Node.js

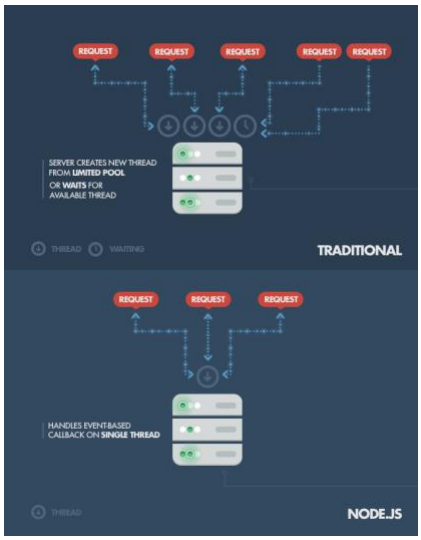
Introduction
Nose.js is a JavaScript runtime environment built on Chrome's V8 JavaScript engine. Ryan Dahl was aiming to create **real-time websites with push capability** inspired by websites like Gmail.

Node.js allows for web applications to have real-time, two-way connections where both the client and server can initiate communication allowing them to exchange data freely.

This is in stark contrast to the typical web response paradigm where the client always initiates communication. Additionally, it's based on the open web stack (HTML, CSS and JS) running over the standard port 80.

Concept
Use non-blocking, event-driven I/O to remain lightweight and efficient in the face of data-intensive real-time applications that run across distributed devices.

Node.js is not used for CPU-intensive operations, using it for heavy computation will annul nearly all of its advantages. Node.js shines in **building fast, scalable network applications** as it's capable of handling a huge number of simultaneous connections with high throughput which equates to high scalability.



Traditional web-serving techniques works where each connection (request) spawns a new thread taking up system RAM and eventually maxing out the amount of RAM available whereas NODE.js operates on a single-thread, using non-blocking I/O calls allowing it to support tens of thousands of concurrent connections

Downfall
Heavy computation could choke up Node's single thread and cause problems for all clients as incoming requests would be blocked until said computation was completed.
Secondly, developers need to be careful **not to allow an exception bubbling** up to the core (topmost) Node.js event loop, which will cause the Node.js instance to terminate, effectively crashing the program.
The technique used to avoid exceptions bubbling up to the surface is passing errors back to the caller as call-back parameters (instead of throwing them, like in other environments).
Even if some unhandled exception manages to bubble up, tools have been developed to monitor the Node.js process and perform necessary *recovery of crash instance* (current state of user session is probably not recoverable), the most common being the **Forever module**, or using a different approach with external system tools *upstart and monit* or even just upstart.

Where Node.js Should Be Used

Chat – Chat is the most typical real-time, multi-user application. Through many proprietary and open protocols running on non-standard ports, to the ability to implement everything today in Node.js with web sockets running over the standard port 80.

The chat application is a perfect example for Node.js as it's lightweight, high traffic, data-intensive (but low processing and computation) application that runs across distributed devices. It's also a great use-case for learning too, as it's simple, yet it covers most of the paradigms you'll ever use in a typical Node.js application.

Use-Case Example

In this simplest scenario, we have a single chatroom on our website where people come and can exchange messages in one-to-many fashion. For instance, we have three people on the website all connected to our message board.

On the server-side, we have a simple Express.js application which implements two things: 1) a GET '/' request handler which serves the webpage containing both a message board and a 'Send' button to initialise new message input and 2) a websocket server the listens for new messages emitted by websocket clients.

On the client-side, we have an HTML page with a couple of handlers set up, one for the 'Send' Button click event, which picks up the input message and sends it down to the websocket, and another that listens for new incoming messages on the websockets client (i.e., messages sent by other users, which the server now wants the client to display)

When one of the clients post a message, here's what happens:

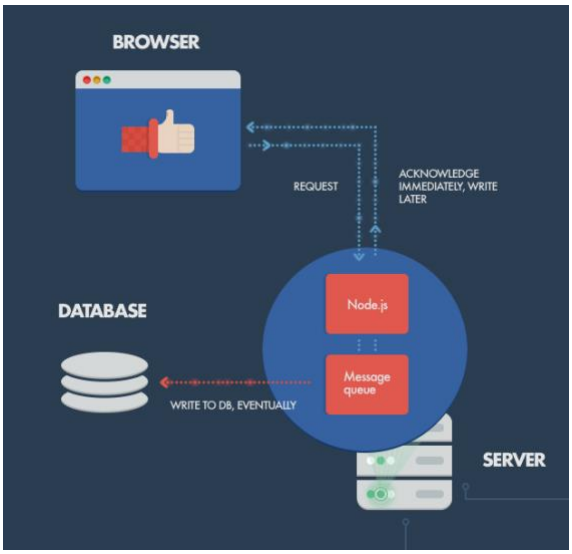
- Browser catches the 'Send' button click through a JavaScript handler, picks up the value from the input field (i.e., the message text) and emits a websocket message using the websocket client connected to our server (initialised on web page initialisation)
- Server-side component of the websocket connection receivers the message and forwards it to all other connected clients using the **broadcast method**
- All clients receive the new message as a push message via a websockets client-side component running within the web page. They then pick up the message content and update the web page in-place by appending the new message to the board.

Queued Inputs

If you're receiving a high amount of concurrent data, your database can become a bottleneck. Node.js can easily handle the concurrent connections themselves however because database access is a blocking operation (in this case), we run into trouble. The solution is to acknowledge the client's behaviour before the data is truly written into the database.

With that approach, the system maintains its responsiveness under a heavy load, which is particularly useful when the client doesn't need firm confirmation of the successful data

write. Typical examples include: the logging or writing of user-tracking data, processed in batches and not used until a later time; as well as operations that don't needed to be reflected instantly (like updating a 'Like' count on Facebook) where *eventual consistency* (often used in NoSQL) is acceptable.



Data gets queued through some kind of caching or message queuing (MQ) infrastructure (e.g. RabbitMQ, ZeroMQ) and digested by a separate database batch-write process, or computation intensive processing backend services, written in a better performing platform for such tasks. Similar behaviour can be implemented with other languages/frameworks but not on the same hardware, with the same high, maintained throughput.

In short: With Node, you can push the database writes off to the side and deal with them later, proceeding as if they succeeded.

Data Streaming

In more traditional web platforms, HTTP platforms and responses are treated like isolated event; in fact, they're actually streams. This observation can be utilised in Node.js to build some cool features. For example, it's possible to process files while they're still being uploaded, as the data comes in through a stream and we can process it in an online fashion. This could be done for real-time audio or video encoding, and proxying between different data sources.

Conclusion

When people run into problems with Node, it almost always boils down to the fact that **blocking operations are the roof of all evil** – 99% of Node misuses come as a direct consequence.

Node.js was never created to solve the compute scaling problem. It was created to solve the I/O scaling problem which it does really well.

If your use case does not contain CPU intensive operations nor access any blocking resources, you can exploit the benefits of Node.js and enjoy fast and scalable network applications.