NODE.JS FOUNDATION FIELD GUIDE:

Building APIs

The dos and don'ts in using Node.js to build APIs

Executive Summary

Node.js is perfect for building application programming interfaces (APIs). It has a short learning curve, especially for developers with JavaScript experience. You can extend your talent base without adding extra hires since developers can work on both the front and back ends. Node.js works in small tasks, which is fast in any case but also ideal for microservices. It also rests quietly while waiting on data returns, which frees resources so they can be used elsewhere in the interim.

This guide explains how to use Node.js to build APIs, and why and where it is appropriate to do so. Several short case studies are also included to further illustrate real-world uses.

Intro to Node.js

Node.js is a JavaScript runtime environment. It is built on Chrome's V8 JavaScript engine and uses an event-driven, non-blocking I/O model. The asynchronous approach of Node.js allows applications that must handle hundreds of thousands or millions of concurrent requests to scale much more effectively.

Lightweight and efficient, Node.js can execute JavaScript code on the server-side and enables a productive, end-to-end experience for JavaScript developers. NPM, the largest ecosystem of open source libraries in the world, is Node.js' package ecosystem.

Endpoint Explosion

Endpoints are the touchpoints where an API can access the resources it needs to perform a prescribed function. On the one end are resources that typically come from a myriad of disparate systems, from legacy applications to cloud systems. On the other end is a growing proliferation of mobile clients and Internet of Things (IoT) devices. Together these developments result in an endpoint explosion.

Calling it an "explosion" is not an overstatement. According to Statista IoT statistics, IoT devices total 23.14 billion in 2018. By 2020, IoT devices will number almost 31 billion. By 2025, the number will skyrocket to more than 75 billion, without even counting phones and tablets.

Statista shows worldwide mobile phone users numbering almost 5 billion in 2018, while the smartphone user headcount is 2.53 billion, and tablet users number 1.32 billion. While growth in mobile devices is slowing due to the maturing market, the segment is still growing. Statista predicts 67% of the worldwide population will own a mobile phone in 2019, up from 62.9% the year before.

Several factors brought us to this point:

1. Legacy systems continue to linger. For years, enterprise computing was done on-premises and behind a firewall. The IT department was valued for its ability to create custom software to automate key business processes. Over time, these became known as legacy systems as companies shifted first from custom builds to commercially-produced "off the shelf" software, and then to cloud services. Thus, the typical modern enterprise's operations are powered by a mix of decades-old legacy systems, myriad on-premises commercial software, and multiple cloud-based services.

2. M&As continue to add software types, brands and versions to the IT mix. Complicating the situation further is the unintended hodge-podge of systems resulting from mergers and acquisitions (M&As) that are common to every industry. Enterprises may also have a mix of software versions in use resulting from M&As and/or an effort to leverage existing investments for as long as possible, and thereby postpone or cut costs.

3. Mobile explodes onto the scene. Smartphones, tablets and wearables further stretched the limits of legacy systems and escalated the demand for integration. Employees were no longer contained to working with hardware provided by the company behind the firewall. Bring Your Own Device (BYOD) programs became common in large and small organizations alike, and so did problems associated with connecting multiple operating systems and form factor limitations with enterprise applications and data resources.

4. IoT devices take over the world. Another set of disruptive technologies blasted onto the scene: the Internet of Things (IoT). Many come preloaded with apps; others are provided with custom enterprise apps.

5. The API economy emerged. Product and services companies are shifting to platforms and seeking to attract developers to build against their data. To do so requires making all products API-ready.

Because of the convergence of these trends, APIs are now central to nearly all business processes. APIs are not new. The need to integrate applications and move information between programs arose early in the IT space and has grown since. APIs open the door to code-sharing and provide limited access to a specific set of features.

All that remained was to determine the best way to perform what could be very complex and time-consuming tasks at scale and across form factors, such as the array of operating systems and screen sizes on mobile devices. That spurred the API-first development movement wherein APIs were built before the applications that would consume them existed – the opposite of building APIs solely to extend content from existing applications. Both approaches served important functions, but the appetite for more APIs to allow freer, and more widespread sharing amongst a growing number of devices and computing schemes proved insatiable.

An API tier is needed to bring together all the endpoints to provide a uniform API for all clients: be that smartphones and wearables, or any number of thousands of IoT devices and sensors. In other words, an API tier bridges older application architectures and services with newer ones and delivers all to any device in the appropriately optimized format. It is the best means with which to modernize the old, leverage the new, and speed delivery of both.

How Node.js Bridges the API Tier

Developers can choose from a number of different strategies using a variety of languages and tactics in the stack for an API tier. But the most successful route, particularly for mobile environments, would be something with:

- A short learning curve;
- the maximum amount of agility and flexibility in both the front and back ends;
- an ability to act with the least connectivity demands and smaller data transfers;
- and strong community support.

Node.js fits that description completely given anyone familiar with using JavaScript can learn it easily, it can be used on both the front and back ends, it transfers small amounts of data, and rests quietly while waiting on data which eases demand on connectivity and other resources. Node.js also has a robust and thriving ecosystem.

Indeed, Node.js is the perfect fit for building APIs to modernize and extend the life of legacy systems.

"In situations where you want to provide a better user experience to your customers, but you don't have the resources or expertise to update legacy services that support it, you can plug in Node.js as a middleware layer, an aggregation tier, to help provide that better experience and somewhat update the overall system to newer technologies," said Brian Clark, a cloud developer advocate at Microsoft.

Node.js' usefulness in such a scenario is readily evident.

"You can reduce the large payload, without having to update the legacy system, by inserting a Node.js layer to massage the data, and only send back to that mobile UI the data that's needed to present to that end user. That's just one scenario," explained Clark.

Besides effectively modernizing legacy systems, Node.js can also bundle disparate data and sources to fuel an entirely new application.

"Node.js is perfect for being the legacy glue to build a singular response back to whatever mobile client is asking for it," says Al Tsang, founder and CEO of LunchBadger, a multi-cloud platform for microservices and serverless computing.
Tsang gave as an example a scenario with a fictional Yelp-like app on a mobile client, where you would want one response coming back from one API with all the necessary information.

"You don't want your client, your mobile phone to be making three different calls, the first call to the legacy system for the name. The second call to the legacy system for the review. The third call to Google Maps with the address to get back a point or a pin on the map. That'd be very inefficient, right?" he explained.

"You want to send it to one API and have that API do all the bird dogging and legwork for you, collate all that information into one payload, and then send that payload across to the phone in one shot when it's ready," Tsang said.

"Node.js is perfect for this, because Node.js is asynchronous."

When and How to use Node.js to build APIs

Moving away from traditional application architectures and operations and toward microservices is an essential step in building APIs for next-generation apps. The old ways are too slow and cumbersome for today's rapid and continuous release cycles. They are certainly too slow to keep the business competitive in a marketplace where disruptors can change the rules overnight.

"The old way of writing applications with .Net and Java was to generally focus on very monolithic, single applications that expose multiple endpoints," explains James Snell, Open Source Architect at nearForm. The software company provides commercial support for its Node.js container images, including OpenShift and Docker distributions of Node.js. NearForm is also one of the largest global contributors to Node.js Core.

"Everything is bundled into one server and that slows down your development. It means everything has to be developed in one step. You're going to need a lot of full cycles, and it will be a year until the application is actually deployable," Snell added.

By contrast, Node.js enables faster development since it works in a series of small tasks.

"Node.js is a perfect fit for building APIs because you can do it and iterate on it very quickly. From conceptualizing and writing the code to actually deploying it into production is a much shorter cycle than on any other platform that we've seen," Snell added.

Node.js: Speeding Development with Microservices and Containers

Breaking jobs into small tasks, i.e. microservices and individual APIs, speeds everything up. Using microservices, you're developing small, single-purpose endpoints. Single-purpose applications can deploy independently of one another, thus enabling a continuous stream of releases, each making changes or adding features as the business needs them. For example, one application might focus on authentication, and another application on serving up the data, and so on.

Containers, which create a basic computing wrapper that can run on any infrastructure, offer a superior way to manage modern microservices applications. The combination of containers and microservices applications allows developers to test new APIs modularly, in variable stacks, and with resources spun up only when demanded.

"Node.js is ideally suited for those small, single-purpose applications, and single-purpose things. Simply because of its ability to process those kinds of singular workloads much more efficiently than something like a web share," says Snell.
Part of the speed is a result of developer familiarity with JavaScript. Almost every developer has had some experience with it. The learning curve is thus short, plus frontend developers can now work on the backend too, and vice versa, effectively optimizing staff availability. Additionally, developers can extend their experience to enhance their credentials and develop their own careers.

The rest of the speed advantage from using Node.js to build APIs comes from reduced obstructions from mechanisms. For example, with Node.js a server can be ready to go in mere minutes, a decided advantage over the day or more it takes to setup a server in more complex environments.

"Anytime you're taking an existing model or application and splitting it into multiple parts, into microservices, the purpose for doing so is to be able to move and iterate faster," says Snell. "That is why you would use Node.js in pulling those pieces out."

The way Node.js works with microservices is an exercise in efficiency that also preserves the initial code.

"It's kind of like superglue for technical applications. Instead of going in and mucking with the messy legacy code itself, glue a layer on top that acts as a clean filter. That filter will help you surface only the pieces that you're interested in. You can take advantage of using that same code base, but you run it for the purpose of that microservice as the Node.js filter on top," Tsang explained.

The Aggregation Tier

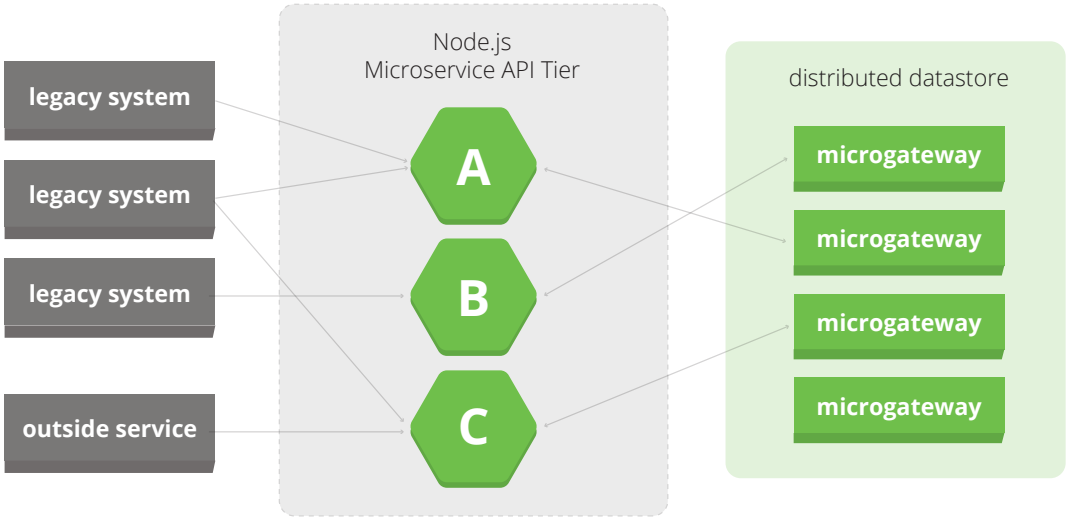
Another advantage in using Node.js to build APIs is found in its asynchronous and event?? nature. It uses resources only when there is something for it to do, and otherwise sits still, freeing those same resources to do other things. In other words, it acts as a perfect aggregation tier.

In short, Node.js fans out to find the data it needs, collects the data efficiently, and intelligently aggregates it into a single, ordered response.

"You send one request from whatever client, whether it is a mobile device, mobile client, or a web client. It's only sending one request across the network, and then the Node.js aggregation tier is responsible for connecting the dots and making those other calls to those individual microservices, and then aggregating that data and sending it back as just one response to the client," explains Clark.

Node.js runs these actions in parallel. It acts again only when it has indication or notification that is has all the responses back.

"Node.js is perfect for what we call 'legacy superglue' because it is really good at transforming data," explains Tsang. "For example, the data coming back from legacy systems one and two, and from Google Maps, may all look different, but Node.js and its libraries for transformation are really good at shaping the response, making it uniform, and making it consumable by the client as an API."



For example, a search for restaurants within five miles of a given location can render a long list. Node.js would handle that by sending back the first five on the list to prevent overwhelming a mobile phone. If a user indicates a desire for more of the data beyond that, then Node.js will send it via paging, in groups of five returns.

"You're not going to send 2,000 restaurants to your mobile app in one singular call. If you did, your mobile would sit there forever trying to process all that information," says Tsang. "You would sit there waiting needlessly. If it sent the first five, you could imagine that you're going to get a much quicker response from that interaction."

"It's perfectly suited for this kind of use case of fanning out, doing it so asynchronously, only acting on events, not holding up compute resources while events are occurring, efficiently taking back data and services and aggregating them into a collective response, into what we call a modern-day API," says Tsang.

Using Node.js in Server Optimization

Using Node.js to separate jobs into microservices enables you to also optimize server use by configuring servers to meet the specific needs of these smaller and well-defined tasks.

For example, consider an ecommerce scenario where there is a catalog microservice, a checkout microservice, and a customer account microservice. Checkout would be use intensive because it's specific to the individual and takes up memory that is only related to that individual. By comparison, catalog is fixed. Users don't get a separate piece of memory just to view the catalog. Instead, everyone shares it and people are simply looking at different portions of it. Customer Account is not as fixed, but changes to credit card information, customer addresses, etc. are typically infrequent and therefore require fewer resources.

"Servers can be optimized accordingly. Use a server with heavy memory sufficient to fit the entire catalog in the allotted space, because everyone is sharing that same catalog," says Tsang. "Server number two that's handling the checkout process needs to have a lot of network availability, so I can optimize that server for network availability and not worry too much about the memory requirements. So long as there's not like a bunch of users stuffing the entire catalog into their cart, because they only want one or two items or however many pieces of the catalog. They want to do a bunch of transactions that require a bunch of network IO."

"If you have a server that's only handling the customer microservice, that box probably can be a quarter of the size or so because you don't need to throw a lot of servers at it," Tsang added. "But imagine if all of these things were to happen on one box. How the heck can you optimize that box? Basically, you are stuck with getting the beefiest box possible for the worst-case scenario of what any number of users are going to do at any given time."

When and How Not to Use Node.js

While the advantages of using Node.js are clear, it isn't a panacea. There are times when you'd be better served using something else.

Here are three of the biggest "don'ts" in using Node.js:

As a reverse proxy

"Node.js is like the big popular hammer everyone wants to hit things with, whether it works great for that or not. One of the examples of this is a reverse proxy," says Snell.

A proxy server sits behind a firewall and directs client requests to the appropriate back end server. A reverse proxy is a type of proxy server that adds another level of abstraction to improve performance and provide a buffer between users and servers on the back end.

"Node.js's way of processing data is not very optimized for a reverse proxy case. It's not designed to receive data and push it back out as quickly as possible. It's designed to receive data, process it, and get a response as quickly as possible, as long as that's a very small piece of work," says Snell.

"Some people use Node.js as a reverse proxy but it would be better to use something like NGINX, which is specifically designed for that purpose," he added.

To combine API gateway logic with application logic

"Node.js can be used for both, and we, in fact, do use it for both. But you don't want the same Node.js code to do the gateway portion as you do the logic," says Tsang.

The gateway should never spot check what the application is doing but rather it should focus on its gateway concerns, such as whether the data is formatted in JSON or XML and whether that fits the client request.

The application logic on the other hand is focused on the application's internal task. For example, it may find an error in the checkout math and correct the error before delivering the information.

Combining the two tasks may sound more efficient but it just makes a mess of things in a hurry.

"You're bogging it down number one, and number two should you decide to update the logic now, you have to update it in two different places," Tsang explained.

Use it for either but maintain the classic separation of concerns.

"You don't want the gateway to worry about having the same pricing logic going out the door, because that's already being done by the microservice," Tsang added. "Similarly, the microservice shouldn't ever care about what the payload is. It just returns back the response, and says, 'The answer is ten. Gateway go put it in whatever format you'd like it to be.'"

For anything requiring heavy computations

"Node.js isn't good for anything that is very computationally complex, by which I mean any operation and set of instructions that takes a long time to complete, or consumes a lot of resources," says Snell.

"In Node.js, it's going to stop anything else from happening. So, say for instance you have 100 people that are all sending requests to a Node.js server. If every single one of those requests requires Node.js to go off and spend 10 seconds processing data, it will only be able to process one of those at a time."

It is important to note, however, that work is underway to give Node.js access to FPGA and GPU acceleration so that it can handle more computationally intensive tasks without burdening the CPU with those tasks as it will instead asynchronously perform the computation on these alternative computing devices[1].

1. Node.js has a thread pool, so that can offload some of the work, but it will still drive up CPU usage on the machine to the point where it interferes with Node.js' ability to deliver the results. The Node.js community is working on giving Node.js access to FPGA and GPU acceleration, whereby it will be able to handle more computationally intensive tasks by virtue of the fact that it will not burden the CPU with those tasks but will rather asynchronously perform the computation on these alternative computing devices.

Case studies

Below are a few notable examples of enterprises using Node.js for APIs.

A. GoDaddy

In 2015, the website builder team at GoDaddy had already transitioned to a complete Node.js API stack. The tech stack is eclectic because in Charlie Robbins' words, "the feature is what matters." GoDaddy offers a wide variety of features from a wide variety of sources. Robbins is the director of UX Platform at GoDaddy, formerly the founder at Nodejitsu and a Gold Director at Node.js Foundation. Watch his presentation on using Node at GoDaddy on this YouTube video.

In 2015, the website hosting giant found more ways to use Node.js and leverage its benefits. The company moved from a monolith, actually a set of monoliths, to microservices and used Node.js to help make that happen.

While GoDaddy's tech stack remains eclectic, it continues to lean heavily to Node.js, using it in myriad ways, including but not limited to:

- CI/CD pipelines
- Website builder product
- Microservices and containers
- Mobile apps
- Big data for small business tools
- Browser builds and frontend ops

The company not only focused on leveraging Node.js at every opportunity, but also in replicating the open source culture internally. They found forming working groups and developing mentorships between people on differing teams to be particularly fruitful.

B. Netflix

Netflix, the now famous and popular video streaming service, counted 85 million subscribers around the globe in 2016. Serving a changing catalog of movies to an ever-growing subscriber base located anywhere on the planet and using any type of device is no small undertaking. The company evolved their API strategy over time in a continuous effort to improve performance.

In its early days, Netflix was only available on browsers for which they used a Java based web server, which is a monolith. That became too unwieldly for the company to expand beyond browsers to other devices, and to change as needed to do newer things. Because of these limitations with a monolith, the company became one of the earliest adopters of microservices and is credited as one of the most successful users of microservice architecture ever since.

Next in their API evolution, they moved to a REST API, which helped breakup the monolith. That also unlocked the ability to support multiple devices but unfortunately meant weeks of delay between API changes, difficult to maintain due to complexity and inefficient to use. That was followed by API.NEXT that allowed Netflix teams to upload their own custom APIs to the service, all crafted in Ruby. But that led back to a monolith and higher costs for some instances.
Today they use a next-generation data access API that is Node.js containers on the front end that connect

with an Edge API on the remote service layer that connects to back end services. This is a set of JS data access scripts running Node.js and restify inside of a Docker container. The focus was to provide an infrastructure-less environment for UI engineers. The Node.js runtime is a production-ready platform-as-a-service so the developers need only bring JavaScript business logic because everything else is already there for them.

The company uses SemVer, from the Node.js ecosystem, for versioning management, which they use for packages as well as services. They built an NPM for services index called the Node.js API Index, which tracks all of the versions of published applications and maps back to a Docker image. This helps with the reproducibility of bugs in applications. SemVer is also used for routing.

If you would like to hear more about how Netflix evolved its API and how it is using Node.js now, check out this talk on YouTube.

C. Node.js API for Native Modules Project

N-API is a stable Node.js API layer that makes native modules work across different versions and flavors of Node.js without the need for recompilations. In short, this means no more breakage in native modules when a new Node.js version is released.

Using N-API aims to eliminate the need to update and recompile native modules with each new release. This added stability cuts costs, time, effort and aggravations for the maintainers in updating modules for new version support. It also spurs adoption of new Node.js versions since module consumers are less afraid to upgrade when the fear of breakage is removed.

N-API enabled modules relieve native code dependencies and became available as Experimental in Node.js 8 in May 2017. N-API is since stable in Node 10.x, and has been back-ported as Experimental to 8.x and 6.x.

Responses to it have been overwhelmingly positive. Within four months of its broader release, there were 1500 downloads and counting of the Node.js add-on API module in NPM.js.

N-API is an evolution of NaN and is expected to replace it entirely. The reasons for the move from NaN to N-API are clear:

- More complete isolation from V8
- Compile once/run multiple versions
- Supports both C and C++

The project team successfully ported several popular modules, including Node-Sass, Canvas, SQLlite, LevelDown, Nanomsg, and IoTivity. In each case the team was able to successfully replace the significant API surface in each module with N-API.

Today the community is continuing to work to make it easier to develop modules, increase backward compatibility, and port more popular modules to reach critical mass.

D. Joyent

Joyent's public, private, and hybrid clouds are all built on microservices written in Node.js, but that is not how the story began.

In 2009, the company built an API written in Ruby and accessible over HTTP to use in managing its public cloud infrastructure. The API, dubbed Master Control Program API (MAPI), rapidly grew into a monolith as features were continuously added and all of the code piled up in one git repository. The usual monolith spawned nightmares ensued: cascading bugs and disruptions from unfettered code sharing and communication failures.

Joyent also found interacting with the cloud API to be unsustainably difficult and turned to creating tools to simplify its use. However, creating CLI tools in Ruby was as, if not more, difficult a task than using the cloud API. The logical solution to these problems was to abandon Ruby and the monolith in favor of Node.js and microservices.

The company hired Node.js creator Ryan Dahl, npm creator Isaac Schlueter, and Bryan Cantrill, Joyent's CTO. Other contributions to code, monetary support and additional talent soon followed.

Early on the company saw the benefits in using Node.js to break the monolith into smaller tasks. Node.js is uniquely suited to microservices builds.

The demand for microservices, and thus Node.js, lies in large part with the reliability of systems built in this way. Reliability that is unmatched given the granular control, increased visibility, and change control that prevents cascading bugs and disruptions.

None of that reasoning meant that using Node.js in the early days was easy. At the time there were no tools for post-mortem debugging or process observability. Such tools are essential in troubleshooting production issues.

To resolve the lack of tools, Joyent's CTO added DTrace support while other Joyent employees contributed by:

- creating Node.js modules and documentation for improved process observability
- built mdb_v8 to aid with post-mortem debugging of Node.js applications using MDB
- added post-mortem debugging support for Node.js executing on Linux

Joyent also benefitted from the contributions of others in the community. The company added better tracing to their microservices by following the OpenTracing specification.

Joyent engineers iterate and deliver at extremely rapid rates because of adopting both Node.js and microservices. Examples include its Manta and Triton systems, and the CLI tools made with Node.js that make interactions from the command-line simpler.

The company soon moved past the more common CLI tools and HTTP services use cases to more uncommon uses. The reasons for extending Node.js use to these other use cases were myriad, but topping the list was usefulness with JSON configuration, API services over HTTP, and its lightweight nature.

Joyent continues to use Node.js throughout its products and services, and to contribute to the community.

Conclusion

Node.js is ideal for API building given it is designed to work in small pieces of tasks suited for microservices, which enables changes without involving an entire shift in a monolith system and without disturbing legacy code.

Thus, it is a perfect fit for building APIs to modernize and extend the life of legacy systems. It is also ideal for jobs aimed at mobile and the Internet of Things devices as it does not overwhelm these systems or small screen real estates.

Join the Node.js and entire JavaScript ecosystem at JS Interactive October 10-12, 2018 in Vancouver to learn how you can get maximum business advantage from Node.js.

References

1. Slaying Monoliths with Docker and Node.js, a Netflix Original by Yunong Xiao, Netflix.com – YouTube video
2. Node.js at GoDaddy, Charlie Robbins – YouTube Video
3. Node Foundation Resources to Help You Succeed with Node.js page
4. N-API – Next Generation Node API for Native Modules – YouTube Video

Acknowledgements

Special thanks to Brian Clark, a cloud developer advocate at Microsoft; James Snell, Open Source Architect at nearForm; and Al Tsang, founder and CEO of LunchBadger, for their contributions to Node.js and to this paper.