Why The Hell Would I Use Node.js? A Case-by-Case Tutorial

Introduction

JavaScript's rising popularity has brought with it many changes, and the face of web development today is dramatically different. The things that we can do on the web nowadays with JavaScript running on the server, as well as in the browser, were hard to imagine just several years ago, or were encapsulated within sandboxed environments like Flash or Java Applets.

Before digging into Node.js, you might want to read up on the benefits of using JavaScript across the stack which unifies the language and data format (JSON), allowing you to optimally reuse developer resources. As this is more a benefit of JavaScript than Node.js specifically, we will not discuss it much here. However, it is a key advantage to incorporating Node in your stack.

As Wikipedia states: "Node.js is a packaged compilation of Google's V8 JavaScript engine, the libuv platform abstraction layer, and a core library, which is itself primarily written in JavaScript." Beyond that, it is worth noting that Ryan Dahl, the creator of Node.js, was aiming to create **real-time websites with push capability**, "inspired by applications like Gmail". In Node.js, he gave developers a tool for working in the non-blocking, event-driven I/O paradigm.

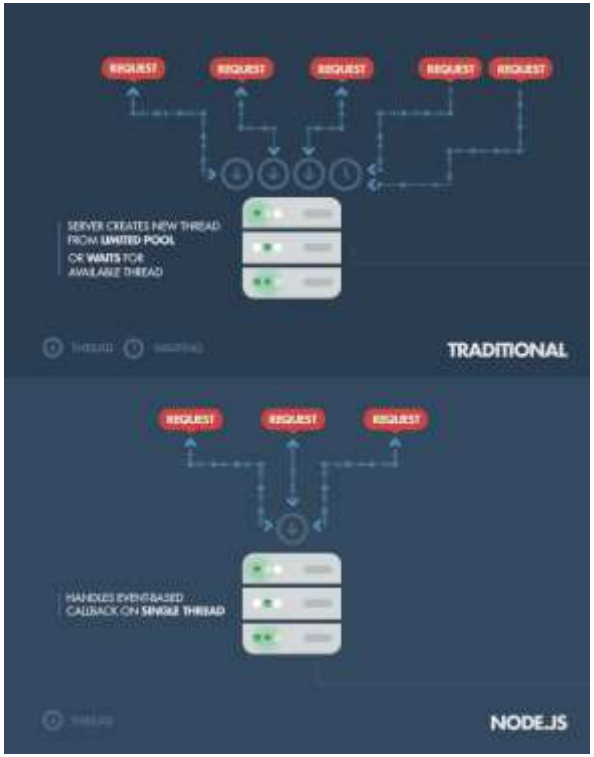
> After over 20 years of stateless-web based on the stateless request-response paradigm, we finally have web applications with real-time, two-way connections.

In one sentence: Node.js shines in real-time web applications employing push technology over web sockets. What is so revolutionary about that? Well, after over 20 years of stateless-web based on the stateless request-response paradigm, we finally have web applications with 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 is all based on the open web stack (HTML, CSS and JS) running over the standard port 80.

One might argue that we have had this for years in the form of Flash and Java Applets—but in reality, those were just sandboxed environments using the web as a transport protocol to be delivered to the client. Moreover, they were run in isolation and often operated over non-standard ports, which may have required extra permissions and such.

With all of its advantages, Node.js now plays a critical role in the technology stack of many high-profile companies who depend on its unique benefits.

In this post, I will discuss not only how these advantages are accomplished, but also why you might want to use Node.js—and *why not*—using some of the classic web application models as examples.



How Does It Work?

The main idea of Node.js: 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.

That is a mouthful.

What it *really* means is that Node.js is *not* a silver-bullet new platform that will dominate the web development world. **Instead, it fills a particular need**. In addition, understanding this is essential. You definitely do not want to use Node.js for CPU-intensive operations; in fact, using it for heavy computation will annul nearly all of its advantages. Where Node really shines is in building fast, scalable network applications, as it is capable of handling a huge number of simultaneous connections with high throughput, which equates to high scalability.

How it works, under-the-hood is interesting. Compared to traditional web-serving techniques where each connection (request) spawns a new thread, taking up system RAM and eventually maxing-out at the amount of RAM available, Node.js operates on a single-thread, using non-blocking I/O calls, allowing it to support tens of thousands of concurrent connections (held in the event loop).

A quick calculation: assuming that each thread potentially has an accompanying 2 MB of memory with it, running on a system with 8 GB of RAM puts us at a theoretical maximum of 4000 concurrent connections, plus the cost of context-switching between threads. You typically deal with that scenario in traditional web-serving techniques. By avoiding all that, Node.js achieves scalability levels of over 1M concurrent connections (as a proof-of-concept).

There is, of course, the question of sharing a single thread between all clients' requests, and it is a potential pitfall of writing Node.js applications. Firstly, heavy computation could choke up Node's single thread and cause problems for all clients (more on this later) as incoming requests would be blocked until said computation was completed. Secondly, developers need to be really 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 callback parameters (instead of throwing them, like in other environments). Even if some unhandled exception manages to bubble up, there are multiple paradigms and tools available to monitor the Node process and perform the necessary recovery of a crashed instance (although you won't be able to recover users' sessions), the most common being the Forever module, or a different approach with external system tools *upstart* and *monit*.

NPM: The Node Package Manager

When discussing Node.js, one thing that definitely should not be omitted is built-in support for package management using the NPM tool that comes by default with every Node.js installation. The idea of NPM modules is quite similar to that of *Ruby Gems*: a set of publicly available, reusable components, available through easy installation via an online repository, with version and dependency management.

A full list of packaged modules can be found on the NPM website https://npmjs.org/ , or accessed using the NPM CLI tool that automatically is installed with Node.js. The module ecosystem is open to all, and anyone can publish their own module that will be listed in the NPM repository. A brief introduction to NPM (a bit old, but still valid) can be found at http://howtonode.org/introduction-to-npm.

Some of the most popular NPM modules today are:

- Express - Express.js, a Sinatra-inspired web development framework for Node.js, and the de-facto standard for the majority of Node.js applications out there today.
- Connect - Connect is an extensible HTTP server framework for Node.js, providing a collection of high performance "plugins" known as middleware; serves as a base foundation for Express.
- socket.io and sockjs - Server-side component of the two most common websockets components out there today.
- Jade - One of the popular templating engines, inspired by HAML, a default in Express.js.
- Mongo and mongojs - MongoDB wrappers to provide the API for MongoDB object databases in Node.js.
- Redis - Redis client library.
- Coffee-script - CoffeeScript compiler that allows developers to write their Node.js programs using Coffee.
- underscore (lodash, lazy) - The most popular utility library in JavaScript, packaged to be used with Node.js, as well as its two counterparts, which promise better performance by taking a slightly different implementation approach.
- Forever - Probably the most common utility for ensuring that a given node script runs continuously. Keeps your Node.js process up in production in the face of any unexpected failures.

The list goes on. There are tons of useful packages out there, available to all (no offense to those that I have omitted here).

Examples of Where Node.js Should Be Used

CHAT

Chat is the most typical real-time, multi-user application. From IRC (back in the day), through many proprietary and open protocols running on non-standard ports, to the ability to implement everything today in Node.js with websockets running over the standard port 80.

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

Let us try to depict how it works.

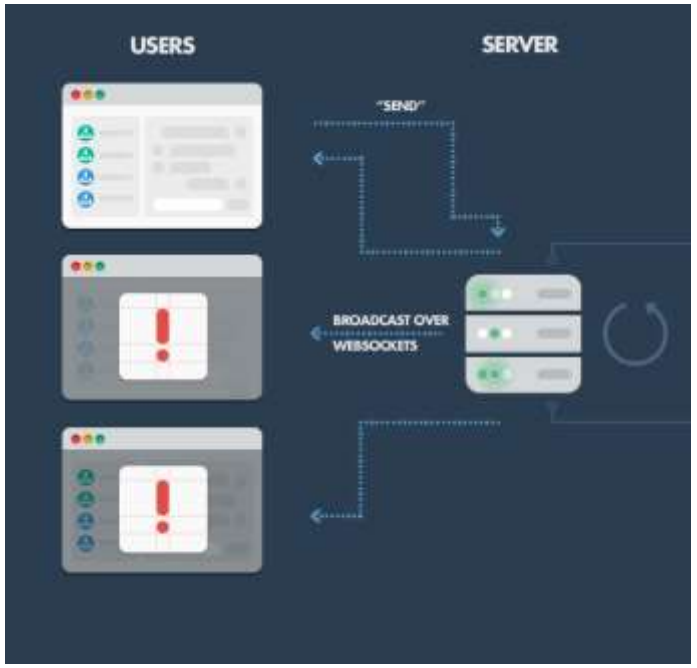
In the simplest example, we have a single chatroom on our website where people come and can exchange messages in one-to-many (actually all) fashion. For instance, say we have three people on the website all connected to our message board.

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 initialize new message input, and 2) a websockets server that 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 the websocket, and another that listens for new incoming messages on the web sockets client (i.e., messages sent by other users, which the server now wants the client to display).

When one of the clients posts a message, here is what happens:

1. 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 (initialized on web page initialization).
2. Server-side component of the websocket connection receives the message and forwards it to all other connected clients using the broadcast method.
3. 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.



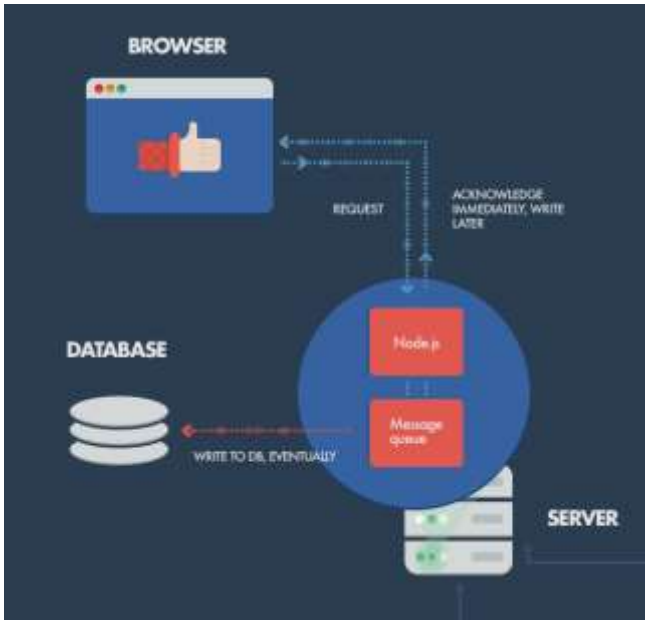
This is the simplest example. For a more robust solution, you might use a simple cache based on the Redis store. On the other hand, in an even more advanced solution, a message queue to handle the routing of messages to clients and a more robust delivery mechanism, which may cover for temporary connection losses, or storing messages for registered clients while they're offline. However, regardless of the improvements that you make, Node.js will still be operating under the same basic principles: reacting to events, handling many concurrent connections, and maintaining fluidity in the user experience.

API ON TOP OF AN OBJECT DB

Although Node.js really shines with real-time applications, it is quite a natural fit for exposing the data from object DBs (e.g. MongoDB). JSON stored data allow Node.js to function without the impedance mismatch and data conversion.

For instance, if you're using Rails, you would convert from JSON to binary models, then expose them back as JSON over the HTTP when the data is consumed by Backbone.js, Angular.js, etc., or even plain jQuery AJAX calls. With Node.js, you can simply expose your JSON objects with a REST API for the client to consume. Additionally, you do not need to worry about converting between JSON and whatever else when reading or writing from your

database (if you are using MongoDB). In sum, you can avoid the need for multiple conversions by using a uniform data serialization format across the client, server, and database.



QUEUED INPUTS

If you are receiving a high amount of concurrent data, your database can become a bottleneck. As depicted above, 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 to the database.

With that approach, the system maintains its responsiveness under a heavy load, which is particularly useful when the client does not 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 need to be reflected instantly (like updating a 'Likes' count on Facebook) where eventual (so often used in NoSQL world) is acceptable.

Data is queued through some kind of caching or message queuing 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 requests and responses are treated like isolated event; in fact, they are actually streams. This observation can be utilized in Node.js to build some cool features. For example, it is possible to process files while they are 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 (see next section).

PROXY

Node.js is easily employed as a server-side proxy where it can handle a large amount of simultaneous connections in a non-blocking manner. It is especially useful for proxying different services with different response times, or collecting data from multiple source points.

An example: consider a server-side application communicating with third-party resources, pulling in data from different sources, or storing assets like images and videos to third-party cloud services.

Although dedicated proxy servers do exist, using Node instead might be helpful if your proxying infrastructure is non-existent or if you need a solution for local development. By this, I mean that you could build a client-side app with a Node.js development server for assets and proxying/stubbing API requests, while in production you'd handle such interactions with a dedicated proxy service (nginx, HAProxy, etc.).