# Reactor Pattern: Thread-Based and Event-Driven

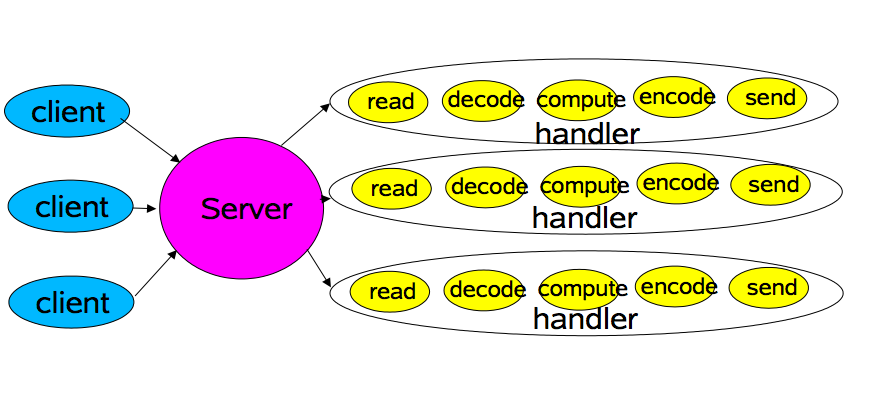
<https://dzone.com/articles/understanding-reactor-pattern-thread-based-and-eve>

To handle **web requests**, there are two competing web architectures: **thread-based** and **event-driven** architectures.

**Thread-Based Architecture**

The most intuitive way to implement a multi-threaded server is to follow the thread-per-connection approach. It is appropriate for sites that need to avoid threading for compatibility with non-thread-safe libraries.

It also uses the best multi-processing modules for isolating each request, so that a problem with a single request will not affect any other.



Processes are too heavyweight, with slower context switching and higher memory consumption. Therefore, the thread-per-connection approach is used for better scalability, though programming with threads is error-prone and hard to debug.

In order to tune the number of threads for the best overall performance and avoid thread-creating/destroying overhead, it is a common practice to put a single dispatcher thread in front of a bounded blocking queue and a thread pool. The dispatcher blocks on the socket for new connections and offers them to the bounded blocking queue. Connections exceeding the limitation of the queue will be dropped, but latencies for accepted connections become predictable. A pool of threads polls the queue for incoming requests, which will then be processed and responded to.

Unfortunately, there is always a one-to-one relationship between connections and threads. Long-living connections like Keep-Alive connections give rise to a large number of worker threads waiting in an idle state, e.g. file system access, network, etc. In addition, hundreds or even thousands of concurrent threads can waste a great deal of stack space in memory.

**Event-Driven Architecture**

Event-driven approach can separate threads from connections, which only use threads for events on specific callbacks or handlers.

An event-driven architecture consists of event creators and event consumers. The creator, which is the source of the event, only knows that the event has occurred. Consumers are entities that need to know the event has occurred. They may be involved in processing the event or they may simply be affected by the event.

**The Reactor Pattern**

The reactor pattern is one implementation technique of event-driven architecture. In simple terms, it uses a single threaded event loop blocking on resource-emitting events and dispatches them to corresponding handlers and callbacks.

There is no need to block on I/O, as long as handlers and callbacks for events are registered to take care of them. Events refer to instances like a new incoming connection, ready for read, ready for write, etc. Those handlers/callbacks may utilize a thread pool in multi-core environments.

This pattern decouples the modular application-level code from reusable reactor implementation.

There are two important participants in the architecture of Reactor Pattern:

**1. Reactor**

A Reactor runs in a separate thread, and its job is to react to IO events by dispatching the work to the appropriate handler. It’s like a telephone operator in a company who answers calls from clients and transfers the line to the appropriate contact.

**2. Handlers**

A Handler performs the actual work to be done with an I/O event, similar to the actual officer in the company the client wants to speak to.

A reactor responds to I/O events by dispatching the appropriate handler. Handlers perform non-blocking actions.

**The Intent of the Reactor Pattern**

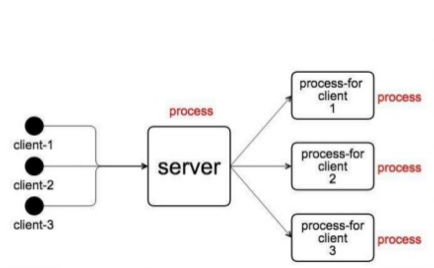
The Reactor architectural pattern allows event-driven applications to **demultiplex** and dispatch service requests that are delivered to an application from one or more clients.

One reactor will keep looking for events and will inform the corresponding event handler to handle it once the event gets triggered.

***The Reactor Pattern is a design pattern for synchronous demultiplexing and order of events as they arrive.***

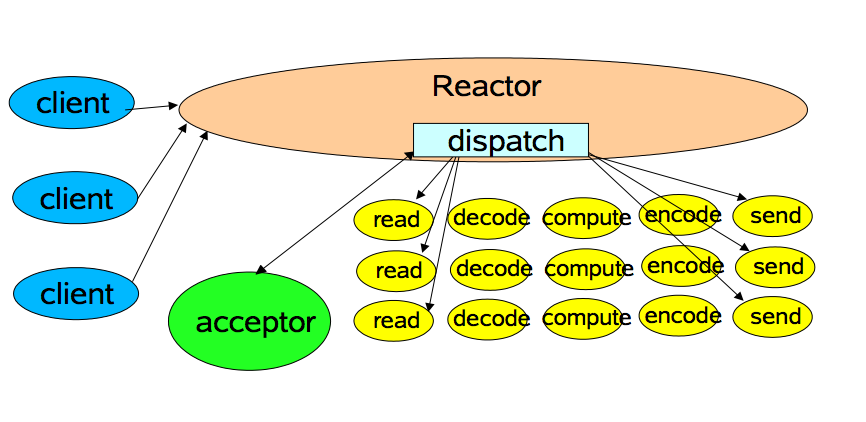
It receives messages, requests, and connections coming from multiple concurrent clients and processes these posts sequentially using event handlers. The purpose of the Reactor design pattern is to avoid the common problem of creating a thread for each message, request, and connection. Then it receives events from a set of handlers and distributes them sequentially to the corresponding event handlers.

[**Avoid this problem is to avoid the famous problem: C10K.**](http://www.kegel.com/c10k.html)



***In Summary: Servers have to handle more than 10,000 concurrent clients, and threads cannot scale the connections using Tomcat, Glassfish, JBoss, or HttpClient.***

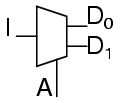
So, the application using the reactor only needs to use a thread to handle simultaneous events.



Basically, the standard Reactor allows a lead application with simultaneous events, while maintaining the simplicity of single threading.

A demultiplexer is a circuit that has an input and more than one output. It is a circuit used when you want to send a signal to one of several devices.

This description sounds similar to the description given to a decoder, but is used to select between many devices, while a demultiplexer is used to send a signal among many devices.



A Reactor allows multiple tasks which block to be processed efficiently using a single thread. The Reactor also manages a set of event handlers. When called to perform a task, it connects with the handler that is available and makes it as active.

**The Cycle of Events:**

1. Find all handlers that are active and unlocked or delegates this for a dispatcher implementation.
2. Execute each of these handlers sequentially until complete, or a point is reached where they are blocked. Completed Handlers are deactivated, allowing the event cycle to continue.
3. Repeats from Step One (1)

**Why Does it Matter?**

Because the Reactor pattern is used by Node.js, Vert.x, Reactive Extensions, Jetty, Ngnix, and others. So if you like the identify pattern and want to know how things work behind the scenes, it is important to pay attention to this pattern.