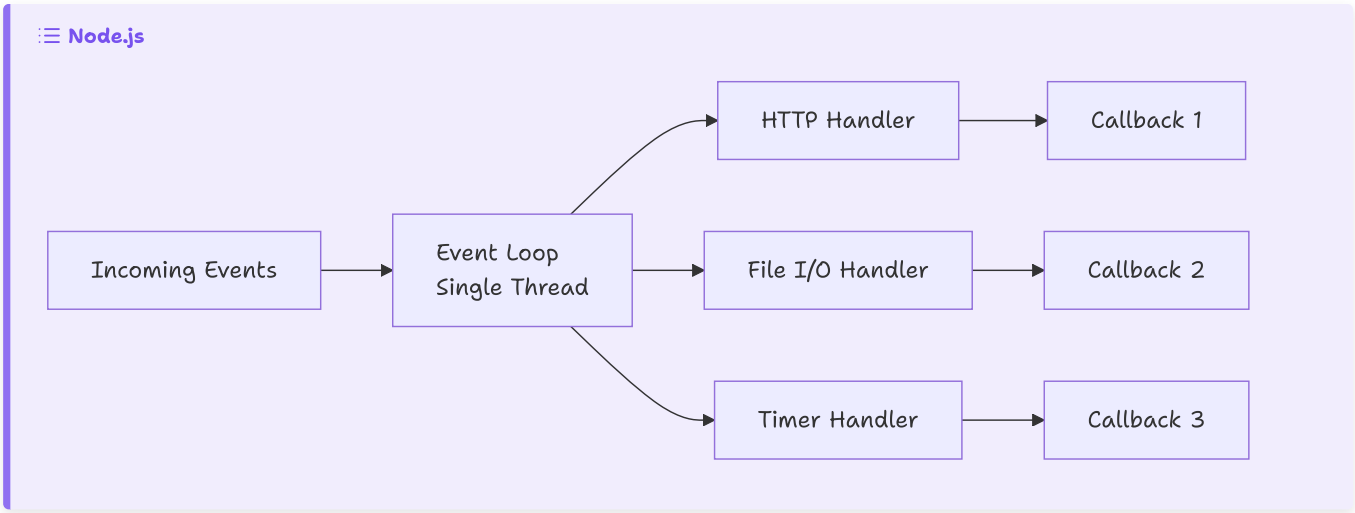
Reactor Pattern

A single-threaded event loop that monitors multiple input sources and assigns events to appropriate handlers. Similar to how interrupts are handled in a computer.

- Single Main **Dispatcher** Thread that polls or waits for changes in inputs (including file descriptors).
- When data arrives, assigns the appropriate handler.
- Handlers are non-blocking and are quick.

Complementary to the Proactor Pattern in terms of workers, as the device synchronously perform all handlers.

Benefits: Efficient for I/O-heavy applications, eliminates context switching overhead.



Node.js uses this pattern. One thread monitors all incoming HTTP requests, file I/O operations, and timers, then calls the appropriate JavaScript callbacks.

Design Considerations

Choosing

- Are you primarily I/O bound with many concurrent connections?
- Can your operations be handled with non-blocking, event-driven processing?
- Do you need to efficiently handle thousands of connections with minimal threads?
- Are your request handlers naturally stateless and quick to execute?
- Do you want to avoid the overhead of context switching between threads?

Building Outline

Event Loop Design:

- What types of events do you need to handle? (network I/O, timers, signals)
- How do you prioritize different event types?

- What's your event dispatching strategy?
- How do you handle long-running operations that could block the event loop?

Handler Design:

- Are all handlers non-blocking and fast?
- How do you handle CPU-intensive operations?
- What's your error handling strategy for handlers?
- Do handlers share any state?

Scalability:

- Do you need multiple event loops for each CPU cores?
- How do you distribute connections across event loops?
- What's your strategy for handling more connections than file descriptors?

Integration:

- How do you integrate with blocking APIs?
- What's your strategy for database operations?
- How do you handle file I/O that might block?