Node JS Basic to Pro Notes.

How v8 JavaScript engine works?

In order to obtain speed, **V8** translates **JavaScript** code into more efficient machine code instead of using an **interpreter**. It compiles **JavaScript** code into machine code at execution by implementing a JIT (Just-In-Time) compiler like a lot of modern **JavaScript engines** do such as SpiderMonkey or Rhino (Mozilla).

Is v8 a compiler?

V8 is a JavaScript engine built at the google development center, in Germany. It is open source and written in C++. ... It compiles JavaScript code into machine code at execution by implementing a JIT (Just-In-Time) **compiler** like a lot of modern JavaScript engines such as SpiderMonkey or Rhino (Mozilla) are doing.

Why is v8 so fast?

Furthermore, **V8** can be **faster** because the only thing that matters to the **V8** people is the interpreter -- they have no standard library to work on, no concerns about language design. They just write the interpreter. That's it. It has nothing to do with intellectual property law.

What is Node.js

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

Node. **js** is a platform-agnostic, free and **open source** JavaScript server framework. It extends the client-side scripting functionality to the server-side too, thus enhancing the Web experience. ... JavaScript (**JS**) is a high-level, interpreted programming language.

With Node.js, you can run your code simultaneously on both the client and the server side, speeding up the whole process of development. Node.js bridges the gap between front-end and back-end development and makes the development process much more efficient.

Node.js is a server-side JavaScript run-time environment. It's open-source,

including Google's V8 engine, libuv for cross-platform compatibility, and a core library. Notably, Node.js does not expose a global "window" object, since it does not run within a browser.

Who

Ryan Dahl

Why

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.

Why is Node.js popular?

Aside from being effective at what it does, Node.js is popular because it has a huge, active, open-source, JavaScript-based ecosystem. Also, it doesn't tend to break compatibility between versions in major ways.

What is the difference between Node.js and Angular/AngularJS?

Node.js executes JavaScript code in its environment on the server, whereas Angular is a JavaScript framework that gets executed on the client (i.e. within a web browser.)

Why is Node.js bad?

Node.js, being single-threaded, may be a bad choice for web servers doubling as computational servers, since heavy computation will block the server's responsiveness. However, Node.js itself isn't bad: The technology is quite mature and widely used for many different types of servers.

Why used for?

Node.js is primarily used for non-blocking, event-driven servers, due to its single-threaded nature. It's used for traditional web sites and back-end API services, but was designed with real-time, push-based architectures in mind.

Why use Node.js?

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. Welcome to the real-time web.

Is Node.js a framework?

No, it's an environment, and back-end frameworks run within it. Popular ones include Express.js (or simply Express) for HTTP servers and Socket.IO for WebSocket servers.

In Simple Words

Node.js shines in real-time web applications employing push technology over websockets.

Is Node.js a programming language?

No, the ".js" means that the programming language you use with Node.js is JavaScript (or anything that can transpile to it, like TypeScript, Haxe, or CoffeeScript.)

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.

Flash and Java Applets

One might argue that we've 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. Plus, they were run in isolation and often operated over non-standard ports, which may have required extra permissions and such.

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.

What Node does ?

Instead, it's a platform that fills a particular need.

When we do not have to use it ?

You definitely don’t want to use Node.js for CPU-intensive operations

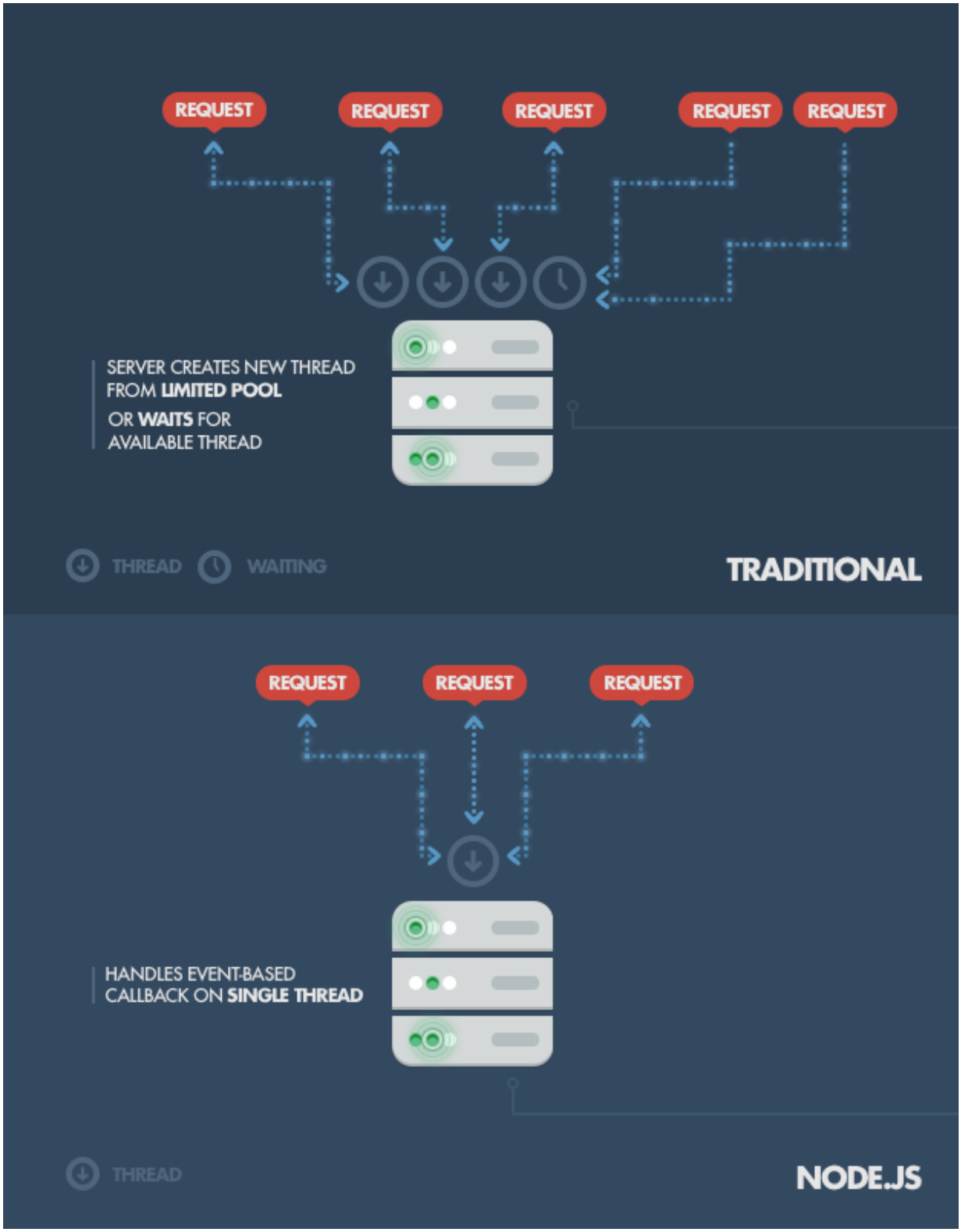
When we have to use it ?

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’s capable of handling a huge number of simultaneous connections with high throughput, which equates to high scalability.

How it works under-the-hood ?

OLD WAY :: 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,

NEW WAY :: 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.



What we need to take care ..

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).

How to avoid - exceptions

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, tools have been developed to monitor the Node.js

process and perform the necessary recovery of a crashed instance (although you probably won't be able to recover the current state of the user session), the most common being the Forever module, or using a different approach with external system tools *upstart* and *monit*, or even just upstart.

— Using upstart to detect node.js app crashes

— Using **Monit** (The watcher) ... The above scripts keeps checking the local host with an http request and if request fails it restarts the **node** app.
—— You should now configure **upstart** to start **monit** when your system restarts. This way most of your **Node**. js App failure scenarios can be handled.

--- NPM: The Node Package Manager — just like pod or gradle

Tools for Node JS

https://opensource.com/article/20/1/open-source-tools-nodejs

Express. **js** is a Node. **js** web application server framework, designed for building single-page, multi-page, and hybrid web applications. It is the de facto standard server framework for node

hapi is a simple to use configuration-centric framework with built-in support for input validation, caching, authentication, and other essential facilities for building web and services applications. **hapi** enables developers to focus on writing reusable application logic in a highly modular and prescriptive approach.

Connect is an extensible HTTP server framework for node using "plugins" known as middleware.

Socket.IO is a library that enables real-time, bidirectional and event-based communication between the browser and the server. It consists of: a **Node**. js server: Source | API. a Javascript client library for the browser (which can be also run from **Node**.

SockJS is a browser JavaScript library that provides a WebSocket-like object. **SockJS** gives you a coherent, cross-browser, Javascript API which creates a low latency, full duplex, cross-domain communication channel between the browser and the web server. ... **SockJS-node** is a **SockJS** server for **Node**. js.

Pug (formerly **Jade)** is a high performance template engine heavily influenced by **Haml** and implemented with JavaScript for **Node.js** and browsers. For bug reports, feature requests and questions, **open an issue**.

mongodb is a **mongodb** driver for node. it is not actual mongobd server. you will need to install **mongodb**

Redis is a fast and efficient in-memory key-value store. It is also known as a data structure server, as the keys can contain strings, lists, sets, hashes and other data structures. If you are using **Node**. js, you can use the node_redis module to interact with **Redis**.

Lodash is a very popular **npm** package which makes the tasks of working with arrays, numbers, objects, strings etc easier. They are great for. Iterating arrays, objects and strings. Manipulating and testing values.

forever is a node.js package that is used to keep the server alive even when the server crash/stops. When the node server is stopped because of some error, exception, etc. **forever** automatically restarts it. From the npmjs

Bluebird is a fully-featured Promise library for JavaScript. The strongest feature of **Bluebird** is that it allows you to "promisify" other **Node** modules in order to use them asynchronously. Promisify is a concept applied to callback functions.

Moment. js is a javascript date library that helps create, manipulate, and format dates without extending the Date prototype.

WHERE WE HAVE TO USE

CASE - 1 —> CHAT App - Node.js best example for node js to be implemented.

The chat application is really the sweet-spot example for Node.js: it's a lightweight, high traffic, data-intensive (but low processing/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.

Server side - Express.js, websockets

GET / request handle

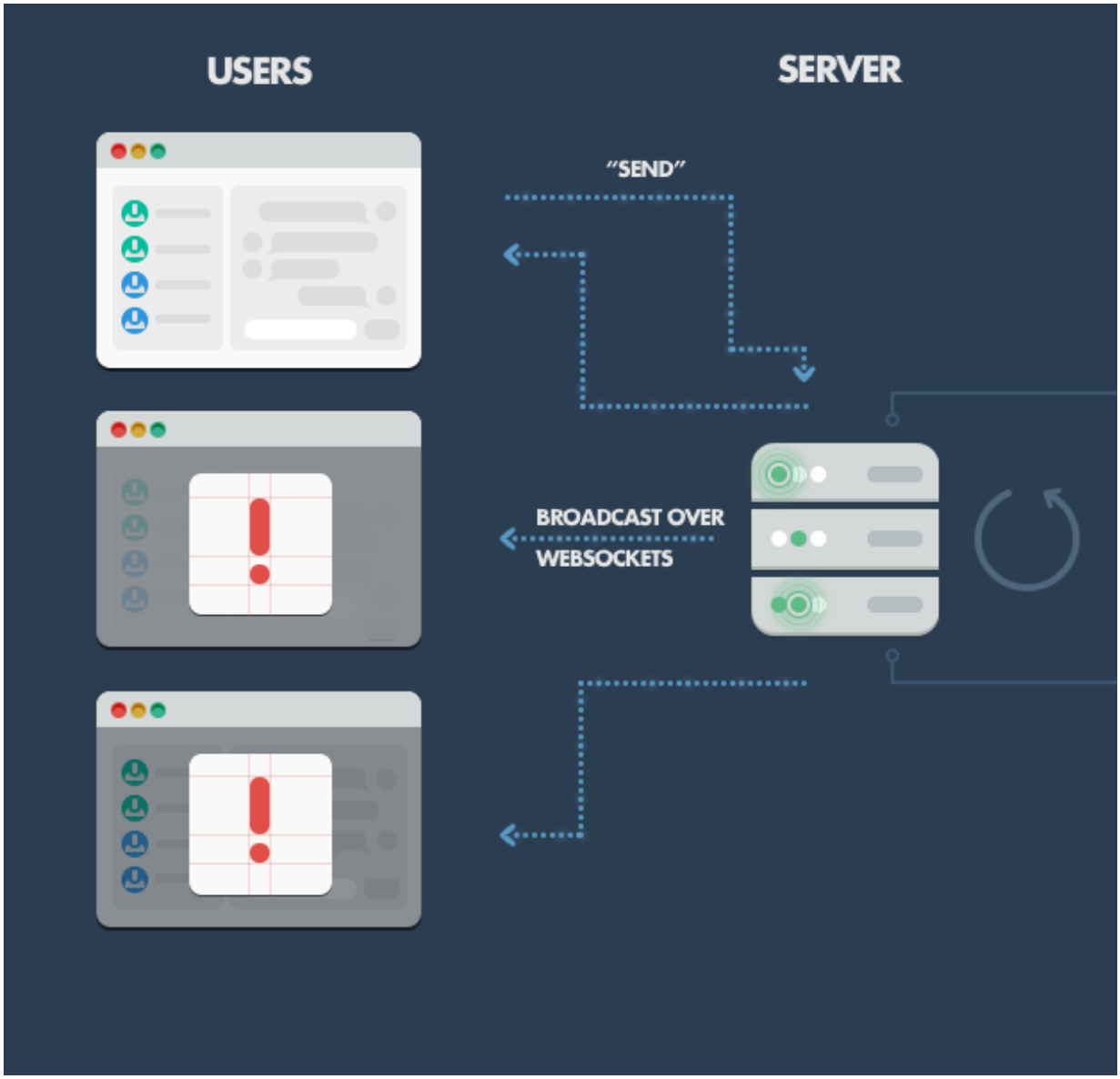
'Send' button to initialise new message

A websockets server that listens for new messages emitted by websocket clients.

Client side - html to handle events send, receive and send data to web sockets and get data from web sockets and all.
(messages sent by other users, which the server now wants the client to display)

When one of the clients posts 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 (initialized on web page initialization).
- Server-side component of the websocket connection receives 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.



you might use a simple cache based on the Redis store. Or 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. But 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.

CASE - 2 —> API ON TOP OF AN OBJECT DB

Although Node.js really shines with real-time applications, it's 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.

PROBLEM B4 Node js

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.

After Node js

you can simply expose your JSON objects with a REST API for the client to consume.
Additionally, you don't need to worry about converting between JSON and whatever else when reading or writing from your database (if you're 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.

CASE - 3 —> **QUEUED INPUTS**

PROBLEM

If you're 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. But because database access is a blocking operation (in this case), we run into trouble.

The solution

The solution is to acknowledge the client's behavior before the data is truly written to the database.

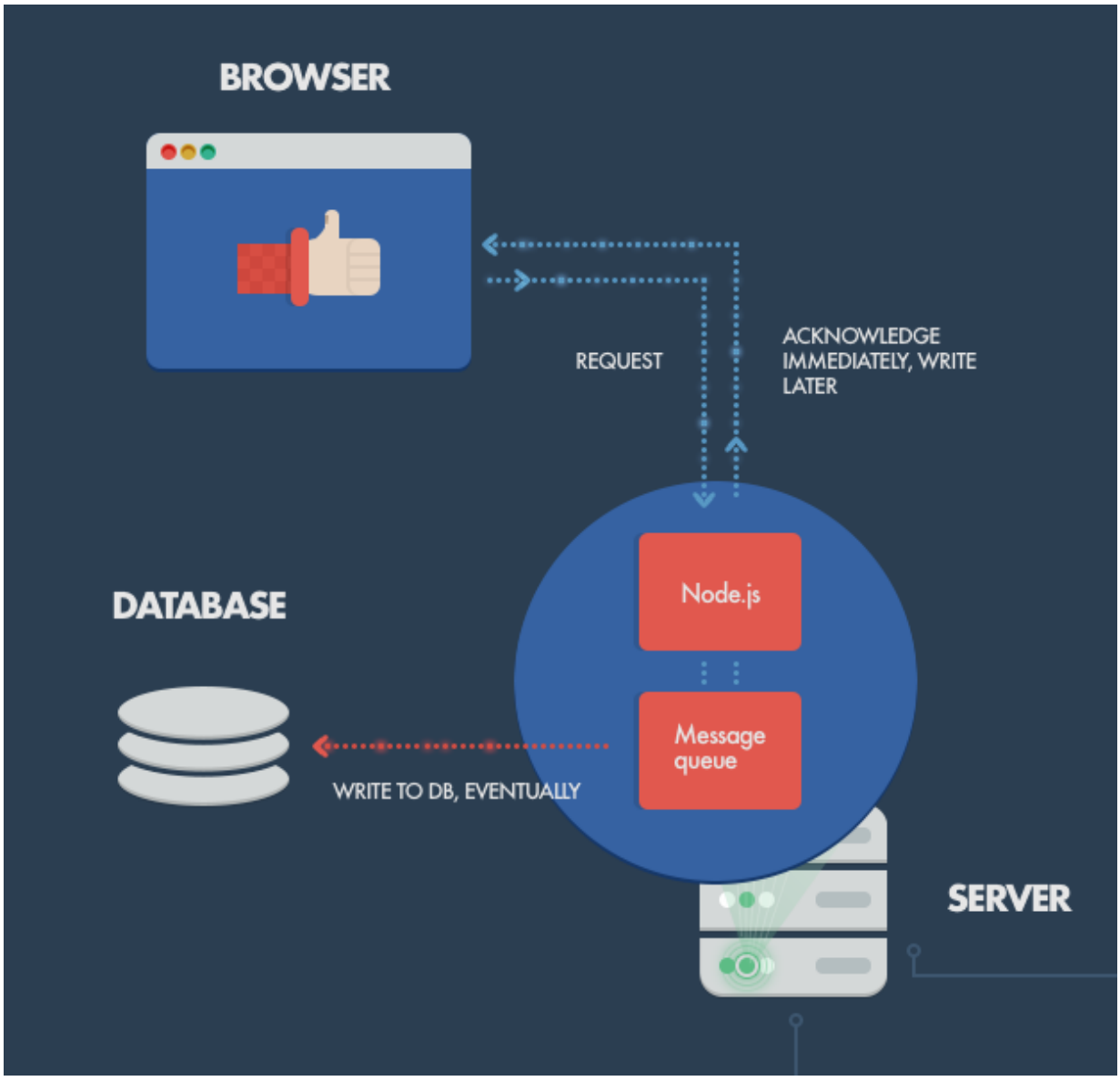
With that approach, the system maintains its responsiveness under a heavy load, which is particularly useful when the client doesn't need firm confirmation of a the successful data write.

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 consistency (so often used in NoSQL world) is acceptable.

Solution

Data gets queued through some kind of caching or **message queuing infrastructure**—like R**abbitMQ or 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 behavior 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.

CASE - 4 —> **DATA STREAMING**

In more traditional web platforms, HTTP requests and responses are treated like isolated event; in fact, they're actually streams. This observation can be utilized 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.

CASE - 4 —> **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's 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.).

CASE - 5 —> **BROKERAGE - STOCK TRADER'S DASHBOARD**

Let's get back to the application level. Another example where desktop software dominates, but could be easily replaced with a real-time web solution is brokers' trading software, used to track stocks prices, perform calculations/technical analysis, and create graphs/charts.

Switching to a real-time web-based solution would allow brokers to easily switch workstations or working places. Soon, we might start seeing them on the beach in Florida.. or Ibiza.. or Bali

CASE - 6 —> **APPLICATION MONITORING DASHBOARD**

Another common use-case in which Node-with-web-sockets fits perfectly: tracking website visitors and visualizing their interactions in real-time.

You could be gathering real-time stats from your user, or even moving it to the next level by introducing targeted interactions with your visitors by opening a communication channel when they reach a specific point in your funnel.

Nice platform for example.
https://www.canddi.com/

Imagine how you could improve your business if you knew what your visitors were doing in real-time—if you could visualize their interactions. With the real-time, two-way sockets of Node.js, now you can.

CASE - 7 —> **SYSTEM MONITORING DASHBOARD**

Now, let's visit the infrastructure side of things. Imagine, for example, an SaaS provider that wants to offer its users a service-monitoring page, like GitHub's status page.

With the Node.js event-loop, we can create a powerful web-based dashboard that checks the services' statuses in an asynchronous manner and pushes data to clients using websockets.

Both internal (intra-company) and public services' statuses can be reported live and in real-time using this technology.

Push that idea a little further and try to imagine a Network Operations Center (NOC) monitoring applications in a telecommunications operator, cloud/network/hosting provider, or some financial institution, all run on the open web stack backed by Node.js and websockets instead of Java and/or Java Applets.

Don't try to build hard real-time systems in Node (i.e., systems requiring consistent response times). Erlang is probably a better choice for that class of application.

Where Node.js Can Be Used

SERVER-SIDE WEB APPLICATIONS

Node.js with Express.js can also be used to create classic web applications on the

server-side.

However, while possible, this request-response paradigm in which Node.js would be carrying around rendered HTML is not the most typical use-case.

There are arguments to be made for and against this approach. Here are some facts to consider:

Pros:

- If your application doesn't have any **CPU intensive computation,** you can build it in Javascript top-to-bottom, even down to the database level if you use JSON storage Object DB like MongoDB. This eases development (including hiring) significantly.
- Crawlers receive a **fully-rendered HTML response**, which is far more **SEO-friendly than, say, a Single Page Application** or **a websockets app run on top of Node.js.**

Cons:

- Any **CPU intensive computation will block Node.js responsiveness**, so a threaded platform is a better approach. Alternatively, you could try scaling out the computation [*].
    - *[*] An alternative to these CPU intensive computations is to create a highly scalable MQ-backed environment with back-end processing to keep Node as a front-facing 'clerk' to handle client requests asynchronously.*
- **Using Node.js with a relational database is still quite a pain** (see below for more detail). Do yourself a favour and pick up any other environment like **Rails**, **Django**, or **ASP.Net MVC** if you're trying to **perform relational operations**.

Where Node.js Shouldn't Be Used

~~SERVER-SIDE WEB APPLICATION W/ A RELATIONAL DB BEHIND~~

Comparing Node.js with Express.js against Ruby on Rails,
for example, there used to be a clean decision in favour of the latter when it came to accessing relational databases like PostgreSQL, MySQL, and Microsoft SQL Server.

Relational DB tools for Node.js were still in their early stages. On the other hand, Rails automatically provides data access setup right out of the box together with DB schema migrations support tools and other Gems (pun intended). Rails and its peer frameworks have mature and proven Active Record or Data Mapper data

access layer implementations.[*]

[*] It's possible and not uncommon to use Node solely as a front-end, while keeping your Rails back-end and its easy-access to a relational DB.

But things have changed. Sequelize, TypeORM, and Bookshelf have gone a long way towards becoming mature ORM solutions. It might also be worth checking out Join Monster if you're looking to generate SQL from GraphQL queries.

HEAVY SERVER-SIDE COMPUTATION/PROCESSING

When it comes to **heavy computation**, **Node.js is not the best platform around**. No, you definitely don't want to build a Fibonacci computation server in Node.js.

In general, any **CPU intensive operation** annuls all the throughput benefits **Node offers with its event-driven**, **non-blocking I/O model because any incoming requests will be blocked** while the thread is occupied with your number-crunching—**assuming you're trying to run your computations in the same Node instance you're responding to requests with.**

As stated previously, **Node.js is single-threaded and uses only a single CPU core.** When it comes to **adding concurrency on a multi-core server, there is some work being done by the Node core team** in the form of a **cluster module** [ref: http://nodejs.org/api/cluster.html]. You can **also run several Node.js server instances pretty easily behind a reverse proxy via nginx.**

With **clustering**, you should still offload **all heavy computation to background processes written in a more appropriate environment** for that, and having **them communicate via a message queue server like RabbitMQ.**

Even though your background processing might be run on the same server initially, such an approach has the potential for **very high scalability.**

Those **background processing services could be easily distributed out to separate worker servers without the need to configure the loads of front-facing web servers.**

Of course, **you'd use the same approach on other platforms too**, **but** with **Node.js you get that high reqs/sec throughput we've talked about,** as **each request is a small task handled very quickly and efficiently.**

Conclusion

In Node, blocking operations are the root of all evil—99% of Node misuses come as a direct consequence.

Remember: 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.