JavaScript

by InfoQ queue
Enterprise Software Development Community
eMag Issue 8 - December 2013

Top JavaScript MVC Frameworks

JavaScript front-end codebases have been growing larger and more difficult to maintain. As a way to solve this issue developers have been turning to MVC frameworks which promise increased productivity and maintainable code. As part of the new community-driven research initiative, InfoQ examines the adoption of such frameworks and libraries by developers.

PAGE 4

JavaScript MVC Frameworks vs. Compile to JavaScript Languages

InfoQ examines the adoption of JavaScript MVC frameworks, compared to languages that transcompile to JavaScript, for front-end development.

PAGE 6

Operating Node.js in Production, with Bryan Cantrill

Bryan talks about the challenges of operating Node.js in real production environments and the experiences he had working with it at Joyent. He also talks about DTrace, SmartOS, V8 and compares with other platforms.

PAGE 9

Ember.js - Web Applications Done Right

This article explains the Ember.js application development model and shows how to use it to build your first client-side JavaScript web application with the framework.

PAGE 20

Contents

BLEEDING EDGE HTML5 & JAVASCRIPT @QCON LONDON

MARCH 3-7, 2014

The Browser is becoming the default platform and this track is about its de facto standards: HTML5 and JavaScript. Leading experts in the field will provide valuable insight into how they leverage HTML5 and JavaScript for building innovative products and services. Presentations will cover several issues like motivations, best-practices, security considerations, deployment patterns and more. Most importantly this track is going to focus on the latest developments and trends that are pushing the envelope for the platform and are shaping the future of web development. Warning: This track will take your HTML5 and JavaScript skills to the next level!

SAVE £50 WHEN YOU REGISTER WITH PROMO CODE "JAVASCRIPT"

Jafar Husain,
Technical Lead at Netflix

Aaron Peters & Andy Davies,
Netflix

Caolan McMahon,
Director at Ground Computing

Andrew Betts,
Director of the Financial Times' Labs division

Top JavaScript MVC Frameworks

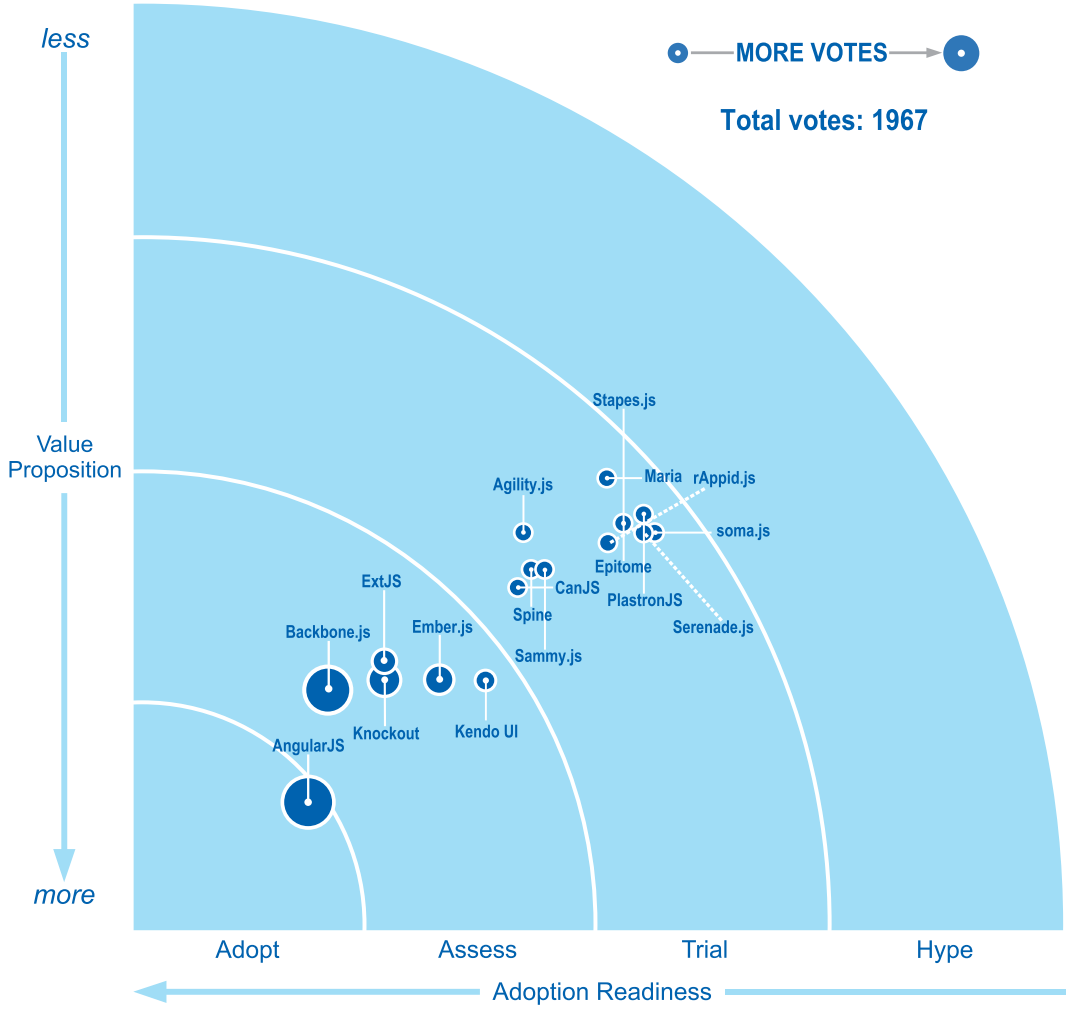
By Dio Synodinos

As the browser ends up executing more and more logic in the browser, JavaScript front-end codebases grow larger and more difficult to maintain. As a way to solve this issue, developers have been turning to MVC frameworks, which promise increased productivity and maintainable code. As part of the new community-driven research initiative, InfoQ examines the adoption of such frameworks and libraries by developers.

- Backbone.js: Provides models with key-value binding and custom events, collections, and connects it all to your existing API over a RESTful JSON interface.
- AngularJS: A toolset based on extending the HTML vocabulary for your application.
- Ember.js: Provides template written in the Handlebars templating language, views, controllers, models, and a router.
- Knockout: Aims to simplify JavaScript UIs by applying the model-view-view model (MVVM) pattern.
- Agility.js: Aims to let developers write maintainable and reusable browser code without the verbose or infrastructural overhead found in other MVC libraries.
- CanJS: Focuses on striking a balance between size, ease of use, safety, speed, and flexibility.
- Spine: A lightweight framework that strives to have the most friendly documentation for any JavaScript framework available.
- Maria: Based on the original MVC flavor as it was used in Smalltalk - aka “the Gang of Four MVC”.
- ExtJS: Amongst other things offers plugin-free charting and modern UI widgets.
- Sammy.js: A small JavaScript framework developed to provide a basic structure for developing JavaScript applications.
- Stapes.js: A tiny framework that aims to be easy to fit in an existing codebase and because of its size is suitable for mobile development.
- Epitome: Epitome is a MVC* (MVP) framework for MooTools.
- soma.js: Tries to help developers to write loosely coupled applications to increase scalability and maintainability.
- PlastronJS: MVC framework for Closure Library and Closure Compiler.
- rAppid.js: Lets you encapsulate complexity into components that can be easily used like HTML elements in your application.
- Serenade.js: Tries to follow the ideas of classical MVC rather than competing frameworks.
- Kendo UI: Combines jQuery-based widgets, an MVVM framework, themes, templates, and more.

READ THIS ARTICLE ONLINE ON InfoQ
http://www.infoq.com/research/top-javascript-mvc-frameworks

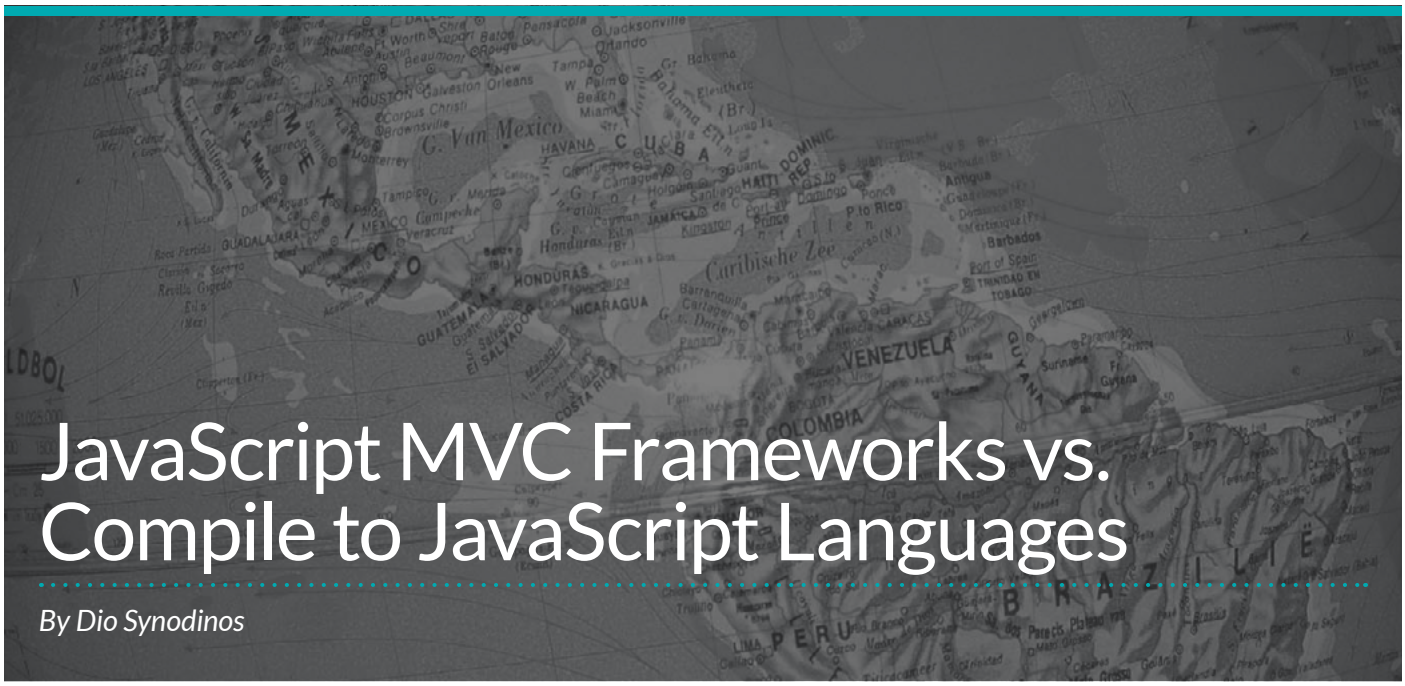
Results



Analysis

Total number of participants: **1969**

Option	Adoption Readiness	Value Proposition	Votes	Details
AngularJS	81%	86%	1063	Heatmap
Backbone.js	79%	74%	936	Heatmap
Knockout	73%	73%	571	Heatmap
Ember.js	67%	73%	498	Heatmap
ExtJS	73%	71%	424	Heatmap
Kendo UI	62%	73%	246	Heatmap
Spine	57%	61%	181	Heatmap
CanJS	58%	63%	151	Heatmap
Sammy.js	56%	61%	145	Heatmap
Agility.js	58%	57%	137	Heatmap
Maria	49%	51%	114	Heatmap
rAppid.js	49%	58%	101	Heatmap
Epitome	47%	56%	100	Heatmap
soma.js	44%	57%	99	Heatmap
Serenade.js	45%	57%	99	Heatmap
Stapes.js	47%	56%	98	Heatmap
PlastronJS	45%	55%	98	Heatmap



JavaScript MVC Frameworks vs. Compile to JavaScript Languages

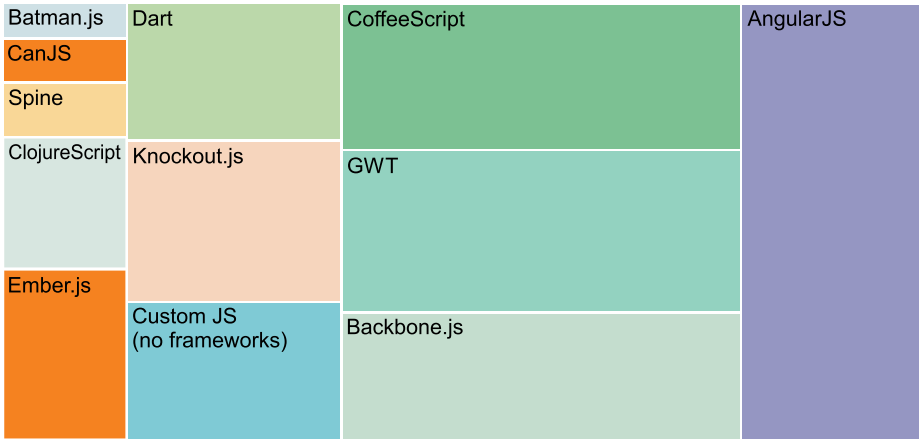
By Dio Synodinos

To avoid spaghetti code on the front end, developers have been turning to MVC frameworks and languages that compile to JavaScript, which promise increased productivity and maintainable code.

As part of a new community-driven research initiative, InfoQ examines the adoption of such frameworks and languages by developers.

Results and Analysis

Total number of participants: **1750**



Option	Dots	Percentage	Standard Deviation
AngularJS	1530	20%	1.64
Backbone.js	1240	16%	1.34
GWT	1231	16%	1.43
CoffeeScript	894	12%	1.12
Custom JS (no frameworks)	704	9%	1.06
Knockout.js	659	9%	0.95
Dart	461	6%	0.79
Ember.js	445	6%	0.76
ClojureScript	300	4%	0.68
Spine	97	1%	0.31
CanJS	79	1%	0.34
Batman.js	61	1%	0.22

READ THIS ARTICLE ONLINE ON InfoQ
http://www.infoq.com/research/js-frameworks-compile

What's Your Next Language on the Javascript Platform?

By Werner Schuster

JavaScript VMs are ubiquitous. Every platform has at least one fast, well-supported JavaScript VM. The JavaScript platform spans servers, desktop/laptops, and mobile - and the embedded space might be next with products like Tessel. Various vendors have been hard at work making existing JavaScript code fast. New projects such as asm.js allow developers to write code that can be compiled efficiently, while WebCL and ParallelJS aim to bring parallelism to JavaScript.

A long list of languages for the JavaScript platform exists - but who actually uses them? Are you planning to use CoffeeScript for your next project or to use ClojureScript to write client logic for your Clojure web application?

InfoQ wants to know: what language will you use for your next JavaScript project?

Languages

Disclaimer: the list of languages was chosen based on the activity and maturity of the respective projects. If the language of your next JavaScript project is not on this list, feel free to tell us all about it in the comments. We also chose to ignore many languages that merely add certain features to JavaScript. See the alt.js website for a full list.

JavaScript-specific languages

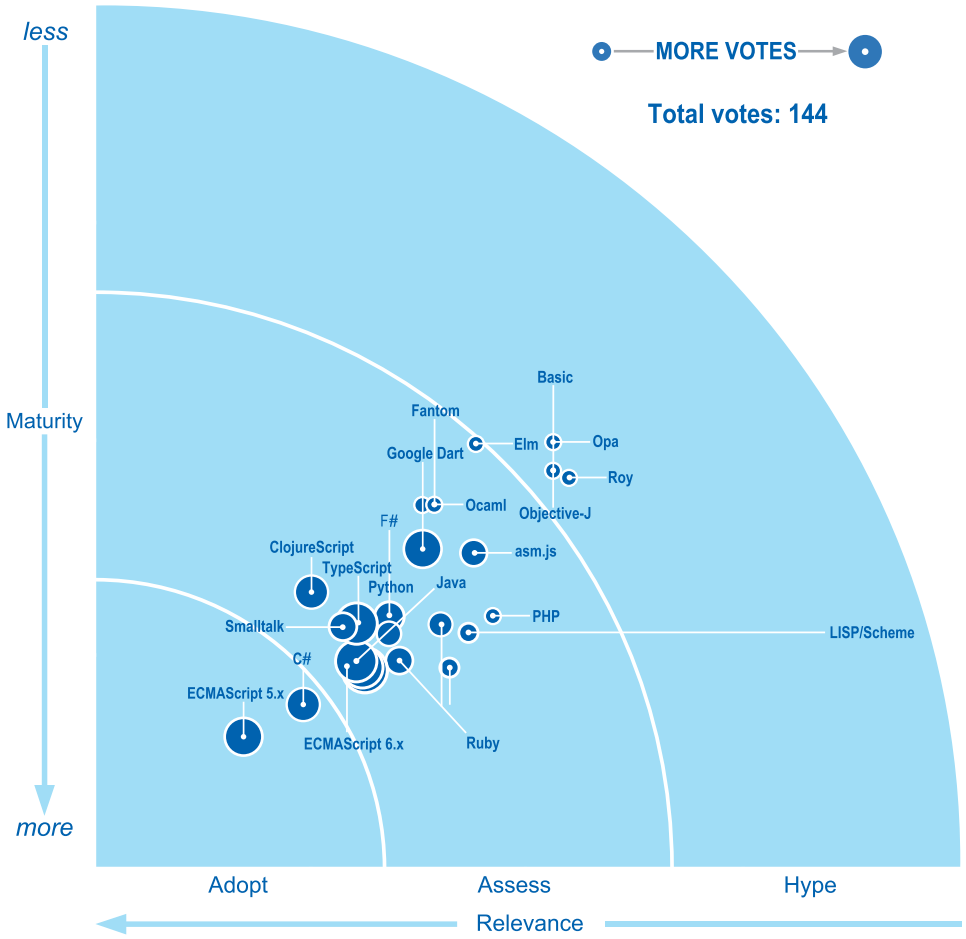
- ECMAScript 5.x
- ECMAScript 6.x - either transpiled or once it's available in Node or browsers
- CoffeeScript
- asm.js - will you write asm.js-compliant code to get maximum performance?
- Elm: Elm site
- Roy
- Objective-J: Objective-J at Cappuccino project
- Opa

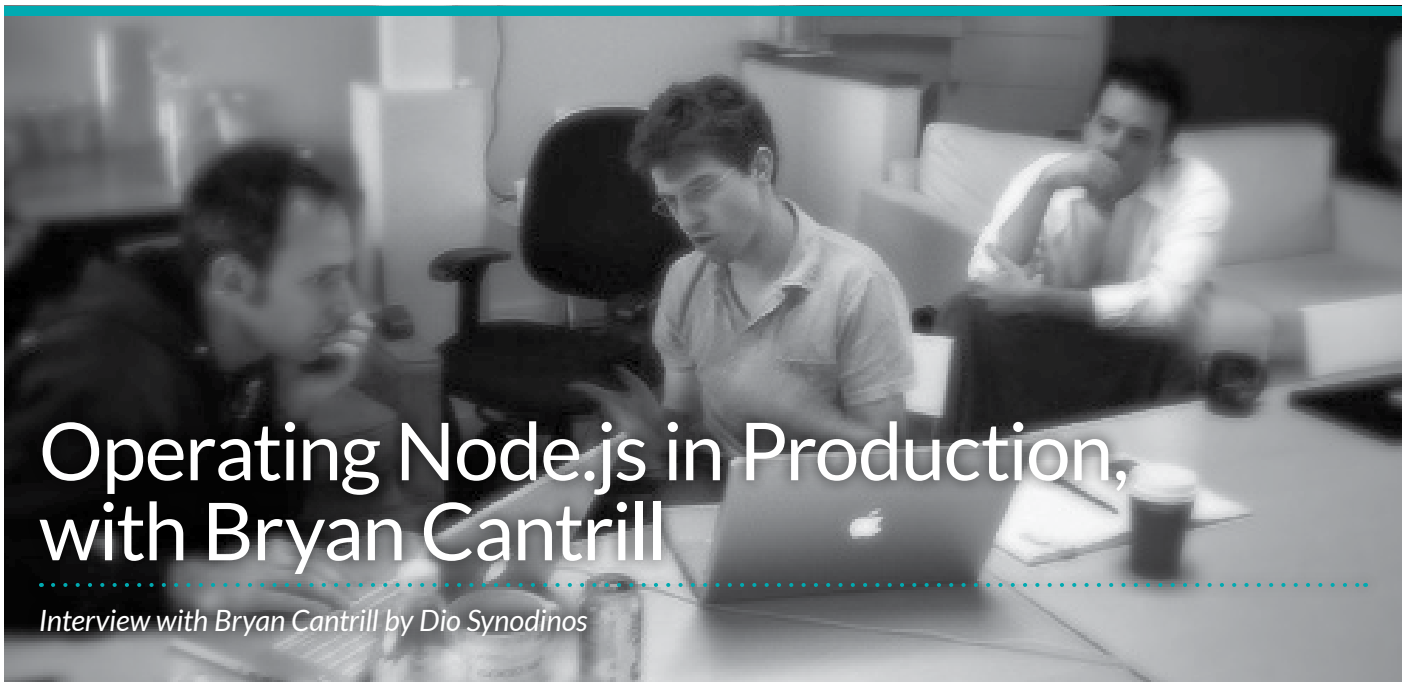
Ports or alternative back ends of non-JavaScript languages

- ClojureScript: ClojureScript on GitHub
- Google Dart: Dart2JS
- Haxe
- C#: SharpKit or other.
- F#: WebSharper, FunScript
- Java: GWT or others
- Python
- Ruby
- Haskell: different backends such as Ghcjs
- Ocaml: js_of_ocaml or others
- LISP or Scheme (excluding Clojure): Parenscript or others
- C/C++ (or other languages with LLVM backends), compiled via Emscripten
- Smalltalk: Amber Smalltalk
- Fantom: Fantom's Javascript backend
- Basic: NS Basic or others
- PHP

READ THIS ARTICLE ONLINE ON InfoQ
http://www.infoq.com/research/languages-on-javascript-platform/

Results and Analysis





Operating Node.js in Production, with Bryan Cantrill

Interview with Bryan Cantrill by Dio Synodinos

We're at QCon San Francisco 2011 with Bryan Cantrill to talk about operating Node.js production systems. Bryan, would you like to introduce yourself?

Sure. I'm Bryan Cantrill. I'm the vice-president of engineering at Joyent; we are a cloud computing start-up here in San Francisco.

What are the main challenges for operating a platform like Node.js? I assume Joyent is probably has the largest deployments of Node.js.

In the words of an old razor commercial, we love Node.js so much we bought the company. We hired Ryan (Dahl, creator of Node.js) because we started to use Node.js and ultimately bought Node.js from him. So Joyent owns Node.js and as such, we are, of course, proponents of it. What's interesting for us is we came to Node by actual using it. We are deploying Node heavily in production not because we own it, not because there's some sort of fiat, but because we're finding it's the right tool for so many jobs. It's really been amazing to me how many things are a good fit for Node, and surprising things.

To give you a concrete example, it is so easy to build a high-performing network service in Node that we're using in most surprising ways. As part of the implementation, the cloud, we have a compute node boot up and it's going to load the image for the OS from a head node. And so the head node is going to serve up DHCP and TFTP, and needs to make some HTTP calls to an API to figure out who the compute node is.

So you end up with this kind of Frankenstein service. It's basically a DHCP server with some TFTP bolted on and some DHCP bolted on and we took IECD HTPD and tried to get it to cooperate and it just was not cooperative. And the engineer who was working on this is like, "You know what? This thing really only wants to be a DHCP server and I think I can actually go write this in Node."

And so, we gave it a shot and three days later, he had a working prototype of a DHCP server that makes necessary DHCP calls and that's our DHCP server. That's what we deployed and that thing got correct so quickly, amazingly quickly, and has had zero issues, effectively, in production.

So we're using Node a lot for those kinds of network placing services over legacy protocols like DHCP. One thing we did recently is LDAP. We're using LDAP internally. And we've implemented LDAP service in Node using arbitrary storage back end. Mark Cavage from Joyent did that and it was funny because Mark confided in me, saying, "You know, I'm looking forward to make this open source." We made it open source and we hope that people will be interested in this.

It was a big story.

It's like LDAP is a jail sentence, no one is actually interested in LDAP. LDAP is something that is inflicted upon them.
Mark asked, "Why just 'we hope people want them'?" And we're trying to talk this out and calm down his expectations, and I answer, "You've got to know this is great work no matter what the rest of the world thinks. Everyone's going to ignore it but this is terrific work."

And sure enough, we put it out there and it became one of the top stories in hacker news. It got a lot of discussion and there's a huge amount of interest for it. And if you look at how long it took Mark to do that work, it is not just that long. I mean, Mark is a terrific engineer and he worked on it very diligently but we're talking like six weeks, eight weeks, 12 weeks to have a high-performing LDAP service - that's pretty amazing. I certainly can't imagine doing that in C. So we're doing that across the board.

Actually many people have tried to do something similar and see they have actually been using the same libraries for more than a decade and copying each other's stuff. What is the Node.js implementation? I think it was pretty much stuff thrown in from scratch.

Yes, it was from scratch. Yes, that's right - you have these libraries that have been dragged along and they kind of imposed complexity and constrained the implementation. So it's been terrific from the speed-of-development perspective. One of the things that we were of course interested to see was how does this thing actually work in production.

Fortunately, some very early production experiences in Node were very positive. We saw much higher load than we thought we would see and nodes seemed to be doing great. We were not seeing runaway memory consumption, which is my concern, just having seen all the difficulty that the GC posed for Java apps as they were first put in production.

I was really braced for some serious GC issues that never really materialized. It's a huge tribute to the V8 guys and a tribute to the fact that the Node model is so straightforward. We have, of course, some early node bugs but they were relatively minor in the grand scheme of things.

The thing that quickly became the issue was not Node itself but when you have a Node program that misbehaves in production, how do you diagnose that? One of the things that I found is that more any other environment I've programmed in, it is easier to get it right in Node. I've been amazed how frequently I've coded something up in Node and said to myself, "Wait a minute, is that it?" It feels like there should be more than that, there should be more work than that, which we haven't experienced as software engineers. Normally, software engineers who are going do something really simple end up with "Oh, my God, this is so excruciating, so complicated. Why is this taking me so long?"

Node was one of the first environments we had that kind of invalid experience like, "Okay, this is very complicated, it's going to take me a very long time…Oh, my God, I'm already done…Wait a minute, something is wrong here!" But still, of course, the program has bugs, so how to actually deal with these things in production? The one aspect that we're having a very hard time with was not programs that died fatally, that blow up with exception and you've generally got the stack phrase there. You can make some sort of progress.

Where things were brutal - and this is unusual - were the other programs spinning out of control. It's unusual because you don't go compute-bound in Node very frequently. So you really do have to have a programmatic error in order to see this, in order to see a program that's spinning out of control. We saw one of these before we went into production with a big release in the spring.

We were doing our testing and one of our demons was spinning out of control and it was very frustrating. You're trying to watch the thing and it is just totally opaque. You know, one of the great things about working at Joyent is that Ryan's down the hall. I kind of shout, "Hey, Ryan, get out here! Help us debug this thing." And all three of us stare and try to debug the goddamn thing and we don't know where it is, we don't know what it's doing. It's very hard to get context... We didn't debug it, didn't find it, which to me is like the worst thing imaginable.

I had the bug in front of me and I got no idea what it was. It's terrifying, and that, to me, is an unresolved disaster. And it's a disaster waiting to happen. Actually, we put a bet on how frequently we were going to see this in production. I thought we were going to see it very frequently. Fortunately, we didn't. We still work with the same production and we never saw it again but we knew it was out there. We had this bug that was out there.

Meanwhile, we were thinking, "Hey, we've got to go tackle this problem because we know this bug and other bugs are out there." We have got to tackle the problem and in particular, we got to tackle the problem of being able to take Node state and V8 state and be able to understand what's going on in the program. As systems guys, we needed post-mortem debuggability in Node. I need to be able to walk up to a Node process, take a core with gcore, take a snapshot of each state, and then go debug it.

It's a really hard problem and it's a hard problem because you take up a core dump with Node, you go to the tracks trace, and it's all meaningless, right? Fortunately, we felt strongly about this and we put some really intense effort into this. Dave Pacheco, on our team at Joyent, was very excited to go tackle this problem in part because it was his software that had the bug. Actually, we didn't know. It was either his code or my code that had the bug.

Dave did all this work, terrific work that I'm very excited to be demonstrating today here at QCon, and we're able to take a core dump now and get a stack trace....
We solved this problem about a week and a half ago. Of course, I'm demonstrating it for customers when it happened and I'm like, "It doesn't seem to be working." I'm sure you've had demos go out on you; it's such a fight-or-flight reaction: "Oh, I'm doing something wrong. Oh, God, no, it's not me. Oh, God, the demo is not working. Oh, it's happening right now. My demo is failing." I feel the adrenaline and the cold sweat and did kind of hand-waving and go demo with another machine.

Meanwhile, we have back-up machines, so it's spinning again but now, after David's done his work, we did a gcore on that thing. He looked at the new stuff of the core dump and we found the bug and sure enough, it was a stupid bug. We are passing an object into routine that has all of its parameters and MIN and MAX are both set on this same number that my routine was never expecting and didn't assert that these things were not different. So my code should have blown up but it didn't; it just went into an infinite loop. Stupid bug.

With this technology, it's immediately debuggable. Without this technology? It doesn't matter who you are, you are not debugging it. And I know that from experience because I imported a lot of resources in trying to debug this thing. Dave and Ryan and I really tried to debug this thing earlier and now we can do it. So we're really excited about that. I think that that kind of technology is the difference between interesting technology and technology that's fun to play with and really rock-solid production-ready technology.

I think Node is actually far more production-ready than other environments that have been around a lot longer. I would deploy Node way before I deployed Ruby. I would deploy Node way before I deployed Python and I know that people are having aneurysms all over as I'm inciting religious wars, but from my practical experience, we can now determine more about what's going on for a Node program than any of these other services.

I think the only environment that has made as much progress on debuggability is Java. Of course, Java has got a 15-year lead. We think that we're more focused on it than Sun was on this particular problem with Java. We are going to continue to flesh it out and continue to make it better and our commitment is really to make Node the best production environment second only to C, of course - the language that God intended. But everything we do is either in C or in Node. So, those two systems we are bound to making absolutely debuggable in production.

Talking about systems that are rock-solid in production, how do you manage the rapid evolution of the Node.js platform? The new versions, the API lacking concrete things like dependencies.... Many of the plugins and the modules that people use have weird dependencies in production that are issues.

Yes and that's been a challenge and we have kind of known. We keep our dependencies under control; what we've been doing is pointing dependencies in the sub repos. So the repo that has the service has got all its dependencies sitting right there and when you want to go pull in the latest in Node, which may require you to update three of these things and may require you to go debug nodes now not working, then at least you got that whole thing as a unit.

So what we have and had is production problems as a result of that; it has slowed down development in certain regards. Fortunately, I would say, more so six months ago, eight months ago, nine months ago, than it is today. Things are settling. A year, a year and a half ago, you go to look for anything and there were five modules that did it and it was very unclear and also did poorly.

That was so demoralizing. I said, "Okay, I need an options parser. Oh, there are five options parsers." I can see after 10 minutes that all of them are wrong and I actually need to go write a sixth. I said, "Great." That was more a year ago or a year a half ago than it is today. these things are now settling and we're seeing more canonical implementation on a lot of this stuff.

I suppose you're using NPM to manage.

Isaac is very fond of saying that if you set out to write a package manager, you might, just might, if you're lucky, end up with a really good options parser as the primary artifact. And you know options are bad examples. I think we've probably used three or four different ones under Joyent's roof. I would try to be flexible in some of that stuff.

The nice thing about Node is it's pretty lightweight and we're not seeing the kind of the Ruby gems kind of monkey-patching, metastasized monstrosities where you're not able to tweak the environment without breaking everything, not yet anyway. I don't think we're going to see that because the ethos is different. The Node community very much to me has the Unix ethos. It's very spartan in terms of interface and really driving towards simple, composable systems and less concerned about - the way I read it, anyway - superficial simplicity and more concerned about things that are deeper simplicity.

Unix is revolutionary because it's a very deep kind of simplicity. It was so deep that we don't recognize now what operating systems used to look like. I think Node is the same way so I think that it has not been kind of the convoluted hairball that it could be.

You mentioned a little while ago about the production issues that Node has. Most of the systems have some inherent issues. With JVM, you usually have memory issues. With Apache plus module environment, you have memory and connection issues. What are the inherent issues with Node.js?

You know, I feel like a jackass saying this but I really can't (point out an) issue. It has been so solid for us in so many ways across so many services. It's hard for me to say, "Oh yes, we'll be running Node so your CPU consumption is going to be pretty high." Never saw it.

We cannot get a node app that's not spinning in infinite loop to even crack CPU; The light CPU load is just amazing. The DRAM footprint is unbelievably small. I mean, we got a free developer on the cloud; we run those 128-MB zones, 128-MB virtual OS containers. You get 128 MB and yet people got lots and lots of Node apps on there because Node actually can do a lot in 128 MB.

So it's not DRAM-limited because of its even-oriented architecture. We're not really seeing complicated threading problems because obviously it's not threaded. We're not seeing IO problems and so on because you don't corked on IO. You're doing IO but you're still servicing requests.

I won't blame anyone for thinking, "Oh, Cantrill went

to the marketing department. I thought this guy was actually like in the trenches, what's going on?" But it's true. For us, the biggest issue has been when you have a Node bug, which fortunately has been very rare. A Node bug that spins on CPU has been historically undiagnosable and that's why we moved on the work that we've done to address that.

Would you like to give us like a short overview about the process of debugging something like this?

For debugging like an infinite loop?

Yes, I mean the tools and the process for someone to debug this.

We're very much using the tool set that we have in the operating system. We have our own operating system in Joyent called SmartOS and so when you are running on no.de you're running in SmartOS. We have a debugger built into SmartOS called MDB and into that debugger we have built modules that understand V8.

And Dave Pacheco has got a terrific blog entry on outlining all the things he's done. We're still working on this and rolling it out so people can look for it out there in the coming weeks and months. To just kind of warn people in advance: it's going to be only a SmartOS feature not because it's not open source, not because we don't want to see it in platforms, but because it builds on effectively 15 years of foundation that we have poured into the debuggability of the system.

So you can't build this kind of stuff into GDB. That's not how GDB is architected. MDB is architected to do this and MDB is itself built on foundational elements very deep in the system that are just not readily portable. We would welcome anyone who wants to port it to different system. We will help them out but it's rocky. So for now, this kind of debuggability is really only going to be on SmartOS.

As one of the creators of DTrace and Node DTrace, would you like to tell us how and why a dtraceable Node.js matters? Obviously it matters if you have poured work in it.

Actually I can be very inverted about that kind of stuff. Again, maybe you'd be right not to believe me when I say that I absolutely don't have a problem. I don't tend to kind of invent things and then use them for their own sake. I tend to be inverted in that I tend to invent the things that I need to solve the problems that I have right in front of me.

If we didn't need Node support for DTrace, if I don't need it, then who cares? But we do need it. That's the reason it's important, because we need it. We need it when these things misbehave and they misbehave with this one pathology of not interacting with the outside system at all and kind of going into the black hole. We've got to be able to understand what that thing is doing and it's not just trip or Node. Actually, if you want to talk about production issues we have just so you know that I'm not in the marketing department. Let's talk about Erlang and RabbitMQ. We can talk about production issues where we did have Rabbit losing his mind off in Erlang Never Never Land - and I mean you got no idea what's going on. Now, the Erlang community is also working on DTrace support which is great. Obviously, we're very excited about that because we need that to be able to debug those kinds of production problems.

It comes in DTrace Node. It's not just, "Hey, it's nice to have," it's a "Hey, if you got this kind of problem in production, you got a production Node system that loses its mind, then you need this technology, you need DTrace, you need MDB to be able to figure out what the hell is going on."

You have talked about deeper support of DTrace within V8 in the past, would you like to elaborate on this?

Yes, we've done a couple of things. One is that we have added some probes to Node itself around points of semantic relevance. There's a terrific module that I'm talking about in my talk. Chris Andrews put together a module that allows a developer to define their own DTrace probes. Mark Cavage at Joyent did this for his old apps, His old apps at Node.js define DTrace probes so that you can walk up to that thing and instrument it.

So not only did Mark develop effectively the most scalable LDAP server, it also is observable because he's leveraging this terrific work. Beyond that, the work that David has done recently lets you take

a stack trace from the kernel and translate it into actual JavaScript frames. That's what we will be demonstrating today. The implementation details are incredibly hairy for that, just unspeakably hairy, the tree of knowledge of good and evil to understand how all of that stuff work, but that actually works.

What I don't know if we'll ever achieve is the ability that you have with DTrace on a system that is in a native environment to instrument an arbitrary function by only changing effectively that function. We don't have the ability to do that in any dynamic environment today; it's an extremely hard problem. I don't know if we'll ever get there, but we hope we're going to get there.

I think we may get there kind of incrementally and it may be that the kinds of things we've done now, the probes, the DTrace and so on, but that's enough to give so much even though you don't have otherwise that to solve that other problem which is 10 times as much work as that software together, it's just not worth it, I don't know, time will tell.

Do you have any information about what the Google guys are doing with V8 with regards to instrumentation?

The VM guys really want to focus on the tools for the developer whereas we focus on production uses. They're very supportive of the work we've been doing and we have great conversations with them but the reality is, it will require them and us to do a lot of work to make V8 truly arbitrarily instrumentable and it's a huge challenge. I mean, they're receptive to it, we're receptive to it in the abstract but I don't know if that works. Again, we've done all other stuff, kind of a periphery, that may be enough.

Would you like to give us a comparison between operating Node.js on your own infrastructure and using a platform like Joyent's? What would be the main differences and the experience for someone who operates Node.js?

Again, I don't mean to sound to like too much a salesman here for Joyent but the reality is, that to stand out Node in production and be able to debug it and understand it, you need these underlying technologies, you need DTrace, you need MDB, you need the stuff we built on it.

So you want to be on SmartOS if you're going to build it yourself. SmartOS is open source, so anyone can go and install that and play around with it. You got DTrace, ZFS, you've got a lot of goodness in there, we've put recorded KVM on SmartOS - there's a lot of reasons why. SmartOS is open source, you don't need to worry about becoming encumbered with Joyent in that regard.

If you were to tell me right now that Joyent has closed its doors and I've got to venture forth in the world and I would almost certainly be developing a Node because that's a great environment for some of the systems tasks, I would absolutely be standing upon SmartOS, no question.

So, you got to set it up on SmartOS. Then the question is, do you want to manage that yourself? Obviously, Joyent would be happy to sell you the orchestration software to manage that. Do you want to go with the cloud provider for that? I think the decisions will kind of fall off from whether you set it up with Joyent, you set it up with a different cloud provider, or you set it up yourself.

To me, the constraint is that you've got to have the tooling to understand how and why this behaves in production. When it misbehaves, which is not frequent necessarily, but when it does, they are really debilitating problems.

How do you see operations involving in general as new kind of applications become popular like real-time stuff and data-intensive applications?

As we go to deploy things that are more real-time into production, we have to be able to understand the behavior of those applications better when in a real-time system. There is a real-time system and a non-real-time system, and timeliness is correctness for real-time system. If it's late, it's wrong.

And if that's the case then you need to be able to know if it's on time or not. And in order to know that, you've got to be able to dynamically instrument the

system. So, I think one of the ways that operations does need to change is that today, largely, operations has a very metric focus on how many of these things that are due for second? And how does that compare historically?

Well, the number of beats per second is not actually that relevant in the real-time system. What's relevant is how long it took. And it's a horror question because it requires you to actually instrument the system to answer it but that's what I think the operations folks need to embrace.

About the Interviewee

Bryan Cantrill is VP of engineering at Joyent, where he has led development of Joyent's SmartOS and SmartDataCenter products. Previously a distinguished engineer at Sun Microsystems, Bryan has spent over 15 years working on system software, from the guts of the kernel to client-code on the browser and much in between.

WATCH THIS INTERVIEW ONLINE ON InfoQ
http://www.infoq.com/interviews/operating-nodejs-production-bryan-cantrill

Contrasting Backbone and Angular

By Victor Savkin

Contrasting ideas and tools is a great way to understand them better. In this article I will go down the list of the things we have to deal with day to day when building web applications, and show how Backbone and Angular can help with each of them.

What We Are Trying to Solve

Most of the things we do as web developers fall into one of the following categories:

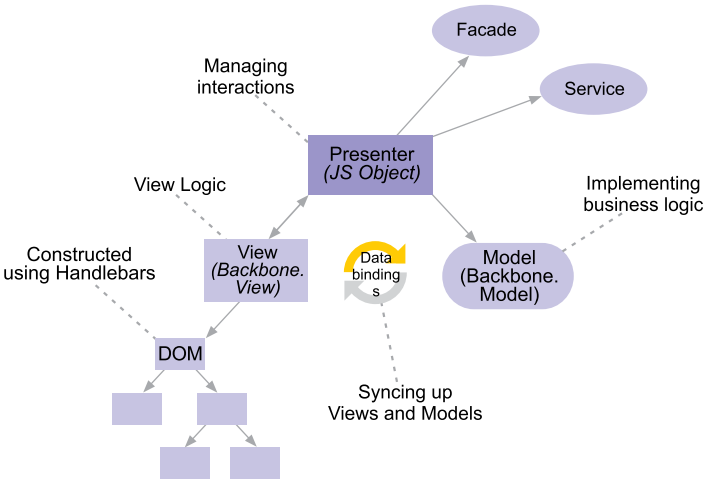
- Implementing business logic
- Constructing the DOM
- Implementing view logic (declarative and imperative)
- Syncing up the model and view
- Managing complex UI interactions
- Managing state and routing
- Creating and wiring up components

Not surprisingly, most client-side frameworks address these in one way or another.

Backbone

Let's start by looking at what Backbone gives us to solve these problems.

Business Logic	Backbone Models and Collections
Constructing DOM	Handlebars
Declarative View Logic	Backbone Views
Imperative View Logic	Backbone Views
Sync up Views and Models	StickIt
Managing UI Interactions	JS Objects or Marionette Controllers
Managing State and Routing	Backbone.Router
Creating and Wiring Up Components	Manually



When I Say Backbone...

Comparing vanilla Backbone with Angular would not be fair. So by Backbone I actually mean Backbone + Marionette + add-ons.

Business Logic

A large chunk of the business logic of the application goes into Backbone models and collections. Often, these objects correspond to resources on the back end. On top of that, they are used for backing up views.

Having to extend Backbone.Model and Backbone.Collection adds quite a bit of complexity.

First, it separates all domain objects into POJOs and Backbone models. POJOs are used when rendering templates and talking to the server. Backbone models are used when observable properties are needed (e.g., setting up data bindings).

Second, it promotes mutability. Since Backbone does not support observing functions, every computed property has to be reset when any of the source properties changes. This adds a lot of accidental complexity, which results in code that is hard to understand and test. On top of that, all the dependencies have to be explicitly specified in the form of on("change:sourceProperty, this. recalculateComputedProperty).

Constructing the DOM

Backbone uses template engines to construct the DOM. In theory, you can plug in any engine you want. In practice, though, Mustache and Handlebars are the ones that usually get used for large applications. As a result, templates in Backbone are often logic-less and string-based, but they do not have to be.

View Logic

The idea of dividing the view logic into imperative and declarative is an old one (it goes back to the original MVC pattern). Event-handling configuration and data bindings are declarative. Event handling itself, on the other hand, is imperative. Backbone does not draw a strict line between the two. Both go into Backbone.View.

Syncing the Model and View

Due to the minimalist nature of Backbone, there is no built-in support for data bindings. That is not an issue for small projects, where the view can be made responsible for syncing the model and the DOM. It, however, can easily get out of control when the application grows.

There are several add-ons available (like Backbone. StickIt) that help unload this burden, so you can focus on complex interactions rather than the trivial model-view synchronization. Most of these add-ons are configured using plain JavaScript, which enables building abstractions on top of them to match the needs of your application.

The downside of using data bindings in Backbone is that they depend on observable properties, whereas template engines use POJOs. Having these two ways of working with the DOM often results in code duplication.

Managing Complex UI Interactions

All UI interactions can be split into simple (managed using the Observer Synchronization) and complex (where the Flow Synchronization is required).

As mentioned above, Backbone.View handles simple interactions with data bindings and event handlers. Since Backbone does not have any prescribed solutions for orchestrating complex UI interactions, you are free to choose the one that fits your application the best. Some use Backbone.View, but I recommend against that. Backbone.View already does too much. Supervising Presenter is the pattern I tend to use for managing complex interactions.

Managing State and Routing

Backbone comes with a simple implementation of the router. It provides no support for managing view and the application state. Everything has to be done manually. That is why in practice other libraries (e.g., router.js) are often used instead of the built-in router.

Creating and Wiring Components

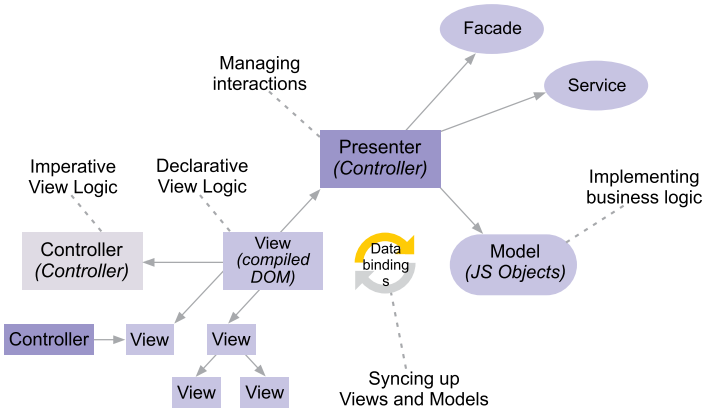
In Backbone, you have the freedom to create and wire components in the way that fits your application best. The downside is the amount of boilerplate you have to write, in addition to the discipline required to keep the code well-organized.

Angular

Now, let's contrast it with how Angular approaches the same problems.

Business Logic	JS objects
Constructing DOM	Directives
Declarative View Logic	Directives
Imperative View Logic	Controllers
Sync up Views and Models	Built-in mechanism
Managing UI Interactions	Controllers
Managing State and Routing	AngularUI Router
Creating and Wiring Up Components	Dependency Injection

For visual people:



Business Logic

Since Angular does not use observable properties, it does not restrict you when it comes to implementing the model. There is no class to extend and no interface to comply with. You are free to use whatever you want (including existing Backbone models). In practice, most developers use plain old JavaScript objects, which yields the following benefits:

* All domain objects are framework-agnostic, which makes reusing them across applications easier.
* They are close to the data that is being sent over the wire, which simplifies the client-server communication.
* They are used to render views, so there is no need to implement toJSON.
* Computed properties are modeled as functions.

The Template and View

The template in Angular is a piece of the DOM before it gets compiled. During the compilation, Angular transforms that DOM subtree and attaches some JavaScript to it. The result of this compilation is another DOM subtree, which is the view. In other words, you do not create the view yourself. Angular does it by compiling the template.

Constructing the DOM

Backbone clearly separates the construction of the DOM from the view logic. The first one is done using a template engine and the second via data bindings and imperative DOM updates. Angular, on the other hand, does not separate the two. It uses the same mechanism, directives, to construct the DOM and define declarative-view behavior.

View Logic

Angular, however, draws a line between declarative and imperative view logic. The view does the former and the controller does the latter.

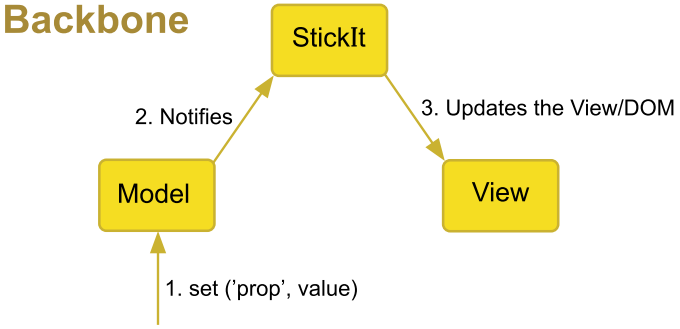
This separation may seem arbitrary, but it is actually quite important.

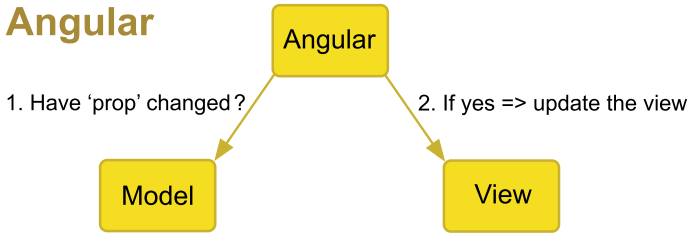
It clearly identifies what has to be unit tested. The declarative logic encoded in the template (such as using ng-repeat) does not need tests. Writing tests for the controller, on the other hand, is usually a good idea.

Note that all dependencies go in one direction: from the view to the controller. Thus, the controller is unaware of the view or the DOM. It enables code reuse and simplifies unit testing. Contrast this with Backbone.View that often manipulates DOM nodes and re-renders large parts of it using template engines.

Syncing the Model and View

Angular has a built-in support for data bindings. In contrast to most client-side frameworks, it does not rely on observable properties and instead uses dirty checking.





The Angular dirty-checking approach has some nice properties:

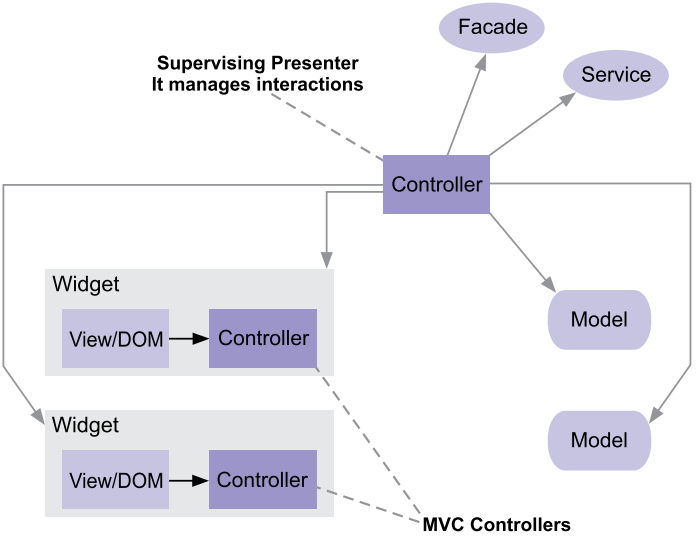
* The model is oblivious of the fact that it is being observed.
* There is no need to specify dependencies between observable properties.
* Functions are observable as well.

But it also has some drawbacks:

* When integrating third-party components or libraries, you have to make sure that Angular sees the changes those make to your models.
* In some situations, it can have a negative effect on performance.

Managing Complex UI Interactions

As mentioned, the controller is responsible for implementing the imperative logic of UI elements. On top of that, it can be used as a Supervising Presenter to coordinate complex UI interactions.



Managing State and Routing

Similar to Backbone's, the built-in router in Angular is basic and insufficient for building real applications. Thankfully, there is the AngularUI Router project. It manages the application state and views and supports nesting. In other words, it does everything you would expect from the router. But you are not limited to it. As with Backbone, you can use other routing libraries (e.g. router.js) as well.

Creating and Wiring Components

Angular has an IoC container that, like dependency injection in general, forces you to write modular code. It improves reuse and testability, and helps get rid of a lot of boilerplate. The downside is increased complexity and reduced control over how components get created.

Summing Up

This short overview looks at how Backbone and Angular address the main problems we deal with everyday when building web applications. The two frameworks have very different solutions to some of these problems. Backbone gives you more options when it comes to rendering templates, setting up data bindings, or wiring up components. Angular, on the other hand, has prescribed solutions for these problems, but is less opinionated when it comes to how you build your models or controllers.

About the Author

Victor Savkin is a software engineer at Nulogy. He is interested in functional programming, the web platform, and domain-driven design. He works on large applications written in JavaScript. Being a language nerd, he spends a lot of his time playing with Smalltalk, JS, Dart, Scala, Haskell, Clojure, and Ioke. He blogs about building large applications in Ruby and JS at victorsavkin.com. You can follow Victor on Twitter @victorsavkin.

READ THIS ARTICLE ONLINE ON InfoQ
http://www.infoq.com/articles/backbone-vs-angular

Ember.js: Web Applications Done Right

By Joachim Haagen Skeie

Introduction

When InfoQ published my article "Ember.js: Rich Web Applications Done Right" last year, the code was based on version 0.9.4 of Ember.js, and Ember.js itself was a fairly young project.
The framework has come a long way since then and it's time to update the article with the state of the art in Ember.js development practices and technologies, in particular the Ember Router.
The Ember.js API has stabilized a lot with the release of the Ember.js 1.0 Release Candidate 1. This article is based on a fresh build of the master branch (March 24, 2013) both for Ember.js and for Ember Data.

My book "Ember.js in Action" will be published by Manning Publications in the second half of this year. Manning has an early-access release that will let you get the first four chapters right away, with new chapters coming out towards the book's release.

So how has the Ember.js landscape improved? Is Ember.js still the framework for developing single-page web applications the right way?
These are a couple of questions that I aim to answer though this article. We have a lot of ground to cover, so let's get started! The source code for the application we are building can be found at GitHub.

What are we building?

We will be building a simple photo-album application which will present users with a row of photo thumbnails along the bottom edge of the page.
When the user selects one of these photographs, the URL will update to reflect this while the application will display the selected photograph above the thumbnails.
We will also be adding a simple slideshow function that will transition to the next photograph in the list every four seconds.
The finished application looks like figure 1 below.

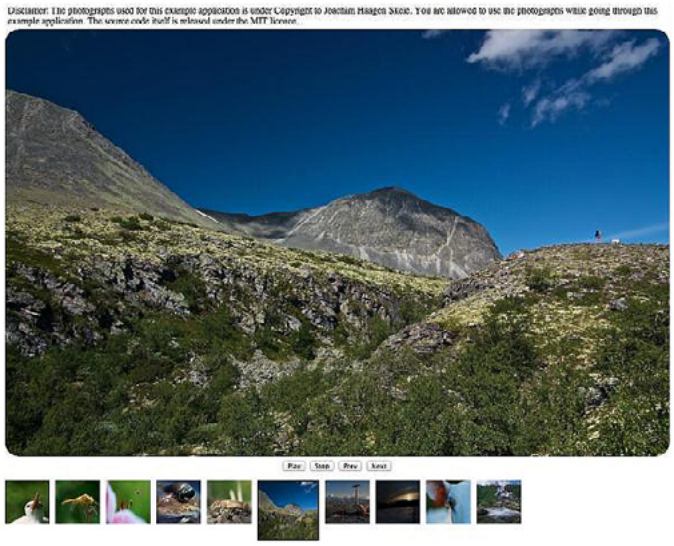


Figure 1 – The finished application

We will start with a clean slate and build the features that we require from our application piece by piece.

Project structure and setup

The finished source code of this application is available via GitHub. The source code for the application is located inside the site directory. You

need to start the application so that it will be available without a context path. There are many ways to do this, and a few simple alternatives are:

- Use the asdf-gem to host the current directory. For further instructions, see GitHub. Once the gem is installed issue the command "asdf" from the site directory
- Any other webserver that will host your application as http://localhost/

The example project have the structure as shown below in figure 2.

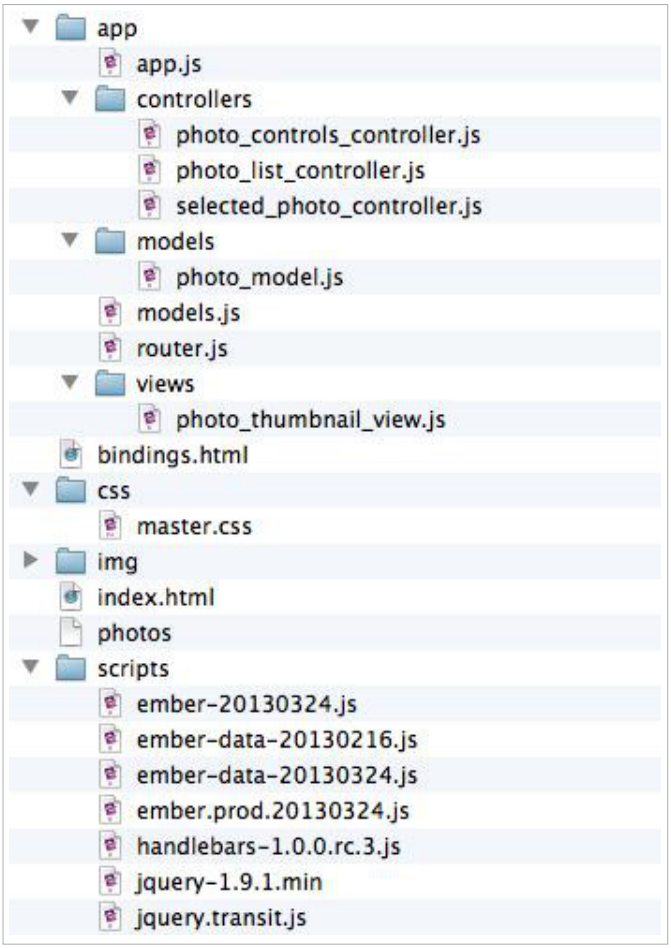


Figure 2 – The Project Structure

All of the external libraries are located in the site/scripts directory. The required libraries are

- Ember.js, built on March 24, 2013
- Ember Data, built on the February 16, 2013
- Handlebars.js 1.0 RC 3
- JQuery 1.9.1

The img directory contains the 10 image files that we will use in our photo album. There is a single CSS file called master.css inside the css directory.

All of our application logic will be split into separate files inside the app directory. We will start out with the single app.js file. The article will tell you whenever you need to add an additional file to the application.

Short introduction to bindings

In Ember.js, bindings are used to synchronize the values of a variable between objects. Consider code in listing 1 below:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">

<html lang="en">
<head>
     <title>Ember.js Example Bindings</title>
     <link rel="stylesheet" href="css/master.css"
type="text/css" charset="utf-8">
     <script src="scripts/jquery-1.9.1.min.js"
type="text/javascript" charset="utf-8"></script>
     <script src="scripts/handlebars-
1.0.0.rc.3.js" type="text/javascript"
charset="utf-8"></script>
     <script src="scripts/ember.prod.20130324.js"
type="text/javascript" charset="utf-8"></script>
     <script type="text/javascript">
          BindingsExample = Ember.Application.
create();

          BindingsExample.person = Ember.Object.
create({
               name: 'Joachim Haagen Skeie'
          });

          BindingsExample.car = Ember.Object.
create({
               ownerBinding: 'BindingsExample.
person.name'
          });
     </script>

     <script type="text/x-handlebars">
          <div id="mainArea">{{BindingsExample.
car.owner}}</div>
     </script>

</head>
<body bgcolor="#555154">

</body>
</html>
```

Listing 1 - Simple bindings example

In the code, we start by loading the CSS file before it includes the JavaScript libraries jQuery, Handlebars, and Ember.js. The application itself is defined in full from lines 10 to 25. It starts by defining a namespace for the application, BindingsExample in this case. The application defines two objects: BindingsExample.person and BindingsExample.car. Here, we are following the Ember.js naming convention, by starting instantiated objects with a lower-case letter.

The name property for the BindingsExample.person object is set to the string "Joachim Haagen Skeie". Looking at the car object, you will notice that it has one property called ownerBinding. Because the property name ends in "Binding" Ember.js will automatically create an "owner" property for you that is bound to the contents of another property, in this case the BindingsExample.person.name property.

If you load this HTML file in the browser, it will simply print out the value of the BindingsExample.car.owner property, in this case being "Joachim Haagen Skeie". The beauty of this approach is that whenever the BindingsExample.person.name property changes, the content on the website will also be updated. Try typing the following into your browser's JavaScript console and watch the contents of the displayed website change with it:

```
BindingsExample.person.set('name', 'Some random dude')
```

The contents of the displayed webpage will now be "Some random dude". With that in mind, it is time to start bootstrapping our example application.

Bootstrapping your Ember.js application

To get a feel for what we are building in this example, you can have a look at the finished application.

To get started, create an empty index.html file and add the directories app, css, img, and scripts. Now, create or copy the following files:

- Inside your app directory, create a file named app.js.
- Download the master.css file and place it inside the css directory.
- Place the scripts files from Github in the scripts directory.

Let's get started with our index.html file. We will start out with a rather empty file that only references our scripts and CSS files as shown in listing 2.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
<head>
    <meta http-equiv="Content-Type"
content="text/html; charset=utf-8">
    <meta name="viewport"
content="width=device-width, initial-scale=1.0,
maximum-scale=1.0">

    <title>Ember.js Example Application</title>
    <link rel="stylesheet" href="css/master.css"
type="text/css" charset="utf-8">
    <meta name="author" content="Joachim Haagen
Skeie">
    <script src="scripts/jquery-1.9.1.min.js"
type="text/javascript" charset="utf-8"></script>
    <script src="scripts/handlebars-1.0.0.rc.3.js"
type="text/javascript" charset="utf-8"></script>
    <script src="scripts/ember.prod.20130324.js"
type="text/javascript" charset="utf-8"></script>
    <script src="scripts/ember-data-20130216.js"
type="text/javascript" charset="utf-8"></script>
    <script src="app/app.js" type="text/
javascript" charset="utf-8"></script>

    <script type="text/x-handlebars">

    </script>
</head>
<body bgcolor="#555154">

</body>
</html>
```

Listing 2 – Our initial index.html file

The first thing we need to do is to create a namespace for our application, which we will place inside app.js. I have decided to create the namespace EME (EMber-Example). Add the following to your app.js file:

```
EME = Ember.Application.create({});
```

Listing 3 – Ember example namespace (EME)

Before we move on to defining the object model for our application, we should add content to the application's CSS file. I would like the application to have a white background with rounded borders, and I would like it to fill the entire screen. Add the following to your css/master.css file:

```
#mainArea {
    border-radius: 25px;
    -moz-bottomleft: 25px;
    padding: 15px;
    margin-top: 5px;
    background: #fff;
    text-align: left;
    position: absolute;
    top: 5px;
    left: 5px;
    right: 5px;
    bottom: 5px;
    z-index:1;
}
```

Listing 4 – Adding the application's mainArea

In order to render a div element with the id "mainArea", we will extend the application template by adding a element to a script tag of type "text/x-handlebars" inside index.html. Listing 5 below shows the end result.

```
<script type="text/x-handlebars">
    <div id="mainArea"></div>
</script>
```

Listing 5 – Adding a div element to hold our application

You can now refresh your page. You should see an empty white area with rounded corners covering most of the screen. With that in place, it's time to start defining a model for our photographs.

Defining your object model with Ember Data

At this point, you need to add Ember Data to your list of scripts in the index.html file, if you haven't done so already.

To use Ember.js, we need to initialize the data store. Open up app/app.js and add the lines shown in listing 6.

```
EME.store = DS.Store.create({
    adapter: "DS.RESTAdapter",
    revision: 11
});
```

Listing 6 – Creating the data store

As you can see, we are creating a new DS.Store object, called EME.store. We need to tell the data store two pieces of information: the Ember Data revision number and which adapter we will use. The revision number ensures that you will be notified whenever breaking changes occur in Ember Data. Because Ember Data has not reached a finalized and stable 1.0 API yet, breaking changes will occur from time to time as the API evolves towards a final release. We will be using the DS.RESTAdapter to interface with the back end throughout this application.

For this application, we will only require one type of data – a photograph – which will have three properties: an id, a title, and a URL. The photograph data model is defined by extending the DS.Model. Create the file models/photo_model.js and add the code from listing 7, below.

```
EME.Photo = DS.Model.extend({
    imageTitle: DS.attr('string'),
    imageUrl: DS.attr('string')
});
```

Listing 7 – The photograph model object

You may note that we haven't actually defined an "id" property. Ember Data will create one for us. In fact, it won't even allow us to specify one manually. We have

defined two properties, “imageTitle” and “imageUrl”, as strings via DS.attr(‘string’).
With that out of the way, it is time to think about how we are going to structure our application and define our application’s URLs.

Structuring our application via the Ember Router

In the previous article, we used the Ember StateManager to structure the application states. Since then, the StateManager has been replaced with a more thorough implementation named Ember Router. Ember Router serves the same purpose but will generate a lot of the boilerplate code automatically for you at runtime using sane defaults that you may easily override when necessary.

First, we need to define the routes that our application will be built with. You can think of a route as a state that the user can be in. Each route will have its own clearly defined URL. Ember.js will conveniently generate controllers, view, and template automatically for you. Whenever you find yourself needing more than the basic pre-defined functionality, simply create your own implementation and Ember.js will automatically substitute your code for the generated code.

For the photo album, we need three routes:

1. An index route that responds to the “/” URL
2. A photos route that responds to the “/photos” URL
3. A selected-photo route that responds to the “/photos/:photo_id” route.

As mentioned, Ember.js will happily generate all of the above for us, but first we need to tell Ember.js which routes we want our application to have. We do this by simply implementing EME.Router.map()shown in listing 8. Create a new file called router.js and add the following code to it:

```
EME.Router.map(function() {
    this.route(“index”, {path: “/”});
    this.resource(“photos”, {path: “/photos”},
function() {
        this.route(“selectedPhoto”, {path:
“:photo_id”})
    });
});
```

Listing 8 – Defining the application routes

As you can see, there are two different constructs that we can use inside the map() function: route() representing final routes and resource() representing a route that can contain sub-routes. We are defining two top-level routes, one named index that responds to the “/” URL and a resource called photos that responds to the “/photos” URL. In addition, we are creating a sub-route called selectedPhoto that will respond to the “/photos/:photo_id” URL. The Ember Router uses URL attributes prepended with a colon to indicate a dynamic part. Ember Router will replace this dynamic part later when it updates the application’s URL.

Note also that in Ember Router’s naming conventions “:photo_id” refers to the “id” property on the EME. Photo model object.
Because we won’t be displaying anything inside the index route, we will simply redirect the index route to the photos route in case the user requests the “/” URL directly. We can do this by creating an EME. IndexRoute class as shown below in listing 9. Add this to the router.js file.

```
EME.IndexRoute = Ember.Route.extend({
    redirect: function() {
        this.transitionTo(‘photos’);
    }
});
```

Listing 9 – Redirecting from the index to the photos route

We’ve used the transitionTo() function for the redirection.

Fetching the photos from the back end using Ember Data

The final piece of code required for our router to photos is to indicate the data the photos route will use. Extend router.js with the code shown below in listing 10:

```
EME.PhotosRoute = Ember.Route.extend({
    model: function() {
        return EME.Photo.find();
    }
});
```

Listing 10 – Fetching photos

In this code, we are relying on Ember Data to fetch all of the EME.Photo objects from the server asynchronously. Once the server has responded, EME.PhotosRoute will populate the automatically generated PhotosController with the data returned. Note that we could have asked for a specific photograph from the server by specifying an id in the find() function as in EME.Photo.find(1) but for our purpose, fetching all photos is sufficient.
Because our model class is named EME.Photos, Ember Data will by default look for the data for this model via the URL "/photos", the contents of which is shown below in listing 11.

```
{ "photos": [
    { "id": 1, "image_title": "Bird", "image_
url": "img/bird.jpg"},
    { "id": "2", "image_title": "Dragonfly",
"image_url": "img/dragonfly.jpg"},
    { "id": "3", "image_title": "Fly", "image_
url": "img/fly.jpg"},
    { "id": "4", "image_title": "Frog", "image_
url": "img/frog.jpg"},
    { "id": "5", "image_title": "Lizard",
"image_url": "img/lizard.jpg"},
    { "id": "6", "image_title": "Mountain 1",
"image_url": "img/mountain.jpg"},
    { "id": "7", "image_title": "Mountain 2",
"image_url": "img/mountain2.jpg"},
    { "id": "8", "image_title": "Panorama",
"image_url": "img/panorama.jpg"},
    { "id": "9", "image_title": "Sheep", "image_
url": "img/sheep.jpg"},
    { "id": "10", "image_title": "Waterfall",
"image_url": "img/waterfall.jpg"}
```

Listing 11 – The sample photo data

This is all well and good, but having the photos loaded doesn't do us any good without any code to display them. So let's get started with defining some templates for our application.

Creating the application templates

Ember.js uses Handlebars.js as its default templating engine. We will define four templates. We have already seen the application template in listing 5, but it is missing one crucial item: an outlet into which it can draw the current route's template. So let's extend the application template with an outlet, as shown below in listing 12.

```
<script type="text/x-handlebars">
    <div id="mainArea">
        {{outlet}}
    </div>
</script>
```

Listing 12 – Extending the application template with an outlet

An outlet is simply defined via the {{outlet}} handlebars expression and it is used to tell Ember.js exactly where that template expects to draw sub-templates. The contents of this template will change depending on which route is the current viewed route. For the index route, this outlet will contain the index template while for the photos resource, this outlet will contain the photos template.

Now that we have a place to draw the list of photo thumbnails, let's define the photos template, shown below in listing 13.

```
<script type="text/x-handlebars" id="photos">
       {{outlet}}

       <div class="thumbnailViewList">
           {{#each photo in controller}}
           <div class="thumbnailItem">
               {{#linkTo photos.selectedPhoto
photo}}
                   {{view EME.
PhotoThumbnailView
                       srcBinding="photo.
imageUrl"

contentBinding="photo"}}
                   {{/linkTo}}
           </div>
           {{/each}}
       </div>
   </script>
```

Listing 13 – Defining the photos template

There are a couple of new things to note in this example. The first is the fact that the text/x-handlebars script tag has an "id" attribute. This attribute tells Ember.js the name of this template and

this name – "photos" – tells Ember.js that this template belongs to the PhotosRoute. As long as you keep to the standard naming convention, Ember.js is nice enough to take care of the rest.

The first thing we define in our template is an {{outlet}} into which we can draw whichever is the selected photograph. Next, we define a div element into which we can draw each of our thumbnails. In order to iterate over each photo, we load into the PhotosController the {{#each photo in controller}} handlebars expression. Inside the each-loop we are using the {{#linkTo ..}} expression to link each photograph to the selectedPhoto route. Note that we are both specifying which route to link to (via the parent-route.sub.route construct), as well as specifying a context that Ember.js will pass into this route. In our case, this context is simply the photograph that the user clicks on.

We have created a custom view to render each of the thumbnails. Create the file views/photo_thumbnail_view.js, the content of which are shown below in listing 14.

```
EME.PhotoThumbnailView = Ember.View.extend({
    tagName: 'img',
    attributeBindings: ['src'],
    classNames: ['thumbnailItem'],
    classNameBindings: 'isSelected',

    isSelected: function() {
        return this.get('content.id') ===
this.get('controller.controllers.
photosSelectedPhoto.content.id');
    }.property('controller.controllers.
photosSelectedPhoto.content', 'content')
});
```

Listing 14 – Creating a custom view for the thumbnails

PhotoThumbnailView starts by specifying that this view will be rendered as an img tag via the tagName property. It then binds the src attribute before specifying the CSS class thumbnailItem for this tag. We want to make the selected photograph larger than the other photographs to make it clear to the user which photograph is selected. In order to append a custom is-selected CSS class to the selected property, we use the classNameBindings property. This will simply append the CSS class is-selected to the view if the isSelected computed property returns true, which it only does if the id of the viewed photo is the same as the id of the selected photo.

The only template missing from our application at this point is the template that renders the selected photo. Because this route is a sub-route of the photos resource, it will have an id of photos/selectedPhoto. Listing 15 shows the selectedPhoto template.

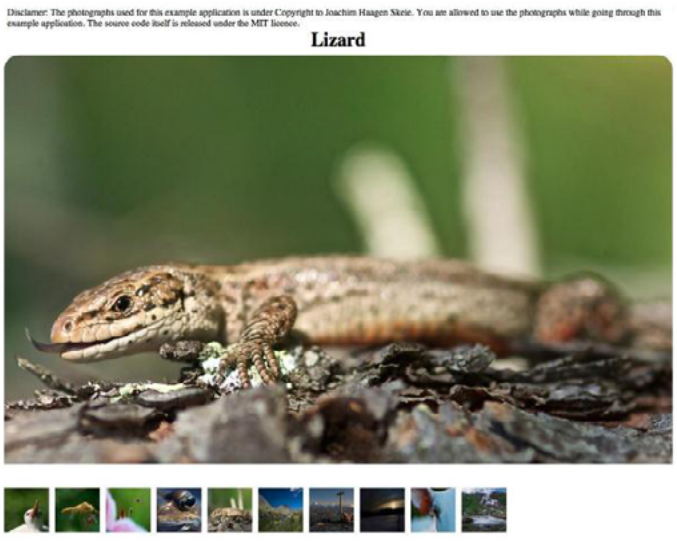
```
<script type="text/x-handlebars" id="photos/
selectedPhoto">
    <div id="selectedPhoto">
        <h1>{{imageTitle}}</h1>

        <div class="selectedPhotoItem">
            <img id="selectedImage" {{bindAttr
src="imageUrl"}}/>
        </div>
    </div>
</script>
```

Listing 15 –The selected-photo template

There is nothing new going on in this template, as it simply renders a div element to hold the selected photograph into which it renders the title and the photo.

At this point, you can reload the application. The end result will look something like figure 3 below.
Figure 3 – The photo album so far
You may notice that the currently selected



photograph is neither highlighted nor larger than

the other thumbails. The reason is that we need to specify a controller for both the photos route and for the selectedPhotos route. In addition, we need to link these controllers together in order for our PhotoThumbnailView to resolve the isSelected computed property correctly. Let's implement these two controllers.

Defining controllers

At this point, we will define customized controllers for the photos resource and for the selectedPhoto route. In addition, we will need to link the photos controller to the selectedPhoto controller. Let's start by creating a new file, controllers/photos_controller.js:

```
EME.PhotosController = Ember.ArrayController.
extend({
    needs: ['photosSelectedPhoto'],
});
```

Listing 16 –The PhotosController

There is not much going on inside this controller. Because we have defined the controller to be an Ember.ArrayController, Ember.js takes care of proxying our list of photos to the template, as well as handling any bindings or other controller issues. The only thing we need to specify is the fact that this controller needs the photosSelectedPhotoController. Because we have specified an explicit relationship between the controllers, we also need to create the controller we are referring to. Create a new file controllers/selected_photo_controller.js, the content of which is shown below in listing 17.

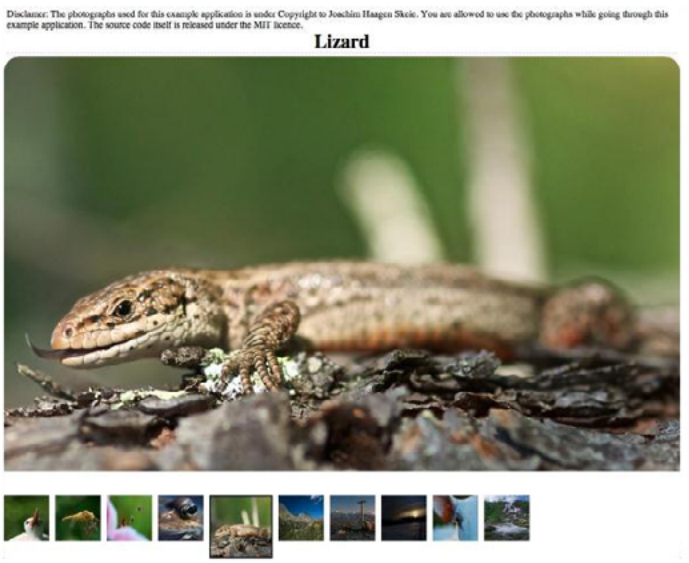
```
EME.PhotosSelectedPhotoController = Ember.
ObjectController.extend({});
```

Listing 17 –The PhotosSelectedPhotoController

Because we aren't implementing any custom logic for this controller, it's enough to declare it and specify that it is an Ember.ObjectController, meaning that it will serve as a proxy for a single photograph.

Refreshing the application, as shown below in figure 4, reveals that the selected photograph is now highlighted as we expected.

Figure 4 – The result after adding the controllers



The final piece of the application is a set of control buttons that allow the user to navigate the sequence of photographs and to start and stop a slideshow.

Adding control buttons and slideshow functionality

We are going to add four buttons below the selected photograph to control which photo is selected. The "Play" button will start a slideshow that will change photos every four seconds. The "Stop" button will stop the slideshow while the "Next" and "Prev" buttons will select the next and previous photos respectively. Let's start by adding a photoControls template to hold the control buttons:

```
<script type="text/x-handlebars"
id="photoControls">
    <div class="controlButtons">
        <button {{action
playSlideshow}}>Play</button>
        <button {{action
stopSlideshow}}>Stop</button>
        <button {{action prevPhoto}}>Prev</
button>
        <button {{action nextPhoto}}>Next</
button>
   </div>
</script>
```

Listing 18 –The photoControls template

This template simply creates four buttons. To each button, it attaches an appropriate action, for which we will implement action functions later. First, we need to tell the application to render this

photoControls template directly beneath the selected photograph. It makes sense to specify this just after the {{outlet}} in the photos template, as shown in listing 19, below.

```
<script type="text/x-handlebars" id="photos">
        <div>Disclamer: The photographs used
for this example application is under
Copyright to Joachim Haagen Skeie.
            You are allowed to use the
photographs while going through this example
application. The source code itself
            is released under the MIT licence.
        </div>
        {{outlet}}
        {{render photoControls}}
        <div class="thumbnailViewList">
            {{#each photo in controller}}
            <div class="thumbnailItem">
                {{#linkTo photos.selectedPhoto
photo}}
                    {{view EME.
PhotoThumbnailView
                        srcBinding="photo.
imageUrl"

contentBinding="photo"}}
                    {{/linkTo}}
            </div>
            {{/each}}
        </div>
    </script>
```

Listing 19 –Adding the photoControls to the application

Next, we need to define the action functions for our four buttons. Because we are using the {{render}} expression, we can create a PhotoControls controller and make Ember.js automatically use this controller as the controller for the photoControls template. Create a new file controllers/photo_controls_controller.js with the contents of listing 20:

```
EME.PhotoControlsController = Ember.Controller.
extend({
    needs: ['photos', 'photosSelectedPhoto'],

    playSlideshow: function() {
        console.log('playSlideshow');
    },

    stopSlideshow: function() {
        console.log('stopSlideshow');
    },

    nextPhoto: function() {
        console.log('nextPhoto');
    },
    prevPhoto: function() {
        console.log('prevPhoto');
    }
});
```

Listing 20 –Adding PhotoControlsController to handle events

If you now reload the application and bring up the browser's console, you should see a new line logged to the console each time you click one of the four buttons.

Note that we have specified that this controller needs both the photosController and the photosSelectedPhoto. The reason is that we don't want to implement the contents of these action functions inside this controller. Rather, we want to delegate some of these to the other controllers, where this responsibility makes more sense. Both the nextPhoto and the prevPhoto make more sense in the PhotosController, so let's update the code to delegate the actions to this controller, shown below in listing 21.

```
nextPhoto: function() {
    console.log('nextPhoto');
    this.get('controllers.photos').nextPhoto();
},

prevPhoto: function() {<
    console.log('prevPhoto');
    this.get('controllers.photos').prevPhoto();
}
```

Listing 21 – Delegating events to the PhotosController

As you can see, we are simply delegating to functions on the PhotosController. But before we have a closer

look at the implementation of those functions, let's finish this controller by implementing the slideshow slideshow functionality, as shown below in listing 22.

```
playSlideshow: function() {
        console.log('playSlideshow');
        var controller = this;
        controller.nextPhoto();
        this.set('slideshowTimerId',
setInterval(function() {
            Ember.run(function() {
                controller.nextPhoto();
            });
        }, 4000));
    },

    stopSlideshow: function() {
        console.log('stopSlideshow');
        clearInterval(this.
get('slideshowTimerId'));
        this.set('slideshowTimerId', null);
    }
```

Listing 22 – Adding slideshow functionality

There are a couple of things to note here. When the user clicks on the play button, the playSlideshow() function will immediately trigger the nextPhoto() function in order to select the next photograph. It then starts a timer using the setInterval() function built into JavaScript. This function returns an id that we can use to clear the interval later on, so we make sure that we store this id in the PhotoControlsControllers slideshowTimerId property.

Because we don't control when the timer executes the callback function and because we want to ensure that we are executing this callback as part of the Ember.js run-loop, we need to wrap the content of the callback into Ember.run().

The final missing piece of our application at this point is the implementation of the nextPhoto and prevPhoto functions. I will only show the implementation of the nextPhoto here, because these two implementations are almost identical. Listing 23 shows the nextPhoto implementation.

```
    nextPhoto: function() {
        var selectedPhoto = null;
        if (!this.get('controllers.
photosSelectedPhoto.content')) {
            this.transitionToRoute("photos.
selectedPhoto",
                this.get('content.
firstObject'));
        } else {
            var selectedIndex = this.
findSelectedItemIndex();

            if (selectedIndex >= (this.
get('content.length') - 1)) {
                selectedIndex = 0;
            } else {
                selectedIndex++;
            }

            this.transitionToRoute("photos.
selectedPhoto",
                this.get('content').
objectAt(selectedIndex))
        }
    }
```

Listing 23 –The nextPhoto() function

If there is no photograph selected, the function will simply display the first photograph in the list. It accomplishes this by transitioning to the photos.selectedPhoto route using the content.firstObject property.

If a photograph is already selected, we need to find out which index the selected photograph has inside the content array. We do this by calling the findSelectedItemIndex() function, the content of which is shown below in listing 24.

Once the index of the currently selected photograph is found, we need to increment the selected index. The only exception is when the last photograph is selected, in which case we want to select the first photograph in the content array.

When the index of the photograph we want to transition to has been determined, we can simply transition to the photos.selectedPhoto using that photograph as the context.

```
findSelectedItemIndex: function() {
        var content = this.get('content');
```

```
        var selectedPhoto = this.
get('controllers.photosSelectedPhoto.content');

        for (index = 0; index < content.
get('length'); index++) {
            if (this.get('controllers.
photosSelectedPhoto.content') ===
content.objectAt(index)) {
                return index;
            }
        }

        return 0;
    }
```

Listing 24 –The findSelectedItemIndex() function

There should be no surprises in the code for the findSelectedItemIndex(). It simply iterates across the content array until it finds the selected photograph. If it is unable to find a photograph, it simply returns the index of the first photograph in the content array.

Normally I would not add my adapter to my app.js file, but rather to its own .js file. This makes it easier to find the adapter code later on when you want to change your implementation.

Conclusion

Ember.js brings to the table a clean and consistent application-development model. It makes it easy to create your own template views that are easy to understand, create, and update. Coupled with its consistent way of managing bindings and computed properties, Ember.js does indeed offer much of the boilerplate code that a web framework needs, and because it is so obvious which technology is in use and how the DOM tree gets updated, it's also easy to add third-party add-ons and plugins.

Ember.js's naming conventions mean that you only have to specify and implement classes when the defaults Ember.js provides aren't sufficient. What you end up with is a clean codebase that is easy to test and, most importantly, easy to maintain.
Hopefully, this article has given you a better understanding of the Ember.js development model and you are left with a clear view of how to leverage Ember.js for your next project.

All of the concepts and constructs that are used in this article, and more, will be covered in detail in "Ember.js in Action". This book is currently available via Manning Publications Early Access Program, meaning that you can get a head start on the book's content right away.

About the Author

Joachim Haagen Skeie is the owner at Haagen Software AS in Oslo, Norway, where he works as an independent consultant and course instructor. He has a keen interest in both application profiling and open-source software. Through his company, he is currently busy bootstrapping the EurekaJ Application Monitoring Tool. He can be contacted via his Twitter account, or via email at joachim(at)haagen-software.no.

READ THIS ARTICLE ONLINE ON InfoQ
http://www.infoq.com/articles/Emberjs-Web-Applications