acmqueue

# Case Study
# Node at LinkedIn: The Pursuit of Thinner, Lighter, Faster

**A discussion with Kiran Prasad, Kelly Norton, and Terry Coatta**

*Node.js, the server-side JavaScript-based software platform used to build scalable network applications, has been all the rage among many developers for the past couple of years, although its popularity has also managed to enrage some others, who have unleashed a barrage of negative blog posts to point out its perceived shortcomings. Still, while new and untested, Node continues to win more converts.*

*In 2011 LinkedIn joined the movement when it opted to rebuild its core mobile services in Node. The professional networking site, which had been relying on Ruby on Rails, was looking for performance and scalability gains. With its pervasive use of non-blocking primitives and a single-threaded event loop, Node seemed promising.*

*Following the creation of Node.js in 2009 by Ryan Dahl (now at Joyent, which sponsors and maintains Node), it didn't take long for developers to seize upon it. Because Node uses JavaScript, a language largely associated with the client side of Web apps, it clears the way for developers working on the client side to also work on corresponding functions over on the server side.*

*Kiran Prasad, who joined LinkedIn as senior director of mobile engineering in 2011, led the company's transition to Node. On the server side, LinkedIn's mobile frontend is now built entirely in Node. Prasad admits Node isn't the best tool for every job, but upon analyzing LinkedIn's system, Prasad and his team determined that what was needed to improve efficiency was an event-driven system. Node also proved attractive because it's thin and light while allowing for the direct manipulation of data objects.*

*Prasad was well prepared for his role in mobile services at LinkedIn, having already accumulated years of experience in mobile apps working on the WebOS platform at Palm and Handspring in addition to stints as an independent developer of mobile Web software (as CEO at Sliced Simple and CTO at Aliaron).*

*He talks here about LinkedIn's adoption of Node.js with Kelly Norton and Terry Coatta. Norton was one of the first software engineers to work on the Google Web Toolkit (GWT) before cofounding Homebase.io, which develops next-generation marketing tools.*

*Coatta is CTO at Marine Learning Systems, which has developed a learning management system targeted at the marine industry. He previously worked for AssociCom, Vitrium Systems, GPS Industries, and Silicon Chalk.*

**KELLY NORTON** Tell us what went into LinkedIn's decision to use Node.js.

**KIRAN PRASAD** We were running a Ruby on Rails process-based system, and it became pretty clear that just wasn't going to scale the way we needed it to. I guess you can always scale something if you're willing to throw enough money at it, but obviously that didn't seem like the right way to go. Also, working with the mobile model—where there are lots of microconnections—we could see that a process-based approach was going to run into difficulty with the Ruby on Rails stack.

We also noticed that there was a pretty big performance hit with Rails since we were doing a lot of string manipulation and the Ruby interpreter version we were using was struggling to garbage-collect all the small-string objects. Nor was it particularly optimized for JSON (JavaScript Object Notation)

translation, which was what our back end was giving us, as well as what our front end was looking to consume.

Clearly, Ruby on Rails was built more as a Web stack in that its real value lies in the templating it offers for that structure, along with some of the framework concepts it provides for the app and the controllers. But the controllers and the views actually move down to the client whenever you're doing client-side rendering, which is what happens with mobile systems.

With the larger, higher-scale stacks you find at places like LinkedIn, you also start breaking apart the model. That is, you're not really inside Active Record on Rails at that point since you essentially end up moving down a level into other servers or services. This means the middle layer starts to get pretty thin and really focused on string manipulation.

So when we really started to look at that, we said, "This sure doesn't feel like such a great fit anymore. It's not designed to do what we're trying to accomplish now." So, what could we replace it with? It had to be something that was evented, was good at string manipulation, and was light and quick and easy to use. We started looking at some evented frameworks in Ruby such as EventMachine, as well as Twisted in Python. But there seemed to be no mainstream evented Java frameworks when we first started checking into this. While Play has become a little more prevalent now, back when we were doing our analysis, in early 2011, it didn't pop up on our radar for some reason.

**KN** But besides being event-driven, what was so compelling about Node?

**KP** We looked at Node and ran some load against it, at which point it would send a request out to something like six other services, grab that data, merge it, and then pop it back out in a fairly simple way. We just ran with that all the way up to about 50,000 QPS (queries per second). Along the way we discovered that Node was roughly 20 times faster than what we had been using and its memory footprint was smaller as well.

Obviously, Node.js also offers other benefits beyond the technical aspects. JavaScript is a language lots of people understand and are comfortable coding in. Besides, it didn't hurt that Node was getting a lot of hype at the time—and still is. In some ways, that makes it easier for me to recruit.

**TERRY COATTA** You mentioned that you had moved away from using Active Record in Ruby as your model representation, but obviously that model had to end up going somewhere. Why didn't you choose to use the infrastructure you already had for the model to handle the middleware as well?

**KP** The thing about models is that they're really designed around objects. They have properties; they have methods; they're very structured and statically typed; and you're trying to create an environment where they're very solid so you know exactly what each object is. This is just the sort of approach we were all taught in our first exposure to object-oriented design.

So you would think a more object-oriented language would be ideal for building systems like that. But then that moves out to the view controllers, and in mobile systems, the view controllers get pushed out all the way to the client. Then the question becomes: What's left in the middle? Basically what you find there is just a bunch of functions that effectively manipulate hashes of data in order to format them. So now you're just down to formatting and a little bit of aggregation.

I'm not sure I really had the clarity at the time to articulate why we chose Node.js. Now after using it for a few years, it has become pretty clear that in this layer that's essentially the glue between your front end (which is literally in the client now) and your back end (which happens to be your data model), a functional sort of language is actually the best fit. At this point, we write all our back-end stuff in Java, and all the Java stacks we're using are more process-based.

**TC** Having had this experience with Node.js in the middle layer, then, would you contemplate using it for the back end?

**KP** Right now, we're working on creating a data store that's mobile-specific, and as part of that we did an analysis of whether we should build it in Node.js and JavaScript. It turned out the team wanted something that was a little more precise—or typed, I suppose. I'm actually trying to steer clear of classic programming stuff such as taking a more statically typed approach, but the team definitely wanted the ability to define an object with certain methods and properties, and for that to be guaranteed so nobody could mess it up. They don't want just anyone out there taking over that prototype or doing anything to it. Still, from a performance standpoint, I would love to try a more event-driven framework. Currently, we're using Rest.li internally to do some of this back-end, data-store stuff because we really do believe the event-driven stuff has transformed our architecture.

**TC** In terms of the performance speedup you observed with Node.js, did you also construct lightweight prototypes for the other languages you were considering for your middleware layer?

**KP** We did some prototyping with Ruby and Python using the evented frameworks EventMachine and Twisted. The bottom line was that Node proved to be 2-5 times faster than both of those in terms of raw throughput. What was even more exciting and really sold us on Node was that it took only two or three hours to write the Node prototype while it took us more on the order of a day or two to write the EventMachine and Twisted ones, just because we had to download so much more stuff.

For example, you need to make sure you're not using the standard HTTP library in Python, but instead the Async HTTP library. That's the sort of thing that pretty much applied across the board. No matter what we wanted to do, we couldn't use the standard Python library. Instead, we had to use the special Twisted version. The same held true with Ruby. Like a lot of others in the community, we found out just how much easier it is to get started with Node, where everything you need is essentially there by default. Further, the fact that we could get stuff up so much faster with Node was really important. That's just another form of performance, right? Developer productivity definitely counts for something.

Memory footprint was also a factor. We looked at how well VMs (virtual machines) worked in each of these languages, and the V8 JavaScript Engine just blew everything else away. We were doing 50,000 QPS with all this manipulation, and we were running that in about 20 to 25 MB of memory. In EventMachine and Twisted, just loading all of the classes necessary to do the async stuff rather than the standard stuff was more like 60 to 80 MB. Just at a raw level, Node was going to run in only about half the memory footprint.

**KN** What sorts of things did you end up missing that you would have had in a more "modeled" environment?

**KP** Because we embraced the functional nature of JavaScript, we didn't think we would first have to translate everything coming in from the back end into a set of objects and a set of methods on those objects. This also meant we didn't have to sort out the hierarchy of those objects, what the subclasses were, what the base class was, how all those things were structured, and what the relationship was among all those different objects. But that's what you would need to do in a model- and object-based system. So for us it was more like saying, "OK, you're going to hit these three endpoints, and then you're going to merge the data and pop out this other object."

In reality, though, it's not an object but a hash, right? You're consuming adjacent things and then

popping out some other adjacent things. It's almost like you've got a filter that you run a stream through. Data comes through and then pops out the other side—which gets you completely around all that thinking about what the objects are, how they work, and how they interact. So that let us get where we wanted to go a lot faster.

Now that we're through that, we're going back and working on some functions. Right now, they all talk to whatever they want. If Function A wanted to talk to Profile and Companies and Jobs and then merge them, then it would just go ahead and talk to Profiles and Companies and Jobs. And then if Function B wanted to talk to Profiles, Companies and something else, it would just go ahead and do that. But the two functions—A and B—wouldn't be using the same functional interface to talk to Profiles or Companies. The problem is that if there was a bug in the way we're talking to Companies, I would have to go in and fix that in two places. If I wanted to add logging so I could see all the instances where somebody was talking to Companies, things are not centralized such that everything would funnel through nicely. While we're not creating an object layer, we are starting to recognize that, at least for RESTful APIs, we need to create a set of functions that sit in front of each resource type we're looking to communicate with. It's kind of like proxying that interface. That's one of the things we've learned along the way and are now starting to fix by creating this new layer of abstraction.

**TC** You mentioned refactoring the code to introduce additional layers. If you consider the amount of time you're putting in now on the refactoring side, do you think you might still end up taking roughly as much time to create your code base as you would have had you taken some other approach?

**KP** I would guess not because it isn't just about how long it takes to do the coding itself. It's also about every other aspect of the process. For example, when you write your app and then type "Node" to run the app, it literally takes only about 20-100 milliseconds for it to come up. With Ruby, just getting the Rails console to come up sometimes takes more like 15-30 seconds. I'm also not ready to say Node comes up short strictly from a coding standpoint in any event. All the way around, Node is just built thinner, lighter, faster. So every last little step, every single nuance of everything I do every day ends up being faster.

With the more structural languages, you've got to account for things like compile times and build times. Then you end up building essentially hot-swap environments where the environment is running but the IDE is also able to connect with and manipulate the runtime. You pretty much have to do it that way, because it takes so long for the thing to boot up, make your changes, and then boot up again. Node eliminates a whole range of these problems just because it's so fast.

Yes, the refactoring has added to the coding time while it has also taken away from the simplicity we enjoyed initially when we were just slashing and burning through things. But I think the overall efficiency we've been able to achieve because Node is so thin and light more than makes up for any minor amount of refactoring we've had to do.

**TC** If you were about to dive into another project based on Node, would you try to build a little more of your structure initially?

**KP** I don't think that's a Node-specific question. It's more a matter of personal preferences. My own view is that front-end UI code tends to last only about one-and-a-half to two years anyway. Very little of that code lasts as long as five to ten years. The reason for this, I believe, doesn't even have all that much to do with the quality of the code, but instead is driven more by the evolution of

technology and the fact that software developers are encouraged to work in four-year increments. You constantly have new people looking over your code, and I know that whenever I look at anything, even if it's just a table, I'm sure I could build a better one. So I believe the natural tendency is that, every year-and-a-half to two years, a new set of people is going to look over some particular code base and decide they can do a better job.

Given that, it's probably faster just to rewrite a whole chunk of that code base so long as it's modularized—or maybe even rewrite the whole thing—and take your one- or two-month hit for that, than it would be to slowly evolve the code base. So, if I had this project to do over again, I'd probably do it the same way. In any event, I'm more inclined to build a project first, get it out, and then extract a platform later if I can—as opposed to building a perfect platform and all the components upfront and then trying to hook them all up in the right order. I don't think you can possibly know what the right order is until you've actually got something running in production and can see where you're hitting pain points. Then you can tell where you actually need to extract libraries and start doing something about it.

*Once the decision had been made to convert to Node, Prasad's team had to figure out the best approach to implementation and maintenance. Because LinkedIn's mobile services team was largely accustomed to Ruby on Rails, they mimicked parts of their previous Rails structure in setting up the Node.js structure, allowing them to jumpstart the project. The team was small, so Prasad was able to monitor the transition to JavaScript and detect problems quickly.*

*Programming in Node.js is event-driven and so required different approaches. Although a minor amount of refactoring was involved, the team didn't have to create new abstractions or additional functions. Most of what they did was syntactical in nature. They also shed layers of abstraction, which significantly reduced the size of the code base.*

**KN** You say your preference is to jump right into a project and do things the fast way. I wholeheartedly agree. In fact, I'd say my own philosophy is that since you don't really know going into a project what it is you're going to need to optimize for, then what you really ought to be setting out to optimize is your ability to change things.

So I'm really curious as to how you structured your code and what practices you used to let you move fast initially and then continue moving fast as the constraints for your project became clearer.

**KP** A lot of our people at the time were Ruby on Rails developers, so they were familiar with that directory-tree structure and the terminology used in the Rails world. We mimicked that terminology, and that gave us a huge jumpstart. It also helped that Node is pretty barebones, much like Rails, but that also was a little scary because you're not sure how to structure stuff, and there were no guides to tell you how to do it.

A second helpful decision we made was to put all the controllers and views on the client side, while the models were placed on the back end. From a directory-structure standpoint, that meant we wouldn't have any files showing up in our model structure or in our views directory. Although we used the term *controllers*, we were really working more with formatters in the sense that you would get something in, make a couple of requests, format the thing, and then pop it out.

This meant we still had this old Rails structure, only most of the directories were actually empty—and, by the way, an empty directory was actually a good thing. We would leave the directory there

just so people wouldn't try to create things in that directory. It was like a code-review tool that said, "We must be really doing something wrong if we start creating things in that directory." And that's just from our server design pattern of how we wanted to use Node.

Another helpful thing we did initially was to set up a basic testing environment and framework. Rails, as well as Python's Django framework, were very big on TDD (test-driven development) and even BDD (behavior-driven development), where you could essentially write your tests and sequences before filling in all the code. It's a model that works really well when there's a testing framework. We essentially took the testing framework that was already there and slapped it on top of our directory structure using some scripts we'd written to get the two to interact. We started off using Vows, but after three months ended up writing all our own tests in Mocha.

You could say that setting up a testing environment and framework was sort of another code pattern, but it was probably more difficult than it would have been with any other language—especially in terms of dealing with the evented aspect of Node. Even though we ended up refactoring the code, we didn't have to do anything major such as coming up with new abstractions or any additional functions. Most of what we did was syntactical in nature. And then there were a bunch of little functional things—for example, Node's concept that in every callback the first argument should be Error, which makes total sense.

But then you couldn't figure out what the second and third and fourth arguments were. Or do you have only one argument? Or is it five arguments? How do you deal with that, and what is it going to be like after you do the callback? Let's say you now want to change the callback signature. How do you tell everyone who was calling the function and expecting a callback that the signatures have changed? I don't know if that's Node-specific or is common to all JavaScript, but it did take us a while to figure it out.

**KN** How did your team communicate interface boundaries? People who are accustomed to working with types wouldn't want to give away interfaces, since they provide a concrete form of documentation that basically allows one team member to say to another, "Here is my intent. I expect you'll be calling me in this particular way." Even more importantly, if that way of calling turns out not to exist, that probably means it's a use case you haven't considered and so is one that probably won't work.

**KP** I think there are interfaces between libraries inside the code and between the client and the server. For the interfaces between the client and server, we used REST (representational state transfer), and we had a very defined model where we had what we called "view-based models" that were returned by the Node server. We would just document those and say, "Hey, here are the REST interfaces, and here's what we support." That's essentially along the lines of a versioned interface structure. It's classic REST, actually.

Within the code base, we heavily used the module systems. Each REST endpoint has a file that represents all the responses for that endpoint, as well as the public interface for the module mapped to the routes. You can have as many functions as you want, but whatever you have will export out of that module. In that way you actually end up specifying the set of functions you're exposing. That's essentially what we use as our interface.

Then, structurally, we did something really simple: Every function inside the module, whether public or private, was defined in an old-school C style. You might put in a comment that said "private" and then list some of your private functions. Or you could put in a comment that said

"public" and then list all your public functions. This is just like what you used to do in C, right? You had a bunch of functions and you'd put the private ones in one group and the public ones in another group. Then your header file would essentially expose your public interface. All of that happened in a single file. We don't have a header file, but module.exports effectively serves as our header.

**KN** Another important communications consideration has to do with helping a team that's not accustomed to writing JavaScript navigate around some of the minefields they're likely to encounter.

**KP** When we started on this, we were a group of only four people, so I was able to watch every check-in. Whenever I saw anything odd, I would ask about it—not necessarily because I thought it was wrong but because I wanted to understand why we had used that particular pattern. It was easy to get to the bottom of why things had been done in a certain way and figure out what we were going to do next. Now the group is much bigger and we're percolating the nuances of the reasoning behind the choices we made, and that's definitely a lot harder. Now we hold a three- to five-day boot camp whenever a new person starts with the group, which gives us a chance to explain, "Here's how we do this. We know that might seem a little strange, but here's why we do it that way." I think that's probably the best way of exposing those code patterns.

**TC** Which of those code patterns do you think are the most important?

**KP** We ended up using Step as our flow-control library. It's like a super-simple Stem, with two main constructs. We augmented those a little and ended up adding a third. Basically, Step has this concept of a waterfall callback, meaning that you pass into Step an array of functions and then it will call each function in order such that as the first function returns, the second one will be called, and so forth. Step guarantees the order of this sequence, so even if the function is asynchronous—meaning it's going to do something evented—that function will be passed a callback, and it has to call once it's finished doing its thing. Independent of whether each function is synchronous or asynchronous, the waterfall will execute in sequence.

Step also includes a group method and a parallel method. We use the group method very heavily. That means you can give it a group of functions and it will execute all of them in parallel, and then return when all of them are done executing.

One nuance has turned out to be a pretty big deal for us. If we have a group that contains three functions and one of them is broken, Step won't capture the responses for the other two. Instead, it will just call the callback and say, "Sorry, I'm errored."

The negative thing about this environment is that if I call six things and two of them are required while four are optional, I'm not going to mind waiting for all of them with certain timeouts. But if one of those optional things ends up erroring out, then it's going to seem like the whole block errored. That's just not ideal for us. We therefore created a function called GroupKeep that runs through and executes everything, and then if there should be an error, it will hold the error in an array. That way, when the calls go out for callbacks, there will be this array of errors. Based on the position of an error, you can get a pretty good idea of whether it relates to something that's required or something that's optional. That makes it possible to write code that can continue a process wherever necessary.

■■

*The light and thin nature of Node.js appealed to Prasad and his team more than anything else. The extent of code reduction this allowed proved to be huge—from 60,000 lines down to just a couple of thousand.*

*They also now have essentially no frameworks, thus eliminating a lot of extraneous code. Moreover, Node's event-driven approach requires fewer resources and moves more functions to the client side. Finally, it takes a functional approach that sheds layers of abstraction. In sum, this all serves to enable support for huge numbers of users on a wide array of devices in realtime.*

**TC** When you were talking about how quickly you managed to get your initial Node prototypes up and running, it made me wonder whether you had also achieved some code reduction.
**KP** Absolutely. Our Node code base has grown a little from the original version, but it's still on the order of 1,000 to 2,000 lines of code. The Ruby code base we were using previously, in contrast, was in the neighborhood of 60,000 lines of code. The biggest reason for that reduction is that our current code base is essentially framework-free, which means there's just not a whole lot of cruft in there.

The second big reason has to do with the more functional approach we're taking now, as opposed to an object-oriented one, which proved to be an important shift for us. In Ruby, the natural tendency is to create an object that essentially wraps every communication and type. Even though Ruby is actually a functional language, it has a much stronger notion of class and object than does JavaScript. So in our earlier code base we had lots of layers of abstraction and objects that had been created under the guise of greater componentization, refactorability, and reusability. In retrospect, however, we really didn't need most of that.

Another significant reason for the code reduction is the momentum behind the MVC (model-view-controller) model, at least for mobile vs. Web-based systems. Before, we had mostly server-side rendering. Now with the move of templates and views over to the client side—along with rendering, of course—a lot of that code has just gone away. Along with that has come a new trust and belief that the back ends, where the models live, are where the validation and all the other more advanced things are going to happen. That means not having to double-check things, which eliminates another huge chunk of code.
**KN** You indicated earlier that one of the insights that led to your rewrite in Node.js was that you realized you didn't really need a deep understanding of the objects you were manipulating, meaning you didn't need to mutate those objects a lot. Basically, you could just do a lot of merging of hashes. Do you think you could have gotten to that same end with some other language, even Ruby, just by working with the hash-map primitive?
**KP** Probably so, but if you look at Ruby, you see that Rails just has so much extra stuff in it, whereas Node at its base has an HTTP server aspect and a client aspect built into the binary. This means you don't need an HTTP Node module and an HTTP listening module.

So, yes, I suppose if we had eliminated all the object hierarchy and just used the hash structures, we might have been able to use Ruby. But then you would still have to listen to HTTP and turn that into a controller, which just gets you back into adding all these little microlayers. While each microlayer gives you a bunch of code you don't have to write, it also adds a bunch of requirements for stuff you do then have to write so everything will work nicely with your framework.
**TC** If you were to talk to someone else who was about to undertake a similar project, what would you point to and say, "Hey, pay attention to this or you're going to be in trouble"?
**KP** Flow control. Exception handling. And while this isn't really specific to Node, I'd say, "Keep it light. Keep it thin." I think there's a natural tendency for people to say, "Well, I need something that does HTTP, so I'll just find a module that does that," and then another 4,000 lines of code drops

into their environment when all they really need is an HTTP request. Instead, they end up with this super-duper thing that gives them that and a whole bunch of other stuff besides.

Basically, the reason why Node is so fast and so good is that it's light and thin. It barely has anything in it at all, so every little thing you add to it, each additional Node module you want to use with it, comes at a cost.

**KN** For those companies that have already launched projects in Node, what would you say are the three things they might want to add to their ecosystems to make them even stronger?

**KP** First would be a good IDE. InteliJ IDEA is pretty good, but outside of that, I haven't really seen a great IDE and toolset for Node.

Second would be to allow for evolving performance analysis and monitoring. Better operational monitoring for Node would be great, but for now it's essentially a black box unless you put your own monitoring hooks into your code. I'd love to see a lot of the stuff a JMX layer in the Java VM provides. You can get out some really useful information that way.

And then the third thing would be something like New Relic for Node—something that can inspect everything your Node system is doing and actually understand your application so it can provide you with detailed breakdowns of where your bottlenecks and slowdowns are. That would be awesome, actually.

**LOVE IT, HATE IT? LET US KNOW**

feedback@queue.acm.org