API TECHNOLOGIES

by InfoQueue
Enterprise Software Development Community
API eMag / Issue 5 - November 2013

The Netflix API Optimization Story

The Netflix API optimization story is an interesting journey from a generic one-size-fits-all static REST API architecture to a more dynamic architecture that lends power to the client team to define and deploy their custom service endpoints.

Page 12

Mike Amundsen on API Design, Governance, and Lifecycle Management

Mike Amundsen talks about API management, versioning, and discovery.

Page 7

Top 10 Internet of Things APIs

At a faster rate than ever before, APIs today are enabling the interconnectivity of everything from your oven to your refrigerator to you car.

Page 17

Adrian Cockcroft on Architecture for the Cloud

In this interview, Adrian Cockcroft, the architect for Netflix's cloud systems team discusses how Netflix combines 300 loosely coupled services across 10,000 machines.

Page 31

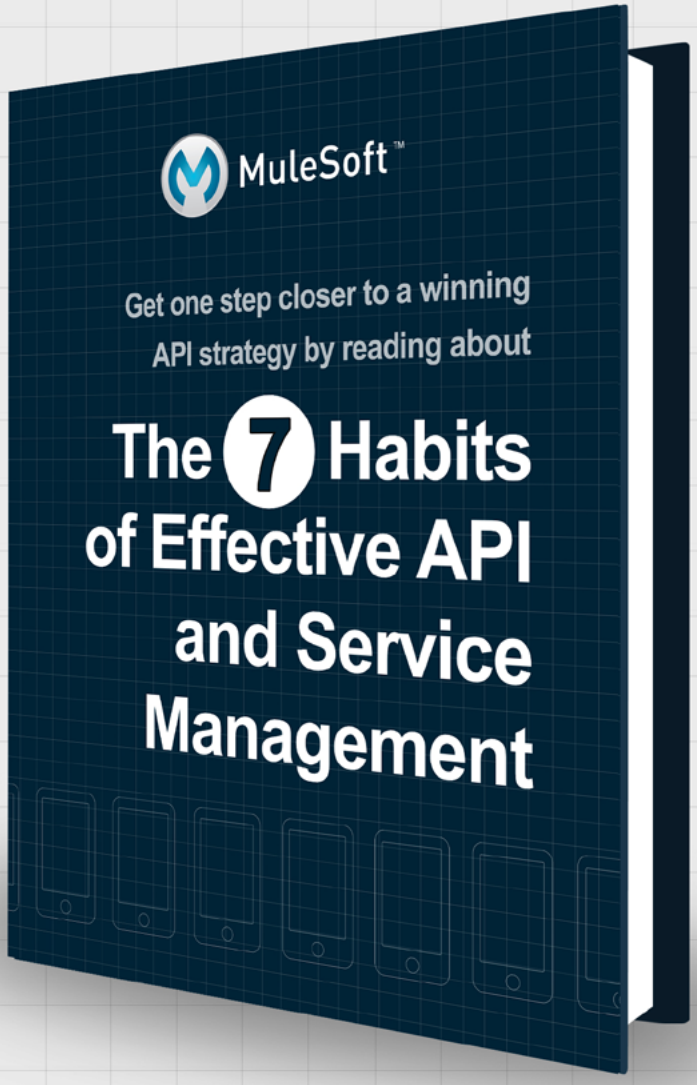
Contents

MuleSoft™

Looking to mobile-enable your services?
Sell your services to create new revenue channels?

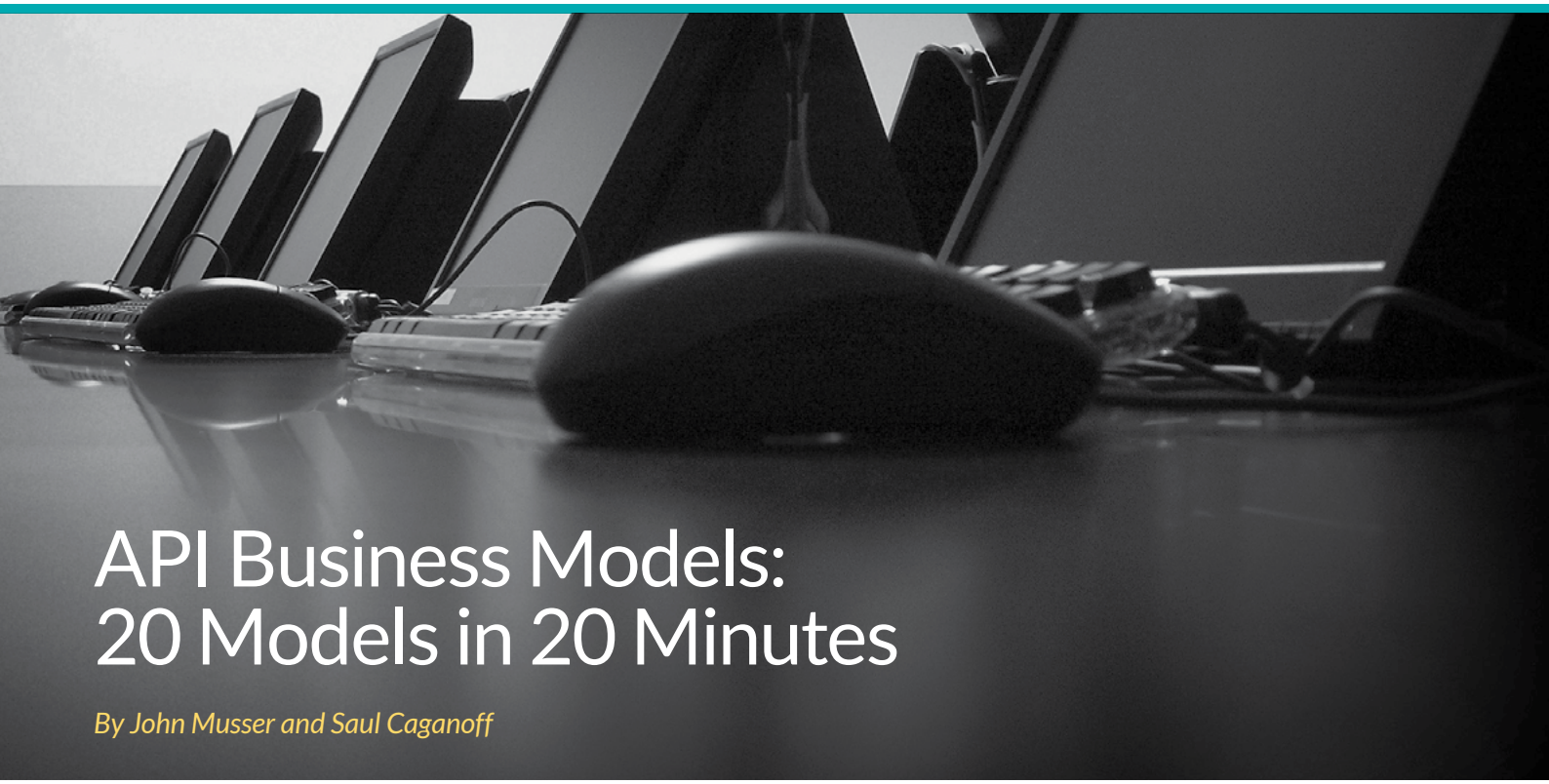
It's not as hard as you think.

The API economy is here and the opportunity to drive concrete business results using APIs is real. But where do you get started? How do you design an API developers will love? And how can you effectively leverage existing services, even if they're not REST-based? It's not as hard as you think.



Get one step closer to a winning API strategy by reading about **The 7 Habits of Effective API and Service Management.**

Download



API Business Models: 20 Models in 20 Minutes

By John Musser and Saul Caganoff

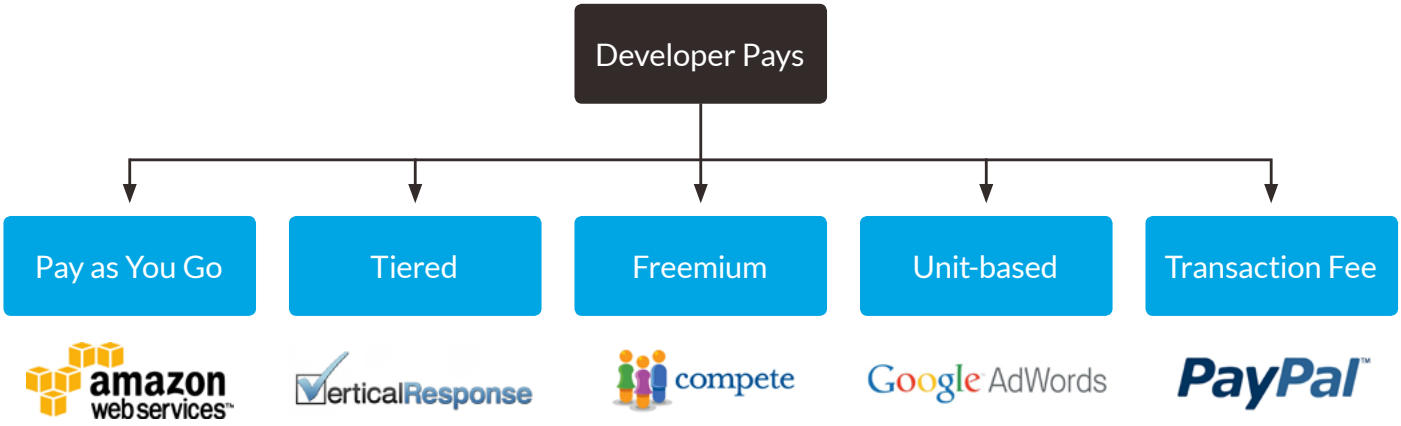
John Musser knows APIs. As the founder of Programmable Web, John has witnessed thousands of APIs and many business models. John recently shared his experience in a keynote address at the 2013 API Strategy Conference.

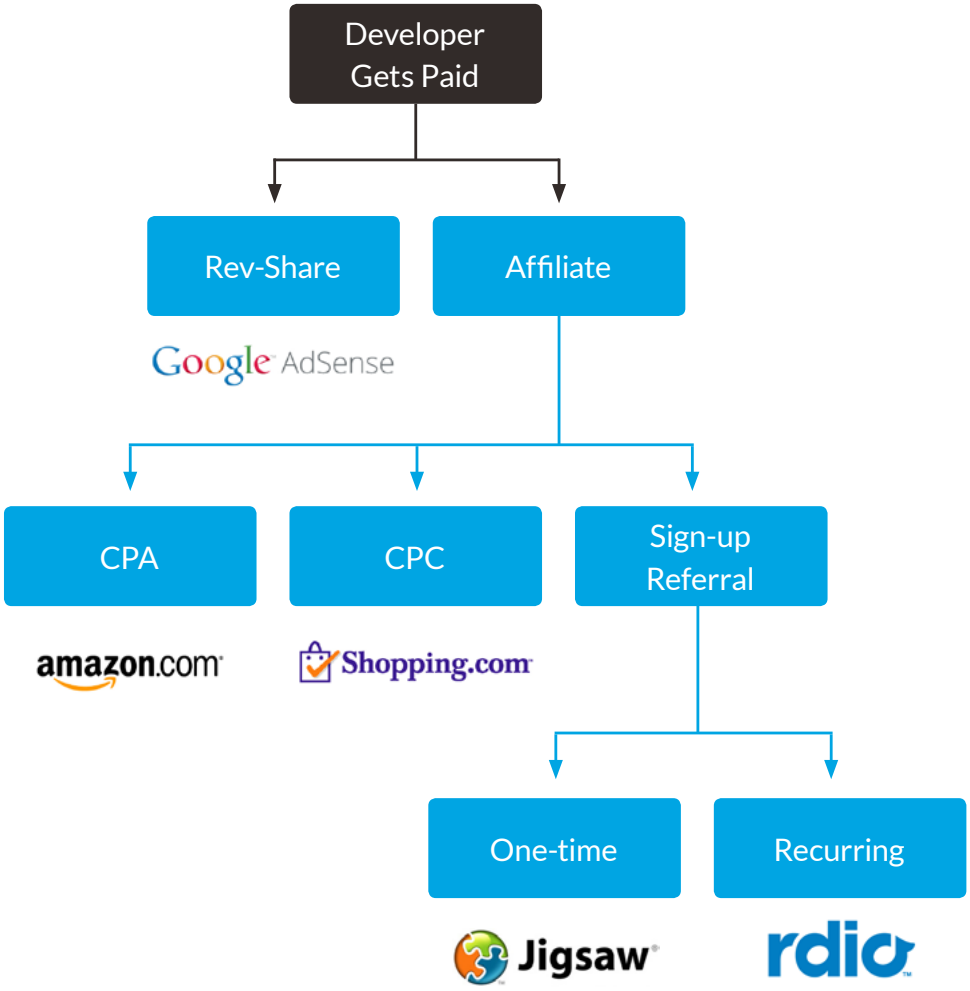
The most frequent question asked of John is "How do we make money from this?" The answer depends on why you'd want an API. An API strategy is not an API business model. An API strategy is "why" you want an API and a business model is about "how" you make money from it.

Looking back to 2005, the beginning of APIs and the time when Google Maps took off, there were four core API business models: "free", "developer pays", "developer gets paid" and "indirect". In 2013, the same four core business models still exist but each has expanded into a variety of different ways to provide and monetise APIs. Most APIs have more than one type of ROI.

Contrary to popular belief, "free" is not the predominant API business model. Facebook and all the government and public sector APIs are good examples of free APIs , but these are only a tiny fraction of all API business models.

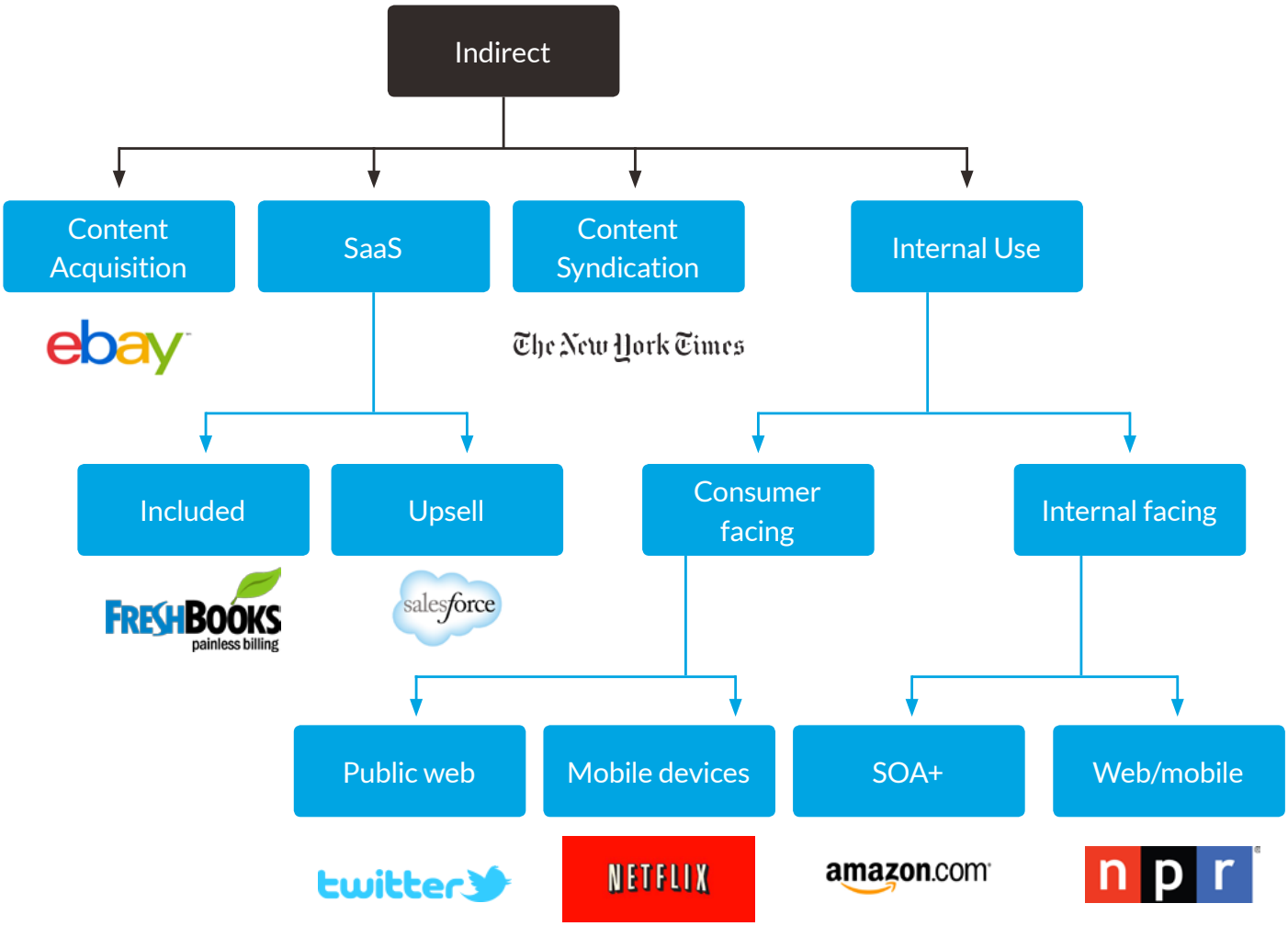
In the "developer pays" category, application developers pay for the services provided by the API. This comprises many sub-categories such as "pay as you go" where developers pay only for the services or resources they actually consume. For example, Amazon Web Services states in its pricing plan: "Pay only for what you use. There is no minimum fee."





A second sub-category, the "tiered" model offered by companies like Mailchimp, provides lower cost per unit as you consume higher volumes of their services. "Freemium" is a well-known model that provides basic features for free but sells services such as additional APIs, higher SLAs, an account representative, etc. Compete and Google Maps use the freemium model. "Unit-based" pricing is another sub-category, such as Sprint's offering where different features in the API cost differently. The final example of "developer pays" is the "transaction fee" familiar to users of payment APIs where a percentage of the transaction is charged. PayPal, Stripe and Chargify are examples of this model.

The converse of "developer pays" is the "developer gets paid" business model. This category also has many sub-categories, such as the "affiliate revenue share" model offered by Amazon Affiliate Program in which developers are paid for customer referrals. In the "revenue share" sub-category, developers are paid a share of the revenue from referred purchases. Mysimon.com and Howstuffworks.com are comparison sites that use Shopping.com APIs to support product comparisons and gain revenue share from click-throughs. Affiliate revenue sharing is not a small business. Expedia derives 90% of their revenue from an affiliate network using Expedia APIs for $2 billion per year. Finally, companies such as Rdio.com use the "recurring revenue share" model in which affiliates who refer new customers for subscription purchases are paid a revenue share for the life of that customer.



An API business-model secret is that you need to bake your business model into your API. This is the biggest point to think about. For example, the Amazon affiliate programme was a natural extension of their retail business model.

Out of the original four core business models, the "indirect" category contains some of the most interesting ways to monetize APIs. The first sub-category is "content acquisition." Businesses like eBay and Twitter need to rapidly acquire content in order to grow, so they use APIs to facilitate content acquisition. eBay APIs allowed power users to create mass listings in the eBay marketplace. Twitter derives and distributes content almost exclusively through APIs and third-party applications. "Content syndication" lets the New York Times deploy APIs to syndicate its content to partners.

SaaS upsell is another business model within the "indirect" category. Salesforce.com offers API access to their platform, but only to companies that pay the enterprise licence. Salesforce.com knows that API integration is an important tool it can use to upsell customers into higher subscriptions. APIs are the glue of any SaaS. Integration adds value to SaaS applications and provides stickiness that dramatically reduces churn.

Note that API business models are not one size fits all. The final business model is "internal use" through which companies use APIs to support their own business. NPR developed APIs to deliver content to websites and mobile apps. Over 99% of Evernote's API traffic is from tablet apps, mobile apps and partner apps. And Netflix uses its APIs to support content delivery to over 800 kinds of devices. Internal use may be the biggest API use-case of them all.

About the Speaker

John Musser is the founder of ProgrammableWeb - the world's leading directory of APIs - and has deep insight into API trends over time including acceleration in the growth of APIs, emergence of Business Models and technology trends.

About the Author

Saul Caganoff is the CTO of Sixtree, an Australian system integration consultancy. He has extensive experience as an architect and engineer in major integration and software development projects in Australia, the United States and Asia. Saul's professional interests include architecture at all levels - enterprise, solution and applications - distributed systems,

READ THIS ARTICLE ONLINE ON InfoQ
http://www.infoq.com/presentations/API-Business-Models

Mike Amundsen on API Design, Governance, and Lifecycle Management

Interview with Mike Amundsen by Jeevak Kasarkod and Nitin Bharti

In this interview, recorded at QCon New York 2013, Mike Amundsen talks about API management, versioning, and discovery. He compares RESTful and CRUD-style APIs, discusses the notion of 'affordance, ' and introduces hypermedia APIs. He examines documentation modelling frameworks for APIs - like Swagger - and also provides his thoughts on API governance, OAuth 2.0, and web single sign-on.

Mike, can you tell us a little bit about yourself?

I work with a company named Layer 7 Technologies, which is based out of Vancouver. We just recently merged with CA, so we are really excited about that. My job title is Principal API Architect. It's a title they made up for me but the coolest thing is my job description is to help people build great web APIs. We have a thing at Layer 7 we call the API Academy: Ronnie Mitra, Alex Gaber, Matt McLarty and I work on providing guidance, gathering research and disseminating it. It's all about helping people build really cool interfaces for the web.

Contracts for RESTful APIs have been a touchy topic. Do you think they'll be necessary given the current proliferation of CRUD style APIs?

Yes. In a general sense, we have a contract, even in a CRUD style API. The contract is, "I'll give you a URL and then you can do these possible things with it." In SOAP, we had a much more extensive contract. We had the WSDL contract which listed a set of functions and arguments, and their types and the kind of return values. So we are used to contracts. In the hypermedia style, our contract is the type of media itself: "When I send you a message, here are the things you can do: you can create a form, you can follow a link and so on and so forth."

No matter what we do in communications, whether machine-to-machine or humans-to-machine, we definitely are going to have contracts. I think the real challenge is figuring out which contracts are effective over the long term and usually that's a contract that is not overly stringent. A contract that lets the system add new features or change features without actually breaking the contract - I think that's the big challenge that we work on.

Tools like Swagger - a specification and framework implementation for describing, producing, consuming, and visualizing RESTful web services – appear to be gaining in popularity. What are your thoughts on Swagger?

Swagger or other products like it, - like I/O Docs, or APIary.io which we actually integrate with at Layer 7 - are all attempts to create documentation modeling for an API that we can present sort of as a

contract. What's really fascinating is, as Ronnie and Alex and I work with customers, we're finding that when we watch people work, we watch developers work, they don't often look at documentation unless they have to. So, one of the things that I am finding very interesting is to figure out how to create ways to communicate developers' information without having them read lots and lots of material, so we are trying to create usable APIs.

Anybody who works with VisualStudio or Eclipse knows about system dot, they type system dot and then they get a bunch of features that's a usable interface. When I am using your CRUD style, I can do API.example.com, hit enter and I get something back. There's links I can follow and I can change the URL. All these documentation things try to solve some of those problems and I think what I find interesting are ways to reduce the amount of reading someone has to do before they are effective with your API. So, Swagger is filling a need, but I think there may be other ways to do that.

How important is affordance in system-to-system integration scenarios?

When you are talking machine-to-machine you need to create affordances that machines understand. To us, a link, a search box - those are all affordances for humans, but they may not be really clear to machines. So, when you start working with machines, we start to do some things where we design a different kind of media type or a different kind of message model that's focused more on machines than humans. We definitely use them. We use them every day.

One of the things I have discovered over the last years is whatever information I don't put inside a response or a message I may have to put in source code and actually have to code into clients. The more information I can include in the message, the less code I need and there's more opportunity for that thing to change over time. I think we are going to find more and more affordance-based message design over the next couple of years. We've had this thing called media type registration and in the last two years we've had more media type registrations that have in-message affordances than we've had in the previous ten. So I think a lot of people are hitting on this idea that affordances are really important.

The first word that comes to mind to describe OAuth 2.0 is "complex". In terms of API security what are your thoughts regarding this?

OAuth2 is definitely more complex than OAuth1 or 1.0.8, but I think we definitely need this kind of authentication modeling. The big advantage of OAuth is this notion that two parties can actually share information when neither one of them holds the actual credentials for the user in case: we have a three-legged patterning. So we definitely need that kind of stuff. The original designer of OAuth, when we worked through the OAuth2 spec, got really frustrated with the process but I think it's leading in the right direction and while there is some built-in complexity early on and some of it might have been avoidable, it's still the best option for us in a widely distributed system.

One of the things we need to do in all kinds of distributed systems is reduce the dependence on single hubs or single points, and I think OAuth does a really good job of creating a model for distributed authentication. I think it's the best model that we have and it's the one that we need to continue to use. You can definitely get tool sets or tool kits or proxy servers to start to handle most of that complexity today; you can hide a lot of that complexity right now even while we work through the details. One of the great things about specs, through the IETF, is that this is an RFC and we can make some changes over time that aren't going to upset the apple cart. We're advising all our customers to definitely consider OAuth, especially if you're in any kind of distributed space.

What about web single sign-on? Do you think there's one approach that might be emerging to be the dominant leader in this space?

I am not sure if there is one approach that is emerging, but what I can tell you from our work with customers is that we're getting more and more requests about it. So, OAuth is this notion that I can bring my credentials to the table. I can show up, I can walk up to your window at your web server and provide my credentials from some other party; in other words, I don't have to actually have credentials with you. I think that what's going to happen is that some third parties are going to become leaders as trusted sources in that space. So, once there are some trusted sources, then lots and lots of services can say, "We'll get a relationship with that trusted source so you can

walk up to the door and you have the same sign-on".

So, the next step, once you signed on here, is to use the same sign-on in multiple places in the same session. I know a company named Okta is very big in the enterprise space on this and this is good when you have all those other services sort of behind a single firewall or a single pattern. I think that is probably the first step that we need to get really good out there. There are some other companies that I can't really think of right now, but I think eventually we need to even get beyond that. I need to be able to carry my credentials around with me in a very distributed way, just as we hide a lot of the complexity of OAuth, I think we will hide a lot of the complexity of multi sign-on or multi service use as well.

Some failures of SOA transformation projects were attributed to excessive governance and process. Do we need governance for APIs? If we do need governance, what would be the right way to approach it?

The first part of that answer is yes, we definitely need governance. Any time we get anything going, you're going to need some kind of governance, some kind of control, some kind of oversight. I think probably what we claim is sort of that the failure of SOA modeling is not just one thing, it's a lot of things. I think the real key is it goes back to contracts. It's about deciding what kinds of things you want to make contractual, what kinds of things you want to govern carefully, what kind of things you want to govern loosely.

One of the powerful things about the web is that we have a lot of loose governance. We have a lot of very general things, the protocol, some media types, a couple of authentication formats and really that's it. After that, you're open to doing what you want, anything you do adds to that and anything that adds to that limits things in the future. So, the longer you run things, the more extensive they are and the more important it is to have some kind of loose coupling.

I think a lot of times when people would try to implement SOA, they'd use a lot of tight coupling, they'd use a lot of tight mining. If you think about it in language terms, they'd have a lot of strong typing rather than loose typing, and we can see a pattern still today, there's still a pendulum that's using a lot of loose typing in languages. I think what we need to do is carefully think about the things we need to govern, which are usually access control lists, identities. The things we don't need to govern quite so carefully would be the actual operations, the function names. So when you apply good credentialing, like OAuth, as a start, and then apply that to a general guideline for a set of functions...I think it's going to work out just fine.

What is a management API and in your opinion, what are its pillars? As an API provider, do I necessarily need a platform for managing my API?

I don't know if you need a platform, but there are definitely some pillars, especially if you are going to share your API outside. It goes right along what we just talked about: I need access control, I need to know who's using my API, I need to know what apps they are building with it, I need to know what kind of traffic they're bringing to the table, what kind of errors they might be bringing to the table, what kind of bandwidth they are using.

So there's lots and lots of pieces. When we talk about API management, we talk about a couple of things. We talk about this notion of identifying the service: we use API keys for that. We talk about authenticating the user who's using the service: that's authentication of some type, it could be OAuth on the outside or lots of other ones. We talk about authorization: now that I know who you are, what are you actually authorized to do? We talk about the idea of encryption: do we use transport-level encryption, do we have field double encryption? Sometimes we even talk about the notion of non-repudiation: do I have to actually make sure that we've validated you to sign this somewhere, that we validate this action? Those are all pillars of managing API interactions.

Then you still have life-cycle issues. If I make a commitment to an API, I make a commitment for a certain amount of time. If I am going to deprecate it, I have to do it in a certain way. If I am going to make changes to it, I have to make sure that the changes don't break clients, especially in a distributed system. If I invest a couple of million dollars in building a client for your API and you change that every six months, that is going to cost me a lot of money. So, that's what API management is about. It's managing all those interactions and then managing that complete life-cycle pattern. So a lot of things that we learn from

SOA we can apply to this API world which means life cycle is still very, very important.

You mentioned versioning as one if the issues. What are some other unique challenges of rolling out and operationalizing APIs as compared to web applications or even an enterprise software?

That does lead right into it. That's part of server management system. There are a couple of things that we talk about, patterns that we see in successful organizations. One of them is great APIs usually tend to be a very thin veneer: they're a true interface into a set of components. Often, an organization will have lots of interfaces to the same set of stable components. Those interfaces may change relatively soon over time, but the components do not, those components very often stay very stable year after year after year.

This notion of having stable components, a little bit of scripted interface and a very thin API veneer is really important for a solid implementation. It means that I can make minor changes to the API without having to percolate Q&A all through the system. It means I have a lot of agility on top and I have a lot of stability on the bottom, so that's a really fundamental part of creating a great implementation for interfaces. And then the other is the one we talked about before which is versioning or life-cycle pattern, leading clients to commit to things that aren't going to change over time.

So our initial guidance to most people is don't version at all. What version of the web are you using, what version of browser do you use today, what version was the page of HTML you just got? A lot of times we don't know. We only worry if they break. So, if you can make changes without breaking, you don't need versioning. This whole concept of implementation details as well as that external guarantee, those are the big issues in creating a functional API library.

So, Mike, UDDI was a failure, but repositories made service discovery possible within the enterprise. With almost 9,000 web APIs out there what are some of the channels for API discovery?

That's actually an excellent question because I think that is our next big challenge. You talk about the 9,000 APIs - we have 9,000 snowflakes. It's not just 9,000 APIs, it's over 100 APIs just for shopping and in the bigger sense of things I don't think we need 100 APIs for shopping. That means I don't think we need 9,000 APIs but we probably need hundreds and hundreds of them. So, I think we need to get used to the notion that every time I create a service I don't necessarily have to do it totally uniquely. I probably can learn from other people, I can probably stand on the shoulders of giants. There's probably a handful, maybe just a few shopping APIs that we really need, so part of it is simplifying that case.

The other part is that even though the notion of discovery system, the UDDI pattern, didn't really take off, it is really solid both inside the enterprise and outside the enterprise. So, this goes back to the API management tool. A good management interface would make it possible for me to search for services, or search for APIs within my collection. Whether you have an enterprise-level management tool or whether we have something that faces outward, those are definite possibilities. It turns out there are a bunch of people working right now on normalizing how to describe these interfaces in a very generic way. There is a project called the JSON Home document through the IETF that Mark Nottingham is working on; there is a process that Leonard Richardson and myself are working on called ALPS, Application Level Profile Semantics; there is one Erik Viel is working on; there is one a guy named Kevin Swiber is working on. All of a sudden we have a whole bunch of people working on this notion of describing the abstract part of APIs. I suspect it's probably not going to happen in the next two years, maybe it's the next three, four or five, but we're going to end up with a generic pattern that works both on the wide distributed web and can be brought into the enterprise for describing APIs in a general way and searching for them and actually using them to generate some code very similar to the way that WSDL has done. So I think that what we might find over the course of these next 10 to 15 years is sort coming right back around and discovering things we really liked about the OAuth model in general but applying it to a much bigger picture in a successful way.

You've been presenting and writing quite a bit about hypermedia APIs; what are they and what kind of reaction has the idea been getting from the community?

Most of the things I have been talking about on the conference track has been about this notion of hypermedia; I still talk a lot about the use of hypermedia. The concept goes a long way back, more than 15 years. It goes back to the Manhattan project, it goes back to these ideas how people follow links. The whole creation of the mouse and the pointer device was because of hypermedia, the whole creation of the web was because of hypermedia, this whole notion of the REST pattern is all based on hypermedia as a fundamental part of it. So there are lots of examples over several generations of doing this but we don't see a lot of it in real life so it fascinated me over the last several years. As a result, I started working with some very smart people that gave me some really great feedback and I've discovered there are things we can do today that make it possible to use this as sort of our contract model.

Most of the talks I do today talk about that. As a matter of fact I did a couple of workshops here at QCon that had hypermedia features to them: one was based on an article series that I am doing with InfoQ about how to implement hypermedia servers and clients. It turns out clients are really the hard part right now. The other one was based on a book that Leonard Richardson and I are going to release in the summer called RESTful Web APIs, and that takes a much broader view. It's not just hypermedia, but it actually lists lots and lots of media types, lots and lots of standards.

So the reaction we are getting from folks, just this week, people are saying, "Ah, I think I see the value, I think I see this pattern, I think I see where this is leading." In Layer 7, Ronnie and Alex and I are working with companies that are adapting this hypermedia pattern internally. Now it turns out this hypermedia model is a little more abstract. There aren't good tools and there's not a lot of public guidance on it, which means it's not really a good first-day strategy for a public API. The CRUD model is a much more accessible and usable strategy, but internally in enterprises I am seeing more and more interest. That's usually a community that is a little more captive, a little bit more focused in a particular domain space and we are finding more and more people asking us about how to bring this to the enterprise, how to bring this to the table. I think the other big driver is going to be mobile. When I have to keep downloading a mobile app every time I want to make a change, that becomes rather expensive. If people can make that change without a download, without waiting through the approval process with hypermedia, I think that they are going to use a lot more of it.

Mike, is there any general advice you'd give to developers or even organizations looking to build out their API strategy?

I think the biggest advice I would give to folks is focus on the message that travels on the network, not on the client or the server. It doesn't matter whether you are doing this in Ruby or Erlang or JavaScript or C# or Java. What matters is the message in between because what is really happening now is we need to distribute this in a very wide way.

I think the other big advice that we are finding valuable is think about distance. We often think about distance in terms of someone is across the world. How are they going to use this API without ever meeting me? How are they going to be able to understand it? But I think what is even more important is the distance in time: I'm going to put this API up, what happens a year from now, two years from now, five years from now? Can somebody still use this API?

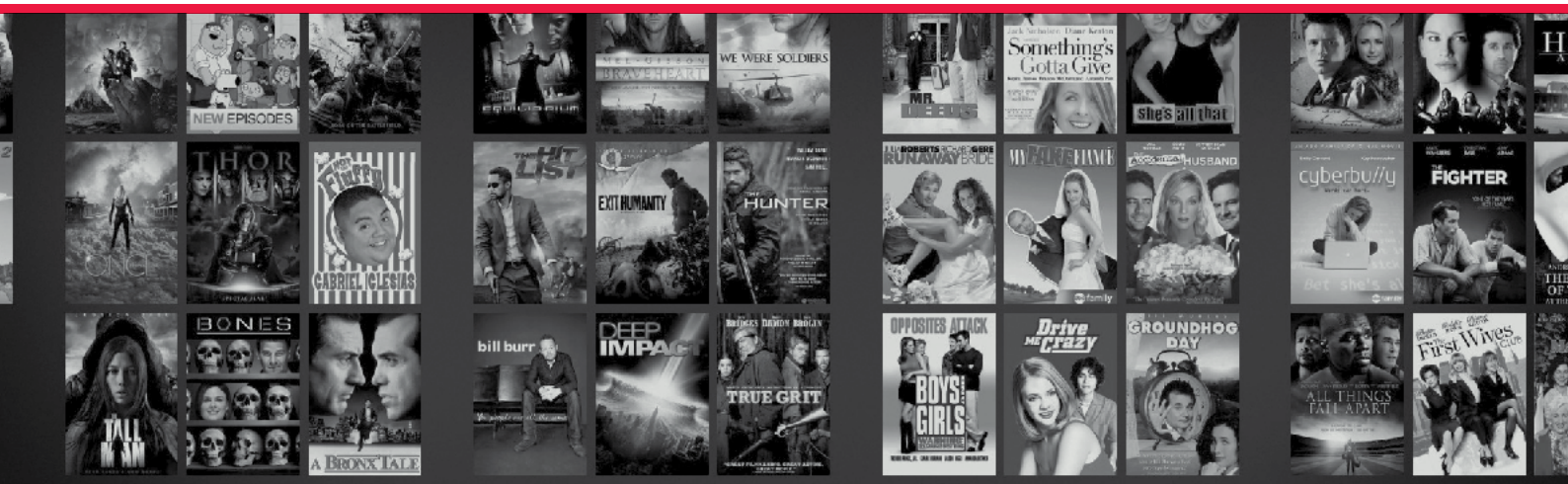
I think this idea of distance and this notion of focus on the message rather than the end points are really the most powerful things you can start to think about to design systems that work in the distance of both time and space and work no matter what kind of language is on either end of it. Now we have an API that is distributable, that's really open, that people can use in lots and lots of ways. I think that's sort of the key.

Donald Norman has this really great idea that a well-designed object is something that allows people to use it in ways the original designer had never intended and I think these are the kind of APIs that we want to build: things that allow people to do new things that we haven't even thought of yet.

About the Speaker

Mike Amundsen is Layer 7's Principal API Architect. An internationally-known author and lecturer, Mike travels throughout the United States and Europe, consulting and speaking on a wide range of topics including distributed network architecture, Web application development and cloud computing.

READ THIS ARTICLE ONLINE ON InfoQ
http://www.infoq.com/interviews/amundsen-api



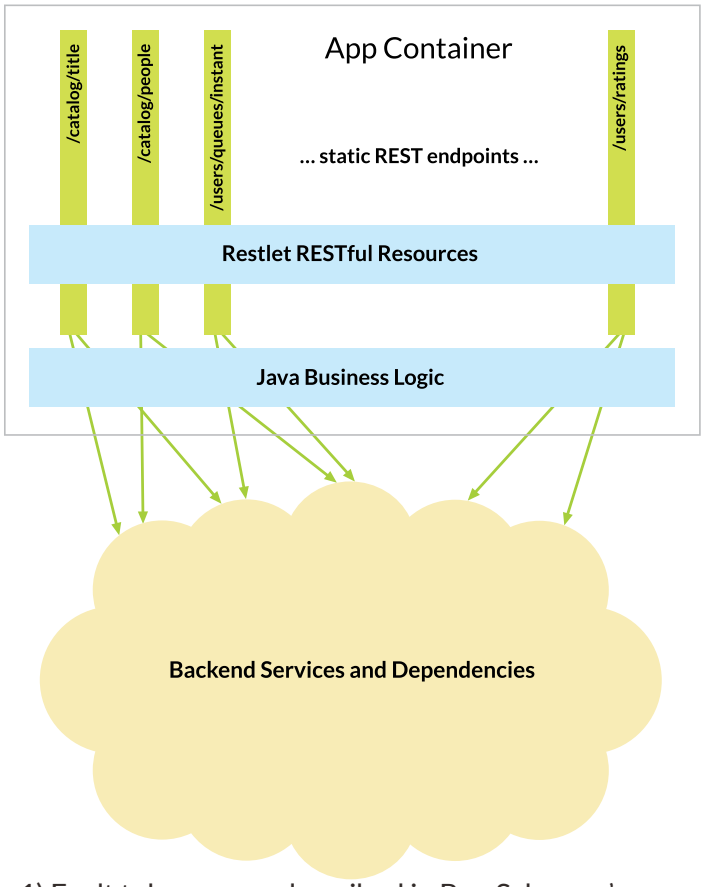
The Netflix API Optimization Story

by Jeevak Kasarkod

Over the past month, Netflix has been sharing stories from the API optimization effort that started a little over a year ago. The initial focus of the performance optimizations was to reduce chattiness and the payload size but as a result of the redesign, API development and operations was distributed and the services layer became asynchronous. The primary redesign feature was redefining the boundary of client and server responsibilities to enable fine-grained customizations such as content formatting to address the differences (memory capacity, document delivery and models, user interactions etc.) among the various client devices that connect to Netflix.
InfoQ caught up with Ben Christensen, Senior Software Engineer at Netflix to clarify and gather further design details.

InfoQ: Can you share a high level architecture of the Netflix API service from a year ago when the optimization effort was commenced and point out the problem areas?

Before the redesign, the Netflix API was a fairly typical RESTful server application. The image below demonstrates how the RESTful endpoints were implemented statically as “Restlet Resources” in a single codebase.



The major problem areas were:

1) Fault tolerance as described in Ben Schmaus’s blog post. Ben is the manager for the Netflix API platform.

Here are some of the key principles that informed our thinking as we set out to make the API more resilient.

1. A failure in a service dependency should not break the user experience for members.

2. The API should automatically take corrective action when one of its service dependencies fail.

3. The API should be able to show us what's happening right now, in addition to what was happening 15-30 minutes ago, yesterday, last week, etc.

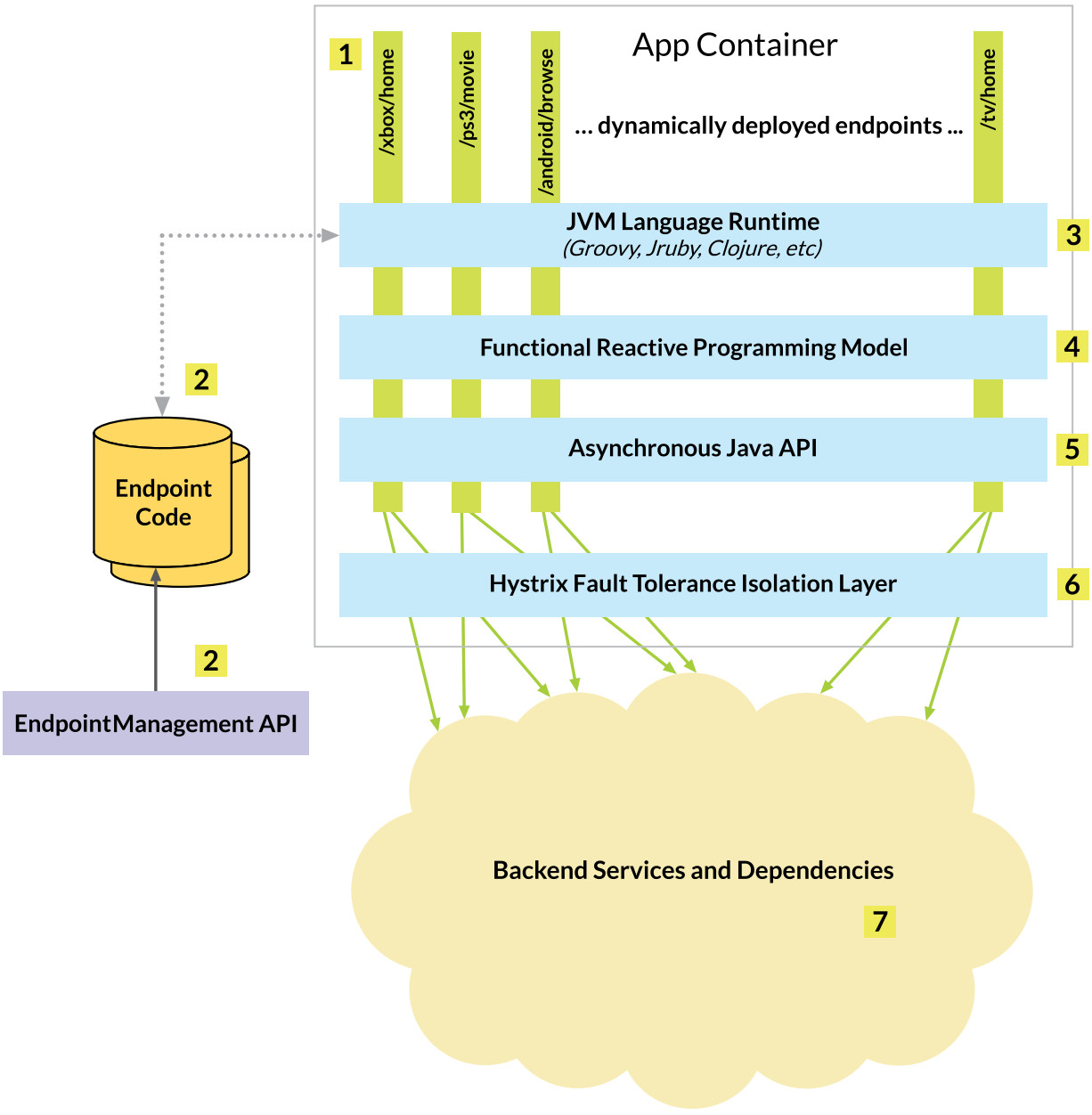
2) Poor match with device requirements that impacted performance and rate of innovation as described in a blog post by Daniel Jacobson, director of engineering for the Netflix API platform.

Netflix's streaming service is available on more than 800 different device types, almost all of which receive their content from our private APIs. In our experience, we have realized that supporting these myriad device types with a one-size-fits-all API, while successful, is not optimal for the API team, the UI teams or Netflix streaming customers.

In short:
- the RESTful API was a lowest-common-denominator solution
- it slowed innovation on each device since each endpoint needed to be used by all clients and thus feature development was synchronized across all of them
- it also slowed innovation because the API team became a bottleneck for client requests, resulting in prioritization and deferment of requests
- it resulted in chatty network-access patterns and inefficient payloads
- complicated feature flags for turning functionality and data types on and off spread through all RESTful endpoints to cater to the differences of each client.

InfoQ: What is the final state for the redesign?

The redesign is depicted below:



[1] Dynamic Endpoints

All new web-service endpoints are now dynamically defined at runtime. New endpoints can be developed, tested, canaried and deployed by each client team without coordination (unless they depend on new functionality from the underlying API Service Layer shown at item 5, in which case they would need to wait until after those changes are deployed before pushing their endpoint).

[2] Endpoint Code Repository and Management

Endpoint code is published to a Cassandra multi-region cluster (globally replicated) via a RESTful Endpoint Management API used by client teams to manage their endpoints.

[3] Dynamic Polyglot JVM Language Runtime

Any JVM language can be supported so each team can use the language best suited to them.
The Groovy JVM language was chosen as our first supported language. The existence of first-class functions (closures), list/dictionary syntax, performance and debuggability were all aspects of our decision. Moreover, Groovy provides syntax comfortable to a wide range of developers, which helps to reduce the learning curve for the first language on the platform.

[4 & 5] Asynchronous Java API + Functional Reactive Programming Model

Embracing concurrency was a key requirement to achieve performance gains but abstracting away thread-safety and parallel execution implementation details from the client developers was equally important in reducing complexity and speeding up their rate of innovation. Making the Java API fully asynchronous was the first step as it allows the underlying method implementations to control whether something is executed concurrently or not without the client code changing. We chose a functional reactive approach to handling composition and conditional flows of asynchronous callbacks. Our implementation is modeled after Rx Observables. Functional reactive offers efficient execution and composition by providing a collection of operators capable of filtering, selecting, transforming, combining and composing Observables. The Observable data type can be thought of as a "push" equivalent to Iterable, which is "pull". With an Iterable, the consumer pulls values from the producer and the thread blocks until those values arrive. By contrast, with the Observable type, the producer pushes values to the consumer whenever values are available. This approach is more flexible, because values can arrive synchronously or asynchronously.
The Observable type adds two missing semantics to the Gang of Four's Observer pattern, which are available in the Iterable type:

1. The ability for the producer to signal to the consumer that there is no more data available.
2. The ability for the producer to signal to the consumer that an error has occurred.

With these two simple additions, we have unified the Iterable and Observable types. The only difference between them is the direction in which the data flows. This is very important because now any operation we perform on an Iterable can also be performed on an Observable.

[6] Hystrix Fault Tolerance

All service calls to backend systems are made via the Hystrix fault-tolerance layer (which was recently open-sourced, along with its dashboard) that isolates the dynamic endpoints and the API Service Layer from the inevitable failures that occur while executing billions of network calls each day from the API to backend systems. The Hystrix layer is inherently mutlti-threaded due to its use of threads for isolating dependencies and thus is leveraged for concurrent execution of blocking calls to backend systems. These asynchronous requests are then composed together via the functional reactive framework.
We chose to implement a solution that uses a combination of fault-tolerance approaches:
1. network timeouts and retries
2. separate threads on per-dependency thread pools
3. semaphores (via a tryAcquire, not a blocking call)
4. circuit breakers

Each of these approaches to fault tolerance has pros and cons but when combined they provide a comprehensive protective barrier between user requests and underlying dependencies.

[7] Backend Services and Dependencies

The API Service Layer abstracts away all backend services and dependencies behind facades. As a result, endpoint code accesses "functionality" rather than a "system". This allows us to change underlying

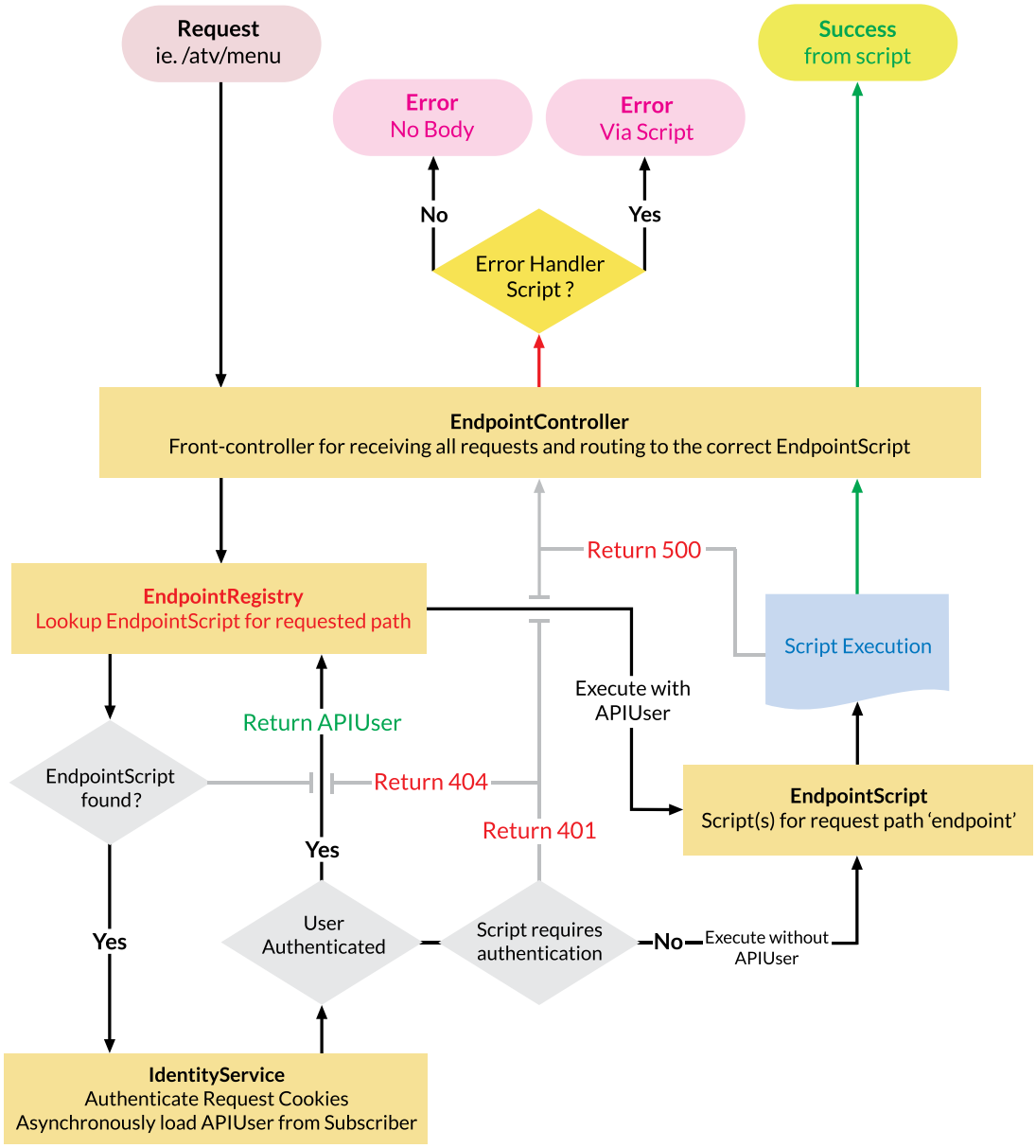
implementations and architecture with no or limited impact on the code that depends on the API. For example, if a backend system is split into two different services, or three are combined into one, or a remote network call is optimized into an in-memory cache, none of these changes should affect endpoint code and thus the API Service Layer ensures that object models and other such tight couplings are abstracted and not allowed to "leak" into the endpoint code.

InfoQ: Can you share any details around the choice of design patterns for the Client Adapter layer?

The client adapter layer was implemented as an HTTP request/response loop.
A client team implements an endpoint and receives the following:
- HTTPRequest
- HTTPResponse
- APIServiceGateway instance for accessing functionality
- APIUser representing the currently authenticated user
- APIRequestContext with environmental variables per request such as device type and geolocation

Since the endpoint is given the HTTP request/response, it can implement whatever style behavior it wishes:
- RESTful
- RPC
- Document Model + Path Language
- Progressive or document response

It also frees the implementation to use request/response headers, URI templating or request arguments and return data in whatever format suits them best such as JSON, XML, PLIST or even a binary format.
The image below shows the front controller flow of an incoming request to lookup the correct endpoint and execute it.



InfoQ: Why choose Cassandra as a code repository for endpoint code over other versioning tools?

First, because we needed global replication (across different AWS regions) and Cassandra offers that. Second, because it is an infrastructure component that Netflix has experience with and supports.
It does not replace standard version-control systems (Perforce, Git, etc) for actual development and versioning of code in that sense. Cassandra is the repository for deployed code used at runtime and we have several different environments such as dev, test, integration and prod each with different Cassandra clusters hosting the code deployed to that environment. The revisions of an endpoint stored within Cassandra allow for multivariate testing, canary testing, gradual rollouts and rapid rollbacks.

InfoQ: Does the dynamic language runtime host the client-adaptor endpoint code and split the request into granular Java API requests which are now asynchronous calls?

Yes, the dynamic language runtime hosts the client-adaptor code. The client adaptor (endpoint) is given an instance of an APIServiceGateway that provides access to asynchronous service calls.
The Netflix API takes advantage of Reactive Extensions (Rx) by making the entire service layer asynchronous (or at least appear so) - all "service" methods return an Observable<T>. Making all return types Observable combined with a functional programming model frees up the service-layer implementation to safely use concurrency. It also enables the service-layer implementation to:

- conditionally return immediately from a cache
- block instead of using threads if resources are constrained
- use multiple threads
- use non-blocking IO
- migrate an underlying implementation from network-based to in-memory cache

This can all happen without ever changing how client code interacts with or composes responses.In short, client code treats all interactions with the API as asynchronous but the implementation chooses if something is blocking or non-blocking.

InfoQ: What is a circuit breaker? How is it used to address fault tolerance?

Ben Schmaus summarized the circuit-breaker pattern in the blog post mentioned above.
We've restructured the API to enable graceful fallback mechanisms to kick in when a service dependency fails. We decorate calls to service dependencies with code that tracks the result of each call. When we detect that a service is failing too often, we stop calling it and serve fallback responses while giving the failing service time to recover. We then periodically let some calls to the service go through and if they succeed then we open traffic for all calls. If this pattern sounds familiar to you, you're probably thinking of the CircuitBreaker pattern from Michael Nygard's book Release It! Design and Deploy Production-Ready Software, which influenced the implementation of our service-dependency decorator code. Our implementation goes a little further than the basic CircuitBreaker pattern in that fallbacks can be triggered in a few ways:

1. A request to the remote service times out
2. The thread pool and bounded-task queue used to interact with a service dependency are at 100% capacity
3. The client library used to interact with a service dependency throws an exception

These buckets of failures factor into a service's overall error rate and when the error rate exceeds a defined threshold then we "trip" the circuit for that service and immediately serve fallbacks without even attempting to communicate with the remote service. Each service that's wrapped by a circuit breaker implements a fallback using one of the following three approaches:

1. Custom fallback - in some cases, a service's client library provides a fallback method we can invoke, or in other cases we can use locally available data on an API server (eg, a cookie or local JVM cache) to generate a fallback response
2. Fail silent - in this case the fallback method simply returns a null value, which is useful if the data provided by the service being invoked is optional for the response that will be sent back to the requesting client
3. Fail fast - used in cases where the data is required or there's no good fallback and results in a client getting a 5xx response. This can negatively affect the device UX, which is not ideal, but it keeps API servers healthy and allows the system to recover quickly when the failing service becomes available again.

About the Speaker

Ben Christensen works on the Netflix API Platform team responsible for fault tolerance, performance, and scale so millions of customers can access Netflix. Specializing in Java and through web and server-side dev, Ben gained an interest and skill in building maintainable, performant, high-volume systems. Before Netflix, Ben was at Apple in the iTunes division making iOS apps and media available.

READ THIS ARTICLE ONLINE ON InfoQ
http://www.infoq.com/news/2013/02/netflix-api-optimization

The Top 10 Internet of Things APIs

At a faster rate than ever before, APIs today are enabling the interconnectivity of everything from your oven to your refrigerator to your car. They're making the "Internet of Things" a reality, and best of all, many of them are open source, allowing developers to harness the Internet of Things for ever more creative purposes. Learn how today's top 10 most innovative APIs are changing the way you connect to the world around you.

IoBridge Data Feed API

IoBridge is an industry leader in Internet-enabling just about any product or device.

Helps users connect sensors to web services and receive alerts

Users can also control devices, like their thermostats, via web service

Smart power meter usage globally is expected to grow:

Evrythng API

Share, manage, and access information coming from your previously unconnected items.

Gives anything its own digital profile

Camera: Suggesting where and when to take great photos

Pill Bottles: Reminding you of refills

Bike: Tracking metrics and location

In 2008, the # of things connected to the web surpassed the # of humans on Earth

50 billion things will be connected by 2020

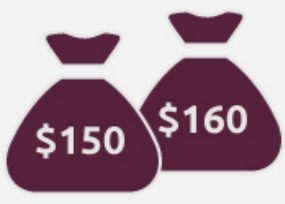
Exosite HTTP Data Interface API v1.0

The interconnectivity of "things" is only as good as the analytics that measure them.

Connects third-party software and the cloud

Functions include data upload, processing, analysis, visualization, and error alerts

The cloud computing market will reach $150-$160 billion by 2013



Cosm API

Collect real-time data and integrate user management tools into your Internet of Things venture.

2 in 3 IT managers will be utilizing real-time delivery and processing by 2015

Big data is growing by at least **50%** per year

42% of all data will be machine-generated by 2020

Up from **11%** in 2005

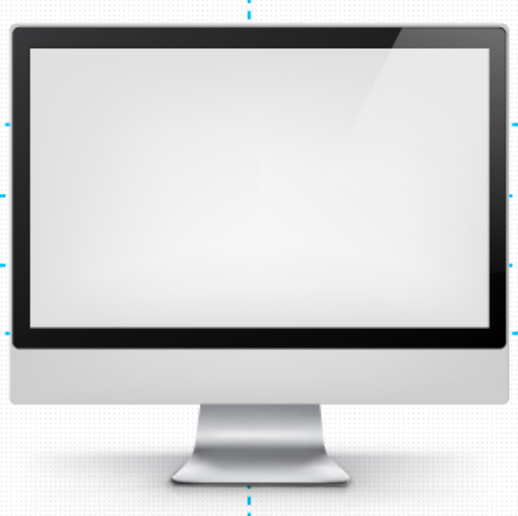
Solution: The Internet of Things offers tools to discover more about data

ThingSpeak Charts API

Once you have have all kinds of data from these "things," you need to make sense of it.

Enables instant data visualization

74% of users say the impact of data visualization on business insight is "high" or "very high"

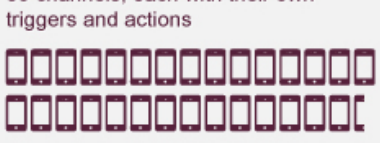


IFTTT API

The company name stands for "If This Then That," which is the type of recipe you can cook up with this API: For instance, if you take a photo on Instagram, then it can automatically sync with your DropBox.

A simple way to connect Google Drive and App.net

59 channels, each with their own triggers and actions



pvSense API

Monitors and works with photovoltaic installations.

Reduces costs by minimizing downtime and monitor performance

Connects subsystems that weren't originally designed to collaborate

Has monitored 300 installations in just 9 months in the solar market

Nest Learning Thermostat API

No more remembering to adjust the thermostat: Monitor and control your home thermostat remotely.

89% of programmable thermostats waste money and energy

Correctly programmed thermostats can cut a home heating and cooling bill by **20%**

Savings: ~$180 a year

Ninja Blocks API

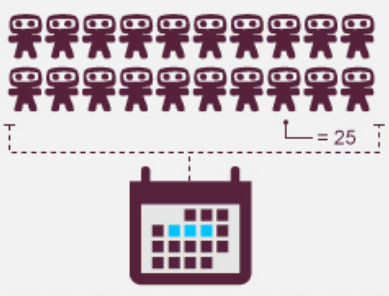
Ninja Blocks helps you leverage the interconnectivity of your devices.

Build a security system that texts you updates

Control temperature and humidity remotely

Install motion sensors that notify you if someone's at the door

>500 Ninja Blocks Kits sold within 3 days after launch



Fitbit API

Personal fitness and quantified self apps are gaining traction.

Fitbit users take an average of **43%** more steps every day

Health-measuring trends

80 million wearable sensors will be used for health-related functions by 2017

Integrated health technologies include:

- Sensor patches
- Wi-Fi scales
- Wristband sensors for exercise metrics
- Stretchable "tattoos" for wirelessly tracking body function

How can you use APIs to maximize your interconnectivity?

http://www.apihub.com/api/iobridge-data-feed-api
http://www.apihub.com/api/evrythng-api
http://www.apihub.com/api/exosite-http-data-interface-api-v10
http://www.apihub.com/api/cosm-api
http://www.apihub.com/api/thingspeak-charts-api
http://www.apihub.com/api/ifttt-api
http://www.apihub.com/api/pvsense-api
http://www.apihub.com/api/ninja-blocks-api
http://www.apihub.com/api/fitbit-api
http://www.apihub.com/nest/api/nest-thermostat-api
- See more at: http://blogs.mulesoft.org/top-10-internet-of-things-apis/#sthash.IrkNwDXA.dpuf

Securely Managed API Technologies Key to Fostering Market Innovation

by Atchison Frazer

Web services are relatively standard approaches enabled by application programming interfaces (APIs) that allow the capabilities of one company's website, application, or internal system to be accessed and used by another company or multiple entities by connecting directly to the underlying data. This approach offers distinct go-to-market velocity in terms of real-time innovation, but requires new standards in the way APIs are secured and managed and the nature in which APIs communicate between organizations at the B2B enterprise gateway level.

How do APIs foster business-model innovation? The most obvious way is through a mashup that leverages an already well-established API platform, such as Google Maps and location-based services. Another, perhaps not so obvious way, is for enterprise companies to create a handful of uber-APIs allowing enterprise applications to perform net-new functions, such as expanding market reach through new channels. Two examples of the most popular enterprise APIs spawning new mashups are Salesforce.com and DocuSign.

In all situations, the APIs must be managed, governed, and secured with essentially the same policies across technology platforms and domain entities, and must take into account the connectivity and transport protocols related to messaging of the relevant applications. The benefits of APIs are even more dramatic when organizations need to share applications and data via cloud or mobile platforms. However, leveraging corporate data through APIs in public/private cloud environments without proper identity, access, vulnerability, and risk management controls in place exposes those sources of data to potential exploits (see also How to Manage and Secure APIs with Unified Services Gateways).

Here are several examples of technology-driven innovation that also require the most rigorous standards of automated API security and compliance:

Retail marketing services: Near-field communications (NFC) allow for smartphone interaction with physical products via RFID; for example, Coke can generate a rewards coupon on the fly in the beverage aisle to a consumer who taps the smartphone's NFC function. The ability to personalize that interaction is dependent upon the consumer's profile consent and security authentication – that and many other NFC functions will be enabled by securely managed APIs. Additionally, NFC APIs deployed in retail channels will need to adhere to Payment Card Industry Data Security Standards (PCI DSS).

Digital video services: WebRTC, or real-time communications, is an emerging open-source video standard being developed by the World Wide Web Consortium (W3C). WebRTC enables browser-to-

browser applications for video chat, voice calling and peer-to-peer file sharing without third-party media players or plug-ins. These video apps, threatening existing real-time collaboration and communications standards such as WebEx and Windows Media Player, will require secure transport and authentication, which will happen in APIs. For meeting-based applications, the API will have to provide for proper authorized use and reuse, privileged executive communications by segmentation of duties, and real-time code controls.

Electronic medical-records services: EMR mandates by the government and necessitated by business demands are well-known; digitizing medical data universally will also create the opportunity for API-based services. The API can interface with an individual's medical profile to enhance both the business and service aspects of healthcare delivery: for example, a scheduling app matching a patient's personal medical needs with available services and specialists or a SMS app alerting the patient to new information or test results. The APIs that interface with EMR will need to incorporate the highest thresholds of data privacy and regulatory compliance.

Financial cloud services: More than 60 percent of securities and commodities trading is now comprised of so-called high-frequency trading (HFT). The analytics data - volumes of trades, successful algorithms, volume pricing, and comparative analysis - are valuable sets of information to the industry. For example, a very large institution may have up to 10,000 professionals who use HFT technologies requiring visibility into the same data at the same time. The ultimate benefit of this type of data-as-a-service platform is to model risk-management analysis for future trading opportunities, thus the API must safeguard the underlying data as robustly as any other financially regulated technology.

Data-sharing encryption services: A growing trend among both mid-market and large enterprises is to allocate certain aspects of their IT infrastructure to a public cloud provider, such as Amazon Web Services. The interaction between enterprise compute, storage and networking clusters and the Amazon Elastic Cloud (EC2) is an enterprise API that communicates with the EC2 API. However, the underlying storage topology, often formed in an open-standard such as Hadoop, doesn't always conform to the individual enterprise's data-storage or data-in-motion standard, and thus enterprise API gateways are required to orchestrate the proper acceptable-use and data-encryption standards in the cloud as if the service were being brokered by the enterprise itself.

API business agility is both a management and security issue. API managers must consider how much API governance can be automated to reduce potential for coding errors. On the flip side, security issues related to governance policies, date protection and compliance adherence all need mitigation to achieve optimal business agility. (for more in depth analysis, see The Reality of API Management Challenges).

Data protection: Unauthorized access to corporate files by developers using APIs carries the threat of corrupting data integrity. Whether it is malicious or accidental, an entity accessing corporate data through an API can potentially change that data and render the business out of compliance with any number of regulatory schemes requiring data integrity for auditing of financial transactions and more.

Compliance adherence: Let us use an example from a global manufacturer. This manufacturer develops an ordering system that is trusted to report the level of incoming sales orders to a host of third-party application developers.By doing this, the manufacturer has potentially put the accuracy of its financial reporting at risk. On a small "c" basis – compliance with internal controls and API governance – the company faces risks that its ERP system might be compromised, with resulting impacts to data integrity and availability. For big "C" or external pressure Compliance, the lack of control and API governance over the ordering system could throw the company out of compliance with Sarbanes-Oxley, which covers internal controls that ensure accuracy in financial reporting.

Risk mitigation: Keeping with the global manufacturer example, uncertainty about who is accessing the back-end ERP system and whether the right API version is running can wreak havoc on internal controls. For example, if a distributor can access a purchase order through an API and change its date – an act theoretically prohibited by an internal control – that control will be rendered deficient. The same risk presents itself across this control procedure. An API lacking proper governance can disrupt the integrity of revenue data.

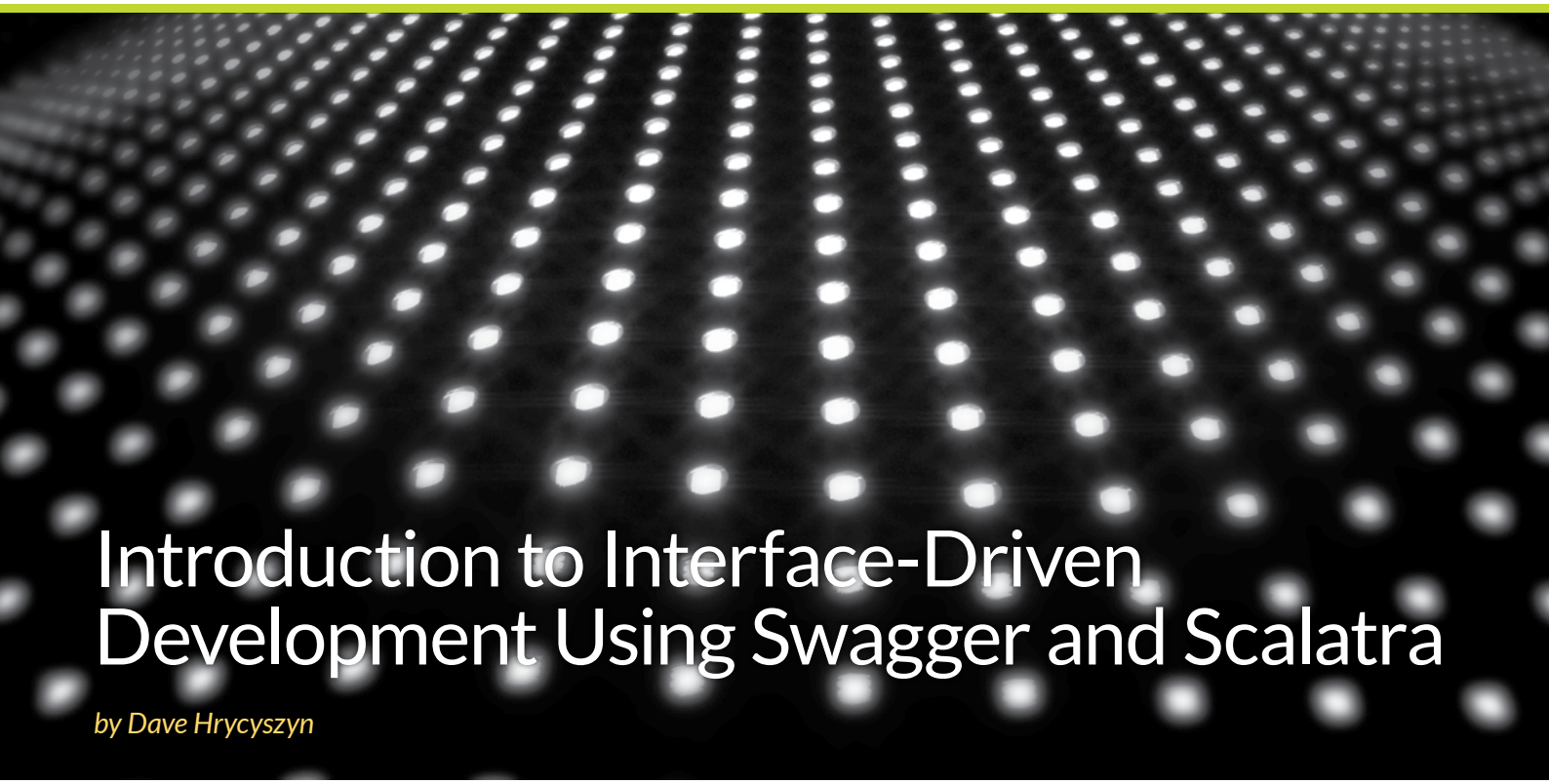
Thus, proper governance involves developing a set of API control objectives based on life cycle and related factors, such as key stakeholders cutting across business and IT roles (just as with ITIL for services and COBIT for processes). Any process that touches an API and is subject to a compliance audit, such as internal or data privacy controls, should meet the following three key governance objectives:

1. The API's life cycle should be subject to control so only permissible versions are in production: at the planning stage, development and test phases, in production and retirement.
2. Key stakeholders, such as lines of business executives, IT managers, information-security staff, and compliance auditors should have visibility into the dynamic, current state of the API. They should always be confident they are looking at the correct version within the proper life-cycle context.
3. APIs should also be subject to authentication and authorization processes to protect enterprise IT assets from misuse, threats to availability, or breaches of privacy, while ensuring monitored and throttled governance mandates are fulfilled.

About the Author

Atchison Frazer is a networking and services strategist with enterprise market-management experience ranging from start-ups Gnodal, ServGate and Fortinet to industry leaders Cisco and HP.

READ THIS ARTICLE ONLINE ON InfoQ
http://www.infoq.com/articles/Security-Managed-API



Introduction to Interface-Driven Development Using Swagger and Scalatra

by Dave Hrycyszyn

Since it began life a little over three years ago, the Scalatra web micro-framework has evolved into a lightweight but full-featured model-view-controller (MVC) framework with a lively community behind it. Scalatra started out as a port of Ruby's popular Sinatra DSL to the Scala language.

Since then, the two systems have evolved independently, with Scalatra gaining capabilities such as Atmosphere integration and Akka support.

It's been used by BBC Future Media for their Linked Data Writer API, managing large datasets in a scalable and manageable way, and has also been used by gov.uk.

One of the things that Scalatra's been most successful at is the construction of APIs. Over the past several years, REST APIs have become the lifeblood of the web. A relatively recent addition to Scalatra's capabilities is an integration with the Swagger toolset, which is produced by the folks at Wordnik.

What is Swagger?

Swagger is a specification that allows you to quickly define the functionality of a REST API using JSON documents. But it's more than just a spec. It provides automatic generation of interactive API docs, client-side code generation in multiple languages, and server-side code generation in Java and Scala. Although they're the most eye-catching component of the project and will impress users, the docs produced by Swagger are also great for fostering communication between API producers and consumers during the API design phase.

Let's take a look at how it all works. We'll build out a small REST API using Scalatra, then use Swagger to document our routes.
The easiest way to install Scalatra is by following the installation instructions at the Scalatra website. See the notes at the bottom for setting up Eclipse or IntelliJ if you use those IDEs, but you should be able to do this tutorial in any text editor.

Once you've got a JVM installed, along with cs, giter8, and sbt, you'll be able to generate a new Scalatra project.

Getting started

Type this at the command line:

```
g8 scalatra/scalatra-sbt --branch develop
```

You'll be asked a series of questions about your project. Answer like this:

```
organization [com.example]:
package [com.example.app]: com.example.swagger.
sample
name [scalatra-sbt-prototype]: flowershop
```

```
servlet_name [MyScalatraServlet]:
FlowersController
scala_version [2.9.2]:
version [0.1.0-SNAPSHOT]:
```

Hit to accept the defaults for the organization, scala_version, and version questions.
Once you answer the last question, a full Scalatra project will be generated. Let's check that it works. Change directory into the flowershop folder, and run Scala's simple build tool by typing:

```
cd flowershop
sbt
```

You'll need the latest sbt 0.12.1 for this. sbt 0.12.0 will work, just downgrade the version number in the file project/build.properties.
It can take several minutes for sbt to care of downloading all of Scalatra's dependencies when you're doing it for the first time, as you're getting a full Scala development environment, an embedded webserver (Jetty), Scalatra itself, and several companion libraries.
When sbt finishes setting everything up, you should be able to start the application by typing the following at the sbt prompt (which looks like a ">").

```
container:start
```

This will start Jetty on http://localhost:8080. Visit that URL in your browser, and you should see a Hello World application.

You don't want to have to manually recompile your app and restart Jetty whenever you make a code change, so type this at the sbt prompt:

This tells sbt to automatically recompile and reload the application whenever you change a file.
So, now we've got a controller for our flower shop. Let's set up a RESTful interface allowing us to browse flowers.

Open up the FlowersController.scala file found in src/main/scala/com/example/swagger/sample. What you start out with is a very simple generated controller. It's got a "hello world" action mounted on the application's root path get("/"), which can be accessed using an HTTP GET to the path "/". It's also got a way to handle 404s, and Scalate templating, which we don't need for our API.

Scalatra allows you to easily add functionality to your controllers by mixing in Scala traits to your class definitions. Let's slim things down a tiny bit by removing the ScalateSupport, since we don't need HTML templating support for our API. Delete the with ScalateSupport part of the class definition, so it looks like this:

```
class FlowersController extends
ScalatraServlet {
```

You can also remove the import scalate.ScalateSupport; it won't be needed. You can remove the notFound and get("/") actions as well. You'll be left with an empty Scalatra controller:

```
package com.example.swagger.sample
import org.scalatra._

class FlowersController extends
ScalatraServlet {

}
```

As you make these changes and save your files, check sbt's output in the terminal. You should see your source code automatically recompiling itself and reloading the application. The terminal output will turn green once you've slimmed down your controller. You've now taken all of the routes out of FlowersController, so you won't be able to see anything in your browser.

Setting up the data model

Let's get some data set up. Since we want to focus on learning Swagger, we won't attempt to actually persist anything in this tutorial. We can use Scala case classes to simulate a data model instead. If you'd like to learn how to set up ScalaQuery, a Scala ORM, check out Jos Dirksen's tutorial at SmartJava.

First, we'll need a flower model. Add a new directory in src/main/scala/com/example/swagger/sample and call it models. Then put the following code in a new file, Models.scala, in there:

Models.scala

```
package com.example.swagger.sample.models

// A Flower object to use as a faked-out data model
case class Flower(slug: String, name: String)
```

A Scala case class automatically adds getters and setters to the class definition, so we get a lot of functionality here without a lot of boilerplate.

Let's add some flower data.
Make a data namespace by adding a new data directory inside src/main/scala/com/example/swagger/sample. Then add a new file in that directory, calling it FlowerData.scala.

The contents of the file should look like this:

FlowerData.scala

```
package com.example.swagger.sample.data

import com.example.swagger.sample.models._

object FlowerData {

  /**    * Some fake flowers data so we can simulate retrievals.    */
  var all = List(
      Flower("yellow-tulip", "Yellow Tulip"),
      Flower("red-rose", "Red Rose"),
      Flower("black-rose", "Black Rose"))
}
```

That gives us enough to work with in terms of data to demonstrate our API's functionality.
Let's make a new controller action to retrieve flowers. Add some imports so that we get access to our models and data in the FlowersController class, by adding this at the top after the package definition:

```
// Our models
import com.example.swagger.sample.models._

// Fake flowers data
import com.example.swagger.sample.data._
```

Retrieving flowers

Now let's make our first API method, a Scalatra action which lets clients browse flowers. Drop this into the body of the FlowersController class:

```
get("/"){
  FlowerData.all
}
```

It doesn't look like much, but this is faking out data retrieval for us, by calling the all method of the FlowerData object we just defined. This is broadly equivalent to calling a static method in Java or C#, or a class method in Ruby.

If you take a look at http://localhost:8080/ in your browser, you should see the following result:

```
List(Flower(yellow-tulip,Yellow Tulip), Flower(red-rose,Red Rose), Flower(black-rose, Black Rose))
```

Scalatra has found all the flowers for us and returned the data. Looking at what we've got so far, though, it's not a very descriptive API. What resource is actually being retrieved? It's not possible to tell by looking at the URL. Let's change that.

Setting the mount path for better API clarity

Every Scalatra application has a file called ScalatraBootstrap.scala, located in the src/main/scala directory. This file allows you to mount your controllers at whatever URL paths you want. If you open yours right now, it'll look something like this:

```
import com.example.swagger.sample._
import org.scalatra._
import javax.servlet.ServletContext

class ScalatraBootstrap extends LifeCycle {
  override def init(context: ServletContext) {
    context.mount(new FlowersController, "/*")
  }
}
```

Let's change it a bit, adding a route namespace to the FlowersController:

```
import com.example.swagger.sample._
import org.scalatra._
import javax.servlet.ServletContext

class ScalatraBootstrap extends LifeCycle {

  override def init {
    context.mount(new FlowersController, "/flowers")
  }
}
```

The only change was to replace the "/*" mount point with "/flowers". Easy enough. Let's make sure it works. Hit http://localhost:8080/flowers in your browser, and you should see the same results as before:

```
List(Flower(yellow-tulip,Yellow Tulip),
Flower(red-rose,Red Rose), Flower(black-rose,
Black Rose))
```

This is a much more descriptive URL path. Clients can now understand that they're operating on a flower resource.

Automatic JSON output for API actions

Take a closer look at the output. What's going on here? Scalatra has converted the FlowerData.all value to a string and rendered its Scala source representation as the response. This is the default behaviour, but in fact we don't want things to work this way - we want to use JSON as our data-interchange format.
Let's get that working. Scalatra 2.2 includes some new JSON handling capabilities which makes this a snap. In order to use Scalatra's JSON features, we'll need to add a couple of library dependencies so that our application can access some new code. In the root of your generated project, you'll find a file called build.sbt. Open that up and add the following two lines to the libraryDependencies sequence, after the other scalatra-related lines:

```
"org.scalatra"% "scalatra-json"%
"2.2.0-SNAPSHOT",
  "org.json4s"%% "json4s-jackson"% "3.0.0",
```

The build.sbt file is somewhat equivalent to a maven pom.xml in Java or a Gemfile in Ruby, insofar as it keeps track of all your project's dependencies and can take care of downloading them for you. Restart sbt to download the new jars. You can do so by first hitting the "enter" key (to stop automatic recompilation), and then typing exit at the sbt prompt. Then type sbt again. You should see some messages telling you that sbt is downloading the new dependencies, and then you'll be back at the prompt. Start the container and recompilation again:

```
container:start
~; copy-resources; aux-compile
```

Add the following imports to the top of your FlowersController file, in order to make the new JSON libraries available:

```
// JSON-related libraries
import scala.collection.JavaConverters._
import org.json4s.{DefaultFormats, Formats}

// JSON handling support from Scalatra
import org.scalatra.json._
```

Now we can add a bit of magic to the FlowersController. Putting this line of code right underneath the controller class definition will allow your controller to automatically convert Scalatra action results to JSON:

```
// Sets up automatic case class to JSON output
serialization, required by
  // the JValueResult trait.
  protected implicit val jsonFormats: Formats
= DefaultFormats
```

Just like its Sinatra forebear, Scalatra has a rich set of constructs for running things before and after requests to your controllers. A before filter runs before all requests. Add a before filter to set all output for this controller to set the content type for all action results to JSON:

```
// Before every action runs, set the content
type to be in JSON format.
  before() {
    contentType = formats("json")
  }
```

Now mix JacksonJsonSupport and JValueResult into your servlet so your controller declaration looks like this:

```
class FlowersController extends
ScalatraServlet with JacksonJsonSupport with
JValueResult {
```

Your code should compile again at this point. Refresh your browser at http://localhost:8080/flowers, and suddenly the output of your / action has changed to JSON:

```
[{"slug":"yellow-tulip","name":"Yellow
Tulip"},{"slug":"red-rose","name":"Red
Rose"},{"slug":"black-rose","name":"Black Rose"}]
```

The JValueResult and JsonJacksonSupport traits which we mixed into the controller, combined with

the implicit val jsonFormats, are now turning all Scalatra action result values into JSON.

Making the flowers API searchable

Next, let's make our API searchable. We want to be able to search for flowers by name and get a list of results matching the query. The easiest way to do this is with some pattern matching inside the / in our controller.

Currently that route looks like this:

```
get("/"){
   FlowerData.all
 }
```

We can change it to read a query string parameter, and search inside our list of flowers.

```
/*    * Retrieve a list of flowers    */
  get("/"){
    params.get("name") match {
      case Some(name) =>FlowerData.all
filter (_.name.toLowerCase contains name.
toLowerCase())
      case None =>FlowerData.all
    }
  }
```

Scalatra can now grab any incoming ?name=foo parameter off the query string, and make it available to this action as the variable name, then filter the FlowerData list for matching results.

If you refresh your browser at http://localhost:8080/flowers, you should see no change - all flowers are returned. However, if you point your browser at http://localhost:8080/flowers?name=rose, you'll see only the roses.

Retrieving a single flower by its slug

The last controller method we'll create for the moment is one that retrieves a specific flower. We can easily retrieve a flower by its slug, like this:

```
  get("/:slug") {
    FlowerData.all find (_.slug ==
params("slug")) match {
      case Some(b) =>b
      case None =>halt(404)
    }
  }
```

Once again, we're using Scala's pattern matching to see whether we can find a matching slug. If we can't find the desired flower, the action returns a 404 and halts processing.
You can see the API's output by pointing your browser at a slug, e.g.http://localhost:8080/flowers/yellow-tulip

```
{"slug":"yellow-tulip","name":"Yellow Tulip"}
```

The nice thing is that since our before() filter runs on each and every action, the JSON format converter is still operating on our output. We get automatic JSON support with no extra effort. Sweet.

Interface-driven development using Swagger

At this point, we've got the beginnings of a REST API. It defines two actions, and offers a way for API clients to see what flowers are available in our flower shop. With our desired functionality achieved, we could stop here. But we're missing two things: human-readable documentation and client integration code. And without these, the API is a lot less useful than it could be.

An API is a way for machines to exchange data, but the process of designing and building an API also requires a lot of communication between people. The best API designs happen when the API's users have a way to get involved and detail what it is they need, and the API implementers boil those conversations down into an interface that works for the desired user stories or use cases. Historically, the fact that you've needed to use cURL or read WSDL to understand what an API does has severely limited the ability of non-technical people to participate in the API design process. The technical complexity of just making a connection has masked the fact that, at their core, the concepts inherent in a REST API are not particularly hard to understand.

Making the API's methods, parameters, and responses visible, in an engaging, easy to understand way, can transform the process of building REST APIs. The people at Wordnik, the word-meanings site, have built a toolset called Swagger, which can help with this.

Swagger is a bunch of different things. It's a specification for documenting the behaviour of a REST API - the API's name, what resources it offers,

available methods and their parameters, and return values. The specification can be used in a standalone way to describe your API using simple JSON files.

The Swagger resources file

If you want to, you can write a Swagger JSON description file by hand. A Swagger resource description for our FlowersController might look like this (don't bother doing this, though, because we'll see how to automate this in a moment):

```
{"basePath":"http://localhost:8080","swaggerV
ersion":"1.0","apiVersion":"1","apis":[{"path":"/
api-docs/flowers.{format}","description":"The
flowershop API. It exposes operations for
browing and searching lists of flowers"}]}
```

This file describes what APIs we're offering. Each API has its own JSON descriptor file which details what resources it offers, the paths to those resources, required and optional parameters, and other information.

A sample Swagger resource file

The descriptor for our flower resource might look something like this:

```
{"resourcePath":"/","listingPath":"/api-docs/
flowers","description":"The flowershop API. It
exposes operations for browing and searching
lists of flowers","apis":[{"path":"//","description"
:"","secured":true,"operations":[{"httpMethod":"G
ET","responseClass":"List[Flower]","summary":"Show
all flowers","notes":"Shows all the flowers in the
flower shop. You can search it too.","deprecated"
:false,"nickname":"getFlowers","parameters":[{"nam
e":"name","description":"A name to search for","re
quired":false,"paramType":"query","allowMultiple":
false,"dataType":"string"}],"errorResponses":[]}]}
,{"path":"//{slug}","description":"","secured":true,"
operations":[{"httpMethod":"GET","responseClass":
"Flower","summary":"Find by slug","notes":"Returns
the flower for the provided slug, if a matching
flower exists.","deprecated":false,"nickname":"find
BySlug","parameters":[{"name":"slug","description":
"Slug of flower that needs to be fetched","requi
red":true,"paramType":"path","allowMultiple":false
,"dataType":"string"}],"errorResponses":[]}]}],"mo
dels":{"Flower":{"id":"Flower","description":"Flo
wer","properties":{"name":{"description":null,"enu
m":[],"required":true,"type":"string"},"slug":{"des
cription":null,"enum":[],"required":true,"type":"s
tring"}}}},"basePath":"http://localhost:8080","swa
ggerVersion":"1.0","apiVersion":"1"}
```

These JSON files can then be offered to a standard HTML/CSS/JavaScript client to make it easy for people to browse the docs. It's extremely impressive - take a moment to view the Swagger Petstore example. Click on the route definitions to see what operations are available for each resource. You can use the web interface to send real test queries to the API, and view the API's response to each query.

Swagger language and framework integrations

Let's get back to the spec files. In addition to enabling automatic documentation as in the Petstore example, these JSON files allow client and server code to be automatically generated, in multiple languages. This means that unless you want to, you don't need to generate these JSON files by hand. There are integrations with a wide variety of frameworks, including ASP.NET, express, fubumvc, JAX-RS, Play, Ruby, and Spring MVC.

The framework integrations allow you to annotate the code within your RESTful API in order to automatically generate JSON descriptors, which are valid Swagger specs. This means that once you annotate your API methods, you get some very useful (and pretty) documentation capabilities for free, using the swagger-ui. You also get the ability to generate client and server code in multiple languages, using the swagger-codegen project. Client code can be generated for Flash, Java, JavaScript, Objective-C, PHP, Python, Python3, Ruby, or Scala.

Setting up the Scalatra Flower Shop with Swagger

Let's annotate our Scalatra shop with Swagger, in order to auto-generate runnable API documentation.

Add the dependencies

First, add the Swagger dependencies to your build.sbt file:

```
"com.wordnik"% "swagger-core_2.9.1"%
"1.1-SNAPSHOT",
"org.scalatra"% "scalatra-swagger"%
"2.2.0-SNAPSHOT",
```

Exit your sbt console and once again type sbt in the top-level directory of your application in order to pull in the dependencies. Then run container:start and ~; copy-resources; aux-compile to get code reloading going again.

You'll now need to import Scalatra's Swagger support into your FlowersController:

```
// Swagger support
import org.scalatra.swagger._
```

Auto-generating the resources.json spec file
Any Scalatra application which uses Swagger support must implement a Swagger controller. Those JSON specification files, which we'd otherwise need to write by hand, need to be served by something, after all. Let's add a standard Swagger controller to our application. Drop this code into a new file next to your FlowersController.scala. You can call it FlowersSwagger.scala

FlowersSwagger.scala

```
package com.example.swagger.sample

import org.scalatra.swagger.
{JacksonSwaggerBase, Swagger, SwaggerBase}

import org.scalatra.ScalatraServlet
import com.fasterxml.jackson.databind._
import org.json4s.jackson.Json4sScalaModule
import org.json4s.{DefaultFormats, Formats}

class ResourcesApp(implicit val swagger:
Swagger) extends ScalatraServlet with
JacksonSwaggerBase

class FlowersSwagger extends Swagger("1.0",
"1")
```

That code basically gives you a new controller which will automatically produce Swagger-compliant JSON specs for every Swaggerized API method in your application.
The rest of your application doesn't know about it yet, though. In order to get everything set up properly, you'll need to change your ScalatraBootstrap file so that the container knows about this new servlet. Currently it looks like this:

```
import com.example.swagger.sample._
import org.scalatra._
import javax.servlet.ServletContext

class ScalatraBootstrap extends LifeCycle {
  override def init(context: ServletContext) {
    context.mount(new FlowersController, "/
flowers")
  }
}
```

Change it to look like this:

```
class ScalatraBootstrap extends LifeCycle {

  implicit val swagger = new FlowersSwagger

  override def init(context: ServletContext) {
    context mount(new FlowersController, "/
flowers")
    context mount (new ResourcesApp, "/api-
docs")
  }
}
```

Adding SwaggerSupport to the FlowersController

Then we can add some code to enable Swagger on your FlowersController. Currently, your FlowersController declaration should look like this:

```
class FlowersController extends
ScalatraServlet with JacksonJsonSupport with
JValueResult {
```

Let's add the SwaggerSupport trait, and also make the FlowerController aware of Swagger in its constructor.

```
class FlowersController(implicit val swagger:
Swagger) extends ScalatraServlet with
JacksonJsonSupport with JValueResult with
SwaggerSupport {
```

In order to make our application compile again, we'll need to add a name and description to our FlowersController. This allows Swagger to inform clients of our APIs name and functions. You can do this by adding the following code to the body of the FlowersController class:

```
override protected val applicationName =
Some("flowers")
  protected val applicationDescription = "The
flowershop API. It exposes operations for
browsing and searching lists of flowers, and
retrieving single flowers."
```

That's pretty much it for setup. Now we can start documenting our API's methods.

Annotating API methods

Swagger annotations are quite simple in Scalatra. You decorate each of your routes with a bit of information, and Scalatra generates the JSON spec for your route.
Let's do the get("/") route first. Right now, it looks like this:

```
get("/"){
    params.get("name") match {
      case Some(name) =>FlowerData.all filter
(_.name.toLowerCase contains name.toLowerCase)
      case None =>FlowerData.all
    }
  }
```

We'll need to add some information to the method in order to tell Swagger what this method does, what parameters it can take, and what it responds with.

```
get("/",
    summary("Show all flowers"),
    nickname("getFlowers"),
    responseClass("List[Flower]"),
    parameters(Parameter("name", "A name to
search for", DataType.String, paramType =
ParamType.Query, required = false)),
    endpoint(""),
    notes("Shows all the flowers in the flower
shop. You can search it too.")){
    params.get("name") match {
      case Some(name) =>FlowerData.all filter
(_.name.toLowerCase contains name.toLowerCase)
      case None =>FlowerData.all
    }
  }
```

Let's go through the annotations in detail.
The summary and notes should be human-readable messages that you intend to be read by developers of API clients. The summary is a short description, while the notes should offer a longer description and include any noteworthy features which somebody might otherwise miss.

The nickname is a machine-readable key which can be used by client code to identify this API action - it'll be used, for instance, by swagger-ui to generate method names. You can call it whatever you want, but make sure you don't include any spaces in it, or client code generation will probably fail: "getFlowers" or "get_flowers" is fine, "get flowers" isn't.

The responseClass is essentially a type annotation, so that clients know what data types to expect back. In this case, clients should expect a List of Flower objects.

The parameters details any parameters that may be passed into this route, and whether they're supposed to be part of the path, post params, or query string parameters. In this case, we define an optional query string parameter called name, which matches what our action expects.

Lastly, the endpoint annotation defines any special parameter substitution or additional route information for this method. This particular route is pretty straightforward, so we can leave this blank.
We can do the same to our get(/:slug) route. Change it from this:

```
get("/:slug") {
    FlowerData.all find (_.slug ==
params("slug")) match {
      case Some(b) =>b
      case None =>halt(404)
    }
  }
```

to this:

```
get("/:slug",
    summary("Find by slug"),
    nickname("findBySlug"),
    responseClass("Flower"),
    endpoint("{slug}"),
    notes("Returns the flower for the provided
slug, if a matching flower exists."),
    parameters(
      Parameter("slug", "Slug of flower that
needs to be fetched",
        DataType.String,
        paramType = ParamType.Path))) {
```

```
    FlowerData.all find (_.slug ==
params("slug")) match {
      case Some(b) =>b
      case None =>halt(404)
    }
  }
```

The Swagger annotations here are mostly similar to those for the get("/") route. There are a few things to note.

The endpoint this time is defined as {slug}. The braces tell Swagger that it should substitute the contents of a path param called {slug} into any generated routes (see below for an example). Also note that this time, we've defined a ParamType.Path, so we're passing the slug parameter as part of the path rather than as a query string. Since we haven't set the slug parameter as required = false, as we did for the name parameter in our other route, Swagger will assume that slugs are required.

Now let's see what we've gained.
Adding Swagger support to our application, and the Swagger annotations to our FlowersController, means we've got some new functionality available. Check the following URL in your browser:
http://localhost:8080/api-docs/resources.json

You should see an auto-generated Swagger description of available APIs (in this case, there's only one, but there could be multiple APIs defined by our application and they'd all be noted here):

```
{"basePath":"http://localhost:8080","swaggerV
ersion":"1.0","apiVersion":"1","apis":[{"path":"/
api-docs/flowers.{format}","description":"The
flowershop API. It exposes operations for
browing and searching lists of flowers"}]}
```

Now for the wonderful part.
Browsing your API using swagger-ui
If you browse to http://petstore.swagger.wordnik.com/, you'll see the default Swagger demo application - a pet store - and you'll be able to browse its documentation. One thing which may not be immediately obvious is that we can use this app to browse our local Flower Shop as well.

The Pet Store documentation is showing because http://petstore.swagger.wordnik.com/api/resources.json is entered into the URL field by default.

Paste your Swagger resource-descriptor URL - http://localhost:8080/api-docs/resources.json - into the URL field, delete the "special-key" key, then press the "Explore" button. You'll be rewarded with a fully Swaggerized view of your API documentation. Try clicking on the "GET /flowers" route to expand the operations underneath it, and then entering the word "rose" into the input box for the "name" parameter. You'll be rewarded with JSON output for the search method we defined earlier.
Also note that the swagger-ui responds to input validation: you can't try out the /flowers/{slug} route without entering a slug because we've marked that as a required parameter in our Swagger annotations. Note that when you enter a slug such as "yellow-tulip", the "{slug}" endpoint annotation on this route causes the swagger-ui to fire the request as /flowers/yellow-tulip.

If you want to host your own customized version of the docs, you can of course just download theswagger-ui code from Github and drop it onto any HTTP server.

A note on cross-origin security

Interestingly, you are able to use the remotely hosted documentation browser at http://petstore.swagger.wordnik.com to browse an application on http://localhost. Why is this possible? Shouldn't JavaScript security restrictions have come into play here?

The reason it works is that Scalatra has Cross-Origin Resource Sharing (CORS) support built-in, allowing cross-origin JavaScript requests by default for all requesting domains. This makes it easy to serve JS API clients - but if you want, you can lock down requests to specific domains using Scalatra's CorsSupport trait. See the Scalatra Helpers documentation for more.

Conclusion

Without much in the way of boilerplate code, you've now constructed a simple REST API, set up model-class-to-JSON output functionality and auto-generated API documentation by annotating your Scalatra routes with Swagger information.
This is one way to use Swagger, but the Wordniks use Swagger differently: rather than starting with the API and using Swagger to just generate the docs,

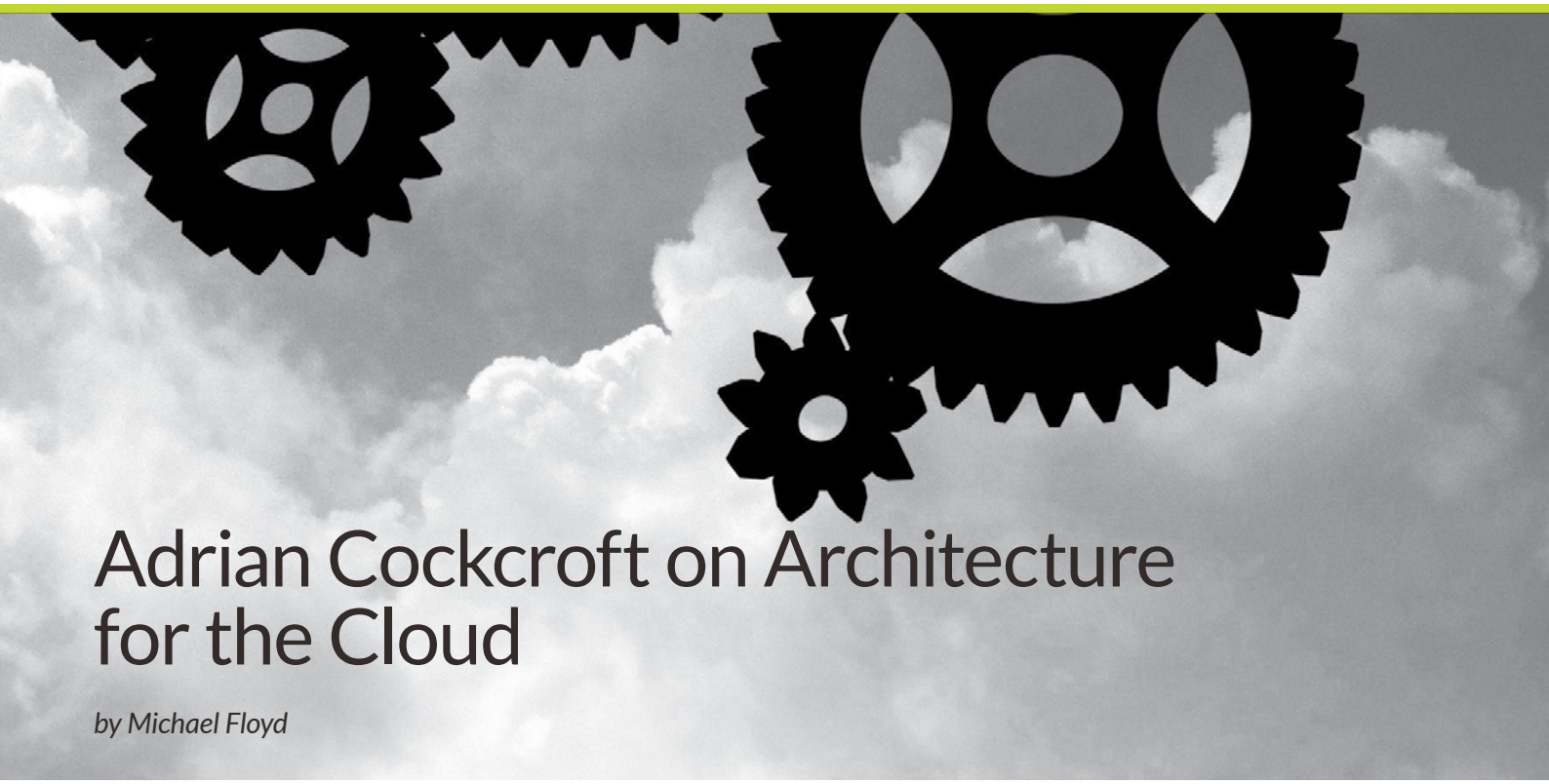
they start by writing the JSON descriptor files by hand. They then look at the API using the HTML docs browser, and have all the parties who are interested in the API sit down and discuss what's needed. After changing the JSON files based on the discussion, they use swagger-codegen to generate the client and server code. This is called interface-driven development, and it's well worth a look. With its ease of use, multi-framework integration and innovative way of involving people in the design process, Swagger is at the forefront of REST API construction tools.

The code

You can download and run a working version of this application by installing Scalatra as detailed at the start of this tutorial, doing a git clone https://github.com/futurechimp/flowershop.git and running sbt in the top-level of the project.

About the author

Dave Hrycyszyn is technical director at Head London, a digital-innovation agency in the UK. He is passionate about APIs and application architectures, and is a member of the team working on the Scalatra micro-framework, which has been used by LinkedIn, the BBC, the Guardian newspaper, and gov.uk. He has a keen interest in organizations and institutions, and the interplay between social structures and software.



Adrian Cockcroft on Architecture for the Cloud

by Michael Floyd

In this interview, Adrian Cockcroft, the architect for Netflix's cloud systems team discusses how Netflix combines 300 loosely coupled services across 10,000 machines. An interesting revelation is that they fully embrace continuous delivery and each team is allowed to deploy new versions of their service whenever they want.

Hi, I'm Michael Floyd with InfoQ and I'm here at QCon San Francisco 2012 with Adrian Cockcroft. Adrian is speaking about architecture for the cloud. Adrian, what's your background?

I've been at Netflix for about five years and I am currently the architect for the cloud-systems team, basically. Before that I spent some time at eBay and then it was Sun Microsystems for a long time working on performance-related things mostly.

You're doing what exactly at Netflix?

I am the overall architect. The way Netflix does architecture is to implement patterns we hope our developers will follow. You can do that by having a big stick and an architecture review board forcing everyone to do something a certain way. We do it by creating patterns and tooling that encourage people to take the easy way and follow the architecture. So it's much more of an emergent system, the architecture emerges from everything that we do, but we are building out lots of tooling that gives us highly available patterns. The talk I am going to give is about the patterns we use to give us highly available services running on a cloud infrastructure that is built out of commodity components that come and go, talk about the kind of outages we get and the way we deal with it and the way we replicate our data in services.

Can you just give us an idea of the scope of the problem that Netflix is trying to solve? For instance API requests - how many API requests would you serve in a day?

We're doing a few billion API requests a day. I think we've published the numbers recently. We do more than a billion hours a month of streaming. Some numbers published this morning basically put Netflix at 33% of the peak-time Internet bandwidth streams to homes in the US. So we're consuming the lion's share of the actual bandwidth that people use in practice, streaming videos to them. That generates a lot of traffic. It's terabytes of traffic, many terabytes, and the requests that support that (traffic) come from currently about 30 million customers of our streaming product. That's a large sum of customers. You can calculate they're watching an hour or two a night each. We have quite a lot of people watching at any one time, so lots of traffic. We run something in the order of 10,000 machines. That's the sort of scale

we're at for handling that traffic.

I was going to ask you about the infrastructure. Can you give us an idea of the scope of that or is that it? Yes, it's all running on Amazon's AWS server services. We have a fine-grained model where basically every engineer or group of engineers develops just one little service with its own REST interface. They all talk to different backends that they have their data in. And we've got maybe 300 of those services - I don't actually keep track of them that well because people keep inventing new ones. So you hit our website and it flows through into this large number of very discrete services. Each of those services is scaled horizontally so we have at least three of anything so that we can lose one and still have two left, and then we have up to 500 or 1000 of our biggest services that some of the big front ends get to that kind of scale and the total number of machines we have there varies between 5 and 10 thousand. It order-scales so at peak traffic it's bigger than when it shrinks and how the numbers work out depends on what you are counting.

So that sets the stage pretty well. We were talking about architectural patterns for high availability, specifically. Can you share some of those patterns with us?

Sure. If you care about high availability then you've got to start replicating. And there's two basic ways of replicating. One way is to have a master and then slaves. The problem with master-slave architecture is if you can't reach the master or it goes down, the slave isn't quite sure whether it's okay to take an update or serve something.

The other alternative is a quorum-based architecture, where two out of three have to be working, so everything we do is triple-replicated. We've got three copies of every service, three copies of all the data in the back end and when you write, you write three copies and as long as two of the copies are still there you can run everything. So it's triple-redundant system. That's the basic pattern. We use the availability zone as our basic unit of deployment. I think of it as a data center, so all our data is written three times, but not to three machines. Those machines are in three different data centers so that we can lose an entire data center, an entire zone, and everything keeps running. That's the basis of the architecture, with a lot of details about how you distribute traffic across that and how you route traffic within them, but that's the basic architecture.

You mentioned continuous deployment. How does that lead to high availability and scalability?

Continuous deployment is really more about the agility and the productivity of the developers. You can either gather all your code into a sort of train model and then every two weeks or whatever you say "Okay, here's a new train." You give it to QA and then they test it for a week and then you launch it and maybe they can figure out what's wrong with it in time before the launch. As the team gets bigger and bigger, that train model stops working. It works for a small team; it's the way we used to work five years ago. The way we work now is completely decoupled. So any developer that owns a service can rev that service as often as they want and whenever they want.

Generally, we try not to change things on Fridays because we like to have quiet weekends. Other than sort of general guidelines like that, if you've got a brand-new thing you've just deployed, you might deploy 10 times in a day just because you are iterating to work through issues that you are finding and you are running a small amount of traffic into it. If it's a service that's mature and it's been sitting around for a long time, it may not be changed for months. Because we've got these fine-grained services, everything is on REST interface and it does one single function, piece of business logic - you can change those as fast as you want. You still have to manage the dependencies and the interfaces versioning and things like that but eventually we figured out how to solve those problems too.

So, deployment wise, how has this really benefited?

How the availability side works? So, we do something a little different to most people since we're running in the cloud. If I have 500 machines running a service, when I want to change it, I don't update those 500 machines. I create 500 new machines running the new code. First of all, I create one and see if it works, smoke test it, we call it the canary pattern, which is fairly well known: you have this thing and if it dies you keep putting it up until you get a good one. But once

you decide you want to deploy it in full, you basically tell the Amazon auto-scaler "Please, make 500 of these machines." It takes the machine image, which contains everything like a root-disk sort of clone, and makes 500 of them.

It takes eight minutes to deploy 500 machines on Amazon. It's a number I've measured recently and then it takes a while for our code to start up, so in a matter of minutes you've got 500 new machines running alongside your old machines. You switch the traffic over to it, watch it carefully, see if everything worked fine. You typically leave the old machines running for a few hours through peak traffic because sometimes systems look okay when you first launch them and then maybe exhibit a memory leak a few hours later or they fall over under high traffic or something like that. We run side by side and we actually automatically clear out the old machines eight hours later or something like that. So if there's an issue with anything, we can in seconds switch all the traffic back to the old machines, which are warmed up, running, just sitting there ready to take all that traffic. So it means that we can rapidly push new code with a very low impact and if we push something that is broken, we can tell immediately and switch it back. So it's about not trying to sort of legislate against everything that could go wrong, it's about being very good at detecting and responding extremely quickly when something isn't quite right.

Would you agree that PAS, platform as a service, for the developer is about giving developers tools to be more productive?

Yes. We built our own PAS specifically to make our developers more productive. When we were building it, it was just our platform and the things we needed to build. As people started talking more about PAS as a thing, we looked at what we built and we decided it's a specific kind of PAS: it's a large-scale, global PAS design for a large development team. A lot of the PAS out there, like Heroku or Cloud Foundry, are really designed for a small team and a small number of developers. You can run it on a VM on your laptop. Our system doesn't make sense until you've got several hundred machines running on Amazon, which would be the smallest deployment that would make sense for us, but it supports scale to tens of thousands of machines, which is the problem we were trying to solve.

So I think what we built is an enterprise-scale PAS and as we were working through it, we started open-sourcing various pieces of it. Our foundation data store is Cassandra, which is already an Apache project. We didn't like the Java client library that's called Hector - who is Cassandra's brother I think in Greek mythology, my Greek mythology keeps getting mixed up - so we rewrote our own client library which is called Astyanax, who is Hector's son, and basically at that point, we had a client library and we decided we should just share it. We started sharing more and more of our code and the code we used for managing Cassandra. Those were the first pieces. We are now in the process of releasing basically the whole platform. All of our platform infrastructure code is either things we consume externally that everyone already has access to or it's things we've put out there and things that we are in process of putting out there.

We launched a service yesterday called Edda, which has something to do with Norse gods' stories or something like that. It's an interesting data store in that it collects the complete state of your cloud infrastructure, so you can go to Amazon and say "Give me all my instances" with one API call. It gives you a list of instances, then you can iterate over those instances saying "Describe, describe, describe, describe", you get back a JSON object.

It continuously does that on a one-minute update and stores the complete history, a versioned history of all those objects. So what you have is for all of the 10 or more different entities that we track on AWS, we have a versioned history going back as far as we want of the state of everything. So I can say exactly what state was everything in three days ago when we had this problem and I can basically interrogate all the instances that are no longer there or I can see the entire structure, everything. And we actually have some tools which clean up things. We sort of do garbage collection on instances, Amazon entities, and things like auto-scale groups. We have a Janitor Monkey that talks to this data and figures out what is old and not being used. In order to figure out what's old, it has to have a historical view of the world. So that's a new service, we just put it up on AWS. It's lots of JSON objects written in Scala, which is a first for us. Most of our code up to this point has been written in Java. It's the first thing we've done in Scala and has a MongoDB back end because that JSON object

manipulation is just the ideal use case for that tool. Most of our high-scale, customer-facing applications are running on Cassandra for its highly available, highly scalable side of things. So that's an example of the kind of thing we're releasing over time.

Making the move to Scala is kind of a big deal. What were the preparatory steps that you took, were there governance things involved, how did this get pushed through to Netflix basically, was there somebody championing Scala?

We don't have governance. We have an engineer that decided to do something in Scala. Basically, there is a series of patterns that are well-supported by tooling. If you run something that runs in a JVM, Scala is similar enough that you run it in a JVM, if you got a language that runs in the JVM that can call Java - Java files basically sit on top of our platforms - that's a relatively small divergence from a platform. And we have a few people that are playing around with Scala; we built some other internal tools with it. So it's one of those emergent things.

We don't stop anybody from trying anything new. If you want to write something in Clojure or anything, you're welcome to try. The problem is that you yourself then end up supporting the platform and the tooling and most engineers will figure out eventually that they don't want to support all the platform and tooling required to support a radically new language.

For example, I think the Go language is really interesting but we have really no way to support it in a very JVM-based environment. So that is going to sit on the side until it becomes a significant enough tool in which enough developers are interested and we decide to figure out how perhaps to increment and add it in. We have other people who think that Scala is interesting. There's no big rule, there's no big process, there are just well-beaten paths with tooling and support and then there are other things that you can do on the side.

The other main language we support is Python, mostly for our ops-automation infrastructure. I don't think we've released any open-source Python projects yet, but we will be doing some. But even there, the back-end platform interfaces that are basically PAS implements are not particularly slow-moving. They are moving forward with functionality really fast. So the challenge then is you've got two language bases and Python is trying to keep up with all the Java-based languages, Java-based platform pieces. So it's actually quite difficult to support more than one very distinct language in a platform, in particular if it's evolving really rapidly. The point of everything is to evolve things rapidly.

So, benefits of making this transition to Scala, what are the benefits that you guys have seen?

So far, I think probably a happy engineer.

So, there were no language features or anything?

I haven't actually had that discussion. We have a few people who have written things in Scala and they seem happy to do it. We're trying to figure what is it good for. I mean obviously every language has its own thing it's particularly good at. So there are some people that have figured out this is a good thing to write in Scala. Then, 99% of our developers code is probably in Java. A little bit of it in Python. So at this point I regard it as experimental but anything that will run in the JVM is basically going to fit into the architecture relatively easily.

You expose certain APIs and other tools. Can you tell us what those are, maybe tell our audience where they can go find out more about? You're on Github, right?

Yes, so github.netflix.com is where we have all our code. We have, I've lost count, 10 or 15 projects up there now. We have at least 10 more that are in flight. We had a big push to try to get everything done for the big Amazon event that is happening at the end of this month, AWS re: Invent, which is the week after Thanksgiving in Vegas. It's a huge show. We have 15, I think, Netflix presentations there all together. Reed Hastings is giving a key note, so it's going to be a massive show for Amazon and Netflix is very involved in this. So what we are trying to do is get as much of our platform out as possible before that show and then at the show gather anybody that's interested to sort of have a little "Are you interested in the Netflix platform?" user-group meeting.

We are still feeling our way, and we don't know how

that's going to work out. We know we have quite a few people. The nice thing about Github is you can see how many people have taken copies of your code and are playing around with it. We've got contributions from people. But the interesting thing about having open-sourcing as a strategy is that it really helps clean code up. This Edda thing was written ages ago and the guy that originally wrote it, I think, left or is working on something else. We hired a new guy and we said, "Well, we need to open-source this thing. Can you figure out how to clean it up for open source?" He looked at it and said, "Nah, I'm going to rewrite it from scratch." And that's why it's rewritten in Scala. It's a version 2 of this thing. The version 1 was something that was thrown together fairly rapidly, iterated over some period of time, so we took that to be the prototype, did a nice clean version of that and then put that out into the open-source project.

So if you're a manager trying to get your engineers to clean up and document their code, probably the best thing you can say is, "Hey, would you like to open-source that?" because then there's a lot of peer pressure. "My code is going to be in public! Anyone can read it!" You know that Github is sort of your future resume if you're a developer and you want your name associated with good projects and good quality code. So that's acted as an extremely beneficial sort of side effect. Internally, we found other teams using some of the things that we've open-sourced for other projects that are not part of the core architecture of what we are building for supporting the customer-facing streaming product. We have a team that's working on movie encoding. It's a completely separate system that pulls in all the main movies from the studios and encodes them. They are now using a project that we open-sourced whereas they could not use the internal version of it because it was too tied in to dependencies in our architecture. So by open-sourcing it as separate projects, you end up componentizing the platform in a way that makes it more consumable and benefits internally, as well as externally.

Is there any one of these projects that is kind of a highlight or a favorite of yours personally?

Well, the user interface is the one most people see. You can talk abstractly about the platform but when you actually see a GUI with an enormous number of options and features on it, you actually realize that there is something there. And that product is called Asgard. Asgard is the home of the gods, I think, control the clouds, it's got a big hammer. Thor.

Yes. So we have that group that's building those tools that seems to have gone on a Norse-mythology naming scheme. We have several names. Every engineer gets to pick a project name. Edda is related to that: it's the stories that describe what the gods did or something like that, so the story is about the cloud. So that's how they came up with that name. And then there's an Odin, which is a workflow manager system for something.

But Asgard is probably the best place to start if you want to see what this looks like, it's a replacement for the Amazon console, so it's a Groovy Grails app, very sophisticated. It's been developed over a long period of time, with some talented engineers. A lot of engineering efforts go into it. It's what everyone at Netflix uses to deploy code, to see what state everything's in, to configure things.

So that's the core front end. We have configuration services, we have a dependency injection system based on Guice, the Google Guice project called Governator. We have some things based off of ZooKeeper, Curator and Exhibitor. We have things for managing Cassandra. Priam is Cassandra's dad, so that's what we used for managing Cassandra. I have to have a slide to keep track of all these things, I have too many, too many projects. But there's more stuff, about once a week we put out new projects. We always do a Tech Blog post at techblog.netflix.com where we put posts up that describe what we're doing in this space. On Tech Blog, we talk about not only open-source projects but technical things, like the personalization algorithms and if there's been an outage, a significant outage, then we talk about what we learned from the outage and responses and things like that. So, I encourage people to look at techblog.netflix.com and if you are looking for a whole lot of Java code, at Github.

About the Speaker

Adrian Cockcroft is the director of architecture for the Cloud Systems team at Netflix. He is focused on availability, resilience, performance, and measurement of the Netflix cloud platform, and is the author of several books while a Distinguished Engineer at Sun Microsystems: Sun Performance and Tuning; Resource Management; and Capacity Planning for Web Services.

READ THIS ARTICLE ONLINE ON InfoQ
http://www.infoq.com/interviews/Adrian-Cockcroft-Netflix