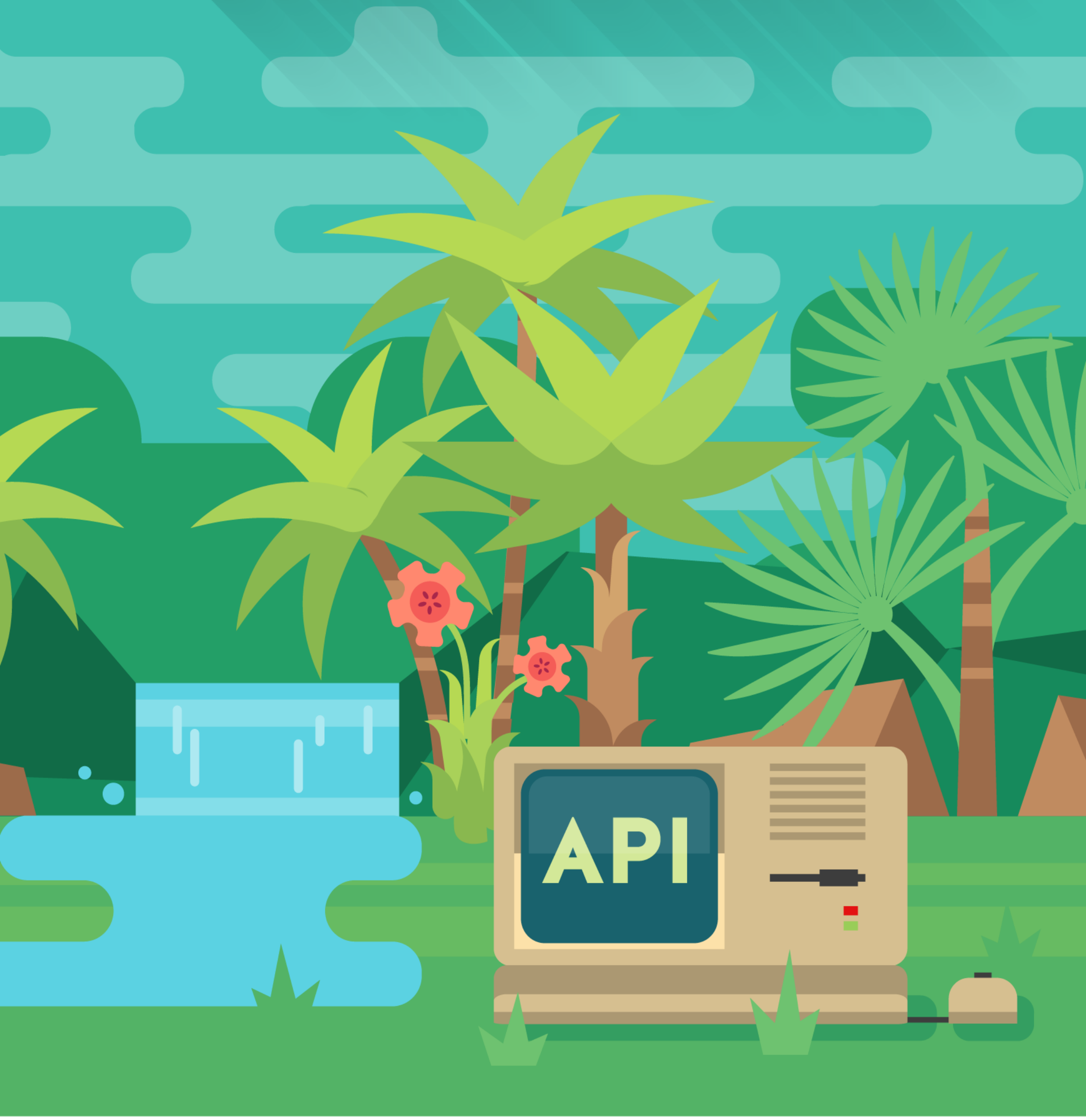
API Design on the Scale of Decades



AUTHORS

Bill Doerrfeld
Kristopher Sandoval
Art Anthony
Chris Wood

NORDIC **APIS**
nordicapis.com

API Design on the Scale of Decades

Learn How to Architect and Design Long-lasting APIs

Nordic APIs

Tweet This Book!

Please help Nordic APIs by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Check out ”API Design on the Scale of Decades” - the latest eBook by the @NordicAPIs team

The suggested hashtag for this book is #APIdesign.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#APIdesign

This book is dedicated to the speakers, attendees, and sponsors that continually make Nordic APIs events wonderful!

Contents

Preface

APIs on the Scale of Decades

Roy Fielding, the creator of the REST standard for API design, once described REST as:

> "*software design on the scale of decades: every detail is intended to promote software longevity and independent evolution. Many of the constraints are directly opposed to short-term efficiency.*"

Essentially, he acknowledged how software developers often execute short term design without long-term design in mind, which causes inadaptability as technology evolves.

In October 2016 Nordic APIs held it's annual Platform Summit centered on this theme, bringing together API industry thought leaders to share their insights on what it means to architect and design Application Programming Interfaces on the scale of decades. The event detailed RESTful design techniques for longevity, operational components of sustaining an API, and topics like DevOps, microservices architecture, developer relations, and new business methodologies for supporting an API platform.

To create *API Design on the Scale of Decades*, we gathered the top 15 most watched sessions from the Summit that were relevant to **API Design**, and drafted companion chapters to dive into each topic. This unique volume thus contains a holistic assortment of insights that our followers have found to be the most current, mission-critical ideas for sustaining an API platform.

For API designers and architects, this eBook release presents a convenient way to tap into a wide range of knowledge that we've been collating on the blog over the last few months. Guided by some of the most significant Nordic APIs presentations, it outlines some of the most helpful advice we've published to date.

As we close the chapter on the last Platform Summit we begin to plan for the future. Stay tuned for updates from Nordic APIs on our 2017 event, which will be on the theme of **API Scalability**. On that note, if you would like to join the legion of past API speakers, consider submitting a session here!

So, please enjoy *API Design on the Scale of Decades*, and let us know how we can improve. If you haven't yet, consider following us, and signing up to our newsletter for curated blog updates and future event announcements.

Thank you for reading!

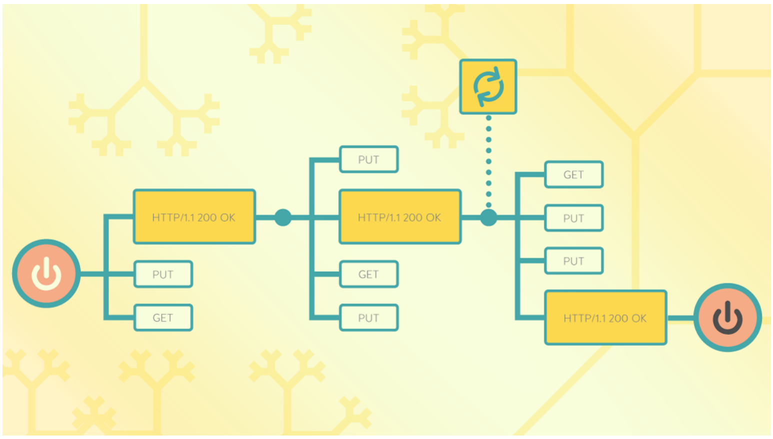
– Bill Doerrfeld, Editor in Chief, Nordic APIs

Connect with Nordic APIs:

Facebook | Twitter | Linkedin | Google+ | YouTube

Blog | Home | Newsletter | Contact

Designing a True REST State Machine



There are a lot of misconceptions surrounding what exactly Representational State Transfer (REST) is. The prime of which is the concept of **hypermedia**, or defined in full context, Hypermedia as the engine of application state (HATEOAS).

Jargon aside, hypermedia is actually a valuable idea that many self-touted "RESTful" web APIs do not adhere to. Hypermedia places heightened operational significance on **resources**, located at **URIs**. With a hypermedia API, when a request is sent to a URI, the response lists information on resource state and acceptable future operations, in essence creating a **state machine** that can be manipulated.

Hypermedia is truly valuable as it can create more powerful APIs that alleviate the need for API versioning. But this is only one of many facets that makes true REST design awesome.

In this chapter, we'll define what exactly REST is and isn't. Led by RESTafarian Asbjørn Ulsberg, we'll uncover the **history** of information design that has led to REST, and debunk some common **misinterpretations** of REST design.

We'll also construct a mock **state machine** and example HTTP behaviors, demonstrating how hypermedia could be used within a REST API to trigger states on an IoT kitchen device. Actually following the REST constraints by building a HATEOAS-compliant API could be very beneficial to advancing the web, so let's give it the focus it deserves.

Speaker: Asbjørn Ulsberg

This chapter was inspired by Ulsberg's session at the 2016 Platform Summit. Watch the full talk here

The History Behind REST and Hypermedia

Can you guess how old the concept of Hypermedia is? It's nearly 80 years old. That's right, in 1941, Agentinian author Jorge Luis Borges wrote The Garden of Forking Paths, a manuscript that contained pages that referenced each other; arguably the first form of **hypertext**. From Bioshock to Goosebumps, choose your adventure style entertainment and relational data have become commonplace, but in his time, the book was unprecedented. Some other important strides that have led to REST are:

- 1963: Ted Nelson coins the terms **hypertext** and **hypermedia**.
- 1968: Douglas Englebart debuts the On-Line System in the Mother of All Demos, which was the first application of hypertext, including mouse pointing, text editing, window environment and more, essentially giving birth to modern word processing.
- 1987: Apple employee Bill Atkinson creates HyperCard, creating the first successful implementation of hypermedia before the World Wide Web.
- 1989: Tim Berners-Lee creates the World Wide Web at CERN, implementing the first successful Hypertext Transfer Protocol (HTTP) implementation between a client and server.
- 2000: Roy Fielding, Co-author of the HTTP and URI specification, writes a doctoral dissertation entitled Architectural Styles and the Design of Network-based Software Architectures. In Chapter 5 he describes Representational State Transfer, or as we commonly call it, **REST**.

How do we Define REST?

So what is REST? Well, it's difficult to tackle all the technicalities in a single blog post, but let's first respond to four misconceptions to understand what REST **isn't**.

Common Misconception #1: REST is just CRUD.

CRUD, or Create, Read, Update, and Delete, has become a hallmark acronym amongst data management professionals as it represents the four basic actions for communicating with a database. Though CRUD maps cleanly with SQL actions, as we can see clearly below, it doesn't map well to HTTP methods:

Operation	SQL	HTTP
Create	INSERT	PUT / POST
Read	SELECT	GET
Update	UPDATE	PUT / POST / PATCH
Delete	DELETE	DELETE

Though `GET` and `DELETE` coordinate well, `POST`, `PUT`, and `PATCH` aren't directly synonymous with CRUD operations. For example, `POST` doesn't necessarily only mean "Create". It's actually a very versatile method — so versatile that the entire SOAP protocol is tunneled through the POST method when used with HTTP.

Since HTTP methods don't map cleanly to CRUD, Ulsberg argues that API providers should consider how they might describe their APIs in a different way:

> "Don't limit yourself to CRUD when you design a REST API. You should read the specification and understand the semantics of each method, and use it properly."

What it comes down to is that REST is an *architectural style*, not a protocol. So, calling an HTTP API behaving with CRUD operations "RESTful" is a fallacy.

Common Misconception #2: Some URI Constructions are more RESTful than Others

Uniform Resource Identifiers (**URIs**) are vital for defining resources and acting on resources, and are a core concept of REST. However, many developers seem to think they can distinguish a REST API simply based on how the URI is structured. Can you tell which URI is more *RESTful* than the others?

```
http://api.nordicapis.com/authors/contributor?author=doerrfeld
http://api.nordicapis.com/blogpost/getPostById?id=47
http://api.nordicapis.com/blogpost/47/edit-form
http://api.nordicapis.com/blogpost/47
http://api.nordicapis.com/128ndoels-8asdf-12d5-39d3
```

Contrary to popular belief, within the REST guidelines there is no such thing as a RESTful URI construction. To REST, these URIs are simply **opaque identifiers** — yes, they are global, unique identifiers that can be used for many purposes. However, without knowing more context and the behavior going on under the hood (what the methods look like, what the request looks like, what the response is, etc.) there is no way to tell whether these are RESTful operations or not. What the URI is actually shouldn't matter, so any of the ones above are as good as the other one.

> "Since only machines should read the URIs, and no human, it shouldn't matter"

The human-readable API design crowd is probably throwing tomatoes by this point. All this isn't to say that you shouldn't give attention to making URIs human readable. However, Ulsberg recognizes that you shouldn't depend on them being in any particular way within the client.

For example, say we have very basic documentation for a Nordic APIs Blog Post API, as identified below:

URI http://api.nordicapis.com	**Method**	**Description**
`/v1/blogposts`	`POST`	Creates a new blog post
`/v1/blogposts/{id}`	`GET`	Retrieves a blog post
`/v1/blogposts/{id}`	`PUT`	Updates a blog post
`/v1/blogposts/{id}`	`DELETE`	Deletes blog post
`/v1/blogposts/{id}/author`	`GET`	Retrieves blog post author
`/v1/blogposts/{id}/comments`	`GET`	Retrieves blog post comments

Just because we've listed a URI, Method, and Description for each one of our API calls doesn't mean that we've created a REST API — we've just documented our URIs as we would document RPC operations. Stefan Tilkov calls these *URI APIs*. According to Ulsberg, this puts a huge burden on the clients to understand how to build these URIs, and removes a lot of flexibility from the server.

Misconception #3: REST APIs Should be Versioned

Imagine if we had a database table called 'Referer'. After much use, we realized that the database name is misspelled, and that we have to change it to 'Referrer'. Since the clients have already hardcoded the table name in their SQL statements, you can't update the table, since this means you would have to update all clients— it would be very messy.

Similarly, if we wanted to change one of our `/blogposts/` URIs from above, we would be in the same tricky situation

— we would have to update all clients. This leads to creating a v2, updating documentation, and asking clients to kindly update everything. In short, having hardcoded versioning in the URI is a painful process.

Upon whether or not to version web APIs, Roy Fielding's keynote presentation for the 2013 Evolve Conference simiply stated:

> "DON'T"

Misconception #4: Hypermedia is Optional for REST APIs

Nope. As Roy Fielding himself stated in a 2014 interview with Mike Amudsen:

> " 'Hypermedia as the engine of application state' is a REST constraint. Not an option. Not an ideal. Hypermedia is a constraint. As in, you either do it or you aren't doing REST."

So what is hypermedia exactly? Well, REST consists of 6 major constraints. Out of these, HATEOAS is arguable the most important and unique to the REST stipulation, but also the least understood.

1. Client-Server
2. Stateless
3. Cacheable
4. Layered
5. Code on demand (optional)
6. Uniform Interface

- Identification of resources
- Manipulation of resources
- Self-descriptive messages
- **Hypermedia as the engine of application state (HATEOAS)**

To understand HATEOAS, Ulsberg recommends we apply the same thinking proposed by Don Norman in *The Design of Everyday Things*. As a cup *wants* to be held and lifted, or a button *wants* to be pushed, hypermedia *wants* to tell you what to do with the resource. Hypermedia is links and metadata for operations that help developers or machines perform additional actions. As Ulsberg says:

> "If you look at hypermedia as a recipe of how the next request is supposed to look like, you will grasp what hypermedia is all about"

Example State Machine: IoT Toaster

So let's delve into what hypermedia actually looks like. To describe hypermedia as the engine of application state, let's take a simple example of a possible state machine — a connected toaster that can be manipulated through the internet.

> A state machine is a concept within computer programming that is used to describe a machine that has a set of states, usually with input and output events.

Our toaster begins in an off state, and when it is turned on eventually reaches a heating state. Then it reaches its upper temperature limit, enters an idle state, which reduces the temperature. By being idle it cools, and goes back into a heat state. It continues this loop, maintaining a constant temperature until the bread is toasted, after which it shuts down.

How could you manipulate such a toaster through REST and hypermedia? Well, as Roy Fielding described, hypermedia is the engine of application state. Therefore, each page on the web represents a single state of a single resource, which can be obtained using a GET call. It is implicit that you are able to retrieve anything with an ID at a URI. So, let's first call our toaster to see what we get back:

```
GET /toaster HTTP/1.1
```

The response looks something like:

```
HTTP/1.1 200 OK
{
        "Id": "/toaster",
        "state": "off",
        "operations": [{
                "rel": "on",
                "method": "PUT",
                "href": "/toaster",
                "Expects": { "state": "on"}
    }]
}
```

As you can see, we have an **id** at the top to denote what resource we are communicating with. The current state is displayed, as well as a list possible operations that we may enact on the machine.

So let's try to actually alter the state of the toaster. We'll send a `PUT` HTTP request for this:

```
PUT /toaster HTTP/1.1
{
    "state": "on"
}
```

Our response will look something like:

```
HTTP/1.1 200 OK


{
        "Id": "/toaster",
        "state": "on",
        "strength": 0,
        "operations": [ {
                "rel": "PUT",
                "method": "PUT",
                "href": "/toaster",
                "expects": { "state": "off" }
        }, {
        "rel": "strength",
        "method": "PUT",
        "href": /fcesj48fl29304d827434j
        "expects": {
                "strength": [1,2,3,4,5,6]
                }
}]
}
```

We have now turned on the toaster! However, as we can see above, the strength is still set to zero, so it isn't heating yet. Let's see what happens when we make a call to affect the strength operation.

```
PUT  /fcesj48fl29304d827434j HTTP/1.1
{
        "strength": 3
}
```

So when we execute this strength operation, we see that the state has changed, and the toaster is now heating:

```
HTTP/1.1 200 OK


{
        "Id": "/toaster",
        "state": "heating",
        "strength": 3,
        "operations": [ {
                "rel": "PUT",
                "method": "PUT",
                "href": "/toaster",
                "expects": { "state": "off" }
        }, {
        "rel": "strength",
        "method": "PUT",
        "href": /fcesj48fl29304d827434j
        "expects": {
                "strength": [1,2,3,4,5,6]
                }
}]
}
```

We still have other options, as laid out in the hypermedia above. We can chose to turn it off, or adjust the strength again. But instead, let's send another `GET` to the toaster ID.

```
HTTP/1.1 200 OK


{
        "Id": "/toaster",
        "state": "idle",
        "strength": 3,
        "operations": [ {
                "rel": "PUT",
                "method": "PUT",
                "href": "/toaster",
                "expects": { "state": "off" }
        }, {
        "rel": "strength",
        "method": "PUT",
        "href": /fcesj48fl29304d827434j
        "expects": {
                "strength": [1,2,3,4,5,6]
                }
}]
}
```

Since our last request, the toaster has entered an **idle** state. You can see that we can still opt to turn it off, adjust the heating temperature with the strength

Wait a few minutes, and another request to the state machine will likely result in a state of “shutting down” which may include no operations, or “off” — the initial resting state with the list of possible operations still in the response.

Conclusion

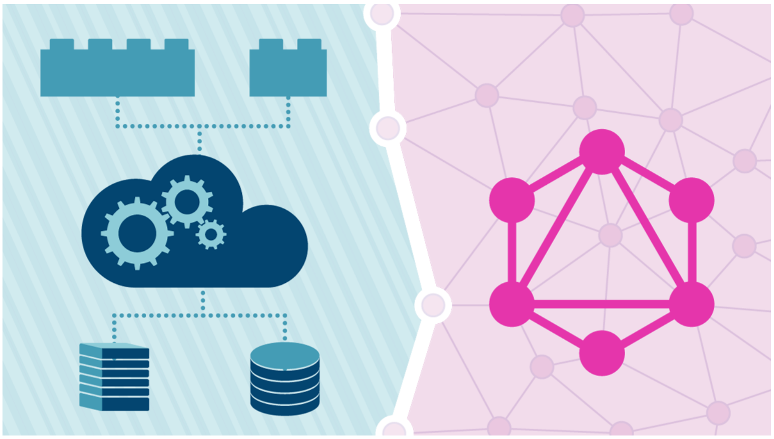
The web functions as a web of interconnecting ideas, linked with hypertext. Ulsberg believes that web APIs should mimic this, and an important facet in doing so is implementing hypermedia within our APIs. For those newcomers to REST, Building a HATEOAS-compliant API is a huge improvement over RPC-style APIs:

> “If you use hypermedia, you can add relations and links, and operations **to** the resources without breaking existing clients, and at the same time, giving new functionality to new clients.”

This also means rethinking the traditional stance on versioning. On versioning, Ulsberg thoughts echo Fielding’s: **don’t**. When did you last see a versioning number on a website? As HTML doesn’t need a version number, JSON shouldn’t either.

By putting more emphasis on the resources and URIs, we can retrieve operations directly from them. This is the power of REST. This is the power of hypermedia.

Is GraphQL The End of REST Style APIs?



The world of APIs is one of innovation and constant iteration. The technologies that once drove the largest and best solutions across the web have been supplanted and replaced by new, more innovative solutions.

That is why it's surprising, then, that many developers have clung to what they consider the bastions of web API development. Such a bastion is the REST architecture. To some developers, REST is an aging and incompleted proposition that does not meet many of the new development qualifications required by the unique challenges faced by modern development groups.

In this chapter, we're going to look at a technology that is poised to replace, or at the very least, drastically change the way APIs are designed and presented — **GraphQL**.

We'll discuss a little bit of history, what issues REST suffers from, and what GraphQL does differently.

Speaker: Joakim Lundborg

This chapter was inspired by Lundborg's session at the 2016 Platform Summit. Watch the full talk here

> "The way we design our APIs structures the way we think about the tools and applications we build."

Defining REST and its Limitations

REST or Representational State Transfer, is an API design architecture developed to extend and, in many cases, replace older architectural standards. Objects in REST are defined as addressable **URIs**, and are typically interacted with using the built-in verbs of **HTTP** — specifically, GET, PUT, DELETE, POST, etc. In REST, HATEOAS (Hypermedia As The Engine Of Application State) is an architecture constraint in which the client interacts with hypermedia links, rather than through a specific interface.

With REST, the core concept is that everything is a **resource**. While REST was a great solution when it was first proposed, there are some pretty significant issues that the architecture suffers from. According to Lundberg, the circumstances have changed, giving rise to the need for new technical implementations:

> "Many things have happened. We have a lot of mobile devices with lots of social and very data

> rich applications being produced…We now have very powerful clients, and we have data that is changing all the time. This brings some new problems.”

Here are some issues Lundberg sees with REST:

Round Trip and Repeat Trip Times

REST’s defining feature is the ability to reference **resources** — the problem is when those resources are complicated and **relational** in a more complex organization known as a *graph*. Fetching these complicated graphs requires round trips between the client and server, and in some cases, **repeated trips** for network conditions and application types.

What this ultimately results in is a system where **the more useful it is, the slower it is**. In other words, as more relational data is presented, the system chokes on itself.

Over/Under Fetching

Due to the nature of REST and the systems which often use this architecture, REST APIs often result in over/under fetching. **Over fetching** is when more data is fetched than required, whereas **under fetching** is the opposite, when not enough data is delivered upon fetching.

When first crafting a resource URI, everything is fine — the data that is necessary for functionality is delivered, and all is well. As the API grows in complexity, and the resources thus grow in complexity as well, this becomes problematic.

Applications that don't need every field or tag still receive it all as part of the URI. Solutions to fix this, such as versioning, result in duplicate code and "spaghettification" of the code base. Going further, specifically limiting data to a low-content URI that is then extensible results in more complexity and resultant under fetching in poorly formed queries.

Weak Typing and Poor Metadata

REST APIs often unfortunately suffer from **poor typing**. While this issue is argued by many API providers and commentators (often with the caveat that HTTP itself contains a typing system), the fielding system solutions offered simply do not match the vast range and scope of data available to the API.

Specifically, this is an argument in favor of **strong typing** rather than weak typing. While there are solutions that offer typing, the delineation between weak and strong is the issue here, not an argument defused by simply stating "well there *is* typing". The strength and quality of typing *does* matter.

This is more a matter of age and mobility rather than an intrinsic problem, of course, and can be rectified using several solutions (of which GraphQL is one).

Improper Architecture Usage

REST suffers from the fact that it's often used for something it wasn't really designed for, and as a result, it often must be heavily modified. That's not to say that REST doesn't have its place — it's only to say that it may not be the best solution for serving client applications. As Facebook says in its own documentation:

> "These attributes are linked to the fact that "REST is intended for long-lived network-based applications that span multiple organizations" according to its inventor. This is not a requirement for APIs that serve a client app built within the same organization."

All of this is to say that GraphQL is functionally the end of REST — but not in the way that terminology implies. Until now, REST has been seen as the foundational architecture of modern APIs, and in a way, the last bastion of classic API design.

The argument here is not made to fully sever REST from our architectural lexicon, but instead to acknowledge that there are several significant issues that are not properly and fully rectified by the solutions often proffered by its proponents.

Therefore, the answer to the question of this piece — is GraphQL The End of REST Style APIs? — is quite simple. Yes, using GraphQL is the end of REST style APIs as we know it — specifically through the extension of base functionality and a reconsideration of data relations and functions. ## 4 Things GraphQL Does Better than REST

> GraphQL declares everything as a graph… You say what you want, and then you will get that.

Now that we've seen the issues with REST, how, exactly, does GraphQL solve them?

REST Has Many Roundtrips - GraphQL Has Few

The biggest benefit of GraphQL over REST is the simple fact that GraphQL has **fewer roundtrips** than REST does, and more efficient ones at that. GraphQL unifies data that would otherwise exist in multiple endpoints (or even worse, ad hoc endpoints), and creates packages.

By packaging data, the data delivery is made more efficient, and decreases the amount of resources required for roundtrip calls. This also fundamentally restructures the relationship between client and server, placing more efficiency and control in the hands of GraphQL clients.

REST Has Poor Type Systems - GraphQL Has a Sophisticated One

While REST can have a **type system** through implementations of HTTP, REST itself does not have a very sophisticated typing system. Even in good implementations, you often end up with variants of type settings — for example, *clientdatamobile* and *clientdatadesktop* — to fit REST standard calls.

GraphQL solves this with a very sophisticated typing system, allowing for more specific and powerful queries.

REST Has Poor Discoverability - GraphQL Has Native Support

Discoverability is not native to REST, and requires specific and methodical implementations of HATEOAS, Swagger, and other such solutions in order to be fully discoverable. The key there is “fully discoverable” — yes, REST has HATEOAS as a “native” discovery system, but it lacks some important elements of effective **discoverability** — namely known document structure, server response constraint structures, and an independence from standard, restrictive error mechanisms in HTTP.

While this and many other points of negative consideration towards REST is often answered with “but you can add that functionality!”, the fact that it lacks it by default only adds to the complexity we’re trying to move away from.

Because GraphQL is based on **relational data** and, when operating on a properly formed schema, is self describing, GraphQL is by design natively discoverable. Discoverability is incredibly important, both in terms of allowing for extensible third-party functionality and interactions and for on-boarding developers and users with an easy to understand, easy to explore system of functions.

REST Is Thin Client/Fat Server - GraphQL is Fat Client/Fat Server

In REST design, the relationship between client and server is well-defined, but **unbalanced**. REST uses a very **thin client**, depending on processing from the server and the endpoints that have been defined for it. Since the bulk of the processing and control is placed firmly on the **server**, this strips power from the client, and also stresses server side resources. Until now that has been fine, but as devices grow in processing power and ability, this client/server relationship may need rethinking.

GraphQL, however, is different. By offloading specification of expected data format to the client and structuring data around that call on the server side, we have a Fat Client/Fat Server (or even a Thin Client/Thin Server depending on approach) in which both power and control are level across the relationship.

This is very powerful when one considers that the data type being requested will be used for specific purposes as regulated and requested by the Client itself — it makes sense, then, that moving from a Thin/Fat relationship to a Fat/Fat or Thin/Thin relationship would improve this functionality on the Client side while freeing up Server resources. Of course, this assumes that the client is capable of handling this burden.

The End Of The Status Quo

There’s a tendency in the tech space for providers and developers of new technologies to proclaim the end of an era with each solution. While it’s common to discussion in the field, the fact is that there are very few complete paradigm shifts that signal an irrevocable end to existing technologies.

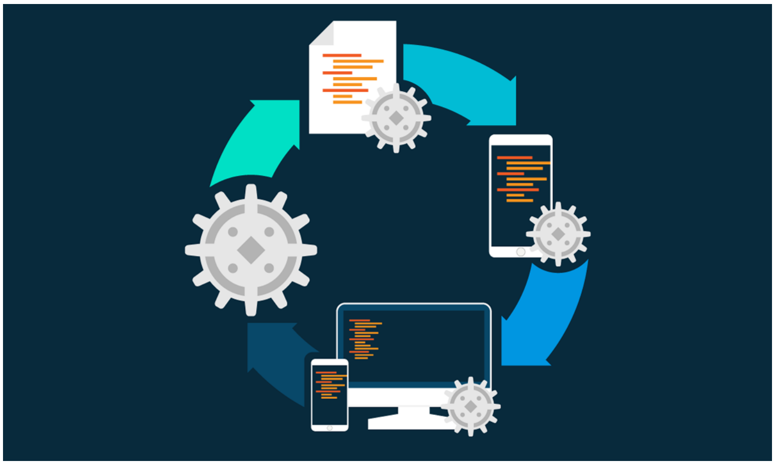
Innovation depends on prior technologies to create new functionality. Therefore, when a new solution is designed, it’s not replacing the solution, but rather iterating. The same is true here. While GraphQL may not be the complete demise of REST, it is the end of the **status quo**. While there are a great many solutions to the issues raised here, they all depend on further integrations and modifications. GraphQL is essentially an overhaul, and one which improves the base level functionality of the API itself.

Conclusion

What we have here is a basic value proposition. GraphQL does what it does well, but the question of integration lies directly on what kind of data you're processing, and what issues your API is creating. For simple APIs, REST works just fine, but as data gets more complex and the needs of the data providers climbs, so too will the need for more complex and powerful systems.

Adopting GraphQL as an adjunct or extension of the REST ideology, while removing REST from the intellectual space of "too big to not use", will directly result in more powerful APIs with easier discoverability and greater manageability of the data they handle.

Continuous Versioning Strategy for Internal APIs



Recently, there has been debate over what the best practices are for **versioning** an API. Many public web APIs are retired as new versions replace them, but if you were to ask Roy Fielding, creator of REST, he may tell you not to version your API at all.

Some companies are taking matters into their own hands, and seeking out innovative ways to handle the cumbersome process of keeping their Application Programming Interfaces up to date in a way that makes sense to their business model. These new strategies place more emphasis on **evolution** rather than **deprecation**.

The typical v1, v2, v3 etc. versioning approach focuses on releasing large sweeping updates to improve the API experience. But the downside of this method is that it causes a major breaking change on the client side. For **internal API-first companies** that have granular control over their various web, desktop, and mobile clients, **continuous versioning** could be a more attractive strategy.

In this chapter, we'll review how public web APIs are typically versioned within our domain, and discuss why companies may want to consider a continuous versioning strategy for handling complex APIs that are subject to continual, iterative evolution. Led by Platform Summit speaker **Konstantin Yakushev**, we'll use **Badoo** as a case study to peek into an alternative approach to versioning. Benefits like feature negotiation, and allowing for experimental development tracks could make continuous versioning strategy a win, especially for private API systems.

Speaker: Konstantin Yakushev

This chapter was inspired by Yakushev's session at the 2016 Platform Summit. Watch the full talk here

Typical Public API Versioning

Within most public scenarios, an API service is updated by creating an entirely new v2 and slowly deprecating the original v1. Problems with v1 are tracked — perhaps a product order is misspelled, or the business logic has changed. All these edits are accumulated and introduced in a v2 that solves these issues, but introduces a complete **breaking change** with the previous version.

An API with an endpoint such as `http//api.example.com/orders`, is typically reworked with a URI extension to something like 'http//api.example.com/v2/orders'. The v1 is then scheduled for **retirement**, usually in accordance with a deprecation policy. Though this is the norm, there are some significant negatives of this approach:

- **Long Timeline**: Instead of incremental edits, with versioning you must wait for all changes to be bundled. This means you can't be that agile in responding to specific user requests.
- **Breaking**: Whether you like it or not, releasing an v2 is inherently breaking the connection, and will require all clients to eventually update their connections.
- **Communication**: Time and resources must be spent to communicate API changes. With a v2, documentation must be updated, and deprecation timeline notices must be sent to consumers.
- **Fielding as a Friend Factor (3F)**: Roy Fielding defines *evolvability* as the ability to change over time in response to changing user needs or a changing environment without starting over. It's actually against Roy Fielding's own recommendation to version your API, saying it's "only a polite way to kill deployed applications."

Many typical versioning strategies focuses too heavily on the URL construction, which to Yakushev, is "the least important step, in my opinion." Instead, it may be better to consider the **entire process** from a more holistic vantage point. When we look at the API update process, we see that perhaps *there is no v2* — after all, much is often salvaged, and introducing an entire new version may not be worth the effort in updating all clients.

Badoo’s Continuous Versioning Strategies

When API-first companies consistently iterate with **continuous versioning**, the problems listed above dissolve. To see how this actually works in practice, let’s consider some specific **use cases** from Badoo, the Russian dating network and app.

Badoo is the largest dating network in Russia, and they have been evolving an internal API since 2010. They’ve never had a breaking change as they’ve been **incrementally updating** all this time. Konstantin frankly admits that the API is not strictly RESTful, rather RPC-style and Protobuf based for mobile clients, and JSON based for their web clients

With nearly 600 commands and over 1,200 classes, the API receives around 9 updates each week, and supports 5 clients (iOS, Android, Windows, Chrome, Safari) with a healthy backwards compatibility for older client versions.

Let’s take a look into Badoo’s internal API strategy to see how they’ve used a continuous versioning mindset for specific updates to avoid major, breaking changes.

Changing the Verification Process

Yakushev describes how Badoo needed to rework their verification process. In the past, if a user signed up with their social account, they received a social checkmark associated with their account. As time grew on, the designers wanted to have more rigorous checks. For example, if a user were to verify with photo verification, they should receive a different badge.

The problem was that the original verification had a **binary logic** that affected other aspects of the app — the users were either verified (true) or not verified (false). Since that was the case, adding a new verification complexity meant instituting a dramatic change to their API behavior.

The Badoo team was able to solve this issue by using GraphQL to list the acceptable fields for clients. Now, when clients request the verification status, they receive more customizable options. Allowing clients to **negotiate** new fields is a way Badoo can update their API while keeping endpoint consistency. The old clients can use old fields, whereas the new clients use new fields.

Updating Banner CTAs for Specific Clients

However, Yakushev recognizes harder challenges in keeping their API updated and consistent across various clients. For large changes, he advises releasing new features on the server, and making clients end supported types explicitly.

For example, Badoo needs to serve various call-to-action banners for different screen sizes and device-specific interactions. If a new banner type is introduced, however, when the client asks for banners, the server could send an unknown or old banner. Typical versioning is not flexible enough here.

To solve this issue, Badoo introduced a list of supported **banner types** to easily decide which banners will be shown to the client. Now, client specific banners, such as swipeable mobile-only logic can be paired with the right receiving device using the same, albeit stateful, API.

Use Flags to Avoid Versioning in Complex Business Logic Changes

What about more complex high-level changes to **business logic**? Yakushev explains how all Badoo profiles have a photo feed attached to them. Over time, the design team wanted to mix in videos with the photos, and add a play button to watch the videos from within the grid view.

To resolve the issue without versioning the entire API, Badoo introduced a supported changes array. This way, the client knows that the server may send videos along with photos. A similar approach can work in many other cases — essentially you release changes behind a **version flag**, and make the client control these flags.

Run Experimental Features

A benefit of Badoo's hands on approach to the entire API lifecycle is the ability to run quick experimental features on select platforms. To do this they create a superset **experimental API** that is only used on a select platform, such as the Windows phone, as it las low usage. Having multiple development tracks enables new features to be tested and engagement monitored.

How Continuous Versioning Could Apply to You

Depending on the situation, continuous versioning could be a powerful ally in developing and scaling agile web APIs. Instead of instigating breaking change, **fields** for new features are added, and the client has a list of supported items to send to the server. Yakushev recommends covering new changes with change flags, and letting the server control enabling and disabling features.

At the end of the day, client developers and product owners are happy. An iterative versioning strategy my put additional pressure on backend developers, but they may like the ability to split their work into parallel tracks for API supersets and experimental features.

In practice, Badoo has nearly 260 feature flags, and 160 negotiable features. Implementing **feature negotiation** at this level of complexity is more easily accomplished within an internal scenario, where communication between teams is united and both client developers and API developers are working toward similar end goal.

In Q&A discussion it was found that continuous versioning still may may not be an ideal method for **public** API providers. Since within continuous versioning new fields essentially equate to new features, some believe this level of feature negotiation is only acceptable when you control both the API and clients. Performing continuous versioning within a **public API** scenario may still need thought, nonetheless is an alluring proposition.

Case Study: Spotify Internal Payment APIs

Adding new layers of complexity within a digital service without sacrificing user experience is a difficult endeavour. Especially for platforms that accept online **payments** and subscription formats, maintaining support for an increasing number of payment methods is deceptively complex.

Users don't want to think too much about the payment process. They simply want payments to work with their preferred, custom method. However, the number of online purchase methods has soared in recent years — digital platforms must now consider support for not only credit cards or bank accounts — but Paypal, cryptocurrencies like Bitcoin, Facebook payments, Google Wallet, Apple pay, Klarna, Boku, Sofort… the list goes on.

All this is expected to behave with zero issues, but under the hood, handling diverse payment methods in an efficient way is becoming increasingly complex, requiring some serious forethought.

At the 2016 Nordic APIs Platform Summit, we were offered an exclusive sneak peak into the architecture that **Spotify** software engineers deploy in order to accept millions of user subscription payments each month. As they build internal APIs to handle the complexities of their payment subscriptions, their platform is an excellent case study into the power **Private APIs** have to streamline internal operations.

Despite architecting a custom system for their own business domain, the Spotify payment model could surely inspire other digital enterprises to introduce similar, API-driven scalable payment subscription formats into their own service framework.

Speaker: Horia Jurcut

This chapter was inspired by Jurcut's session at the 2016 Platform Summit. Watch the full talk here

The Evolution of Spotify Payments

To begin with, let's consider how Spotify evolved. In 2006, Spotify was developed by a small team in Stockholm, and was launched to the Scandinavian market in October 2008. At that time, they only accepted credit/debit cards as a payment method.

By the end of 2011 their market increased again, spreading throughout Europe and into the United States. It was then that they added support for Paypal. In 2013 their market size increased again, and they added Boku, Sofort, Klarna, Google in-app purchases (iAP), and Facebook payments as optional payment methods.

Within this two year period their growth was impressive. As Spotify was emerging into many local markets, they decided to shift their focus from only accepting **global** payment methods to adding support for more **local** ones. To do this, in 2015 they partnered with the payment service Adyen, which allows companies to support over 250 different payment methods and 150+ currencies — truly a globalized, yet locally aware solution. They added 9 other payment methods to their stack, including bank transfer and preloaded cash cards, which increased the brand engagement into areas that have seen low credit card penetration.

In 2016, they now have roughly 40 million worldwide subscribers in 60 markets. Beside delivering great musical playlists and audio experiences, their remittance goal remains consistent across the globe — to **help users select most convenient payment method**.

What's Really Going on with Online Payments?

Things like subscriptions seem like an easy process, but the workflow required for even a simple credit card payment has hidden complexity, with many working actors as well as unique edge cases to consider. First, let's break down the workflow of a typical online payment made using a Visa credit card.

1. The user opens a line of credit with a **Bank** and is given a card.
2. They type their card number into a **Checkout** screen that ties into a **Payment Backend**.
3. Next, the **Payment Backend** will attempt to authorize the payment details with the **Payment Provider**.
4. If it succeeds, the **Payment Provider** will contact the **Bank** through the **Credit Network** to set aside money for a subscription.
5. Finally, the **Bank** confirms that the purchase is possible, and the response comes back to the **Payment Backend** through the **Credit Network**.

After this common flow is complete, an online subscription will be engaged. How is this communication architected? For Spotify, the process is supported by three main actors: the **Client** with the checkout page, the **Payment Backend**, and the **Payment Provider**. In reality, however, Spotify supports **16 different payment providers**, each with unique APIs that have different implementations.

So how should a platform manage this complexity? To do it Spotify developed two smart tools: The **Checkout API** to help build flows that make it easy for users to enter payment details, and the **Billing API** to interface with the various details of Payment Providers and Credit Networks, enabling the Payment Backend to determine if they can charge a user for a subscription with a single call.

The Checkout API

The first main component is the **Checkout API**. To begin a subscription, the Payment Backend must try to validate payment details, and initiate the workflow mentioned above. However, in order to do this, it first needs to gather information from Spotify internally, which is in effect sourced from user input on clients like mobile or desktops. Since the type of clients are manifold, Spotify uses a **state machine**.

Therefore, their Payment Backend dictates the operations, controlling the workflow and requesting data from the clients. The Checkout API thus facilitates the communication between the Payment Backend and the clients.

This is necessary for the fact that extra steps may exist in a check out workflow. For example, in some countries, filling in tax information may be required. The Spotify Payment Backend needs to inform clients of the existence of this step, and the clients need to respond with a nuanced form, and collect other data as well. Other special cases may involve an offer code from an email invite to join. For all these scenarios, the backend decides what data is required, and the client must be populated with the correct form.

The key API that is driving these experiences is the Checkout API, sitting in between the Spotify clients and Payment Backend. This API enables them to create customized forms in an infinite number of iterations within Spotify clients.

Simple Checkout API Flow

1. The **Client** initiates a purchase, which triggers the **Checkout API**.
2. The **Checkout API** asks the **Payment Backend** what the next step is.
3. The **Payment Backend** responds saying that credit card information needs to be collected.

4. The **Checkout API** sends this to the **Client**, who uses this information to display a custom form for the user to input data into.
5. The **Checkout API** then sends the card data to the **Payment Backend**.
6. Then the **Payment Backend** processes the payment, confirms with the **Checkout API**, which initiates the **Client** to display the confirmation page.

These steps could easily be extended for custom form scenarios. For this to work, the business logic is always resting on the backend to dictate processes and for the clients to respond with appropriate information.

There are a few benefits to architecting a payment process this way. Having the Checkout API between the backend and client means that you can alter the checkout experience quite easily. Clients themselves can also now create native experiences, meaning that data can be moderated throughout a device to alter other apps or device-specific functions.

The Billing API

After the Payment Backend receives payment information, it needs to communicate with the Payment Providers to charge the user. Spotify's Billing API is the bridge between the Payment Providers and their Payment Backend; the second interior API necessary for this payment flow to occur.

Since Payment Providers have different business logic (a credit charge is instant, whereas a bank transfer may take 3-5 days, for example), the Billing API is very useful. It translates these different implementations into usable data, hiding the complexity of 16 different Payment

Providers from the Payment Backend, consolidating things into a **single call**.

The Billing API is also responsible for handling financial data, settlement, monitoring, and other duties. This allows the team to compare different payment methods, such as seeing the usage differences between Paypal and a method where SMS confirmation is required.

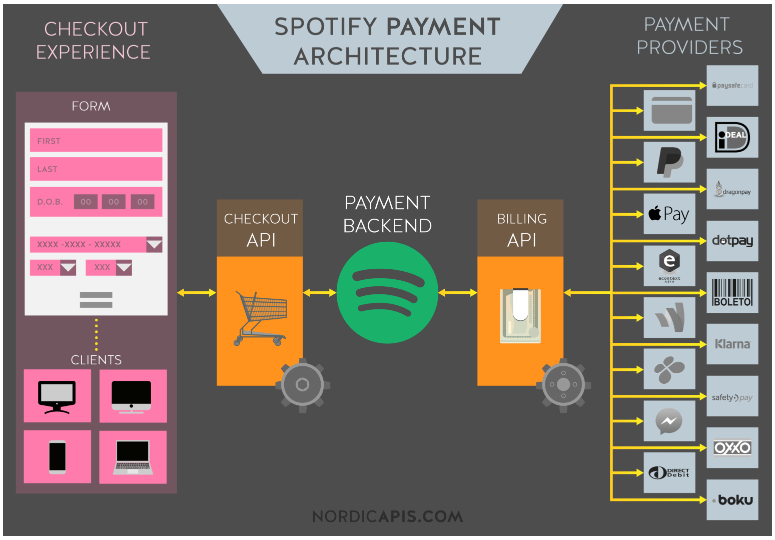
Sample Use Case: Automatic Alerts

A main facet of the DevOps approach is having automated alerts to allow for faster response times to system issues. Spotify leverages the internal Billing API to provide rich platform **monitoring** features, that are then repurposed throughout the enterprise in what they call their "anomaly detection system". The Billing API is able to turn actions from the Payment Providers into consistent strings of events, which can then be monitored and filtered.

The varying actions for all remittance methods are boiled down into JSON exchanges with fields such as `message`, `amount`, `provider`, `failure_code`, and more. Producing standardized events for all types of payment methods enables alerting which can inform development and operations.

Keeping track of received payment transactions, and modeling this with past transaction histories, means that the system can detect trends in **payment flaws**. So, when a developer deploys a change that breaks the production environment, the team receives an alert. Using this same data to populate visualizations, the team can also detect if a certain payment provider is having an issue, as the data allows them to isolate individual provider activities.

Having a monitoring system for a billing API allows a company to follow trends, and become aware of local events, like bank holidays. It also can help automate internal accounting; Spotify's approach is so fine-grained that if a small payment provider misses a due date/time for sending batch requests, the Spotify alert system will immediately notify the team.



Repurposed from original diagram within Hurcut/Spotify presentation

4 Reasons Why API Design is Critical to Subscription Services

Of all the approaches to handling payments, why should subscription services design their own internal APIs? There are a few benefits:

- **APIs are platform independent**: First, web APIs are designed to be used by any client. This helps hide complexity when dealing with a large number of clients and payment providers. Since the purchase method landscape is still evolving, adaptable interfaces are very important.

- **Internal integrations**: Other benefits include integrating with accounting platforms, payment security, or user activity monitoring. Having well-defined APIs for subscription services enables powerful data integration capabilities with other internal systems.
- **Scale for growth**: As a business grows, the need to build more complex products will become a concern. Therefore, having good API design helps prepare for compatibility with new business logic that will affect the company.
- **Customization potential**: There are many payment providers, aggregation services, or payment gateways within the market, but building your own APIs is a way to ensure customization and efficiency.

Horia Jurcut, Software Engineer at Spotify, also describes his personal affection for APIs and the positive reverberations they create within a modern business:

- APIs make it easier for multiple teams to collaborate
- APIs help you take a hard problem and divide it into more manageable domains
- APIs require documentation
- APIs enable you to rapidly test, experiment and learn.

Analysis: Scalable API-Driven Infrastructure will Power the Future of Online Payments

Spotify has impressively balanced the small and large with their payment architecture. They have successfully arbitrated complexity that has arisen out of a need to serve customization options to their users, and in doing so, have created a scalable architecture for accepting multiple payments that could be modeled at other companies.

By building consistent constructs that interface with malleable components (payment providers and clients) through APIs, they are able to both scale for global use while integrating with small payment providers, allowing their brand to cater to very local, niche audiences in specific markets.

On the user side, adding support for local payment methods is one way Spotify has honored a pledge for great consumer experiences. As the Spotify music service excels at offering custom, curated musical playlists based on user behaviour, it makes sense that their payment models would allow for custom methods as well. **Platform-wide consistency with an end goal to improve UX** should, similarly, be the end goal at other companies. It seems that a significant portion of Spotify's rapid, global growth is due, in part, to a well-architected payment backend.

The Spotify payment strategy is conformable within other businesses, and can act as a great architectural model for new end-user facing digital services that intend to expand to specialized markets with **local payment options**. Since the number of payment providers continues to climb, digital companies will very soon need to plan for how they can most efficiently accept and monitor new payment types within their subscriptions.

As the decisions made now regarding online payments will be crucial for digital platform **longevity**, consider how your business could parcel settlement functionality more efficiently. Whether or not you need a custom built solution is up to you. Exciting emerging financial technology APIs like Klarna, Plaid, Stripe, Transpay, or other FinTechs will likely aid SMBs, unlocking differing payment methods so they may become globally recognized and supported services.

Though fine-grained payment options may not currently be necessary for all digital services, their ubiquity is on the rise. At the very least, from this case study we can see the benefits of creating distributed internal microservices: abstracting functionalities is key to scalability.

The Benefits of a Serverless API Backend

Imagine if your backend had no infrastructure. No permanent server, nowhere for your API to call home. Sounds a bit bleak, doesn't it? As it turns out, **serverless** backends could be the next big thing for implementing truly scalable cloud architecture.

How do you make an Application Programming Interface lightweight on the client side, yet scalable to heightened traffic demands? SaaS vendors have been migrating to **serverless** architecture to solve this dilemma, as well as many other operational issues found in hosting their web applications.

In this chapter, we'll identify what the serverless craze is, and why some providers may want to consider having a **serverless API backend**. Led by Rich Jones of Gun.io, we'll define what we mean by **serverless API backends**, provide an example of one practice today, and aim to outline some potential benefits and pitfalls of adopting this approach.

Speaker: Rich Jones

This chapter was inspired by Jones' session at the 2016 Platform Summit. Watch the full talk here

What Does "Serverless" Mean?

Traditional cloud hosting is permanent. As in, you pick a server provider, and they run you software on multiple servers worldwide. There are precise, recurring physical locations for where your data is stored and functionality processed.

A serverless backend scales to the needs of API requests.

Serverless computing is a strategic deviation from this model — it is an event-driven setup **without permanent infrastructure**. This doesn't mean servers are no longer involved, rather, it means that servers are auto-created on a per-need basis to scale to the demands of your app.

But for developers, what serverless *really* means is less time spent on operations, since they no longer have to worry about traditional server maintenance. The benefits of a serverless infrastructure really add up:

- No more over capacity issues
- Servers are autoscaling

- You don't pay for idle time
- Consistent reliability and availability
- No load balancing, no security patches

In general, serverless simply equates to peace of mind — but perhaps not for some, Operations may need to find another job altogether.

From Traditional to Serverless Environments

To understand the subtleties between traditional and serverless approaches, let's walk through a basic step-by-step sample implementation of each.

Traditional Web Request

An interaction with a traditional web server will often occur in a format similar to this:

1. An Apache or NGINX web server listens for events as they come in.
2. The server then converts this to a Web Server Gateway Interface (WSGI) environment.
3. This is sent to an application to process the request.
4. Then the web server sends the response back to the client.
5. The web server resumes listening.

There are a few drawbacks of this approach. For one, if you encounter a huge spike in traffic, this system will deal with the requests as they came in. If the end user isn't ahead in the queue, they will likely experience a timeout, and the page will look like it's down. Medium to late visitors to the queue face very slow speeds. Secondly, when it's not processing a request, the web server is left in an idle state to poll, wasting valuable resources that could be used elsewhere.

Serverless Web Request

Within a serverless infrastructure, each request corresponds to it's own server. After the server processes the function, it is immediately destroyed. For example, here's how Jones's Zappa handles a web request:

1. The request comes in through an API Gateway.
2. The API request is mapped to a dictionary using Velocity Template Language (VTL).
3. A server is created.
4. The server then converts the dictionary to a standard Python WSGI and feeds it into the application.
5. The application returns it, and it passess it through the API Gateway.
6. The server is destroyed.

Astoundingly, all this occurs under 30 milliseconds, so that "*by the time the user actually sees the [content appear on the] page, the server has disappeared… which is actually a pretty zen thing if you think about it*," says Jones.

So what are the advantages to spawning servers on a moment's notice? To Jones, the top reason is **scalability**. Since a single request matches to a single server creation, this relationship can be scaled indefinitely, on a scale of literally trillions of events per year.

Second is **cost** savings. Paying by the millisecond means that you are only spending money on actual server processing. AWS Lambda charges around $0.0000002 per request. But since Lambda tier offers 1M free requests per month, this means it could stay **free** for small projects or young **startups**.

This infinite scalability make serverless infrastructure a boon for both small breadth projects like microservices, APIs, IoT projects, or chatbots, but also for larger traditional enterprise content management systems like Django as well.

How Get Started: Understanding the Serverless Vendors

Sound interesting? An easy way to get started is with a serverless framework like Zappa, Serverless Framework, or Apex (more here). With some frameworks, like Zappa, you can adopt serverless computing for existing APIs. All three are built around AWS Lambda, Amazon's cloud computing service, but other significant serverless computing providers are within offerings by Microsoft Azure Functions, Google Cloud Functions, and IBM Bluemix OpenWhisk. However, according to Jones:

> "AWS Lambda is by far the leader in the space… it's just far more capable in pretty much every regard. The others are still playing catchup."

Designing Event-Driven Serverless Applications

Within a serverless environment, a main design element that will be novel to newcomers is that code is going to execute only in response to **events**. Since building a robust, **event-driven** application means designing in-response to events, what can we define as our *event sources*?

An event may be related to **file operations** — for example, say a user uploads an image and the application needs to resize a large picture into a small avatar. Using a serverless architecture, you could have a thumbnail service execute a response in an asynchronous and non-blocking way. Instead of setting up an entire queuing system, having a native cloud hosted queue can handle this.

Support **notifications** like receiving an email, text, or Facebook message could also be interpreted as events. Rather than polling for new emails to come in, an action could be executed specifically in response to these. Where it gets really interesting is how you can treat **HTTP** requests as an event. This paired with other event trigger types is usually called a **hybrid architecture**.

Database activity could also be used as an event trigger. A change to a table's row could trigger an action to happen, for example. However, Jones reminds us to treat the "API as the primary source of truth in your application" — don't make SQL calls inside your event functions, rather, funnel this through your API.

Jones reminds us that **time** is also an important factor that can be used as an event source, and will be needed to initiate regular occurring tasks or updates. Throughout these varying event sources, instead of creating machines that constantly poll your resources for changes, you're essentially setting up triggers within your applications to execute a response.

5 Serverless Pro Tips:

All this sounds awesome, but what are the downsides of building applications with serverless backends? In his presentation, Jones covers some ground on potential downsides, how to avoid them, and some general tips for getting the most out of a serverless arrangement:

- **Avoid vendor lock-in**: This can be a big issue when adopting any new technology. To avoid vendor lock-in Jones recommends integrating software that provides open source compatible offerings, and to decouple vendor interactions from your application. Rather than hardcoding interactions, Jones recommends decoupling this logic — creating a dispatcher inside of a function to add an item to the queue is one way of doing so.
- **Mock your vendor calls for testing**: When writing a mock or sample app that behaves as if synced to the cloud, you may want to test your cloud functions. Placebo is an interesting package that will record your actions with AWS and replay them as if you were interacting with the server.
- **Think "*server-lessly*" and avoid infrastructure**: It can take a while to develop the serverless mindest. When developing, consider if you actually *need* a database, or if a queue can be adopted instead.

- **Stage different environments**: When testing and staging, Jones recommends using CI for multiple production environments (Blue/Green Deployment).
- **Deploy globally**: Using a geographically distributed server arrangement can increase speed and security. AWS Lambda services can host on 11 regions, so that anywhere on the planet can reach a 20 millisecond ping.

Example: Kickflip SDK

So how do we build an authenticated API, with low-latency, low cost, that is infinitely scalable, without having to worry at all about server operations? Let's turn to an example serverless implementation in action.

Kickflip is an SDK that brings live streaming video to mobile applications. A "**live stream**" is essentially just a combination of separate MP4 files, along with a manifest that determines the order of the videos. Since a real-time video stream service wouldn't need to keep large amounts of video data around for later use, it is an ideal application for a serverless environment.

Kickflip uses a hybrid architecture of HTTP and non-HTTP event sources to trigger server creation from a mobile phone upload, which updates the manifest file so that end users view the latest video chunks. To do all this, Kickflip uses a combination of services: API Gateway for authentication, an API constructed with Lambda, Zappa, and Flask, file storage using S3, and CloudFront for global content delivery. The simplified flow is as follows:

1. The **client** authenticates with the **API**. Kickflip uses Amazon's authentication API key generation service,

but a custom identity access management handler could work here as well.
2. The **API** returns a short-lived federation access **token** which can only be used to upload a file into a specific S3 bucket.
3. The **client** receives the token, and uses it to upload the video.
4. An AWS Lambda **server** is executed in response to the new video upload, and the stream manifest is updated. This upload acts as the event-source.
5. Content is served on the **CloudFront** delivery network for low-latency.
6. Users see the latest video stream on their device.
7. The server is destroyed and temporary access token revoked.

Jones demonstrates that with the strategic pairing of technologies, a serverless video streaming service can be developed in only 42 lines of Python.

Building Serverless API Backends

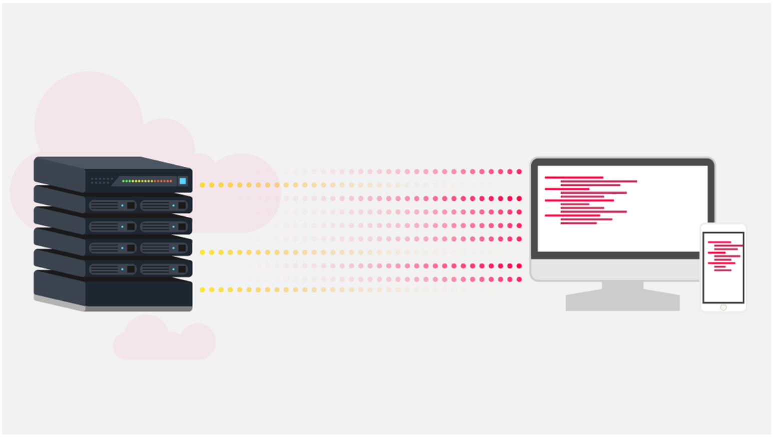
The serverless movement represents a profound paradigm shift in our ability to create impressively scalable web services. Rethinking how events can spark temporary server iterations can be an extremely cost effective solution for microservices and large projects alike.

With all the small connected services being deployed in this manner, the serverless arrangement also reiterates the rise of composable enterprises that depend on many different services to thrive; cementing the web API's position as an important cog in modern and future web communication.

Additional Resources

- Awesome Serverless: Helpful curated list of serverless tooling and helpful information
- What is Serverless Computing?
- Ten Attributes of Serverless Computing Platforms
- The Serverless Stack: Step-by-step tutorials for creating serverless React.js apps

Putting an End to API Polling



APIs have been around for a long time. While that means there's a lot of great tools from a lot of amazing developers, it also means that, as a community, the API space has held on to some practices for a long time – some would argue too long in many cases.

One such holdover, according to some developers, is the concept of **polling**. While polling itself is not a bad thing – after all, it's a simple implementation of an endpoint call – many argue the effects of constant polling require a solution, and an immediate one at that.

Enter REST Hooks. Today, we're going to take a look at the concept and application of REST Hooks, and exactly why some argue for their necessity. We'll discuss some objections to the idea of polling, and the responses from their supporters.

What is the Polling Madness?

Polling Madness is a concept championed by API provider Zapier that simply states the pattern of calling an endpoint for new data, or "polling", is wasteful. The idea is that the constant polling of an endpoint is wasteful in terms of resources committed to the action from the developer, in terms of the traffic seen by the vendors, and in terms of actual result to effort. While there are some objections to this concept, it's got some merit to it, especially in the modern day of lean, efficient processing.

> *Polling is the same as the refresh button. It's not a viable solution*.

Part of why polling has stuck around for so long is because it is ubiquitous – until recently, there were very few effective ways to limit the negatives of polling, and because everyone was doing it, it was a very hard proposition to get other developers to move away. At best, you'd get a new solution that was a distant third option, and at worst, you'd get a powerful but ignored product.

The problem is that polling is essentially just hitting a refresh button – and, just as if the user was hitting the refresh button themselves, depending on the refresh button for vital functionality is simply unacceptable.

One Solution: REST Hooks

A solution from Zapier is actually quite simple – POST a subscription /api/hooks that collates hooks at a target URL, which then pings the resource requester when a change is noted. It's a subtle change, but it moves resources from constant rechecking in active fashion to passive waiting for updates.

Counter Arguments

Many argue that this issue is not really an issue at all - many seem perfectly fine with the current status, or at the very least, the current solutions on hand.

One of the chief arguments against REST Hooks is that it flies in the face of REST itself. REST is specifically supposed to be session-free, and so the idea of creating a constantly polling, static URL on a session-less system seems counterintuitive.

Further issues are raised with the idea that there's no current solution which does what REST Hooks is intending on doing. Some would argue that this, too, is false – TCP/IP websockets should be capable of doing what is being asked of the REST Hook.

The issue really, then, is one of effectiveness. Yes, it is true that REST is session-less in most implementations, but that does not mean that one can't benefit from the positive attributes of sessionful communication while maintaining the overall benefits of REST. So too is the argument for TCP/IP somewhat flawed - yes, TCP/IP websockets can somewhat do what is being asked, but there are issues (loss of control for the user, less customizability, etc.) in those approaches.

Watch: Audrey Neveu

This chapter focuses heavily on REST hooks. For more advice on real-time streaming APIs, watch Audrey Neveu present at the 2016 Platform Summit. Watch her talk here

So REST Hooks should be seen for what it is - one of many solutions, unique in its application, that can be used to solve a complex, consistent issue in a new, unique way.

Implementing REST Hooks

Conceptually, the implementation is rather simple, though it does come with specific concepts that must be adhered to if the implementation is to be considered properly as a REST Hook. REST Hooks are essentially URLs which collate changes that would usually be polled, providing a URL that can be actively monitored.

Create a Subscription

In order to escape the issues of polling a webhook, the webhook itself needs to be tied to a subscriber URL.This can be done with a simple POST element:

```
POST <subscribe_endpoint> \
    -H Authenticated: authenticationSolution \
    -H Content-Type: application/json \
    -d '{"target_url": "https://hooks.zapier.com/hooktest>",
        "Event": "user_created"}'
```

This chunk of code sets up the URL by which you can direct polling webhooks to the REST Hook. It uses the standard authentication/authorization solution you've already implemented. Zapier responds with three elements:

- Authenticated User
- target_url
- event

All three of these items are stored in local persistent data, which can be called upon when accessing the REST Hook. Of note is that Zapier recommends a few status codes for certain behaviors. For successful subscriptions, a 201 code, signifying “Created”, should be returned. Likewise, when a non-unique subscription URL is set, a 409 code should be returned, signifying “Non-Unique”.

Sending Hooks

Now that the hook is setup, we need a way for the API to actually send the data. We can do this by implementing the following POST code:

```
POST https://hooks.zapier.com/testhook \
    -H Content-Type: application/json \
    -d <json payload>
```

Zapier notes that they typically expect an array of objects – if the API sends a single object only, it needs to be wrapped in the following element array:

```
[ {“element1”: “Content1”, “element2”: “Content2”} ]
```

Unsubscribing and Setting Up a Global URL

Finally, a DELETE call is made to unsubscribe using the following code:

```
DELETE <unsubscribe_endpoint> \
    -H Authenticated: authenticationSolution \
    -H Content-Type: application/json \
```

In order to define a polling URL permanently, there is an option to set up a trigger. This trigger will allow a user to set up a permanent data point, rather than having to create a new one to poll the endpoint. The polling URL alleviates this issue, creating a permanent location to draw information and data from.

Alternatives

> WebSockets are bi-directional - you tell both the client and the data to send that data to each other. Server-Sent Events is unidirectional - its an open channel where data can be streamed from the server to the client.

There are some alternatives, of course. **WebSockets** can provide a constant back and forth connection, negating polling in a way by moving the relationship from single-directional “pull” to “push”. This is questioned by some, though, as being essentially “polling 2.0”, in which the polling is from two sources rather than one.

Server-Sent is yet another solution. In this approach, the server does not respond to a poll request, but instead constantly “pulses” changes to the client itself. The problem here is that, while it removes the polling technically, it also removes much of the control from the user, and forces them into a passive state.

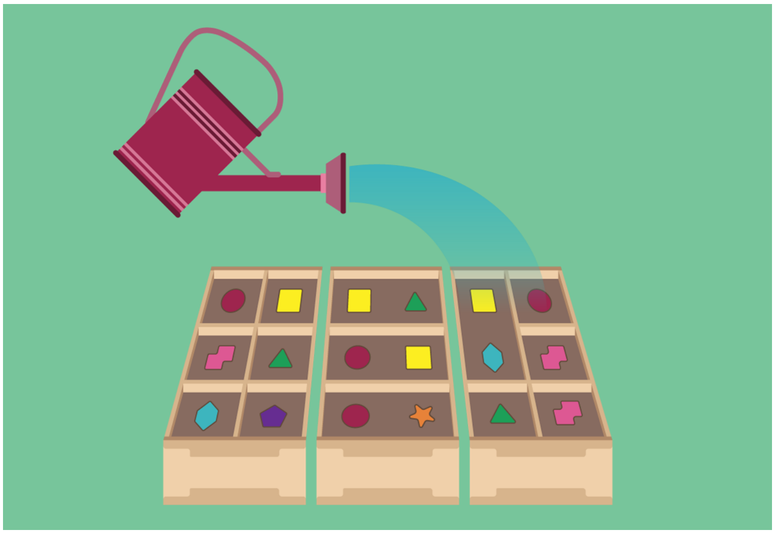
Because of this, many feel that both solutions are essentially inverse versions of polling, and are thus not acceptable solutions. Given the use case, they might be appropriate, and in others, they could be opposite of what is needed.

Conclusion

Implementing REST Hooks can solve a huge problem, but it rests on the developer to figure out how much an issue it truly is - on the one hand, polling is a consistent resource dedication that can result in slower response times when multiplied over many hundreds of connections.

On the other hand, there are many solutions that already exist, though they each have strengths and weaknesses opposite that of the Zapier REST Hook solution. If a developer does not have a polling solution, this is a fine one indeed - if, that is, the developer believes they have a problem to begin with.

6 Ways to Become a Master Microservice Gardener



The API space is an extremely interesting one, largely due to it straddling a dichotomy of opposing requirements. On the one hand, an API must be innovative, quick to change, and ever evolving. On the other, consumers demand stability, and with it, the security that comes with proven solutions.

The constant battle between old and new, centralized and distributed, has led to a unique range of solutions and approaches to solving the dilemma. One such solution is the concept of **microservice gardening**, the idea of being a "landscaper" for an API or suite of microservices.

Microservice gardeners must use innovative build techniques that embrace a distributed architecture, avoiding the Kafka-esque monolith at all costs.

In this piece, we're going to discuss what it means to be a "microservice gardener", and present **six ways** API providers can excel as masters.

Speaker: Eric Wilde

This chapter was inspired by Wilde's session at the 2016 Platform Summit. Watch the full talk here

6 Ways to Become a Master Microservice Gardener

> "Companies are increasingly seeing themselves as restructuring themselves into something that has organizational boundaries that have technical alignments. […] The idea is that you set up your company in a way that what you're doing is also aligned with how you're doing it. Then it's easier for you to restructure and rebuild yourself when you have to."

Imagine a garden — a range of flowers, each with their own needs and purposes. This is the **API microservice landscape**. Just as each flower or vegetable needs specific fertilization and specialized care, it is the role of the provider to give attentive care to API microservice consumers as an **API gardner**.

As a landscape manager, it is your duty to control and augment the environment to allow for the greatest growth and prosperity. Failing to do so does not just hurt the environment, it could possibly destroy it. Within an API landscape, not supporting users where they are can choke your system and any potential user base you have yet to develop.

Both of these issues arise from a fundamental shift in how APIs are produced and managed. Whereas APIs used to be monolithic and shackled to larger systems, we are now in an era of microservices, all designed to do specific things in concert with other solutions, which forces a containerization and decentralization of the API model.

Now that we know what a Microservice Gardener is, let's look at some techniques we can apply to master this art.

1: Use Bimodal IT to Avoid Stagnation

> "Imagine that you have the castle, which is your legacy system. Now you have these new things coming around where people say 'oh we should do microservices, and it will be fast, and we can do interesting things.' They're highly enthusiastic at what they're doing, but they also need guidance."

Bimodal IT is the concept of dealing with disparate styles of work that nonetheless function perfectly well separate from one another. "Bimodal" means two modes, and in this case, the term matches the function — one focused directly on *predictability*, and the other on *exploration*.

The **exploratory** type focuses on unique and innovative solutions that result in a requirement to explore rather than to depend on known, constant functionality. The second development track is the inverse, a **dependable codebase** that is known, proven, and well documented, thus resulting in predictable results.

While an API provider could simply direct their API to only support one of these types of content and the resultant users through the effect of eschewing innovative, unproven solutions, developers would be excising a large portion of their potential power and user base by doing so. Bimodal IT approaches are designed to support both concepts and approaches. By combining predictable product evolution and documentation with innovation and exploratory tools, developers can harness the "best of both worlds."

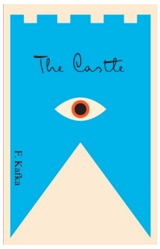
2: Avoid the Kafka-esque Monolith

Erik provides a very good example of what a centralized monolith results in by example of *The Castle*, a novel by Franz Kafka. Throughout the events of *The Castle*, the protagonist, known only as K., suffers the tribulations of bureaucracy from a government which erroneously called him into their village for a job that was non-existent.

Redirected from contact to contact, ignored by higher-ups with the simple instruction to talk to their subordinates, and never given explanation for any action of the officials, K. exposes the bureaucratic nonsense of the village surrounding the castle, much to the villagers' chagrin.

Poorly designed and implemented APIs (and even the well designed and implemented ones) sometimes result in this sort of behavior. The user, attempting to use the API for whatever function they deem, are lost in the structure of subordination and redirection. The API design methodology used can somewhat augment this, but giving APIs too much structure could end in a bureaucratic result.

It doesn't have to be this way, of course. APIs need structure, but just as important as structure is the need for **guidance**. Designing an API to not only get the job done, but to get it done in an understandable way while allowing for innovation and explorability is a key tenant of bimodal IT. Essentially, you're joining the old — that is, the bureaucratic — with the new.



To Wilde, monolithic architecture is similar to the nightmarish bureaucracy of Franz Kafka's The Castle

3: Design In a Way That Promotes Further Iteration

First and foremost, consider that all of these issues, even those in *The Castle*, derive from the principle foundations upon which the system is built. No single API specification, documentation, and design approach is perfect, and as such, avoiding dependence on a single monolithic solution will do a lot to resolve this.

When developing an API, keep in mind not only the intended functionality, but the ability to extend that functionality. If an API is designed to perfectly fit only the current requirements, it will be a perfect API — but only for a moment in time.

This is much more of a "top level" concept, but it's one that needs to be adopted as a mantra for any API provider. When considering an API, don't think about what the user today wants, or even what the user a year from now will want — future proofing is difficult as technology fluctuates so rapidly. Rather, Wilde recommends the best solution is not to code the functions you expect to need in the future, but to enable the easy additions of those features to an extensible code base. Allowing further functionality through extensible classes and hooks to combine functions creates powerful, longevity-driven solutions.

4: Harvest Concepts and Incorporate

Harvesting and introducing new concepts should be easy. In *The Castle*, K. not only runs into confusion, he also encounters people actively resisting his attempts to resolve the confusion. Your API should not do this.

Part of a solution is simply allowing your API to be explorable. Don't lock everything down via obfuscation and layers of haze — explain how an API does what it does, how this can be extended, and how to call this functionality.

Essentially, it should be easy to identify what an API does and to use it for new purposes — failing to provide a system to do this kills any innovation. This has been seen time and time again, and is a prime reason open source has been as successful as it has been. A locked down ecosystem works perfectly — but only if everything is kept internal.

5: Distribute the Seeds: Adopt True Microservices Arrangement

The greatest way to become a master microservice gardener is to move away from centralization. Give teams of developers more **autonomy** in how solutions are developed and implemented, and allow for greater service discovery on the user side. Essentially, move away from a centralized API approach and more towards a true microservice arrangement.

Developers have the tendency to thinks of themselves as the "central authority". The developer has made the code, has restricted it, has documented it, and by this process, has complete control. While this is fine for single use APIs or small teams, this is not how an interactive API platform should function. When a provider functions this way, they are being despotic — and with despotism comes inefficiency and bureaucracy.

Instead, **approach API development as a benevolent democratic leader**. Allow for more personal use, and do everything you can to support growth and understanding. If a gardener cuts too close to the root, they can kill a flower — if a provider does not provide any amount of support for anything other than their specific user case, they'll do the same to the API.

6: Prune the Service Surface

When it comes down to it, what truly matters to the microservice consumer is the **surface** of the service. The surface is unique to each offering, but may consist of varying degrees of public developer portals, API metadata, documentations, or other functional summaries and integration methodologies.

While it's tempting to take the advice here and reveal the entirety of your API, that's a bit too far in the wrong direction. Consider the simplicity of offering a service surface and making well-defined concepts that are fundamentally shallow in their functionality. By doing this, you’re creating an extensible solution while mitigating the potential damage of innovation for innovation’s sake.

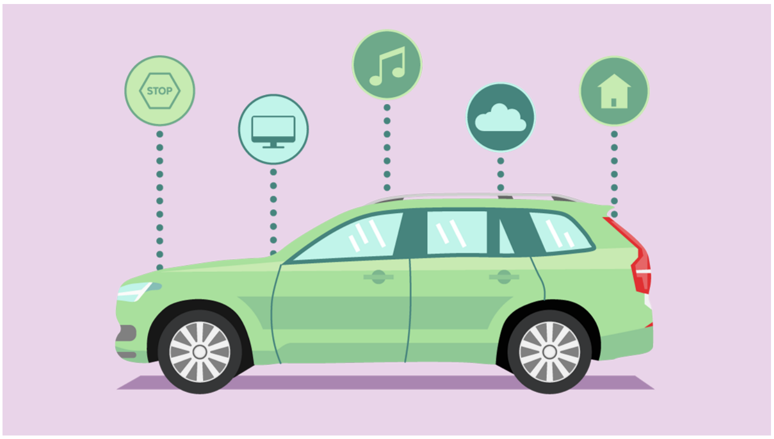
Nurture API Ecosystem Growth

> “Instead of having a top-down process to design and document and describe APIs, use more of an ecosystem-like approach… Try to build things in such a way that you make the conditions under which good things will happen.”

As providers and developers create API ecosystems, they need to have a sea change in concept. To do so, API providers must eschew centralization in favor of innovation and new developments.

The best a developer can hope for is this — create a system in which the API is understood and extensible, and from there, create the conditions for success. Even if that success does not immediately happen, preparing should mitigate those possible issues while magnifying the potential for success.

Is Your API Automotive Grade?



Imagine that a user has plugged into your API and triggers a request, only to be met with an error message. Now imagine all of this is happening while they're traveling at 70mph on the highway. For Henrik Segesten, cloud domain architect at Volvo, making sure that this sort of thing doesn't happen isn't the stuff of bad dreams — it's part of his day to day routine.

Segesten and co. can't afford to make the same mistakes that API developers working on a plugin for, say the Nest thermostat can. Rather than cause a minor inconvenience for the user, the potential impact of flaws in an automobile scenario could be life-threatening. From the transmission to the software powering onboard electronics, **automotive grade** equates to rigorous testing and extremely high reliability standards.

But this is about more than just ensuring that automobile APIs are robust and bug-free – after all, that should be the aim of any API developer. Rather, it's about improving **longevity** (something we're writing extensively about currently) and compatibility.

Right away, we can see that there's something of a disconnect here: it's extremely difficult, if it's even possible at all, to ensure that **every** step in the API process can be monitored with the sort of obsessive quality control demanded by true automotive grade standards. Imagine, for example, telling API consumers that they're only allowed to use your APIs if they do all of their coding on a MacBook Pro, produced no later than 2015, running OS X 10.10.5 or above.

But, as we'll see below, there are a number of useful concepts relating to **automotive grade** that can make a lot of sense in the context of how we design and update APIs as well.

Speaker: Henrik Segesten

This chapter was inspired by Segesten's session at the 2016 Platform Summit. Watch the full talk here

Test It Like it Has to be Automotive Grade

We've previously written about the importance of testing, both your APIs themselves and your documentation, yet it's something that some API developers continue to overlook.

Think about some of the processes that a car, and its component parts, need to go through before the latest model rolls off the production line:

- Testing for extreme temperatures
- Analysis of how the vehicle performs on different terrains
- Adequate warning when something isn't working properly
- Lots of redundancy built in

There's an important lesson from automotive grade in there for API developers: it's highly likely that the tests outlined above are **far** more thorough than anything you put your APIs through…unless you work at Volvo with Segesten, where they're trying to establish API architecture that can remain solid for decades.

While "bank grade" and "military grade" are often used in relation to security and construction materials respectively, **automotive grade** is a term that's a little more tightly defined.

Segesten outlines automotive grade as being a **hardware concept** with "a measure on the quality level of all parts in a car…with an expected lifespan for each and every component, down to every nut and bolt, that is equal to or greater than the expected lifespan of the car." That usually equates to around **30** or **40** years.

In the context of software development, we need to loosen up that definition a little bit. We aren't necessarily trying to create an API that will last for that period of time so much as, as Segesten jokes, giving some thought to "the poor girls and guys who need to re-implement your API in twenty years' time." He highlights the fact that "automotive grade translates to longevity, so you need to design with longevity in mind." In other words, just because something seems to work *for now* doesn't mean that it's the right choice.

Automotive Grade and Futureproofing

Segesten advises looking backwards in order to plan for the future to consider what the main drivers have been in web-oriented software and API development throughout the past 30 years. It's worth reproducing that list here:

00s:	**90s**	**80s**
SOAP	HTTP	FTP
JSON	CORBA	SMTP
XML	DCE	TCP/IP
	PGP	Telnet
	X509	PPP
	ONC/RPC	

But this doesn't mean that all API developers should be shunning RESTful development in favor of standards that have been around for 30+ years. Rather, it's an argument to ensure that your API uses technology that **can** stand the test of time. In their lifespan, typical APIs will need to be "re-implemented several times along the way … because the old system has died for some reason. Pick your technology wisely."

Now, let's think about the protocols/languages etc. commonly used in the API space in 2017:

- JSON
- REST
- SOAP

- XML
- JavaScript
- Open API/Swagger
- GraphQL

It's interesting to note that, of the above standards listed by Segesten, only JSON, SOAP and XML appear across multiple decades. This doesn't mean that you should only consider using these data formats when building an API. Rather, it's evidence that ten years isn't a long time in the world of software development; developers can ill afford to use the latest, hippest languages and standards **unless** they're pretty confident that they have what it takes to stand the test of time. Based on what we've seen so far, REST appears to fit that bill.

But it's worth remembering, as Segesten says, that there's more to an API than just the technology you use; the best APIs can thrive regardless of the technology that power them.

Automotive Grade and Backwards Compatibility

As important as it is to futureproof an API, it's just as important to ensure that it's **backwards compatible** too. For example, what would happen if a car buyer disables all the connectivity features in a car, uses it for ten years, then sells it? When the next buyer turns on a connectivity feature, the car will be consuming the associated APIs for the first time.

The lesson here is that **old versions** of APIs never die, and you'll inevitably need to maintain them because:

- You can't be sure that all users will migrate to later offerings
- Certain products/hardware may not be capable of communicating with your newest API
- You need to remain compatible with hardware that will go offline, i.e. asynchronous communications.

Designing APIs with backwards and forwards compatibility in mind from the start – using extensibility, optionality, and so on – is much easier than trying to simultaneously manage tens of different versions … as the folks trying to support users with versions of Windows that are decades old will probably tell you. In fact, there is a strong case that versioning shouldn't even happen at all with web APIs.

Lastly, Segesten reminds us that it's a good idea to keep things as simple as possible. "If you have to have complexity…try to put all of that complexity on the server side, because you can update that all the time but the API should never have to be updated." When that hypothetical ten year old car we mentioned above finally hooks up to a Volvo API, it should be more or less good to go if you make most of your changes server-side.

Final Thoughts

As we've seen above, particularly given the lack of control API developers have over how and where their products are used, *actually* producing an automotive grade API is extremely difficult. If we take the term at face value, i.e. remaining valid for 30 or 40 years, it may not be possible at all.

Any experienced API developer knows that there are too many variables to control to guarantee that a service will still be working in 2047. With that in mind, it might be better to consider employing the principles of "automotive grade", such as:

- Using **futureproof**, or at least as futureproof as possible, languages and methodologies
- Have **compatibility** strategies in place – how do you version handle your APIs?
- Keep things **simple**, both in terms of functions and documentation, and testing as thoroughly as possible to make sure it all works as expected

The good news is that, for the majority of us, our API consumers will never be using our products or services at 70mph. But that doesn't mean that it's not a good idea to prepare like they might be…

Why OAuth 2.0 Is Vital to IoT Security

The internet is fundamentally an unsafe place. For every service, every API, there are users who would love nothing more than to break through the various layers of security you've erected.

This is no small concern, either — in the US alone, security breaches cost companies in excess of $445 Billion USD annually. As the **Internet of Things** (IoT) grows, this number will only climb.

The problem is our considerations concerning security are for modern web services and APIs — we rarely, if ever, talk about the coming wave of small connected and unconnected IoT devices that will soon make this an even greater concern.

From the connected fridge to the smartwatch, the **IoT** is encompassing many new web-enabled devices coming to the market. As we're designing new API infrastructures, Jacob Ideskog believes that the "IoT is going to hit us hard if we're not doing anything about it."

Thankfully, there's a great solution by the name of OAuth. **OAuth 2.0** is one of the most powerful open authorization solutions available to API developers today. We're going to discuss OAuth 2.0, how it functions, and what makes it so powerful for protecting the vast Internet of Things.

Speaker: Jacob Ideskog

This chapter was inspired by Ideskog's session at the 2016 Platform Summit. Watch the full talk here

What is OAuth 2.0?

OAuth 2.0 is a token-based authentication and authorization open standard for internet communications. The solution, first proposed in 2007 in draft form by various developers from Twitter and Ma.gnolia, was codified in the OAuth Core 1.0 final draft in December of that year. OAuth was officially published as RFC 5849 in 2010, and since then, all Twitter applications — as well as many applications throughout the web — have required usage of OAuth.

OAuth 2.0 is the new framework evolution that was first published as RFC 6749 alongside a Bearer Token Usage definition in RFC 6750.

What Does OAuth Do?

While by definition OAuth is an open authentication and authorization standard, OAuth *by itself* does not provide any protocol for **authentication**. Instead, it simply provides a framework for authentication decisions and mechanisms.

> "OAuth does nothing for authentication. So in order to solve this for the web, we need to add some sort of authentication server into the picture."

That being said, it does natively function as an **authorization protocol**, or to be more precise, as a *delegation* protocol. Consider OAuth's four actors to understand how it accomplishes this:

- **Resource Owner (RO)**: The Resource Owner is the entity that controls the data being exposed by the API, and is, as the name suggests, the designated owner.
- **Authorization Server (AS)**: The Security Token Service (STS) that issues, controls, and revokes tokens in the OAuth system. Also called the OAuth Server.
- **Client**: The application, web site, or other system that requests data on behalf of the resource owner.
- **Resource Server (RS)**: The service that exposes and stores/sends the data; the RS is typically the API.

OAuth provides delegated access to resources in the following way. Below is a fundamental flow in OAuth 2.0 known as **implicit flow**:

- The Client requests access to a resource. It does this by contacting the Authorization Server.
- The Authorization Server responds to this request with a return request for data, namely the username and password.
- The Authorization Server passes this data through to an Authentication solution, which then responds to the Authorization Server with either an approval or denial.
- With an approval, the Authorization Server allows the Client to access the Resource Server.

Of note is that OAuth 2.0 supports a variety of token types. WS-Security tokens, JWT tokens, legacy tokens, custom tokens, and more can be configured and implemented across an OAuth 2.0 implementation.

Unique IoT Traits that Affect Security

Now that we understand OAuth 2.0 and the basic workflow, what does it mean for securing the Internet of Things (IoT)? Well, the IoT has a few unique caveats that need to be considered. Ideskog notes that IoT devices are typically:

- **Battery powered**: IoT devices are often small and serve a particular function, unlike server resources which have massive calculation-driven platforms and consistent, sanitized power flow.
- **Asynchronous**: They are partially or completely offline, connecting only **asynchronously** via hub devices or when required for functionality.

- **Lean**: Lastly, IoT devices usually have limited calculation capabilities, and depend on central devices and servers for this processing functionality.

Despite all of these caveats, **IoT devices, are extremely attractive targets to attackers** due to their known **single use functions** and relatively **lax security**.

Proof of Possession

Due to all of these caveats, the OAuth workflow is strikingly different — we are, in fact, using a methodology called **Proof of Possession**. Consider a healthcare scenario, in which a doctor must access an EKG IoT device. Since the IoT device cannot perform the same authentication process as a full client device can, we need to do a bit of redirection.

The start is normal. The Client sends an access request to the Authorization Server. From here, the Authorization Server contacts the Authentication Server, which prompts the Client with Authentication Data. When this is provided, the Authentication Server authenticates to the Authorization Server, which issues an authorization code to the Client:

```
authorization_code = XYZ
```

From here, we deviate from the standard OAuth workflow. The Authorization code is a one-time use proof that the user is Authenticated, and this code can be used to further contact that IoT device as an authorized device. The code is not something that can be used to directly access data as other OAuth tokens are, it is simply proof that we are who we say we are and that we've been authenticated and authorized.

The Client then generates a key (though this key can also be generated server side) to begin the connection process with the IoT device, sending a packet of data that looks somewhat like this:

```
Client_id = device123
Client_secret = supersecret
Scope = read_ekg
Audience = ekg_device_ABC
authorization _code = XYZ
…
Key = a_shortlived_key
```

With the data in hand, the Authorization Server now responds to this packet by providing an `access_token`; a reference to data held in the Authorization Server memory to serve as proof of possession of both authentication and authorization:

```
Access_token = oddfbmd-dnndjv…
```

This is the final step — the client is now fully and truly authenticated. With this `acess_token`, the client can start a session on the IoT device. The IoT device will look at this `access_token`, and pass it to the Authorization Server (if it's a connected device) asking for verification to trust the device. When the Authorization Server accepts the verification, it passes a new key to the IoT device, which then returns it to the Client, establishing a trusted connection.

Disconnected Flow

What happens if a device is unable to ask the Authorization Server for verification due to power or calculation limitations? In this case we can use something called **Disconnected Flow**. A key point for Disconnected Flow is unlike other OAuth 2.0 solutions, this eschews TLS (Transport Layer Security) by nature of the Resource Server being a disconnected device with intermittent connectivity and limited communication and processing power.

In this case, we're actually shifting the parties around somewhat. The EKG machine is now the client, and another IoT device, a test tube, is the Resource Server. First, the EKG machine authenticates and authorizes in the same way as before:

```
Client_id = ekg_device_ABC
Client_secret = supersecret
Scope = read_result
Audience = connected_tbie_123
Token = original_token
...…
Key = a_shortlived_key
```

Once this is received by the Authorization Server, the server replies not with the access token in the former structure, but instead an access_token in JWT (or JSON Web Token). This token is a by-value token, meaning it contains the data fed to it and the response. Whereas our first string referenced a memory location in the Authorization Server, the JWT has all of the data in a single key string.

From here, the JWT can be converted into other formats for easier reading by the test tube. By design, the test tube is crafted to trust the Authorization Server in a process called **Pre-provisioning**. Because of this, when we send the Client token in JWT (or whatever format that's been chosen), the tube implicitly trusts the key as long as it originated from the Authorization Server, and begins a connected session with the Client.

Ideskog notes there would technically be 2 token types involved in the flow above: a signed JWT would contain an encrypted token (JWE), which has a key in it that is later used for the communication channel. The JWS (commonly called JWT) isn't necessarily encrypted, and is usually in plain text and signed.

Real Worth Authorization Failure

To see exactly why this is all so important, consider some **real world authorization failures**. One of the most visible failures is known as The Snappening, a leak of over 90,000 private photos and 9,000 private videos from the Snapchat application.

Most of the blame for the Snappening came from users using **unauthorized third party applications** to save Snaps. These third party applications did not utilize OAuth solutions, meaning when remote access users attempted to use the undocumented Snapchat URL that the third party application relied on, they were able to spoof as authorized users and retrieve content without proper token assignment.

This a great example of OAuth 2.0 vs. no implementation, as we have essentially a "control" application (Snapchat secured by OAuth) and a "test" application (the unauthorized applications tying into the undocumented API). With improper authorization integration, content was allowed to leak through an insecure system with relative ease.

Had the third party application properly implemented an authorization scheme, this would never have happened.

This issue is only relevant to non-IoT things, though — it’s just a photo sharing application, right? Wrong. Consider now this same fault of security for something like an IoT button that triggers replacement business items. Attacking this device can result in man-in-the-middle attacks to capture addresses of order processing servers and even spoof orders to the tune of thousands or hundreds of thousands of dollars.

OAuth Embeds Trust into the IoT

Applying OAuth to the IoT makes it truly extensible and customizable. Using OAuth, we can build systems based on trust that use fundamentally secure resources and communications protocols. OAuth is, by design, all about **trust**.

This trust is key to securing the IoT. For connected devices, Proof of Possession can solve most security issues. For constrained environments by either connectivity or processing calculative power, devices can be secured using pre-provisioning that is independent of TLS, and does not require the device to be online at all times.

It should be noted that JWT, JWS, and JWE are all helpful tools, but all work with JSON. For lower processing environments, sibling binary tokens such as **CWT**, **CWS**, and **CWE** can be used as they cater well to building on **low power** and **limited scope** devices.

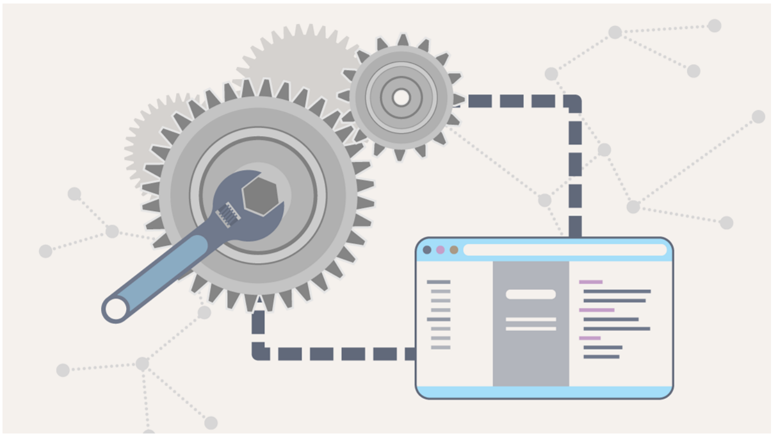
Conclusion

This isn’t a game — though having lax security can be convenient for innovation and experimentation, when it comes to the IoT, this is a dangerous approach. IoT devices might be underpowered and single use, but they, as a network, are powerful.

Remember that a network is only ever as secure as the sum of its parts and the weakest point of entry to its ecosystem. Failing to secure one IoT device and adopting a security system based on inherited security can result in a single IoT device comprising every device connected to it.

OAuth 2.0 can go a long way towards solving these issues.

4 Design Tweaks to Improve API Operations



We've previously discussed best practices when it comes to designing an API with quality developer experience. But what will the long term **operational** repercussions be for the design moves we make now?

For example, if URLs are designed without metadata to describe actions, later on, product owners will have a difficult time staring at unintelligible logs. Or, if microservices aren't orchestrated correctly, you run the risk of long load times queuing multiple API calls in a mobile environment. These are only two of the many operational consequences that many API owners overlook while designing their APIs.

Today, we'll consider some methods to make web Application Programming Interfaces more operationally efficient and responsive to the way that clients will consume them. Led by Nordic APIs veteran Jason Harmon, we'll cover the most common **operational anti-patterns** that could break an API. We'll offer ways to remedy these poor design situations, that if not addressed, could cause some serious migraines later on in the API lifecycle.

Speaker: Jason Harmon

This chapter was inspired by Harmon's session at the 2016 Platform Summit. Watch the full talk here

From simple, yet often overlooked issues, such as using the appropriate HTTP method, to more complex composition advice, let's consider four **anti-patterns** found in the wild all too often.

> "To some extent, the way you design your API can set you up for failure."

1: HTTP GET instead of POST

API providers commonly implement an HTTP `GET` call where a `POST` should have been used instead. A reminder to use the correct HTTP method is by far the most basic piece of advice here, yet surprisingly, it's a common error that can cause some major damage.

Harmon related a story from his experience at Typeform, where a small subset of users in web browsers were hitting the "back" button, causing their sessions to lose all data. By changing the call type from `GET` to `POST`, Typeform was quickly able to solve their caching issue, as the HTTP RFC states, `POST` is not allowed to be cached by anything.

The lesson here is to watch out for cached calls from browsers or proxies. If you do encounter unexpected behavior, Harmon recommends to first "look for the `GET`s… using `POST` instead is an easy fix." If you're stuck in a situation with erratic cache issues, he adds that adding an extra query string with a randomly generated cache-buster to the `GET` call (i.e. `?cache_buster=[random]`) could solve a recurring issue.

2: Letting clients constantly poll APIs

Many API providers should reconsider how they allow clients to update data. Too often, the client sets up constant polling on API endpoints. When a large dataset is involved, and queries are occurring continually, such as every 30 seconds, the number of calls can really add up. Large volumes of calls to an AWS server can be expensive as well.

Typeform is not immune to the API polling issue. Typeform end users want to know if there is updated data on their form, and therefore set up a constant polling service to see if the data has been updated. Since Typeform is among the Zapier compatible apps, meaning that users can create customizable "zaps" that tie in Typeform functionalities, the number of services continually requesting new data skyrockets.

To avoid constant polling, Harmon recommends you build and launch **webhooks** for your service, and convince your consumers to use them. But first, **find out what the rate of change for your data is**. For example, The average rate of change for Typeforms are typically 1-2 weeks. Identifying a pattern for the rate of change in your data can help you design your webhooks to be as lean as possible.

Harmon recognizes one catch to this approach — he recommends still using a **polling API** alongside the webhook payload to carry out sporadic system checks and perform large downloads.

3: Rigid hierarchy in microservices causes latency

Lately, many SaaS providers have been transitioning to the microservices architectural style. Microservices are lean, well-bounded components dedicated to processing a specialized functionality. This is great for structural segmentation, but if microservice communication is not orchestrated well, requesting data can cause serious latency.

> “Build your microservices to be externalized from the ground up”

The problem with microservices is that a client requesting a large number of functions could end up sending dozens of separate calls, leading to a **massive query load**. This is especially problematic in mobile environments, where calls must be queued in series instead of being executed in parallel to one another. For some environments, this process is simply too slow.

According to Harmon, the solution lies in a **BFF**. No, not a best friend forever, but a **Backend-for-Frontend** that acts as a shim to help compose microservices.

A BFF is a lightweight layer that acts as an orchestration API. It could be built as a Node.js service, or however internal developers see fit. The goal of such a shim is to, with one call, have the client receive all the resources packaged together in the way the client wants. That way, to put it in Harmon's words:

> "They're not gluing together a model in the browser, the model's already given to them the way the wanted it in the first place."

insert picture of BFF[A BFF has been implemented to solve microservice design issues at both Paypal and Typeform. By decreasing JSON packages, a BFF can cut down JavaScript processing.]

Allowing clients to call a microservice directly without any composition layer is poor design as it doesn't consider the use cases and client limitations. Constructing a BFF layer is a possible solution, but it should be noted that GraphQL is another potential solution as well.

4: Generic actions

When we design URLs, Harmon reminds us that detailing **actions** matters. He recommends to note state transitions within the URL, and to pass along short descriptions as metadata. If you don't describe state transitions outside of the typical CRUD verbs, you run the risk of having very generic, unreadable logs.

This problem is often called **protocol tunneling**; with generic URL names you often lose perspective, and then when an action breaks, it's hard to locate the affected calls and analyze trends. If we design URLs as generic phrases that tell us nothing about the entire story, then error diagnosis will be difficult.

> "URLs are a key component in how you operationalize the API"

Harmon notes this is especially important for product owners using tools such as the ELK stack to visualize data sources to determine insights. When designing URLs, visibility is a plus, so consider a good method of identifying these URLs with naming schematic, along with metadata for the actions that are being taken.

```
/resource/:id/generic-name + {action:process}
```

Harmon-ious Mantras to Live By

We've covered much ground with some specific issues to avoid. For more generic design advice, the wise Harmon provides some mantras to design by:

- **Use cases first, then design**: Ensure your HTTP methods are correctly ascribed, and that user behavior won't adversely affect the outcome.
- **Design can influence performance**: Put an end to polling. Rather, design with subscription webhooks and insist that clients use them instead.
- **Structure is good, but be prepared to blur those lines**: Though microservices architecture is about separate division, having too rigid of a structure can result in "unhappy clients and crappy performance."
- **Design can put out fires**: Having a DevOps approach to design early on can put out fires in the long term. Try to make performance more visible.

- **It's all about the logs**: Ultimately, your logs are the transcript of what clients are actually doing. Consider how you can make this performance more intelligible by crafting URLs and metadata that will aid metric analysis and make product owners happy.

Developer experience is the mainstay of most API design discussion, but we mustn't ignore how design moves will affect API **operations** as well. As Harmon points out:

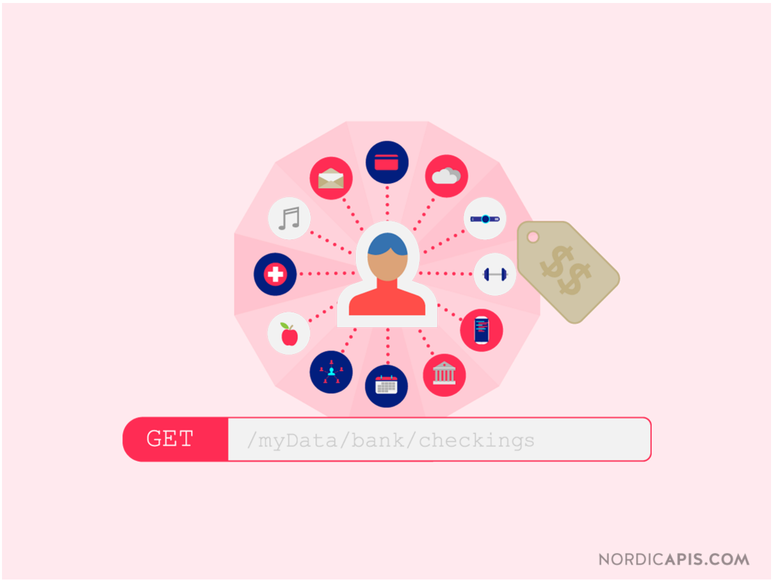
> "API design is not just fanciful usability discussions and lot of fluffy emotions about developers… Developer Experience really is just the first layer."

Following these guidelines, we can improve specific areas to help design on the scale of decades, as operational efficiency is intimiately correlated with platform longevity. By planning for operational improvements now, we can "design in a way that writes sentences, to tell a bigger story later on."

Speaker: Jason Harmon

Also watch Nordic APIs veteran Jason Harmon present his tips on scaling API design. Watch the full talk here

The API of Me



Here's an interesting fact: If you live in the EU your **personal data** is yours. You are the owner of your data, a fact enshrined in law under the General Data Protection Regulations (GDPR). As owners of data and citizens of many economies — internet, application, information, API — we have a myriad of tools and technologies available to mine, mash up, and generally manage our data as we see fit. Moreover, we can exploit our data for our own benefit, selling it to the highest bidder for our own profit.

However, as consumers we don't often consider how we can *exploit* our data. Typically we are only concerned with data security and are not as conscious of using it for our ends. Perhaps this is because the banks, insurance companies, and other corporations that hold our most valuable data are not naturally disposed to making it easily accessed or shared with whomever we see fit.

Enabling the sharing of our personal data in a manner controlled by us is at the heart of some unique concepts. This would allow us to create a **Personal Data Store** to better visualize personal data, and would also enable the **API of Me**, the endpoint by which we can share our valuable personal data in a digital world.

The data revealed by the API of Me is not a single plane; there are different types of personal data from a variety of sources that make up our dataset and describe who we are, what we do, and what we touch. They come with varying connotations and likelihoods of being exposed to an external audience. In this chapter, we'll explore these types of personal data that if unlocked could finally enable the **API of Me** — the programmable endpoint to our digital lives.

Speaker: Chris Wood

This chapter was inspired by Wood's session at the 2016 Platform Summit. Watch the full talk here

Confidential

Confidential data is the data most important to us. These attributes uniquely describe us and allow important things to happen in our life, from our birth to our death. It also includes subsets of information that we would prefer to be extremely closely guarded; data about our health and ongoing health records, for example.

It goes without saying that confidential data is **extremely sensitive**, and when obtained by miscreants can be used to impersonate us and falsely make applications for bank accounts or mortgages without our consent. Most countries and regions have established extensive laws and regulations, like GDPR, to safeguard confidential data. Though safeguarding this data is crucial, *selectively* making it available can aid our daily lives by simplifying the process of data exchange.

In general, due to data privacy constraints, organizations that hold our confidential data limit it to a specific, authorized realm: The **owner** of data themselves, or a **third-party** who is expressly authorized to access this data. The majority of APIs that access this information are therefore either private or partner APIs, with little or no public access. Enrollment, authorization and access are tightly controlled. Such private APIs include the closed APIs that power our online banking platforms, or partner APIs that are the heart of price comparison websites for insurance or other products.

This closed approach to confidential data introduces considerable friction to us as consumers, and requires us to endlessly confirm our identity using physical artifacts like a driver’s license, passport, or proof of address. Only where a formal scheme exists for sharing our identity and confidential data digitally (such as the schemes in the Nordics like NemID and BankID) does that friction disappear. The technology also exists to extend the scope of the platforms that hold this information and to share this data on our behalf; both in terms of protocols or products like Trunomi. It remains to be seen whether we as consumers have the appetite and trust to allow this information to be shared via an API to unlock it for our own benefit.

Financial

Our **financial data** consists of the transactions we make, our banking history, and creditworthiness. Like confidential data, it is of considerable value to us as consumers. Exposing this data to applications like **personal financial management (PFM)** tools can provide helpful visualizations that the atomic data doesn't necessarily deliver.

Our financial data also has intrinsic value to **businesses** who attempt to sell us products and services based on what we spend our money on. Such data is essential to **market analysis** and demographic trends. Obtaining such data is easier for major retailers, especially those with loyalty schemes who can easily correlate spending over time with a real individual.

Financial data and APIs presents an interesting juxtaposition for consumers: Having this data available to PFMs or accounting packages like Xero that help us make sense of our spending is extremely valuable, but often only private APIs (such as online banking) expose it. Gaining access to the data therefore becomes a question of using workarounds such as **screen-scraping**, which can contravene the terms of use for the services we access. Moreover, this data is also not available to businesses who might sell us a product or service we are truly interested in. This can negatively affect our perception as consumers of businesses who resort to spamming us based on either spurious correlation deduced from limited data or no analysis at all. Controlling how we expose financial data via APIs is therefore highly valuable to us, but again we are frequently thwarted by those who hold our data on our behalf.

Tactile

It's more difficult for consumers to comprehend the value of **tactile data**: It's what we 'touch', both in the physical and virtual world. Tactile data includes:

- The sensors we activate when travelling on public transport, using either a travel or contactless card or a mobile or smart device (which may indeed present a cross-section with our financial data)
- Our GPS movements as tracked by our smartphones;
- The items we browse at a clothes store that are labeled using smart tags;
- Our browsing history that shows what we view online;
- Qualified data we catalogue about ourselves, such as calorie information collected through apps like MyFitnessPal.

These events tie us to particular **locations** and **activities** and provide a clearer view of us as an individual. If we chose to share tactile data with the organizations we want to do business with, we could exploit unique offers specifically tailored to our daily activities, which could bring about fantastic consumer experiences.

At the time of writing tactile data is only sporadically available to us as consumers and indeed subsets of that data are tightly controlled in most countries, such as the **geographic** tracking of an individual. There are exceptions in certain subject areas of course, like the Garmin Connect API (although being able to access this data carries a considerable price tag!). **Browsing data** is also becoming the subject of intense scrutiny with more tools being created to protect it. Few, however, are allowing consumers to exploit it for their own purposes. As consumer awareness of tactile data increases we will value it more and thus our desire to exploit it will become greater. However, exploiting it and making it available via APIs, given the regulations and laws involved, may be extremely difficult for the foreseeable future.

Aggregate

Aggregating our confidential, financial, and tactile data helps to build a picture of our lives that is extremely valuable to us: It could be a way to exploit the **Intention Economy**, a concept described by Doc Searls that he later expanded upon on his book on the topic. The Intention Economy focuses on the fact that consumers come ready made, and that **advertising is unnecessary**: As buyers of goods know what they want, they can make rational decisions in approaching the market to make a purchase, allowing sellers to “bid” for them. This is incongruent with the vast majority of markets today, as they are seller-orientated and based largely on advertising.

The Intention Economy has yet to come to fruition in a universal sense, but pockets of behaviors that loosely follow the idea have emerged. Examples include price-comparison websites for insurance, energy supplies or credit cards, but these only bid in silos i.e. the companies offering deals don’t overtly attempt to outperform each other. The concept and implementation of Personal Data Stores is becoming a reality through software like Meeco and Mydex, enabling us to corral, organize, and allow specific access to our data.

We are on the cusp of being conscious of our real value as consumers. However, the architecture, protocols, and networks to support the ready aggregation and interchange of our personal data is in a nascent state, with some of the key tools like **vendor relationship management** (VRM) software (a means to manage our relationship with the vendors we wish to do business with) under relatively early development. The API-of-Me concept will need to mature for us to truly elicit the value of our personal data.

Final Thoughts

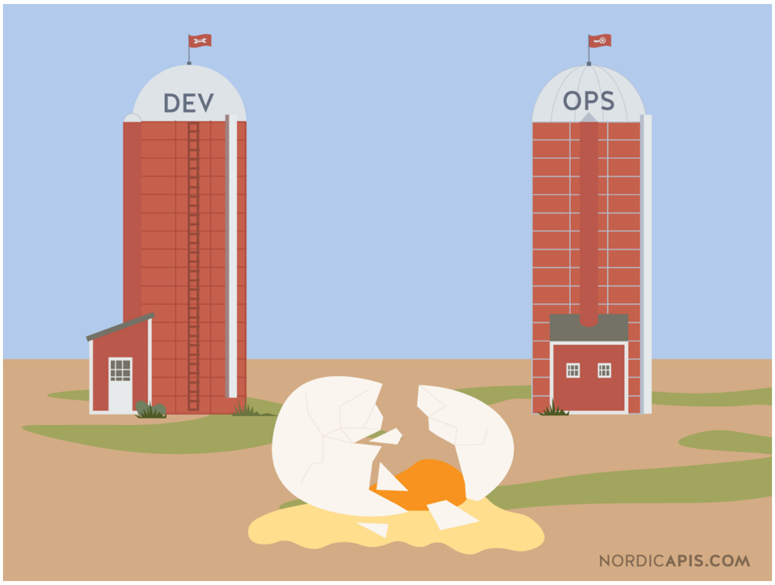
The ownership and stewardship of consumer data is a subject much wider than the API economy: It covers all areas of technology, and for the majority is a subject that only surfaces in conscious thought when a business or organization uses their data in a way that they should not.

However, as the population becomes more tech-savvy, consumers will begin to understand that their data (whether exposed by an API or another means) has intrinsic value to them and controlling it can deliver them significant benefits. Generally speaking, we are at a standoff between the desire to access this data for our own means, the willingness of the organizations that harbor our data to make it available, and the manner in which we can correlate it, understand it, and disseminate it for our own benefit.

The cross-section of this data forms a picture of us as individuals that is hugely interesting to potential consumers of this data. That value must be controlled and safeguarded for our own benefit.

We are on the verge of utilizing the data we create for completely new ventures — both the API of Me and the APIs that allow us to access our data will certainly provide a vehicle to help this opportunity become a reality.

Avoid Walking on Eggshells and Use DevOps



Humpty Dumpty sat on a wall, Humpty Dumpty had a great fall. *All the king's horses and all the king's men Couldn't put Humpty together again*.

That sure sounds like the last time your site went down and every team in the office started panicking, right?

When things go wrong, everybody starts pointing fingers. "The developers are saying it's a server issue, IT is saying it's a code issue. It's absolute chaos," says Emily Dowdle, of Wazee Digital.

The real truth? No-one is quite sure what's wrong. Otherwise, they'd already be doing what's needed to fix the problem and get the site back online. In many cases, the problem is **miscommunication**: person A knows something that person B doesn't, and that's preventing person B from coming up with a fix.

The idea behind DevOps is that facilitating interactions between teams, particularly development, operations and QA, can alleviate some of that chaos.

Speaker: Emily Dowdle

This chapter was inspired by Dowdle's session at the 2016 Platform Summit. Watch the full talk here

Silos Are Bad For Business

There's a problem with the modern workplace that not many people talk about. Namely, that everybody works in **silos**. And, where that's the case, no amount of team-building exercises or open office space will change things.

Let's talk about developers and operations: they don't always get along. Emily Dowdle, of Wazee Digital, describes the situation like this:

> "There's tension, could be described as a friction or an attitude … a general inability to tolerate each other without eyerolls and audible sighs."

She talks about how the Operations side of things is evaluated not by the features they release, as developers are, but by quantifiable measurables like site uptime. Ops teams strive for five nines (or better) uptime, i.e. a site being up for 99.999% of the time, which works out to 5 minutes a year of downtime.

> "Fundamentally, we have different priorities. It means we have conflict, we butt heads. It's no wonder we're natural enemies."

But it doesn't have to be this way.

Key DevOps Concepts

With the rise of agile development – which, of course, results in more frequent releases – embracing DevOps as a concept is not just advantageous, but a necessity.

But – like "big data", "agile development" and other mots du jour – DevOps is such a large, often poorly defined, concept that it's easy to get lost in.

Dowdle has a few ideas for practical implementation of DevOps in the workplace:

1. Bridge the skills gap

Acknowledging that development and operations teams don't work in the same way, and don't have the same priorities, is key to making changes.

No developer is ever going to fully understand the ins and outs of operations, and vice versa, but some comprehension of *why* the other team would want to take a certain approach is extremely valuable.

2. Share information

Everyone hates meetings, but they do have their uses.

If a new **feature** is in the works, it's important for development and operations to discuss the safest way to iterate without breaking… well, everything.

From the other side, it's helpful for developers to have access to **read-only logs** that let them see the impact of their deployments and any other changes they've made to a site. Adding ChatOps, colourfully named so by GitHub, into the mix isn't a bad idea either.

3. Consistency

Whether we're talking about the process of deployments themselves – which should be such an intuitive process that anyone on the team can do it – or how closely a staging area replicates its live equivalent, **consistency** is key.

Making small, incremental changes – no more bloated feature releases – is the best way to make sure everything stays recognizable while still moving forward with development timetables.

Continuous integration/delivery is, as we've written before, here to stay. A cornerstone of agile development, it's also incredibly useful for trying to keep things as consistent as possible.

4. Increased accountability

In most workplaces, teams are guilty of passing the buck sometimes.

Sales and marketing teams might close deals on projects that don't technically exist in the product yet (but the devs can take care of all that!), while developers might make a deployment that really needed more QA in order to meet a deadline.

Including devs in on-call rotation, i.e. involving them in that 4am "the site is down" conversation, is one example of a way to change the way developers think about deployments.

Again, this is as much a culture thing as it is a process thing: in her presentation, Dowdle talks about the fear of heads rolling when something goes wrong, and that fear makes people *want* to divert responsibility.

Making sure everyone is up to speed with best practices (e.g. security) in the workplace is key, but it's just as vital to make sure that employees feel able to do their job without worrying that one mistake will cost them it.

5. Embrace failure

Let's think about those mistakes a little more. So the site went down? That's bad. But it doesn't need to be the end of the world.

It's more important to figure out what went wrong, which automated alerts can seriously help with, and figuring out how to make sure it doesn't happen again. Assigning personal blame isn't helpful, but enabling a team to see where they went wrong definitely is.

Expecting and preparing for mistakes or accidents, and not setting unrealistic uptime targets of 99.99999%, can help to ease some of the tension between development and operations.

If (or maybe that should be when) you perform a post-mortem on what went wrong, figuring out why and whose fault it was — without a lot of pointing fingers – is the best way to make sure that it doesn't happen again.

DevOps and APIs

Some of the problems outlined above become even more significant when APIs are thrown into the mix: According to Dowdle, in most cases of API problems, there will be at least two development teams and two production teams in the mix.

In that respect, APIs actually offer a great jumping off point for an approach that takes DevOps more seriously; when it comes to creating a great API, creators are keen to avoid issues because they know that multiple teams will inevitably be involved.

That's especially true if we're talking about a public API, in which operations and testing are even more critical than one designed for internal use.

As a result, the following are key in successful API development:

- A very specific approach to requirements e.g. parameters, response codes etc.
- Thorough documentation to walk through a number of potential problems
- Need for rigid testing frameworks that functions exactly like what's live

What API developers don't necessarily realize is that everything they've applied to the development of their API is the perfect recipe for a balanced DevOps-style approach to building a core software offering too.

So, next time you're working with your team on a new feature, ask yourself the following question: "is this how we'd be doing it if we were making changes to our API?" If the answer is no, then you might find the concepts above useful to consider.

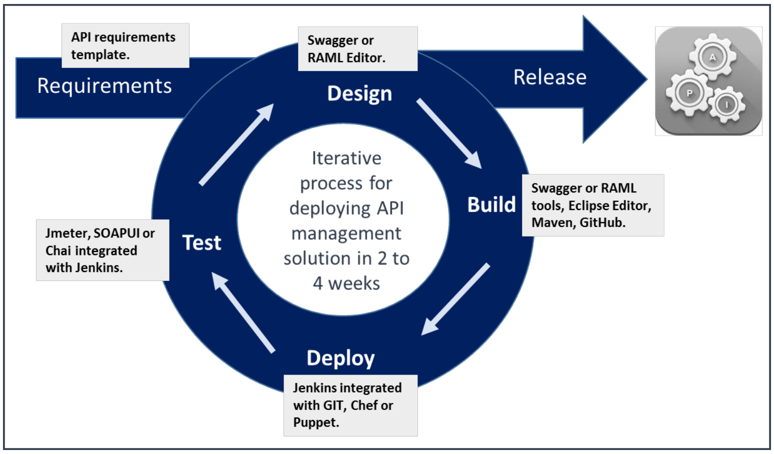
Dowdle doesn't come right out and say it, but her strong focus on documentation, interdisciplinary communication and parity between environments means that it's difficult not to reach this conclusion: the way we build APIs is an effect a model for the entire software ecosystem.

Guides to API Development Management

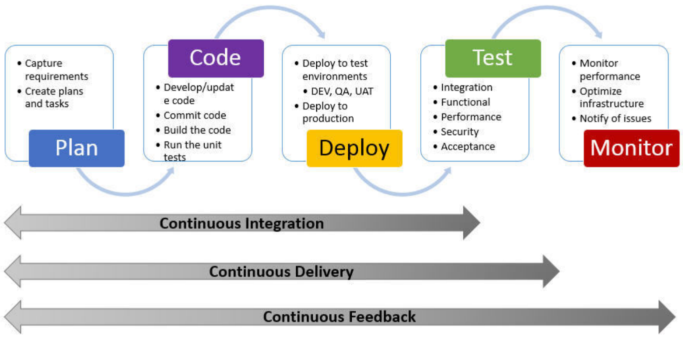
API developers aren't always given the same freedoms as traditional software developers. Or perhaps we should phrase that another way: API development is typically directed by feedback and requirements, or a product backlog.

In some – but not all! – workplaces, developers make additions based on what marketing/sales asks them to add, what features THEY think a product should have and sometimes (let's be honest here) what's trending that week on HackerNews or Reddit.

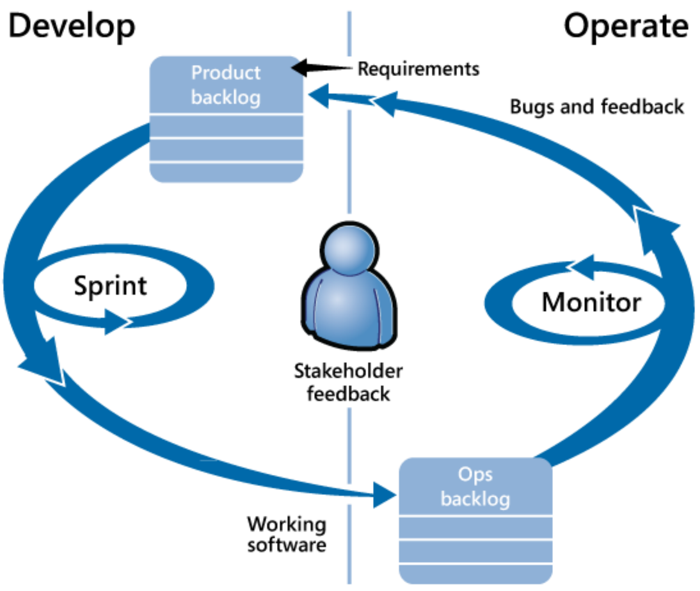
Let's consider the following three diagrams for slightly different development models:


API requirements template.
Requirements
Swagger or RAML Editor.
Design
Release
A
P
I
Iterative process for deploying API management solution in 2 to 4 weeks
Build
Swagger or RAML tools, Eclipse Editor, Maven, GitHub.
Jmeter, SOAPUI or Chai integrated with Jenkins.
Test
Deploy
Jenkins integrated with GIT, Chef or Puppet.

Accenture DevOps diagram



IBM continuous integration process



Microsoft development management cycle

The first of the above diagrams is a DevOps approach to building and testing API features outlined by Brajesh De, of Accenture, and the latter two show continuous integration/delivery/feedback DevOps approaches used elsewhere.

You'll notice that, while the order and the specifics may be a little different, the overarching ideas are basically identical – requirements are designed, deployed, tested and feedback is gathered, combining both the development and operation sides in synchronicity.

The fact that the Accenture API management framework is so similar to other DevOps templates only seems to further the idea that the approach adopted during API development – which typically involves rigorous testing and continuous feedback from Operations and/or those who use the API – effectively offers a valuable DevOps model.

Conclusion: Putting Humpty Back Together Again

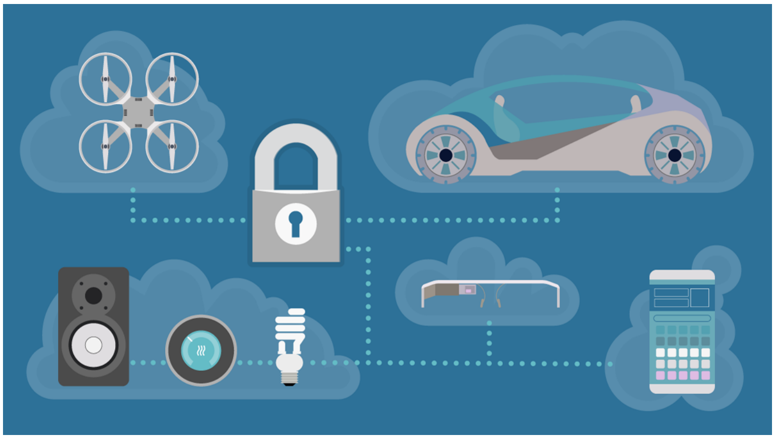
The bottom line is that much of this comes down to culture: a DevOps approach, or one that mimics the style of API development, won't be successful if your Development and Operations teams are completely unwilling to co-operate.

The DevOps mindset isn't a new one but it's one that, while hugely valuable, can still be difficult to adopt because of problems with coordination or company structure. Still, it's one that's increasingly important as agile development and continuous delivery/feedback becomes the norm.

Adopting an approach that works is even more critical when it comes to APIs because, when API development is done badly, it's not just your own business that's at risk; you're also compromising the businesses of everyone who uses your API.

And if there's one thing harder than putting Humpty back together again, it's putting a thousand Humpties back together again with an angry mob shouting at you the whole time.

Securing the IoT for Decades to Come



In 2007 Kevin Kelly gave a TED talk in which he forecasted how the World Wide Web would look 5000 days into the future, prophesizing the emergence of the IoT and AI. He envisioned a more connected planet where all manufactured goods tap into a single, global, intelligent network.

At the time, the Internet of Things (IoT) was only a nascent idea thrown around by futurists. Now, however, it's a hot potential market, accelerating large corporate moves and entrepreneurial initiatives throughout the world. Gartner predicts there will be over 20 billion IoT devices by 2020.

So, let's model Kelly's thought process and take the long view yet again. What will the *next* 5,000 days bring? What will the world look like in **2030**? Drone deliveries, self-driving cars, and more innovation is surely coming, but so are unprecedented vulnerabilities to the user. In 2030, the way IoT sensors and devices *share* data **must be secured**, or else we are creating our own doomsday.

According to Travis Spencer, founder of Twobo Technologies, the IoT can only exist if **identity** checks are properly implemented at the **API** level — as any IoT device must know *who* is requesting data, and *what* they are allowed to do with that data.

The creation of a global device-oriented network is only possible by using APIs for standard information design, as well as adopting internally recognized **open standards** (like OAuth, OpenID Connect, SCIM, etc.) as a foundation for the **identity management** that is so critical for protecting user safety and data.

We've discussed at length the need for standardized APIs within the IoT, but scaling IoT platforms from a security perspective is another beast entirely. In this chapter, we'll look decades into the future to see how IoT and API security must unite, so that developers can begin to scale their platform and security measures accordingly.

Speaker: Travis Spencer

This chapter was inspired by Spencer's session at the 2016 Platform Summit. Watch the full talk here

Looking into the Crystal Ball: The World in 2030

The world is changing rapidly, and IoT is gaining true momentum. By 2030, we'll see the ubiquity of driverless cars and drones. Highways are already being designed for driverless cars, drones have expanded beyond hobbyist usage into numerous applications, and sensors are driving the fog computing movement.

As the IoT leaves the realm of hobbyism and integrates into our daily lives, Spencer sees connected apparel, space tourism, 3D printing, smarter healthcare, and secure electronic voting as realistic implementations.

> "The IoT is not hype. It's everywhere — it's in the room."

There will also be major restructuring within our industry and workforce. Entire companies will vanish and new ones will emerge. Futurist Thomas Frey predicts that by 2030, 2 billion jobs will disappear from the planet.

But this transformation will also reinvigorate the workforce; It's been said that by 2030, 1 in 5 jobs will require **programming** as a basic skill, and the need for programmers is growing 12% faster than other areas.

Today's tech will appear absolutely archaic in comparison to future devices, yet the **software architecture** could remain constant if open standards are adopted. Let's see what IoT evolution will mean for scaling security architecture for decades to come…

Identity is the #1 Impediment to Safe IoT Connections

As we embed devices into our bodies and immediate surroundings, Spencer sees political, religious, and social systems being reinvented and questioned. But something that won't ever change is the human desire to form **relationships**. People will always want to know *who* or *what* it is that they are interacting with. > "Within social, big data, and all different aspects of computing impacting our lives, we need to know who we are talking to — who's on the other end of that API?"

Without identity control in place, trusting an IoT gadget poses a serious threat. Consider a skiier using smart goggles that describe upcoming conditions on the slope. If someone hacked this platform and edited the terrain, this could lead to a fatal crash. To prevent this, we need to intimately understand **who** is accessing IoT devices.

Whereas humans intimately understand relationships, all a computer understands are 0s and 1s. So how do we train IoT devices to recognize **identity**? Spencer sees the solution lies in incorporating an open standards-based identity management system within your API management makeup, using specialized systems to authorize and federate access across API connections.

Building on Open Standards will Secure Future Identity in the IoT

The **open standards** we're talking about are communication specifications defined and generally agreed upon by the international web development community. According to Spencer, deploying these open, internationally recognized standards will enable long-lasting IoT/API platforms to be constructed.

The **Neo-security Stack** achieves this, and is built on the following open standards:

- **Authentication**: U2F & Web Crypto
- **Provisioning**: SCIM
- **Identities**: JSON Identity Suite
- **Federation**: OpenID Connect
- **Delegated Access**: OAuth 2.0
- **Authorization**: ALFA

Within the Neo-security stack, these standards are combined within three subsystems: An **Identity Management System**, **API Management System**, and an **Entitlement Management System** — three fundamental systems that should be recognizable and supported within all IoT organizations. All are cleverly powered by open standards to ensure platform longevity.

For example, the goal of an **Identity Management System** is to answer the fundamental question of *who is calling*. Once it figures out the identity, it then forwards this information to other subsystems. An identity management system should be comprised of:

- A **user management service** to perform Create Read Update Delete (CRUD) operations on things like user information, groups, or connections between users. It uses SCIM, the standardized identity API by IETF.
- A **federation service** to allow you to reuse credentials in another trusted domain. It brokers cryptographically secure credentials using OpenID Connect.

- A **security token service** that issues a security credential token, implementing the OAuth standard.
- An **authentication service** is another important facet of the overall identity management system. It covers many bases — it can log you in, collect credentials, act as a self-service registration, and provide Single-Sign On (SSO) capabilities. It also integrates with different directories such as SQL databases, other integration providers, social networks, or exterior identity providers. An authentication service is often built to the specific requirements of the organization, and in doing so may need to deal with older, non-standard protocols.

Acknowledgement: Open Standards Fluctuate

Spencer recommends that developers build on new security standards that will stand the test of time, yet they should loosely couple them in a way that allows for revising or adding new standards to the stack. This is because standards, and the products they support, will inevitably evolve:

> "The world is changing and shifting… if we don't build in a loosely coupled way where the interfaces between these systems are governed by open standards then it's going to be hard to replace those products."

By 2030, your API platform may not be replaced in it's entirety, but it will be likely be reiterated and extended. Ensuring the platform is open from the start, based on standards you can implement yourselves or find replacements for, goes a long way to extending lifespan. It also ensures that security will evolve alongside this evolution.

The Nuances of *IoT*-Based Communication

Much of what we've discussed so far has been related to internet-based communication. In which, the API security management system is talking to APIs that are accessing the internet over HTTP or HTTP 2. However, IoT devices may be working with other protocols, such as CoAP, a specialized web access protocol designed for machine-machine dialogue.

We typically use JWT (JSON Web Token) as the standard token, as it creates lower technical barrier, is easier parse JSON over XML, and integrates well with lower level languages. With a little maneuvering, we can send an OAuth message inside of a CoAP stream.

Since HTTP 2 redesigned HTTP as a **binary** protocol for compactness, it would make sense to use the same message format but as a binary representation. To do so, you can use CBOR Web Tokens to create a compact profile of the JWT. This could be used within an IoT setup as a way to optimize throughput.

5 Actionable Steps Toward Improving IoT Security

We glimpsed into the near-distant future, and discovered a rapidly changing world greatly impacted by the advent of the IoT. In the Internet of Things, identity control will be paramount for protecting users, but to get there, we need a long-lasting API security architecture founded on open standards.

With all this information, where do we go from here?

Spencer recommends 5 solid next steps to begin implementing better API security:

1. **Gap analysis assessment**: Could your systems benefit from a more secure identity architecture? How important are the components for your business?
2. **Gauge the impact on the platform**: Assess how a new security service will impact your existing databases. How would this alter how clients must call the APIs?
3. **Pilot and deploy in baby steps**: Assemble a proof of concept and test it with your users, assess the strengths and weaknesses, and iterate.
4. **Go live with HTTP 2!**: We should all embrace HTTP 2 as it'll make the internet a speedy, happier place!
5. **Research IoT-specific protocols**: To make data transfers truly compat, consider CoAP as an IoT specific protocol, and CBOR for distributing your JWT tokens.

Our future digital world will be truly incomprehensible compared to the present, and perhaps, somewhere in the cloud exists the world's biggest internet company, yet to emerge. Now that the IoT has become a reality, we must future proof the API security platform at it's core. As identity will be pivotal for understanding relationships between devices, the best we can do now is to build identity management systems that embrace open standards that are poised to stick around, decoupling them so that as products evolve we can synergize them with new components.

5 Ways the OpenAPI Specification Spurs API Agility

The API lifecycle is a topic of much discussion — and rightfully so. The API space is agile, ever changing, and its participants must continually meet the shifting kaleidoscope of user needs and demands.

Speeding up and securing this lifecycle is seen by many, then, to be the Holy Grail. Being able to leverage proven API specifications is a powerful method, as proper specification usage opens up a whole new realm for developers. Whether this be in efficient, automated documentation, improved usability and developer experience with dynamic sandboxing, or decreasing speed to market through faster iteration, the benefits can't be overstated.

Today we'll look at a wonderful solution to help reach this lofty goal: the **OpenAPI Specification**. We'll talk a little bit about what it actually is, and how it can enable faster, more secure API development.

Speaker: Arnaud Lauret

This chapter was inspired by Lauret's session at the 2016 Platform Summit. Watch the full talk here

What is the OpenAPI Specification?

The OpenAPI Specification is a specification and framework implementation designed to create **machine-readable** interface files for describing, producing, consuming, and visualizing web services in a RESTful architecture. As Lauret describes, the use cases are numerous:

> "The OpenAPI specification universe extends way beyond generated documentation. It's boundless. The OpenAPI specification can be used to accelerate and secure API creation and evolution."

Before there was OpenAPI, there was **Swagger**. Swagger was originally designed for use on the Wordnik word discovery service, a web service that was touted to "find meaning in words." Swagger was then designed under this same guideline, though with a slight deviation — to "find meaning in APIs."

In 2015, API provider SmartBear acquired the open-source Swagger framework, and began to expand it out of the confines of Wordnik's use case. In 2015, SmartBear helped to create a new organization under the banner of the Linux Foundation called the Open API Initiative, working with Google, IBM, and Microsoft to craft new tools and solutions. As part of their efforts, they released the Swagger Specification to the group, dubbing it the OpenAPI Specification. It should be noted that that the tooling around the specification still retains the Swagger brand (Swagger Editor, UI, Codegen, etc). At the time of this writing, OpenAPI Specification is in a stable v2 with a version 3.0 in draft status.

Why OpenAPI Specification?

The OpenAPI specification has some really special features that make it so well-received in the API world. First and foremost, OpenAPI is **language-agnostic**. Unlike other solutions, this specification has no language preference and can work across any language with minor instruction.

Another huge benefit of the OpenAPI Specification is that, as a result of its specific dependence on declarative resource specification, clients have a much larger, more robust and **holistic view of functionality**. Instead of depending on restricted server code or documentation, the OpenAPI Specification, in many ways, *describes itself*.

Wrapped up into the OpenAPI Specification is the **Swagger-Codegen**, a piece of Swagger technology that utilizes a template-driven engine to generate documentation, server foundational stubs, and clients.

For many reasons, OpenAPI is often seen as the **best solution** to date for API development. With this in mind, how specifically does it help us toward our goal?

API Fastness

> “API fastness is probably what every API provider aims to do — being able to deliver an API quickly but securely.”

Lauret defines **API fastness** as accelerating the development of an API and its resources while maintaining security and quality control. Essentially, you’re aiming for the best of both worlds, and as part of that, you’re accepting some limitations.

In the classic development lifecycle, you’re playing a game of balance — you sacrifice quality for speed to market, and you give up some speed when you focus on securing resources. Though not every project will have the same pitfalls, slow and methodical generally counters quick and innovative. As Mark Zuckerberg once said, “move fast and break things.”

Luckily, the OpenAPI Specification takes away a lot of the “break things” part of that motto and focuses more on the “move fast”. Here are 5 ways it does that.

1: Proper Design and Approach

Any good engineer can tell you that structures don’t fail because of the last brick, they fail because of the first. If you start with a shaky foundation, or bad bedrock, you’re going to collapse, and the same is true of an API.

The OpenAPI Specification provides perhaps one of the strongest foundations upon which one can build an API. It does this not by solely providing tools or approaches, but by providing a proper **design approach** towards how your API will be interacted with.

> "A beginning is a very delicate time, especially for an API. Whether an API is internal or external, its design is crucial. A bad design can lead to a total disaster. It can kill projects and even companies."

Perhaps the most powerful example of how the OpenAPI Specification does this is in the synchronization of source code and client libraries. Failing to synchronize these elements can result in divergent code sets and awful interactions for the consumer.

On the other hand, adopting a proper methodology for ensuring that content is properly synchronized can make for better functionality. The greatest part of all of this is that the three main ways this specification generation is done — Codegen, automatic generation (swagger-node-express and swagger-play), and manually — are completely adaptable to specific API functionalities and design aesthetics.

2: Complete Documentation and Description

SwaggerUI, an element of OpenAPI, is supremely powerful and capable when it comes to generating **descriptive documentation**. Due to where the OpenAPI Specification came from — specifically, Swagger — the main selling point of "accurate, easy documentation" that came with Swagger is also shared within OpenAPI. As Lauret describes:

> "The OpenAPI specification is able to easily and efficiently describe an API. Every single aspect

> can be described easily in a simple but structured document."

That's not the end, either — the OpenAPI Specification is **open** by design, which has given rise to some serious contenders for functionality mirroring that of the SwaggerUI. For instance, APIs.guru have released their ReDoc solution as an alternative system.

That's the real power here. Any documentation service worth its salt will get the job done, but the OpenAPI Specification is specifically built around the idea of opening up an API to multiple solutions, languages, and approaches in an effort to reach the best solution possible for **any given permutation**.

3: Rapid Testing and Iteration

The OpenAPI Specification utilizes a declarative resource specification, and as part of this, allows for easy exploration of the API without access to the server code or an understanding of the server implementation.

This means that, unlike other solutions where the dev has a great sandbox, but the average user is restricted, the API can be sandboxed in a way that allows for both developers *and* non-developers to understand and explore in an interactive sandbox.

This is all done via that magic implementation called the SwaggerUI. SwaggerUI reads the API description that is stated as part of the OpenAPI Specification, and renders it as a web-page.

While this is great from a consumer viewpoint, the real benefit here is in testing and iteration. By being able to actively test and modify API content during development in a sandboxed environment, endpoints can be tested against any permutation, resource access can be tested in a multitude of environments, and the API can be tested in interactions with other APIs.

This drastically aids in iteration, and enables truly rapid, dynamic testing with minimal overtime.

4: Shorter (More Secure) Time to Market

Due to the aforementioned drastic reduction in resource and time dedication for testing and iteration, this also naturally results in a decrease regarding **time to market**. By taking less time to iterate and using less resources between initial revisions, this results in a much quicker cycle as features are rolled into the main codeset dynamically.

A key sidenote here is that this also drastically increases the security level of the API. Because the developer is able to test and iterate with relative ease and quickness, issues in the code base and its interactions are discovered earlier in testing, and bugs are attached directly to stated resources.

Imagine building an engine. When putting the pieces together, you have to test each element, and upon the addition of a new part, you must test the engine as a whole. Now imagine the same situation, but with a super computer that allows each individual part to be tested no matter how deep the part is, how surrounded by other parts it is, and how core or ancillary to the overall functionality it is.

This is functionally what the OpenAPI Specification allows you to do — test anything dynamically and with low overhead and time between testing.

5: Machine and Human Readability and Translation

> "The OpenAPI specification can also be used to create human-readable documentation."

The OpenAPI Specification is a Rosetta Stone for the API space. Whereas other solutions generate machine-centric data for the purpose of functionality and then derive from this human readable documentation and specification (or vice versa), the OpenAPI specification starts from a different position — a **progenitor language** from which all other content is derived.

By functioning as a **specification** which describes the API, the OpenAPI Specification can then derive both **machine** and **human readable** content in a better, more complete way. This approach led Swagger to become a household name in the API space, and it continues to influence documentation trends to this day.

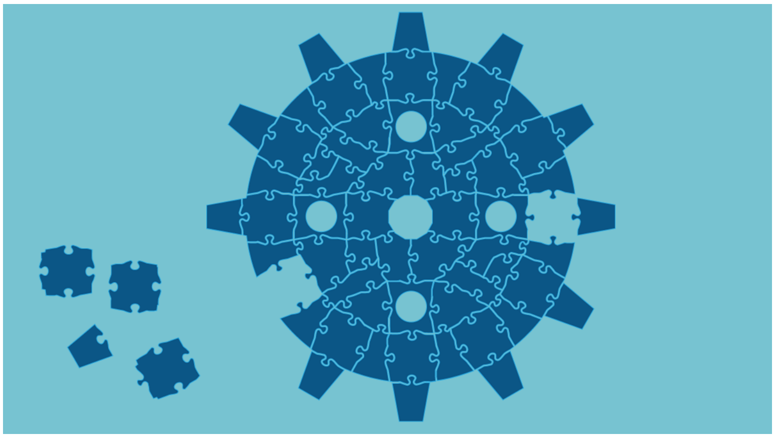
The OpenAPI Specification leverages its powers to create the most complete and useful content in both machine and human readable format; an important ability that simply cannot be overstated.

Conclusion: OpenAPI Enables an Agile API Lifecycle

As developers, API providers, and evangelists, the biggest issue we face in this industry is moving at a speed quick enough to iterate and innovate, while securing the resources our consumers depend on. It's no easy task — many have tried and failed (and quite spectacularly so).

The OpenAPI Specification is possibly the best implementation at this moment for this exact purpose. Due to how it generates documentation, APIs from a description, and does so in such a dynamic, diverse way, the OpenAPI Specification bridges both worlds, enabling, as The API Handyman calls it, *API fastness*.

Case Study: Bosch Ongoing Enterprise API Management Saga



Have you ever tried to implement a new process in a big company? Believe us when we tell you that it's not an easy (or fun) task. From obtaining initial approval all the way through to disseminating information, it takes time and there are often bumps along the road.

This is something that Josh Wang, project manager at Bosch Automotive Aftermarket, knows all about. Recently tasked with implementing an **API management solution** at Bosch, he talks about the incredible amount of data associated with Bosch's products, which range from mobility solution and industrial technology through to consumer goods and building technology.

The aim of the project – to implement an API management **platform** that would allow them to use their APIs more effectively and make more useful products for customers – was a slam dunk for Bosch. With billions of data points being generated via end user usage of their products, it made perfect sense for them to want to streamline and solidify their API management practices.

This chapter will cover some of the lessons learned from **enterprise API management** implementation — though that's not to say that there's nothing to garner if you're working at a much smaller company — many of these best practices apply whether you're managing a very large number of different APIs or creating and maintaining just a couple.

Speaker: Josh Wang

This chapter was inspired by Wang's session at the 2016 Platform Summit. Watch the full talk here

Is building everything yourself always the answer?

For developer consumers, there's a certain stigma associated with building your property on someone else's land – What happens if they shutter the API? How will downtime affect our app? The question posed above is an important one to ask.

When beginning a developer program, you must consider whether or not there's already an API out there that can do the type of thing that you're looking for. For example, there's no need for a website looking to give personal recommendations for movies to build their own API when they can just use IMDB's data instead.

That's a basic example, but the idea holds true even in large enterprises – it's much easier to convince your team to use an **existing product** than it is to get C-level execs, developers, marketing and operations on board with building one of your own.

There are certain parallels here between using a **third party API management platform** and trying to build your own API in that making use of what's already out there can save a lot of time. In the case of Bosch, however, one thing was for sure – whatever the solution was, it needed to be extremely robust. As Wang says, "we're building stuff that can potentially burn your house down, crush you or do harm to you if it's not done 100% correctly."

In this context, it's clear that relying on many different developers to maintain APIs that need to work seamlessly together wasn't the best idea for Bosch. They settled on using a third party API management solution and, while that might sound like a luxury for smaller organisations, it makes good sense when planning for **scalability**. We'll talk more below about cases in which that might be particularly important.

Have an effective strategy

Regardless of what you're doing with APIs, it's best to start with a list of requirements – what do you need to be able to do with your API(s)? How will existing infrastructure impact what you can do? Can your current setup handle the extra load?

In Wang's case, this list comprised more than 200 requirements. Large enterprises have to deal with multiple security zones, each with clearances, manual registration processes AND different users required to complete those processes. "This is far, far away from continuous delivery," Wang jokes.

Smaller companies are lucky in that they're generally better equipped to develop in a more agile fashion, but security, data protection, and uptime of existing services are still things that need to be taken into account when you're planning your next API move.

And, as much as we hate to bring it back to this, **organizational complexity** will always be a huge factor in API deployment and management. "Within Bosch, for example," says Wang, "there are so many divisions, business units and people with authority, that you need a platform that is flexible enough to facilitate the entire API lifecycle workflow."

Without that buy-in from others in the company, you're going to be spinning your wheels every step of the way during any API-related activity.

Be ready for that strategy to fall apart

"Life is what happens to you while you're busy making other plans", said Allen Saunders, and he might as well have been talking about software development when he did so.

Wang admits that he was naive in coming up with a timeframe for the execution of his project to implement a third party API management system from Axway that could handle the many APIs used by Bosch. "The team we were working with said it would take two days, maybe five. Ok, let's say ten days." Just to be on the safe side!

The reality was that implementing a management system, designed to simplify the maintenance of and interaction between existing APIs, took far longer than this: management nodes couldn't communicate properly with the current security setup, necessitating a design of exception, firewall issues, etc. The infrastructure for the new domain needed to be changed four or five times. Even tiny problems like Bosch's existing system not allowing the execution of scripts in the temp folder, which was required by Axway, seemed to snowball.

> "And those are just the small problems!" says Wang.

Whether we're talking about API management or implementing a new API, **flexibility** is key. On the former, Wang prompts enterprises to ask big questions like "how can you define your API management architecture in a way that supports on-premise hosting and cloud solutions?" and to "think about the different use cases with security."

But he's eager to point out that platform providers have plenty of work to do as well, saying that there are "challenges that providers of API management solutions and gateways need to address if they want to offer their solutions to large enterprises." It's important to note here that this doesn't mean that management providers like Axway are necessarily doing anything wrong, only that the relationship between them and large organizations are so (relatively speaking) new that it's common for unpredictable issues to arise.

For example, Wang talks about some of the legal requirements associated with API management, joking that this is something most developers don't like to think about:

> "In Germany, you need to record the version and date of when somebody has agreed with the terms and conditions in the privacy statement. If the system does not support a dedicated database that can store this kind of information, it's very difficult to convince the legal department that you can go live with the platform."

That's just one example, but it's a compelling one that highlights the difficulty creators of API management platforms face: they don't only need to measure up to the technical requirements proposed by developers, but must address a number of other issues as well.

Think big, even when you're small

Some of this may feel like overkill for startups and smaller businesses, but APIs are like Pringles – once you pop, you just can't stop. Salesforce, for example, has 10 APIs and Microsoft Cognitive Services have 23. Google has such a large suite of APIs that they have an API explorer to navigate them. And, with companies increasingly using freemium or paid API models, the trend of multiple APIs and microservices will only become more prevalent.

Wang highlights the way in which the size of a company doesn't always correlate with the complexity of an organization anyway: "Even small companies have very complex authorization structures in roles sometimes, depending on what kind of domain they're working in, and it's really important to have a platform that supports different roles, additive roles etc." Even smaller companies need to think about all of this, sooner rather than later, if they're thinking about building a suite of APIs.

There's a certain element of **future proofing** inherent in the idea of building or implementing an API management platform: the insinuation is that APIs will become, or at least have the potential to become, a key part of the business. That can be a smart move since lots of companies develop a single API then, before they know it, they're hosting multiple APIs that probably aren't synchronised as well as they could be.

To go one step further, API Evangelist suggests that "personal APIs" are not only on their way, but already here. As employees bring their own APIs to the table, potentially using them in their own individual workflow, an approach that's very open to API management platforms makes sense.

It might seem far-fetched for some companies to start thinking about API management, and the problems associated with it in large enterprises before they've even built their first API, but it could just end up being one of the smartest business moves they ever make.

In Summary

From REST to securing the Internet of Things, in this volume we've covered a lot of ground. To summarize, here are the key takeaways that API designers and enterprise architects can glean from each chapter:

TL/DR - 15 Important Takeaways

- **Understand true REST API design**: We responded to misconceptions of REST API design, reviewed hypermedia, and explored what it takes to create a HATEOAS-compliant state machine.
- **But consider GraphQL**: GraphQL performs select functions better than REST, but it means a significant reversal of modern REST API design standards.
- **Private APIs benefit from continuous versioning**: Eradicating the typical URI versioning schematic (v1, v2, etc) could withhold the server to client bond, equating to consistency and better API agility, however is largely unproven in public scenarios.
- **API-fy internal processes**: Spotify brilliantly uses Internal APIs to streamline their varying payment type subscription options. Consider how internal APIs can bring platform-wide consistency to improve *your* UX.
- **Have a serverless API backend**: Serverless architecture offers an infinitely scalable cloud backend for APIs and web applications, equating to a lean platform and cost reduction.

- **Put an end to polling**: Allowing clients to continually poll APIs can be a huge, wasted drain on your resources. Instead, use REST Hooks, or alternative means such as websockets or Server-sent.
- **Master microservice gardening**: API providers must eschew monolithic centralization in favor of innovation and new developments. This includes using Bimodal IT for parallel tracks, and a microservices architecture.
- **Model automotive IT for API longevity**: API providers could take a lesson or two from automotive grade manufacturing - automakers must build long-lasting, reliable IoT-centric APIs that stand the test of time.
- **Use OAuth 2.0 to secure the IoT**: The IoT is coming, and OAuth 2.0 is the way to secure it.
- **Avoid common API design anti-patterns**: Always consider the operational repercussions for the design moves we make now. Avoid improper HTTP method usage, protocol tunneling, polling, and rigid microservices structure.
- **Personal data is valuable**: With the rise of open banking, programmatic accessibility to the "API of Me" is becoming more realistic. Keep in mind the value of user data, and the government regulations mandating its liquidity.
- **Use DevOps**: When Humpty Dumpty falls and cracks, instead of pointing fingers, development, operations and QA should work together. This means having a DevOps mindset.
- **Secure the platform for decades**: IoT and API security must unite, so that developers can begin to scale their platform and security measures accordingly. This means building on open standards.

- **Use the OpenAPI Specification**: For growing and securing the API lifecycle, use a powerful API specification format. The OpenAPI Specification is a great solution to boost platform agility.
- **Enterprise API management techniques**: For lessons in enterprise grade API management, we studied Bosch's experience implementing Axway's API management solution across billions of data points.

What's Next?

The 2016 Platform Summit focused on **API longevity**, emphasizing the strategies architects must deploy to construct everlasting, holistic API programs. As we gear up for the 2017 Platform Summit, our focus will shift to **scalability**. If you are interested in joining the lineage of great Nordic API speakers, consider submitting a speaker session!

We hope you enjoyed *API Design on the Scale of Decades*, and let us know how we can improve. Our eBook releases usually come paired with an announcement for a new title, and this release is no different. Guess what… it's a GraphQL book!

As we've covered before, **GraphQL** is the query language making ripples throughout the economy. *GraphQL or Bust* will aim to once and for all determine the position of GraphQL within the API ecosystem. We'll explore things like the benefits of GraphQL, the differences between it and REST, nuanced security concerns, extending GraphQL with additional tooling and GraphQL-specific consoles, making a transition to GraphQL from an existing web API, and much more. Follow our progress on Leanpub!

Stay Connected

Thank you again to our readers, event attendees, and event sponsors and partners. If you appreciate what we're doing, consider following @NordicAPIs and signing up to our newsletter for curated blog updates and future event announcements.

Nordic APIs Resources

2016 Nordic APIs Summit Talks

You can watch full videos of the talks featured in this eBook below. In order of appearance:

1. The REST and Then Some (Asbjørn Ulsberg, PayEx)
2. Is GraphQL the end of RESTstyle APIs? (Joakim Lundborg, Wrapp)
3. Versioning Strategy for a Complex Internal API (Konstantin Yakushev, Badoo)
4. How Spotify Payments Creates APIs to Manage Complexity (Horia Jurcut, Spotify)
5. Building Serverless API Backends (Rich Jones, Gun.io)
6. Stop the Polling Madness and Adopt REST Hooks (Audrey Neveu, Streamdata.io)
7. Bimodal IT for Microservice Gardening (Erik Wilde, CA Technologies)
8. Look to Automotive IoT for a Lesson in API Longevity (Henrik Segesten, Volvo)
9. OAuth 2.0 and the Internet of Things (IoT) (Jacob Ideskog, Twobo Technologies)
10. API Design Anti-Patterns for Microservices (Jason Harmon, Typeform)
11. Will Open Banking Trigger the API of Me? (Chris Wood)

12. Avoid Walking on Eggshells and Use DevOps (Emily Dowdle, Wazee Digital)
13. Platform Security that will Last for Decades (Travis Spencer, Twobo Technologies)
14. Secure and Accelerate the API Lifecycle with Open API Specification (Arnaud Lauret, AXA Banque)
15. Assembling The API Management Jigsaw In Big (And Small) Companies (Josh Wang, Bosch)

More eBooks by Nordic APIs:

Endnotes

Nordic APIs AB Box 133 447 24 Vargarda, Sweden

Facebook | Twitter | Linkedin | Google+ | YouTube

Blog | Home | Newsletter | Contact