

Developer Portals

WINTER 2018

A selection of articles
about developer portals
and API documentation



Table of Contents

Introduction

Welcome

API Docs theory

A Brief History of API Docs	4
Good API Documentation Is Not About Choosing the Right Tool	5
API Documentation	14
API Design Guidelines	23
A Guide to RESTful API Design: 35+Must Reads	25
What is the Difference between API Documentation and a Developer Portal?	28
CIDM Ideas 2018	30
	43

Developer Portal Analyses

The Best Developer Experience KPIs	45
What is the Role of Blogs in the Developer Journey?	46
The Function of API Use Cases and Case Studies on Developer Portals	50
Find out How Typeform is Building the Ultimate Developer Experience	69
What is the MVP for a Developer Portal?	88
API the Docs	97
	111

Toolchains and docs as code

Case Study: Switching Tools to Docs-as-Code	124
Building a Developer Portal? Here are Four Key Questions to Answer First	125
8 Common Customizations for Drupal-based Developer Portals	143
Tool the Docs	147
	151

Authors

162

INTRODUCTION





Kristof Van Tomme
CEO and co-founder of
Pronovix

Welcome to the Developer Portals Winter 2018 articles collection

Last year, we experimented with publishing an e-magazine about API documentation and developer portals. In [Developer Portals Summer 2017](#) we collected our recent articles and featured highlights from the past semester's publications on API documentation.

This was a great exercise. Our team greatly benefits from anthologies and compilations made by others, and we now give you ours, hoping it will be useful to you too. We hope our digest gives you new insights about developer portals and API documentation. We thank all writers and presenters contributing to this magazine!

This winter's Pronovix e-magazine is a selection of our developer portals newsletter content from September 2017 until February 2018 as well as a curation of recent publications we think had great impact on the developer portal scene.

Topics featured in this digest:

- Theory behind creating great API documentation
- Analyses of developer portals to find out about best practices
- Recommendations on toolchains and docs as code
- Event recaps in the same topics: CIDM Ideas, API the Docs Amsterdam, Tool the Docs Fosdem

Subscribe to our newsletter

If you're interested in developer portals and API documentation, make sure you [subscribe to our newsletter](#) to receive a copy of our Developer Portal Components white paper and our future research publications. We also regularly share video recordings of conference talks and workshops. Be the first to hear whenever we have a new blogpost about API documentation, Developer Portals best practices, Developer Evangelism, or about technology that will help you optimize your API's developer experience.

We hope you'll enjoy this e-magazine. If so, stay tuned for the next edition that will come to you this summer!

Kristof and the Pronovix team

API DOCS THEORY



A Brief History of API Docs

<http://docsbydesign.com/2017/09/20/a-brief-history-of-api-docs/>

Bob Watson



I published my first API around 1988 for a peripheral to the IBM PC in which the API consisted of software interrupts to MS-DOS. (A software interrupt is similar in function to a procedure call, but used for operating system and device driver functions. I didn't write the documentation (at least not the published version), but a couple of co-workers and I wrote the interface.

Later, I want to say in 1997-ish, I wrote the API provided by the Performance Data Helper (PDH) dynamic link library (DLL) that shipped in Microsoft Windows NT 4.0 (IIRC). I didn't write the documentation for this API either as I was still developing software at the time. A super technical writer did the heavy lifting of making sense out of the functions the PDH.DLL provided.

While this isn't a particularly impressive resume of API development, it shows that APIs and I go way back. They are a very useful tool for solving many problems. Yet, over the past 30+ years that I've been using, developing, and documenting APIs, there hasn't really been much written about their documentation—until recently.

In my informal count (i.e. what I could find on Google while having my morning coffee a few days ago). I came across 17 articles and blog posts on the topic in just the past seven years (full list at the end of the post).

For the curious, here is some of the research that supports what I've read in some of these articles.

It's not really a surprise that APIs are getting more attention lately—their popularity and notoriety have soared in recent years; however, as with so much in [technical writing lore](#), some of what's been written is based on research, some on personal and anecdotal experience, and much on hearsay (when someone quotes or references someone else's personal, anecdotal experience).

In [Academic paper references](#), I list the API documentation-related references that I've cited in my publications over the years. The good news is that there is now quite a collection of academic research out there, and the better news is there have been many

new studies on the topic in recent years (which reminds me, I need to update that list). This is in stark contrast to 2008, when I was getting started in researching API documentation seriously. At that time the list of API documentation-related studies would literally fit in one hand (as in one per finger).

Here are some of the studies I've reviewed on the subject, organized by theme in a sort of annotated bibliography.

API reference topic content

The study in a recent article, [API documentation – What software engineers can teach us](#), is very similar in methods and findings to a study conducted at Microsoft by an academic and a Microsoft researcher in 2008 and published in 2009 with a longer version published in 2011.

- Robillard, M. P. (2009). What makes apis hard to learn? answers from developers. *Software, IEEE*, 26(6), 27–34.
- Robillard, M. P., & DeLine, R. (2011). A field study of API learning obstacles. *Empirical Software Engineering*, 16(6), 703–732.
- A related study, though with a slightly different focus was published a few years later, as well.
- Maalej, W., & Robillard, M. P. (2013). Patterns of Knowledge in API Reference Documentation. *IEEE Transactions on Software Engineering*, 39(9),

The reason I remember those was because they took a critical (and unflattering) position towards API documentation (which, at the time, I was writing a lot of, so I was probably a little sensitive to the topic). Honestly, the studies were fair and reasonable, however another related article from around that time took a more provocative tone:

- Parnin, C. (2013, March 4). API Documentation – Why it sucks [blog]. Retrieved November 2, 2014, from <http://blog.ninlabs.com/2013/03/api-documentation/>

As a practitioner at the time, it was frustrating to hear, in scientific detail, how API documentation (like what I was writing) presented all manner of learning obstacles to developers, while seeing, first-hand, that many of the problems cited in those articles

could be solved easily and quickly just by adding a few (hundred) more technical writers to the job. My thought at the time was, “So, what’s the big deal? You just discovered that technical writing is understaffed.”

What I felt was missing from those articles, as a practitioner, was the context to know, working with the assumption that the writing will be severely under-resourced, how much of what to write in any particular circumstance. This inspired my [Audience-Market-Product](#) thread and later webinar.

API usability

Ten years ago, as a technical writer tasked with documenting hundreds (ultimately, thousands) of interfaces and methods, I noticed how the same amount of documentation seemed overkill in some cases and woefully insufficient in others. It was then I learned about the notion of API usability. I used my yarn example to explain it:

To warm your hands, a ball of yarn and a pair of knitting needles need much more documentation than a pair of gloves made from the same yarn.

Some API methods and interfaces were, essentially, what a ball of yarn would look like if it were implemented in C#, while others were as easy to use as a pair of gloves. Likewise, the audience’s skill and knowledge is critical to how much you document and about what. You wouldn’t, for example, describe the step-by-step process of knitting to an expert, you would cover it at a higher level of abstraction by referring to specific stitch styles.

It turns out, I wasn’t the first to realize this—some of those who got there before me, include:

- Bloch, J. (2006). How to design a good API and why it matters. In Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications (pp. 506–507). ACM.
- Clarke, S. (2005). Describing and measuring API usability with the cognitive dimensions. Presented at the Cognitive Dimensions of Notations 10th Anniversary Workshop, Citeseer.

- Cwalina, K., & Abrams, B. (2008). Framework design guidelines: conventions, idioms, and patterns for reusable .net libraries. Addison-Wesley Professional.
- Henning, M. (2007). API design matters. Queue, 5(4), 24–36.

API usability as a topic in the press comes and goes (mostly goes), but in 2005 it was all the rage (-ish). Nowadays, I hear, it's just "good practice." Good practice, agreed, but not practiced consistently. In any case,

Any API documentation advice or guidelines that don't take the API's usability into account are going to be as comfortable to use as shoes you buy without taking the size of your foot into account.

The next time you tackle documenting an API, consider if it's a ball of yarn or a pair of gloves.

Software developer personas

The developer personas, such as those mentioned in [API documentation – What software engineers can teach us](#), were identified by a Microsoft researcher in 2004.

- Clarke, S. (2004, May 1). Measuring API Usability. Retrieved December 13, 2016, from <http://www.drdoobbs.com/windows/measuring-api-usability/184405654>
- Clarke, S. (2007, February 7). What is an End User Software Engineer? [InProceedings]. Retrieved October 26, 2014, from <http://drops.dagstuhl.de/opus/volltexte/2007/1080/>

Clarke wrote many other related blog posts and articles during the 2004-5 time frame. At the time, Microsoft was doing a lot of research into API usability while the .NET Framework was growing thousands of new interfaces and methods each month.

I talked with Steven after reading his articles and he told me that those personas didn't really represent people, per se, as much as how developers might approach a problem. The same person could, for example, be a detail-oriented, systematic developer on one problem (say, for example, while researching an API to use in a system design) and, at the same time, be a quick-and-dirty, opportunistic developer while writing some code to test it. (Personas described in: Clarke, S. (2007, February 7). What is an End User Software

Engineer? [InProceedings]. Retrieved October 26, 2014, from <http://drops.dagstuhl.de/opus/volltexte/2007/1080/>

Developer use cases

There was a flurry of research around 2008-9 into how software developers used documentation. My favorite article about this to cite is:

- Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M., & Klemmer, S. R. (2009). Two studies of opportunistic programming: interleaving web foraging, learning, and writing code (pp. 1589–1598). Presented at the Proceedings of the SIGCHI.

Specifically,

“Several participants reported using the Web as an alternative to memorizing routinely-used snippets of code.”

If you can't do your own user research and can only read one article, I would recommend this one as the one to read—it's not very long and has some interesting visualizations of their data. At the same time, I would also recommend that you always read more than one study about any subject. So, here are some more on this subject:

- Brandt, J., Dontcheva, M., Weskamp, M., & Klemmer, S. R. (2010). Example-centric programming: integrating web search into the development environment (pp. 513–522). Presented at the Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ACM.
- Brandt, J., Guo, P. J., Lewenstein, J., & Klemmer, S. R. (2008). Opportunistic programming: How rapid ideation and prototyping occur in practice (pp. 1–5). Presented at the Proceedings of the 4th international workshop on End-user
- Ko, A. J., Myers, B. A., Coblenz, M. J., & Aung, H. H. (2006). An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *Software Engineering, IEEE Transactions on*, 32(12), 971–987.
- Stylos, J., & Myers, B. A. (2005). How Programmers Use Internet Resources to Aid Programming. Submitted for Publication.

- Stylos, J., & Myers, B. A. (2006). Mica: A web-search tool for finding API components and examples. In Proceedings of the Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on (pp. 195–202). IEEE.

Why?

After my preliminary research into [the TC Myths project](#) and reading some recent articles on API documentation, I'm getting the sense that we (technical writers) have a habit of reinventing and rediscovering the same thing over and over as though we were stuck in some sort of geeky adaptation of Groundhog Day—[an observation made 13 years ago](#) (and probably earlier, as well), which refers to the fact that academic journals (such as the one that contains this observation) frequently live behind some rather expensive pay walls and many are written in “academic.” Which is unfortunate.

Those who cannot remember the past are condemned to repeat it.

–George Santyana

However, if technical writers—people who practice a profession that exists to record things for others—cannot record their past in an accessible way, which is most unfortunate. In academia, our writing has to build on past research and theory to both situate a given article and help keep track of the foundations on which a new argument is made. I don't see that as much, in articles written by professionals. Professional articles seem to focus on the now and the new—without realizing (or without crediting) that what is described is something from a forgotten past (less than 10 years ago, in some cases). Tom Johnson explores the academic-practitioner gap in more detail in his blog post [Why is there a divide between academics and practitioners in tech comm?](#)

At the same time, I get it. Sifting through the bucket of academic journals I've listed here and the many more I had to read to arrive at this list is (i.e. it was for me) much more work than reading a 1,000-word blog post. For a practitioner, the 1,000-word blog post is probably sufficient—until it isn't, of course. Unfortunately, it is often hard to tell where the limits are in a 1,000-word blog post. But, the fact that it's free to access and a tiny fraction of the academic-journal word count give the blog post a tremendous advantage.

Yet, it still seems like there must be a better way.

Recent API Documentation Articles (non academic)

As promised, here are the recent articles on API documentation that I found while having coffee, sorted by date.

Title & Link	Author	Date
What do you consider good API documentation?	Various	4/8/2010
Web API Documentation Best Practices	Peter Gruenbaum	11/12/2010
Designing Great API Docs	James Yu	1/11/2012
REST API Documentation Best Practices	Irene Ros	8/22/2012
How to Write “Good” API Documentation	Natalie Kerby	4/2/2015
API Tips — How to Write API Documentation	Ajitesh Kumar	5/17/2015
Best Practices in API Documentation	Keshav Vasudevan	5/17/2015
The Best API Documentation	Brad Fults	11/13/2015
Best practices and UX tips for API documentation	Katalin Nagygyörgy	11/13/2015
Tips and Considerations for Documenting REST API Documentation	Dana Fujikaw	5/3/2016
Best Practices for Writing API Docs and Keeping Them Up To Date	(Not specified)	9/19/2016

Title & Link	Author	Date
<u>Five Questions Every Technical Writer Faces with API Documentation</u>	Ed Marshall	10/6/2016
<u>API Best Practices: Documentation</u>	Kin Lane	11/1/2016
<u>Best Practices in API Documentation</u>	Keshav Vasudevan	6/20/2017
<u>The Importance of API Documentation</u>	(Not specified)	9/7/2017
<u>The Ten Essentials for Good API Documentation</u>	Diana Lakatos	9/19/2017
<u>Documenting APIs: A guide for technical writers</u>	Tom Johnson	No date

Good API Documentation is Not about Choosing the Right Tool

<https://blog.algolia.com/api-documentation-choosing-right-tool/>

Maxime Locqueville



As a member of Algolia’s documentation team, I am often asked: “What tool are you using to build your documentation?” Of course, I answer the question, but I am often tempted to say that it’s probably the least valuable piece of information I can provide.

In this post, I am going to give you some of that information: what things you should care about when building your docs, and how those things will make the choice of tool the least important thing.

Documentation is usually composed of two big parts: the content and the software rendering it. You might have guessed where I am going with this: a quality README.md stored on GitHub can be far more efficient than over-engineered documentation that is well displayed but has issues with content.

The perfect tool is not out there

There are plenty of ways to build API documentation: static website generators ([Jekyll](#), [Hugo](#), [Middleman](#)), web frameworks ([Ruby on Rails](#), [Laravel](#), [Django](#)), dedicated doc tools ([Docusaurus](#), [Read the Docs](#), [Sphinx](#)), SaaS products ([HelpDocs](#), [Corilla](#)), and that’s just the tip of the iceberg — there are so many more.

Depending on the one you choose you’ll have more or less flexibility, and more or less work to build and maintain. All tools will let you decide to a certain extent what you can do, and constrain you on the other end. I don’t believe there is a tool that can fit 100% of the needs in the long term. Documentation is something that needs to evolve, and you may have to change your tools several times as you outgrow certain constraints and have new needs.

Two years ago, we moved away from an internal tool that was aggregating content from GitHub ReadMes, a database and an external software managing our FAQ. This change took us full two months, and this is not counting the months of preparation prior to making the change.

By far the most time consuming task was to undo the formatting that our original tools required us to make. We had no consistency — some were Markdown, some were Markdown with custom extra features, some were plain HTML — and so while moving

away from our previous tools, we had to edit thousands of files manually in order to unify everything.

Put the focus on making the tool fit your content (not the other way around)

Instead of focusing on which tool to use, a better option is to focus on whether you are doing everything possible to be as little software-dependent as possible. If you can respond to: “Can I switch easily to a new tool?” with a “Yes!”, then you are on the right track.

Build all components to be software-independent

While developing custom components, it’s good to keep in mind that they should be as little dependent on the software as possible.

Now, to answer that question from the beginning of the article...It’s been two years since we’ve been using [Middleman](#) for our [main documentation](#). It’s doing the job, but it has some downsides and we’ve had to customize it quite a bit to our needs.

Here are some of the things that we added/modified:

- Custom [sitemap](#)
- Custom [data files](#) system
- Custom snippet generation
- Injections of custom metrics from Google Analytics for ordering purposes (for example the FAQ entries listed in <https://www.algolia.com/doc/faq/> are the most viewed ones)
- Ability to have a snippet file that can be auto-injected into any content file
- An Algolia indexer

These customizations are done in a way that we can reuse them in any project. The modifications represent ~800 lines of custom code, which is rather small for documentation like ours, but it enables us to be able to move data files to any other software in a matter of days rather than months; this is us adapting the tool to our content.

Keep the content properly structured

What is even more important is how you organize your content so that you can re-organize it programmatically when needed, or transform it so it fits another tool.

The more structured the information is, the easier it is to:

- reuse it across different parts of the documentation
- change the organization system itself when needed

Two tips on that front:

1. Keep the content centralized

As mentioned earlier, our documentation comes from a system that was split in many different parts. Today, we have documentation in a single repo that you can run independently from the main website. This removes dependencies and allows us to focus on content and doc-specific modifications. It also give us the ability to iterate more quickly both on content and code parts.

2. Choose the right file format

Also very important is where you write your content. When using a static website generator, it is “the norm” to put all content inside the Markdown files. This can work for small docs, but when your documentation starts growing above hundreds of pages, with different types of content (guides, tutorials, reference, FAQ), and you are seeking consistency, using structured data files is a better approach.

That’s why at Algolia we documented all methods and parameters in yml files and not Markdown files. While we ended up with Markdown inside yml, which can seem a bit counter-intuitive, it is quite useful. It also allows you to reuse the content in different ways across the website.

The two pages above are generated from the same yml file. When editing, this makes it very easy to keep consistency between different parts of the website.

■ Add objects



Description	Add new objects (records) to an index.
PHP name	<code>addObjects()</code>
Required ACL	<code>addObject</code>

More details

There are two ways to add a record to an index:

1. Supplying an `objectID`.
 - If the `objectID` does not exist in the index, the record will be created
 - If the `objectID` already exists, the record will be replaced
2. Not supplying an `objectID`.
 - Algolia will automatically assign an `objectID` and you will be able to access it in the response.

In both cases, once a record is added, it will have a unique identifier called `objectID`. This ID can be used later by methods like [Update Objects](#) or [Partial Updates](#).

Using your own unique IDs when creating records is a good way to make future updates easier without having to keep track of Algolia's generated IDs. The value you provide for `objectIDs` can be an integer or a string.

Description

Add an object to the index, automatically assigning it an object ID.

Example

```
sh

curl -X POST \
  -H "X-Algolia-API-Key: ${API_KEY}" \
  -H "X-Algolia-Application-Id: ${APPLICATION_ID}" \
  --data-binary '{
    "name": "Betty Jane Mccamey",
    "company": "Vita Foods Inc.",
    "email": "betty@mccamey.com" }' \
  "https://${APPLICATION_ID}.algolia.net/1/indexes/contacts"
```

Upon success, the response is 201 Created, with the `objectID` attribute containing the ID assigned to the added

```

name: Add objects
rest_name: Add an object without ID

method_name:
  javascript: addObjects
  php: addObjects
  python: add_objects
  ruby: add_objects
  csharp: AddObjects
  java1: addObjects
  java: addObjects
  android: addObjectsAsync
  go: AddObjects
  swift: addObjects
  scala: index into

acl: '`addObject`'

short_description: |
  Add new objects (records) to an index.

description: |
  There are two ways to add a record to an index:

  1. Supplying an `objectID`.
     - If the `objectID` does not exist in the index, the record will be created
     - If the `objectID` already exists, the record will be replaced
  2. Not supplying an `objectID`.
     - Algolia will automatically assign an `objectID` and you will be able to
       access it in the response.

  In both cases, once a record is added, it will have a unique identifier called `objectID`.
  `This ID can be used later by methods like [Update Objects](/doc/api-reference/api-methods/update-objects/)
  or [Partial Updates](/doc/api-reference/api-methods/partial-update-objects/).

  Using your own unique IDs when creating records is a good way to make future updates easier without having to keep track of Algolia's generated IDs.
  The value you provide for objectIDs can be an integer or a string.

examples: |

  Example with automatic `objectID` assignments:

  <%= snippet(batch_new_objects_auto) %>

  Example with manual `objectID` assignments:

  <%= snippet(batch_new_objects_manual) %>

  To add a single object, use the following method:

  <%= snippet(add_new_object_manual) %>

rest_description: |
  Add an object to the index, automatically assigning it an object ID.

rest_path: /1/indexes/{indexName}`
rest_http_verb: POST

rest_errors:
  - code: 400
    reason: "Invalid `indexName` or JSON"

```

So by focusing in this way on content first – its needs, structure, maintainability – and then finding and customizing the tool, you can come up with a documentation that is easy and quick to evolve.

Once this is in place, a good next step is to have your team and customers contributing content. Which leads me to ...

Bonus tip: get more contributions from the team and your customers

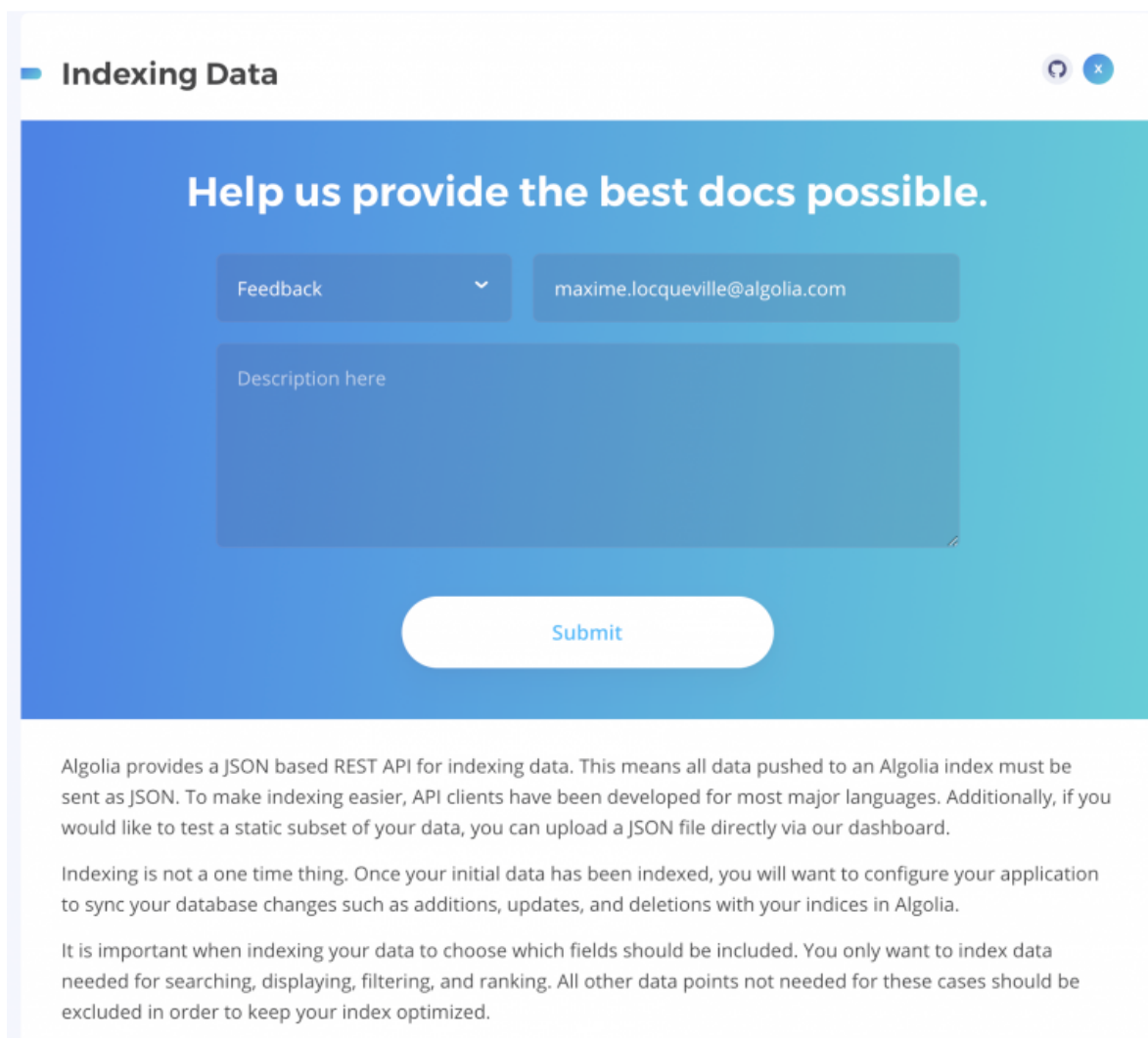
....and make those contributions as frictionless as possible.

There are a few actions we took to achieve this. When logged in as an admin, next to every section or code snippet we have an edit button that links to the correct file in GitHub, enabling admins to modify the content he or she is currently viewing.

Let's take an example of a new developer joining the team. One of the first things they are going to do is learn about the product by working with it. The easiest route for that is using the documentation. While they are using it, they will notice typos, unclear bits, undocumented features...if they have to think about where to provide feedback, or how to edit the file, there is a high chance that they will do nothing and switch to another task.

And if that's true for your team, it's even more true for customers. It is very unlikely that a user who noticed a typo will look around the website for the support email to tell you that they found something wrong.

This is one of the reasons we have the following big form at the bottom of every documentation page and also accessible from every section of the content:



The screenshot shows a feedback form titled "Indexing Data" with a blue header and a white "Submit" button. The form contains a dropdown menu for "Feedback", a text input field for an email address (maxime.locqueville@algolia.com), and a large text area for a description. Below the form, there is a paragraph of text explaining the Algolia REST API and indexing process.

Indexing Data

Help us provide the best docs possible.

Feedback

Description here

Submit

Algolia provides a JSON based REST API for indexing data. This means all data pushed to an Algolia index must be sent as JSON. To make indexing easier, API clients have been developed for most major languages. Additionally, if you would like to test a static subset of your data, you can upload a JSON file directly via our dashboard.

Indexing is not a one time thing. Once your initial data has been indexed, you will want to configure your application to sync your database changes such as additions, updates, and deletions with your indices in Algolia.

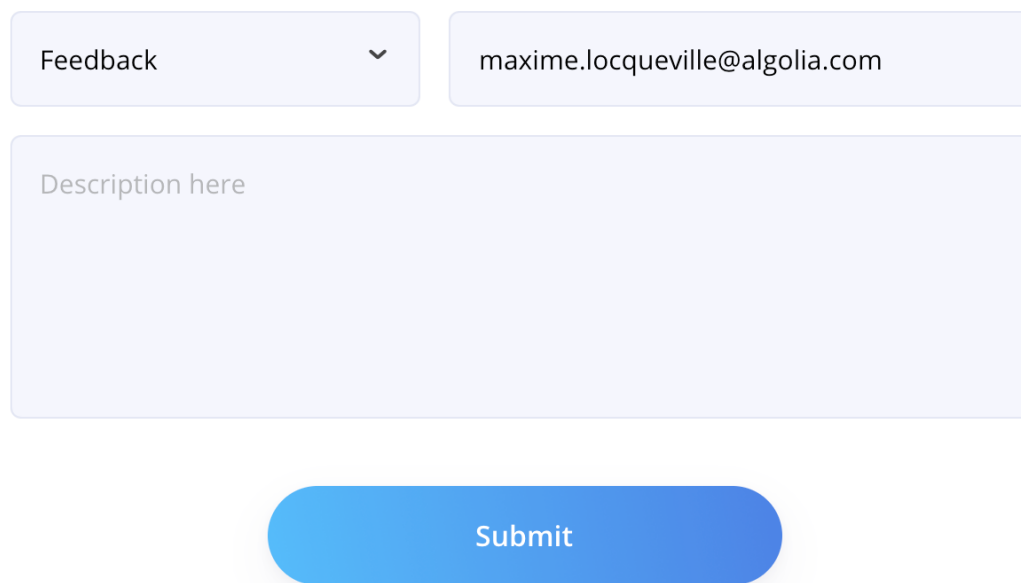
It is important when indexing your data to choose which fields should be included. You only want to index data needed for searching, displaying, filtering, and ranking. All other data points not needed for these cases should be excluded in order to keep your index optimized.

Did you find this page helpful?

We're always looking for advice to help improve our documentation!

Please let us know what's working (or what's not!).

We're constantly iterating thanks to the feedback we receive.

A feedback form interface with a light blue background. At the top, there is a dropdown menu labeled 'Feedback' with a downward arrow, and a text input field containing the email address 'maxime.locqueville@algolia.com'. Below these is a large text area with the placeholder text 'Description here'. At the bottom center is a blue rounded rectangular button labeled 'Submit'.

The customer cannot miss it and that's the point: if they see it once they know it's there and the friction to contribute is low.

This also has the benefit of giving a great developer experience to our customers. When someone reports an issue on the doc, it goes straight to our regular support channel, where we have a good response time. We also fix and deploy a majority of the issues within the same day. A company that takes immediate actions on feedback gives an impression of care (one of our core values), and that's exactly what we are aiming for.

When someone in the team creates or updates a PR, the change will be rebuilt and deployed to a new domain. To achieve that, we use the [continuous deployment feature of Netlify](#), which brings several benefits. The main one is ability to preview, not only for the person doing the PR, but also for the person reviewing it, because they don't have to deal with running the doc locally for small changes.

This is just one example of how to reach out to your readers. It creates a virtuous cycle where everyone (team + customers) contributes more and more, and enjoys doing it.

In short...

There are so many things to consider before worrying about which tool to use. Naturally, you do have to start somewhere and choose a tool, so I advise you to choose the one you and your team are most comfortable with. Just keep in mind to focus on the content, and adapt the tool to the content, not vice versa.

We'd love to hear your feedback and other experiences with this topic: [@maxiloc](#), [@algolia](#).

API Documentation

<https://alistapart.com/article/the-ten-essentials-for-good-api-documentation>

<https://alistapart.com/article/ten-extras-for-great-api-documentation>

Diana Lakatos





A LIST APART

Diána Lakatos, senior technical writer at Pronovix, wrote two guest articles for [A List Apart](#), on the essentials and the extras for your API documentation.

Read the articles here:

[The Ten Essentials for Good API Documentation](#)

and here:

[Ten Extras for Great API Documentation](#)

API Design Guidelines

<https://github.com/paypal/api-standards/blob/master/api-style-guide.md>

**Sanjay Dalal, Jason Harmon, Erik Hogan, Jayadeba Jena,
Nikhil Kolekar, Gagan Maheshwari , Michael McKenna,
George Petkov, and Andrew Todd**



Introduction

The PayPal platform is a collection of reusable services that encapsulate well-defined business capabilities. Developers are encouraged to access these capabilities through Application Programming Interfaces (APIs) that enable consistent design patterns and principles. This facilitates a great developer experience and the ability to quickly compose complex business processes by combining multiple, complementary capabilities as building blocks.

PayPal APIs follow the [RESTful](#) architectural style as much as possible. To support our objectives, we have developed a set of rules, standards, and conventions that apply to the design of RESTful APIs. These have been used to help design and maintain hundreds of APIs and have evolved over several years to meet the needs of a wide variety of use cases.

We are sharing these guidelines to help propagate good API design practices in general. We have pulled extensively from the broader community and believe that it is important to give back. The documentation is as generic as possible to make it easier to incorporate into the guidelines you use in your projects. If you have any updates, suggestions, or additions that you would like to contribute, please feel free to submit a PR or create an issue.

Document Semantics, Formatting, and Naming

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC 2119](#).

The words "REST" and "RESTful" MUST be written as presented here, representing the acronym as all upper-case letters. This is also true of "JSON," "XML," and other acronyms.

Machine-readable text, such as URLs, HTTP verbs, and source code, are represented using a fixed-width font.

URIs containing variable blocks are specified according to [URI Template RFC 6570](#). For example, a URL containing a variable called `account_id` would be shown as https://foo.com/accounts/{account_id}/.

HTTP headers are written in camelCase + hyphenated syntax, e.g. Foo-Request-Id.

Contributors

[Sanjay Dalal](#) (former member: PayPal API Platform), [Jason Harmon](#) (former member: PayPal API Platform), [Erik Hogan](#) (PayPal API Platform), [Jayadeba Jena](#) (PayPal API Platform), [Nikhil Kolekar](#) (PayPal API Platform), [Gagan Maheshwari](#) (former member: PayPal API Platform), [Michael McKenna](#) (PayPal Globalization), [George Petkov](#) (former member: PayPal API Platform) and Andrew Todd (PayPal Credit).

Read the whole article here:

<https://github.com/paypal/api-standards/blob/master/api-style-guide.md>

A Guide to RESTful API Design: 35+ Must-reads

<https://techbeacon.com/guide-restful-api-design-35-must-reads?amp>

Bill Doerrfeld



When it comes to designing web APIs, no other style is more respected than REST. Outlined by Roy Fielding in [his famous dissertation](#), REST, or representational state transfer, has become the go-to method for [designing powerful APIs](#) that run over HTTP.

Resource naming, hypermedia, proper HTTP method usage, caching, [idempotence](#), versioning, and other API design elements all come with best practices. For API developers, learning the nuances of RESTful API design is critical if they want a high adoption rate for their APIs.

However, a lot has been said on the topic, so I've assembled this collection of more than 35 top resources on REST API design—eBooks, tutorials, and articles—that will not only get you up to speed, but act as a guide throughout your API development lifecycle as well.

Check out the whole collection here:

<https://techbeacon.com/guide-restful-api-design-35-must-reads?amp>

What is the Difference between API Documentation and a Developer Portal?

<https://pronovix.com/blog/what-difference-between-api-documentation-and-developer-portal>

Kathleen De Roo



A developer portal is more than just the documentation for an API. As a sort of self-service support hub, it is a key DevRel tool that helps an organization to provide the best possible developer experience for its APIs.

A developer portal has a role in support, marketing, sales, and engineering. It is an environment that:

- Provides all necessary materials and services needed to reduce friction when working with an API (onboarding, registration, API key provisioning, payments),
- Generates [trust](#) and gives an indication if your business will be committed to an API over a long enough period,
- Helps potential API customers find the developer portal and the API products it contains through Search Engine Optimization and other web traffic generation,
- Has content for all API [stakeholders](#) no matter where they are in their user journey,
- Has tools to manage and maintain the relationship with API customers.

At least that is how we think about developer portals at Pronovix.

A conversation on the [#documenting-apis WTD slack channel](#) sparked the idea for this blog post. [@rosewms](#) asked this exciting question and documentarians ([@docsbydesign](#), [@ellispratt](#), [@hangingwater](#), [@jrondeau](#), [@ikoevska](#), [@kvantomme](#), [@lemay](#), [@melissamahoney](#), [@monique.semp](#), [@neal](#)) shared their unique insights.

What follows is a summary of the discussion.

API documentation and its components

What is the role of a developer portal?

One key question was the role of a developer portal. [@lemay](#) points out that, on the one hand, the way we label API documentation has evolved. On the other hand, the scope of the developer portal also widened: having a [list of endpoints is not enough, give users the opportunity to learn and understand how an API works:](#)

@lemay

Traditionally API documentation was narrowly defined as just the reference docs — just the list of classes, methods, etc. Additional guides, getting started, best practices, FAQ, etc were other parts of the overall developer docs set. These days “API docs” seems to mean “all docs related to a developer API”. Also traditionally the entire developer docs set were rolled up into a package with samples, starter code, libraries, etc, called the SDK (software development kit). Depending on the library (and especially for web APIs) the SDK could very well have been just all docs. So for a while the idea of the developer portal was just the SDK and API docs, put online. That’s still in many cases what a developer portal is. It’s a place where developers go to get information about the thing they’re developing for. More recently as the developer audience itself has grown and there are more services around developer support, the idea of a developer portal has grown a lot. It can still be just docs but IMHO typically includes forums, blog posts, support resources, video, etc, etc, etc. For some developers — mobile apps come to mind — the developer portal may also be the place where you submit your code to an app store, manage different versions of your app, get information and analytics about how your app is doing in the market, and get paid.

@ikoevska specified the relation between portal and documentation:

So, a Developer Portal would contain API docs but not the other way around I think.

@ellispratt indicated the different roles a portal can fulfill:

I'd see a portal as training, reference, task/help info, and discussion

We can also interpret the following comments in this light:

- @rosewms highlighted the community:

other factor might be a dedicated, connected community/forum space as a part of the site

- @melissamahoney put the role of sandboxes for developer portals forward:

maybe that one is more traditional documentation and the other is more like a sandbox?

- @jrondeau also mentioned sandbox environments:

the portals I've helped develop include sandboxes

- @ikoevska drew attention to tutorials and code resources:

API docs, to me, mean a reference guide with all the classes, properties, yada, yada. And tons of examples. Maybe a tutorial or two thrown in for good measure. And a Hello World app how-to probably

The different roles of a portal are reflected in the documentation types.

API docs is more than reference docs

This fact was pointed out by @jrondeau (“often “API documentation” is shorthand for only the reference docs - plenty o’ other docs also needed”) and @monique.semp (“A “developer portal” is just a portal to all sorts of docs. The docs likely are API focused, but should also likely have other things, too”).

Nevertheless it is a truth universally acknowledged, that API docs are often not much more than, say, a Swagger UI, notices @rosewms (“many people just do the swagger/raml/api blueprint and call it a day”). While [autodoc tools help to produce documentation more swiftly and error-free, they miss the human touch](#). A combination of both worlds will create an environment that addresses every type of user.

What documentation types do developers need?

@kvantomme

Most of the portal will still be some sort of documentation, but it will be organised to help developers move through the different stages faster. A developer portal can also help to accentuate the role of documentation in the marketing and sales process. As such I think there is a much larger potential for ROI.

Developer portals have several [stakeholders](#):

- Developers
 - developers inside the company that provides the API product
 - consumer developers of the API client and end-consumers
- Product owners
- Marketers
- Salespeople
- Developer evangelists
- Support team members
- Documentarians

Stakeholders go through user journeys. Each stage in those journeys will provoke new information needs.

With developers as key audience, their user journey includes the following phases:

@kvantomme:

- discovery/research (landing pages)
- evaluation (worked examples)
- getting started (tutorials)
- development/troubleshooting (guides & API reference)

- celebration (showcases e.g. XYZ did this amazing app)
- maintenance (e.g. API usage,...)

We could interpret **API documentation as the information on the portal that users might need during their user journey**. The docs can be categorized into types or components, like:

- Support resources, like FAQ pages, knowledge base articles, contact forms, community sections
- Onboarding documentation, like tutorials, (topic) guides, worked examples, how-to guides, try out sections (e.g. via a sandbox environment), SDKs
- Troubleshooting documentation, like reference docs
- Showcase options, like case studies, use cases, blogs
- Trust docs, like API status, uptime status
- ...

The listed categories intermingle of course - and there are other ways to list the different documentation components. At Pronovix, we are currently working on a method to portray them efficiently.

Terms: documentation, portal, documentation portal, developer portal

A next topic was about terminology and labeling: what to call the docs and the portal?

@rosewms

I'm in the midst of documenting an API so I'm trying to figure out if and what value add for positioning it as documentation vs developer portal and a lot of the hosting companies/sites use it interchangeably or don't show huge differences regardless of the term used

We came across these synonyms for API documentation during our [developer portal components research](#):

- Documentation
- Developer documentation
- Developers
- (API) docs

The label “developer portal” causes confusion. Some contributors brought up these solutions and examples:

@ikoevska

I'd say that Developer Portal is for when your readers are developers. It should contain various resources and some way to communicate with the community, also support channels, etc. There should be a downloads section (if anything is available for download, etc.) While Documentation Portal stands for documentation only.

@docsbydesign

I've seen the term "portal" used when a product has multiple audiences (e.g. users and developers). "Dev Portal" is then to distinguish developer-related stuff from regular user-related stuff. As such, "portal" content is whatever is interesting to the dev audience but not other audiences. The challenges come when the lines between the audiences are not sharp. In which case you need to make sure each reader can tell they are in the right "portal".

@jrondeau

if devs can get keys, auth, actual access (whether to sandbox or production) using your content, maybe call it portal

Developer portal types

At one point, the contributors of the discussion started to talk about public and private portals. Some suggested (like @jrondeau above) to call the platform that is hidden behind

a login/paymentwall or authentication process the actual portal. Others pointed out that, nevertheless, most portals make quite a few documentation types publicly accessible.

@lemay

Most of the developer portals for apps require a payment to view them.

@hangingwater

Years ago I worked at a company that sold a graphics SDK and we had what we called a "support site" - this was an area of the website that only customers could access (they had to login) where they could find downloads, SDK docs (API reference, user guides, tutorials, white papers) and also a support query form and half-hearted FAQ. Nowadays we would definitely call that a "developer portal".

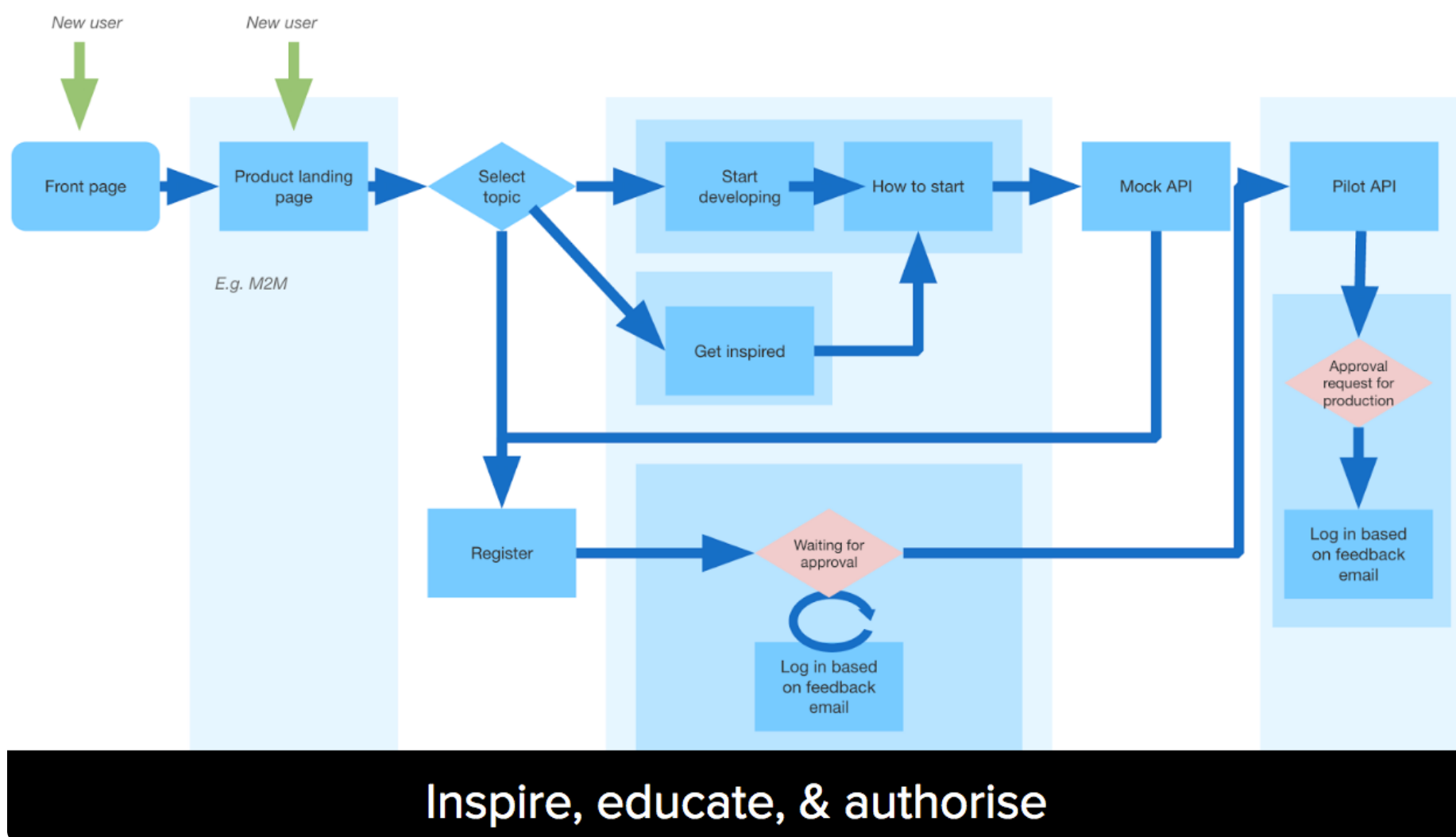
Basically everything that someone interested in your API/SDK would want access to, with a landing page explaining where to get everything. I'm guessing the "developer portal" term came from either Apple or Android? As @lemay said, the developer centre for Apple iOS devs has tools for uploading your app to the store, getting stats, etc., so not necessarily all just to do with programming with the API.

we definitely used to send docs to the senior techies at prospects (or grant them access to the support site / dev portal). I suppose that does illustrate that a dev portal may include both public and private aspects - for example I think anyone can read the iOS developer docs, but you need to be a member of the Developer Program (and have paid your \$99) to get onto the Dev Centre. Which is the "portal"? In that context is it just the landing page that orients you to the API docs and also to the Dev Centre (and instructions about how to join?)

We believe there are several types of developer portals, and, moreover, the types can intermingle:

- **Flat access portals:** public or firewalled developer portals: The site might be behind a firewall or you might have to log-in, but its structure is flat: users have access to all API documentation without segmentation.
- **API Catalogues:** submission workflows & discovery: Internal agility type of portal.
- **Partner & customer APIs:** Developer portals with differentiated access permissions: These portals have personalized restricted access.
- **Utility APIs: portals for services with metered access (Pay as you go):** Measurements are based on use.

The diagram below shows how portals can organize a chain of actions to inspire, authorize and educate.



Infographic: organize a chain of actions to inspire, authorize and educate (by Pronovix)

Role-based access control with Drupal developer portals

Login requirements also connect to keeping specific information private, for several reasons, like:

- Making a distinction between users or customers
- Business policies

@neal

“a dev portal may include both public and private aspects” — sure, I can imagine cases where you’d want to protect certain info (if it could be abused, I assume?)

@hangingwater

well for example in the case of an app store, you can only submit apps to it or see stats on sales if you are a member of the dev program. Or you may want to make the documentation relatively freely available, but only paying customers should be able to actually download the SDK

Drupal developer portals have role-based access control customizations, which enables API providers to give distinct access. RBAC is able to control the accessibility of the API products and the corresponding API documentation based on the groups created and managed within the system. With this system, developer portal administrators can create groups, assign content to groups, add members (users) to them, and manage group visibility or the visibility of specific group content individually.

@kvantomme

Role-based access control is a really big feature for the portals we build, e.g. to hide docs of APIs that are only accessible for partners or internal developers. In fact I think that is one of the biggest features that a CMS based dev portal really excels in, in comparison to static site generators.

@hangingwater

Some companies do tend to want to keep all their developer / SDK documentation secret unless you pay - I don't personally favour that approach (and perhaps it is changing). As *@kvantomme* says, a CMS based portal with different levels of access may be a good solution

But which services and documentation components should go behind the login or paywall?

@neal

Excellent things to note: what parts of a dev portal can be left open (less annoyance/no need to sign in, good for marketing), which pieces are OK to be behind a login/paywall

Keeping certain content private implies a different marketing strategy.

@kvantomme put it like this:

if your docs are behind a login, you lose all the SEO juice... I think it makes sense to have a short description for all your APIs public, so that you have those keywords out there. The actual reference docs you might want to hide, and then some APIs that you don't want your other partners to get mad about (e.g. why do they get that and we don't), you might want to hide completely.

One of the best practices to introduce your developer portal documentation types is to provide a landing page: a front page that gives [an overview of all available documentation](#). It is the place where users usually land, therefore it is important to communicate the API product(s) well, based on who your primary personas are and on what they expect of your site. There are several classification systems to structure information: by documentation type, by product, by programming language, by objectives. The chosen systems influence the way users will find information and will work with the portal.

Read more about the [common customizations for Drupal based developer portals](#).

Developers and marketing

@neal

and to beat that drum again: documentation is marketing if your dev portal includes docs, community, and what @kvantomme calls “celebration showcases”, that’s exactly the sort of thing your potential customers need to see before they buy

It is said that developers hate marketing. But if you, as pointed out by [Adam DuVander](#), “share knowledge, not features”, you still market your product, but in a language developers know and like. Good documentation makes information developers are looking for available, especially if you optimize it for search engines.

Give developers a reason to share and recommend your portal:

- have a space where developers with successful API integrations receive attention (e.g. via a community section, a community portal, forum or as a guest author for a blog post or use case),
- feature their integrations (it increases their market value),
- provide gamification tools that will spread the word about your portal and attract new users.

If developers can easily share their work, your portal will get visibility among their peers and drive others to adoption.

@hangingwater

Am I being cynical in thinking the term "dev portal" is largely to distinguish the dev facing site from the main "dull and boring marketing heavy corporate website"?

Our [own research results](#) show that companies often opt for a developer portal separated from the marketing site (where, of course, they provide a link to the portal or its documentation).

Final thoughts

- Developer portals initially mainly focused on developers and reference documentation, but have evolved towards:
 - Addressing other stakeholders, like decision makers and technical writers
 - Including documentation types that fit in with the journeys those stakeholders undertake.
- Organize the documentation on a portal in a way that will help your users move through the different user journey stages faster.
- API documentation as the medium between the API and the user: combine the splendors of automatically generated docs with content that addresses users more personally.
- Provide a landing page and short descriptions about the API, especially when parts of your portal are behind a login/firewall, in order to optimize search. Make sure the landing page answers initial questions ad hoc, like:
 - what is the product about?
 - how can you use it?

Thanks to the people in the WTD slack channel for the idea, comments and suggestions!

CIDM IDEAS 2018

| <https://pronovix.com/blog/api-documentation-best-practices>



API Documentation Best Practices

By Diána Lakatos

PRESENTATION AT THE CIDM IDEAS 2018—WRITING WELL ONLINE CONFERENCE

Organized by the Center for Information–Development Management, IDEAS is a two-day industry conference including five 60-minute concurrent sessions in two tracks. The theme of this winter's conference was Writing Well, where we spoke about API Documentation Best Practices.

In this session we talked about different aspects of API documentation:

- Why API documentation plays an important role in the API economy
- The difference between API documentation and developer portals
- The target audience of API docs
- Developer portal components and best practices for arranging and writing content for each
- Usability considerations, like interactions, readability and personality
- Ways to edit and keep your documentation up-to-date, workflow considerations, and the docs like code approach

By the end of the presentation, we hope to have given Technical Writers a solid starting point for seeing how they can convert their technical writing skills to API documentation writing, along with some resources that they can use to learn more. Although we aimed the presentation at Technical Writers, we think that all stakeholders of an API documentation project can benefit from watching it.

Watch the presentation [here](#).

DEVELOPER PORTAL ANALYSES



The Best Developer Experience KPIs

<https://pronovix.com/blog/best-developer-experience-kpis>

Jenny Wanger



I spoke with various Developer Experience product managers about how they measure success—what KPIs they use and why. It became apparent that the responsibilities of each DX team shaped the KPIs they use. There are four potential domains that developer experience teams are responsible for, with different KPIs for each domain.

The Four Domains				
	Get people to the developer portal	Convert people on the developer portal to customers	Retain developers	Operate the API gateway or program
Primary KPIs	Increasing quality people landing on the dev portal	Conversion rate	Retention rate	Uptime and latency
Leading Indicators	<ul style="list-style-type: none"> • Site analytics • Time on site • Click-thru rates • Bounce rates 	<ul style="list-style-type: none"> • Usability testing and developer satisfaction • Time to first hello world 	<ul style="list-style-type: none"> • Developer satisfaction • Adoption and use of tools • Deflected queries • Pull requests to open source projects 	

Of the four areas, everyone I spoke with was responsible for the middle two sections. Depending on the company, the team also had one of the responsibilities on the ends, but no team I spoke with was responsible for all four domains. While not everyone tracked the same KPIs, I will highlight the ones that I found to be the best fit. When discussing the KPIs below, I define them as primary KPIs and leading indicators. I do this because each domain has one KPI that is the ultimate measure of success. Because the KPI can be hard to measure or slow to gather, many teams use leading indicators as secondary metrics to support their work.

Get people to the developer portal

Not every DX team is responsible for marketing to developers, but plenty of them have responsibility for developer evangelism. As [Cristiano Betta](#) explained to me, if you're responsible for traffic to your website, then you're really responsible for getting potential

customers to the point of sale. Given that, your KPI should be focused on increasing the number of quality visitors that arrive on your site. This requires defining your target audience, figuring out how to target them, and serving the right content at the right time. A good analytics program is critical at this point.

Convert people on the developer portal to customers

Once you have people on your site, it's time to convert them and, just as in ecommerce, your main KPI becomes your conversion rate. The KPI is easy to understand but hard to break down, and so leading indicators are especially helpful here. [Bandwidth](#) uses usability testing, time to first hello world, and site analytics together as leading indicators. The combination of qualitative and quantitative measures here seems especially helpful.

Retain developers

Once you've got your customers, it's time to retain them. As with the conversion piece, all the product managers I spoke with talked about this domain in one way or another. Some phrased it as "supporting the developer community" or "working with customers", but it comes back to retention. The main retention rate is straightforward enough, but leading indicators are less concrete. Possible metrics include adoption and use of tools, an increase in volume of API calls, and pull requests on open source projects. Returning to the ecommerce analogy, it's equivalent to size of shopping cart.

Operate the API gateway or program

The least common responsibility of a DX team is actually managing the operations of their developer program. In one case the product manager explicitly told me that this is a DevOps role, and it was not the primary focus of any DX team I spoke to. However, for those who need to operate the program as a part of their responsibilities, standard DevOps metrics apply: uptime, latency, etc. The general consensus was that the more mature the company, the less likely this role would be part of the DX responsibilities.

Developer Satisfaction

While each specific area has corresponding KPIs to match, the one piece that touches the whole spectrum is developer satisfaction. Given that it's an emotional goal, it's very hard to measure with anything concrete. Some use net promoter score, but that gets fuzzy—it's hard to separate out satisfaction with your developer experience program versus the company overall. [Slack](#) creates a few leading indicators for developer satisfaction by keeping an eye on documentation quality and adoption of tools. They're also great at the most important part of measuring satisfaction-- listening and talking with their developer community, whether through online tools or events, which allows them to really capture the qualitative side of this metric. Others mentioned deflected questions—questions that get answered through a knowledge base or support flow without requiring a person to intervene to help the developer find resolution.

Choosing your KPIs

Defining your KPIs is an opportunity to define the scope of your team's responsibilities. Take the time to connect across your organization to figure out who is taking care of your customers and at what point; once you know your responsibilities, the KPIs should follow.

What is the Role of Blogs in the Developer Journey?

<https://pronovix.com/blog/what-role-blogs-developer-journey>

Kathleen De Roo



Companies use blogs as an informal, unstructured **communication tool** that can engage current and future customers and members of their wider community. Blogs are often used to start and maintain relationships, to update an audience, discuss new functionalities, and to inform about company decisions. Because blog posts are published as a sequential series of updates, they create a fleeting feeling of urgency, serendipity and newsworthiness, which makes it much more likely visitors will share blog posts through social networks.

Blogs also function as a **content incubator**: an unstructured area in the content architecture where you can develop new types of content for your developer portal, a place where new content types “incubate” before they get their own place, like a company's first "getting started tutorial" or "how to do XYZ" guides.

Most API teams use blog posts on their developer portal or business site as a collection of documentation formats (tutorials, testimonials, interviews, guides etc). Blog posts can:

- **Educate users:**
 - Demos and mashups help users explore the API and its functionalities.
 - Blog posts often contain code samples and help with problems in specific areas.
- **Build trust:**
 - The frequency of blog posts, and the time since the last update are an indicator of the health of an API.
 - Blog posts can communicate policies and team culture.

Blogs can help fulfill these needs throughout the whole API implementation process. In this post I'll explore how blogs can serve the portal users' needs throughout the different stages of the developer journey. I'll discuss their labels and subcomponents, and extract best practices. To finalize, I'll list some questions to keep in mind when deciding whether, when and how to plan a blog on your developer portal.

To write this post, I reviewed the blogs of [Adyen](#), [Amazon](#), [Apigee](#), [CenturyLink](#), [DigitalOcean](#), [Dropbox](#), [Dwolla](#), [Facebook](#), [GitHub](#), [Google](#), [IBM](#), [Instagram](#), [Keen IO](#), [LinkedIn](#), [Mapbox](#), [Orange](#), [Pinterest](#), [Slack](#), [Spotify](#), [StackOverflow](#), [Stripe](#), [Twilio](#) and [Twitter](#). These companies have active communities and/or provide a wide variety of developer resources. I explored how they organize their blogs and what purposes the blogs serve.

Blogs in the developer journey

The blogs in our research sample announced events, hooked their users, explained functionalities, communicated company decisions, linked to other documentation resources, provided support in certain areas, explained concepts (domain language), provided code samples, and encouraged users to think out of the box when applying the API. Blogs are generally filled with videos, images, screenshots and explanations in everyday English.

Developers go through 6 stages when implementing an API into an application. These 6 stages are **discover, evaluate, get started, troubleshoot, celebrate, and maintain** :

1. Discover


Optimize the API for search engines: blog posts are often keyword sensitive, they might help users to find their way to the developer portal and its documentation.

Role: The blog as a marketing tool in the developer's exploration phase


- Hook users, get them interested in the API product (e.g. add code samples for developers).
- Add user stories and case studies to reach out to other decision makers, like product owners and marketing engineers).
- Show interesting integrations that require out of the box thinking.


DWOLLA Products Developers Log in Sign up Let's talk

All Updates Resources Developers Use Cases Search updates

 **Case Study: Nomad Health integrates ACH-optimized Access API for Same Day ACH payouts**
 BY: CAITLIN, MARCH 9, 2017

A modern healthcare staffing platform leverages Dwolla's Access API to accept ACH payments from medical facilities and payout directly to clinicians' bank accounts. Download the Case Study PDF According to Staffing Industry Analysts, the temporary healthcare staffing industry is projected to reach \$15 billion in revenue in 2017. Nomad ... [Read more](#)

 **Case Study: Patch of Land leverages Access API for Real Estate Crowdfunding**
 BY: MARIAH YOUNG, MARCH 5, 2017





Use cases or case studies (Dwolla blog)


Slack Platform Blog Follow Sign in / Sign up

HOME ANNOUNCEMENTS HOW TO APP DIRECTORY OP-ED DEVELOPER STORIES

TAGGED IN
Developer Stories

 Slack Platform Blog
 Several bots are typing...
 More information
 FOLLOWERS 4.4K
 ELSEWHERE
 MORE, ON MEDIUM
 Developer Stories

 **Workstreams in Slack Platform Blog**
 May 9 · 4 min read

 4 replies Last reply 6 days ago

- workstreams
- alex
- isabelle
- bianca (you)


chris 11:43 AM
 We should do this as soon as possible

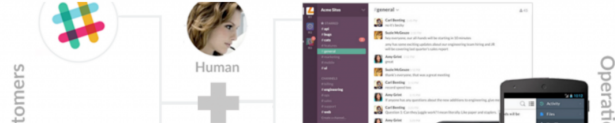
Workstreams APP 1:38 PM
 Task archived by Chris

Bringing bot interaction to a new level
 How we used Slack's most recent updates to interactive...

Read more...

23 1 response

 **Josh Barkin in Slack Platform Blog**
 Apr 4 · 7 min read



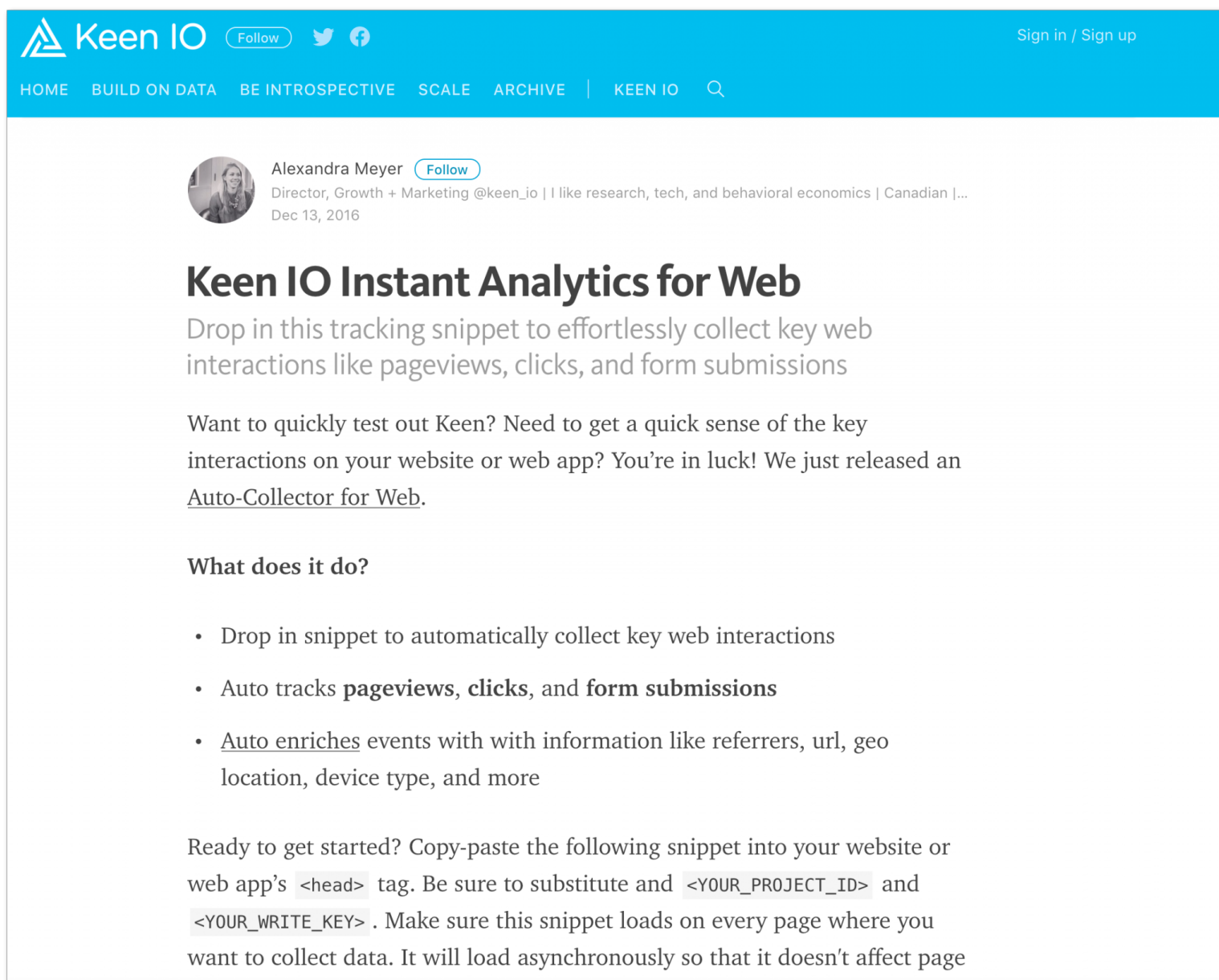
Developer Stories (Slack Blog)

2. Evaluate

Enable people to evaluate what's on site, via mock APIs, test accounts, tutorials, sample apps: blog posts and articles can explain how an API works, and how it can be implemented in plain English.

Role: Blog posts as a collection of (best) practices

- Show expertise: blog post writers bundle knowledge about a specific topic.
- Highlight interesting or popular integration examples and tutorials.



The screenshot shows a blog post on the Keen IO website. The header is blue with the Keen IO logo, a 'Follow' button, and social media icons. The navigation bar includes links for HOME, BUILD ON DATA, BE INTROSPECTIVE, SCALE, ARCHIVE, and KEEN IO. The post is by Alexandra Meyer, Director of Growth + Marketing at Keen IO, dated Dec 13, 2016. The title is 'Keen IO Instant Analytics for Web'. The main text describes a tracking snippet for collecting web interactions like pageviews, clicks, and form submissions. It mentions the release of an 'Auto-Collector for Web' and lists its features: automatically collecting key web interactions, tracking pageviews, clicks, and form submissions, and enriching events with information like referrers, URL, geo location, and device type. The post concludes by providing instructions on how to use the snippet in a website or web app's <head> tag, including placeholders for project ID and write key.

Test Keen IO (Keen IO blog)

Build a smart doorbell with Twilio and Android Things



by Marcos Placona on June 6, 2017

Like Tweet Follow @twilio

Working from home and being an avid Marilyn Manson fan means I usually miss out when someone's knocking on my door. We have a doorbell but it's not nearly as effective as the noise cancelling capabilities on my headphones.

My headphone is paired to my phone which still means I will miss someone at the door, but never any calls. How about we build a smart doorbell that rings my phone when someone's at the door?

CenturyLink provides a blog topic "tutorial", but this page is also directly available from the top menu (CenturyLink blog)

Examples of an integration: complete use case/tutorial with code snippets, written in plain English (Twilio blog)

CenturyLink | Developer Center

Blog Tutorial Events SDKs & Tools Support

Self-Hosting Git Repositories with Gogs and CenturyLink Cloud

READ MORE

FILTER BY CATEGORY: TUTORIAL SUBSCRIBE TO OUR BLOG

Enter Term

Self-Hosting Git Repositories with Gogs and CenturyLink Cloud
Change tracking and version control are essential tools for software development. In fact, they are one of the pillars u...

Deploying WordPress with Janrain Social Login on CenturyLink Cloud
Keeping track of hundreds of

Example of a sample app article (Dropbox developer blog)

Dropbox Developer Blog

Topics Subscribe Dropbox blogs

Category: "Sample apps"

Try out Dropbox Business endpoints in .NET

Stephen Cobbe | March 14, 2016

Want to see the Dropbox Business API in action? We've built a .NET sample app that shows you how to link to Dropbox business teams, and use the activity endpoints to get statistics about the members. Check it out in our .NET GitHub repo, [here!](#)

This simple dashboard offers a visual overview of some of the team data that Dropbox endpoints expose. Using the [Team member file access](#) permission level, data on team information, team daily active users, distinct apps that team members have linked, shared folders with activity in the last week and team member rosters,

Filed under: [Sample apps](#), [Tips & tricks](#)

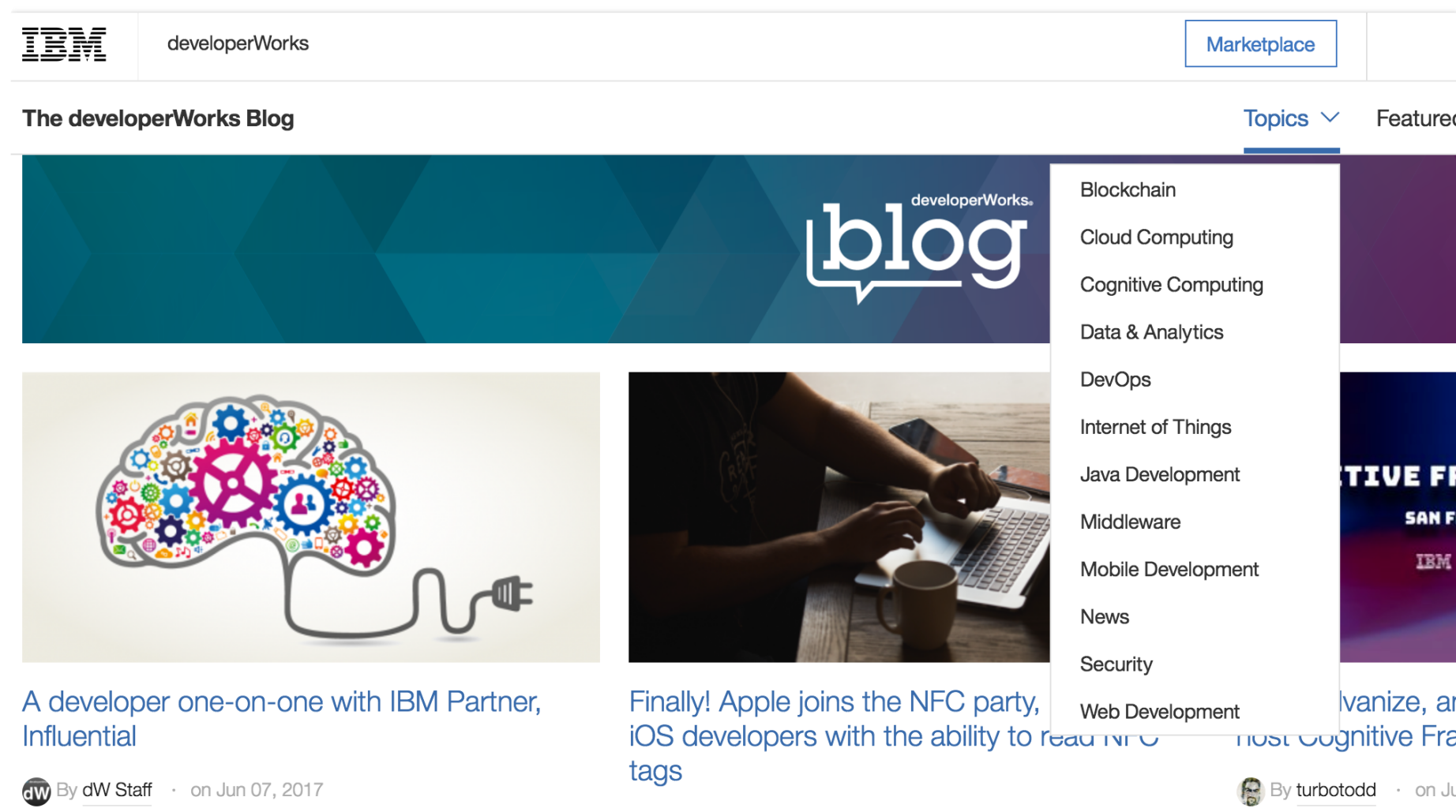
[Read more](#) →

3. Get started

Blog posts can include code samples, test cases and user stories that might inspire fellow developers to get started with their implementation.

Role: Blog posts as onboarding tools

- Include articles for both beginner and experienced users.
- Provide information about your products, e.g. via a topic selector.
- Link to the portal's knowledge base to find more information on certain topics.
- Explain concepts that your users might need to know before implementing.



The screenshot displays the IBM developerWorks blog interface. At the top left is the IBM logo and the text 'developerWorks'. A 'Marketplace' button is visible in the top right. Below the header, the page title 'The developerWorks Blog' is shown, along with 'Topics' and 'Featured' navigation options. A large banner features the 'developerWorks. blog' logo. A dropdown menu is open, listing various technology topics: Blockchain, Cloud Computing, Cognitive Computing, Data & Analytics, DevOps, Internet of Things, Java Development, Middleware, Mobile Development, News, Security, and Web Development. Below the banner, two article thumbnails are visible. The first article is titled 'A developer one-on-one with IBM Partner, Influential' by dW Staff, dated Jun 07, 2017, with an image of a brain composed of gears. The second article is titled 'Finally! Apple joins the NFC party, iOS developers with the ability to read NFC tags' by turbotodd, with an image of hands typing on a laptop.

Topic selector (IBM developerWorks blog)

8 Common Customizations for Drupal-based Developer Portals

by Kristof Van Tomme & István Zoltán Szabó, Pronovix (@szabosteve)
MAY 23, 2017

In the [previous post](#), we covered four questions that help identify the purpose of a developer portal and help to facilitate the decision-making process. Answering these questions has helped us, in collaboration with the Apigee team, custom fit the Apigee Drupal-based developer portal to a variety of Apigee's customers' individual requirements (when this post refers to developer portals, it is referring to the Drupal-based portal, not the [new, lightweight portals](#)).

Here, we'll discuss the most commonly requested custom implementations. You'll see it's a diverse set, both in scope and in function.

SSO implementations

Single sign-on (SSO) is an authentication process that enables users to employ one set of login credentials to access multiple services or applications. It simplifies the authentication process, because if a system or a service (for example Gigya, Okta, or Google) already authenticated a user, then the users don't have to login again at every visit.

Pronovix customers often have a large group of websites. With an SSO implementation, it's possible to log in only once on one website and enable a user to access the other sites of the particular group without having to log in again.

Role-based access control

Role-based access control (RBAC) provides a scrupulously customizable access system implemented on the Drupal developer portal. RBAC is able to control the accessibility of the API products and the corresponding API documentation based on the groups created and managed within the system.

With this system, developer portal administrators can create groups, assign content to groups, add members (users) to them, and manage group visibility or the visibility of specific group content individually.



SOA
API Management
OAuth
BaaS
Analytics
API Monetization
API Security
I Love APIs

Microservices



Download the e

Contact Us

Watch Demo

Guest article on customizations, explaining several concepts (by Pronovix, Apigee blog)

4. Troubleshoot

Blog posts explain and communicate about problem areas and function therefore as support tools. Blog posts are also an internal evaluation tool: e.g. to explain a product works you will also be to testing it at the same time.

Role: Blog posts as support resources

- Explain problem areas in plain English.
- Include code snippets.

Create an Express route to accept new inbound calls

Start by setting up a boilerplate Express app in index.js:

```
JavaScript
1  const express = require('express');
2  const bodyParser = require('body-parser');
3  const request = require('request');
4  const twilio = require('twilio');
5
6  // Set up and configure the server.
7  const app = express();
8  app.use(bodyParser.json());
9  app.use(bodyParser.urlencoded({ extended: false }));
10
11 // Twilio will load this route when you receive a new call.
12 app.post('/new_phone_call', function(req, res) {
13   // New phone call logic will go here
14 });
15
16 app.listen(3000);
```

Note that you'll have to `npm init -y` followed by `npm install -save express body-parser request twilio`. This will create a `package.json` file with default values and then write our dependencies to it.

Ask the user for their zip code

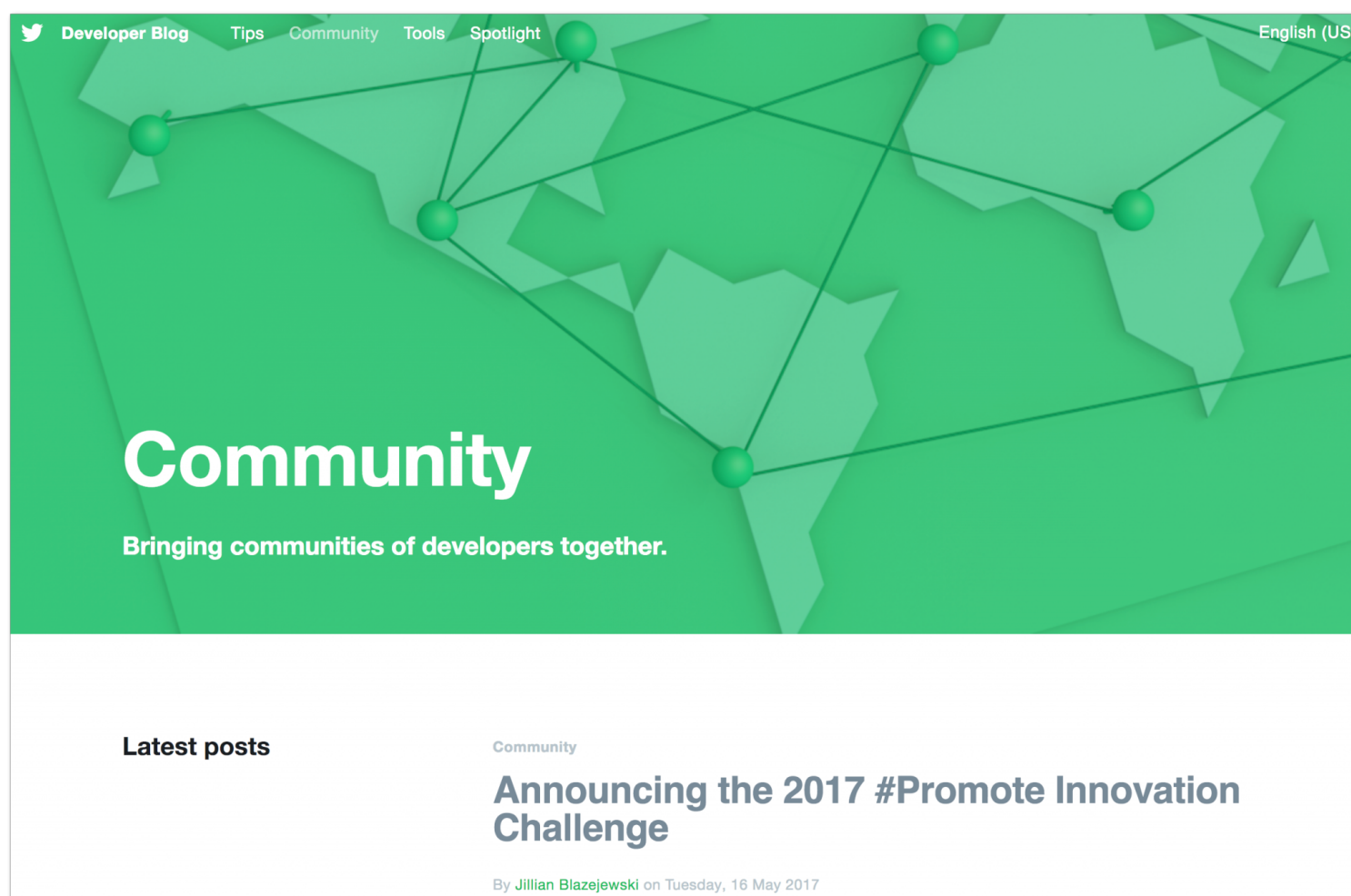
Example of an article with code snippets (Twilio blog)

5. Celebrate

Show developers that you care about their work: offer them a place on your portal, e.g. via guest posts or in interviews.

Role: Blogs can help grow a community


- Add Call-to-Actions to trigger readers.
- Write series on certain topics, in order to make users return to an interesting story.
- Include real life examples (via interviews, podcasts, guest posts, user stories).
- Announce events, write recaps.
- Provide articles that deal with everyday life tips (e.g. [Keen IO's Culture blog section](#)).
- Make community sections.
- "[Put a URL on it](#)": turn questions from the community into URLs, and provide blog posts to make it easier for users to find the content they are looking for.





Community section (Twitter blog)

CenturyLink® | Developer Center [Blog](#) [Tutorial](#) [Events](#) [SDKs & Tools](#) [Support](#)

FILTER BY CATEGORY: **INTERVIEW**


 SUBSCRIBE TO OUR BLOG

Enter Term 




CloudTalk: Customer Care - Skynet


Chris Welkenstein of the Customer Care Team within CenturyLink Cloud sat down to speak with us about...

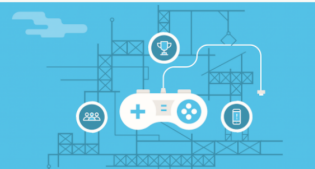


A Day in the Life - Steven Landow, an Emerging Developer


Our "Day in the Life" series gives you a glimpse into the everyday lives of team members a...








Interview section (CenturyLink blog)

 [Products](#) [Developers](#) [Industries](#) [Company](#) [Pricing](#) [Sign in](#)

BLOG

Filter: **EVENTS**



Security on the AWS edge at Enigma 2017

JAN 23 2017 by Ian Ward

I presented at AWS re:Invent in December, sharing how Mapbox uses different AWS edge services to achieve high availability and low latency for our customers around the world. As a...

[Security](#) [Platform](#) [Events](#)

Event news (Mapbox blog)

6. Maintain

Companies can use their blog to communicate about the API health.

Role: Blog posts indicate API availability and reliability

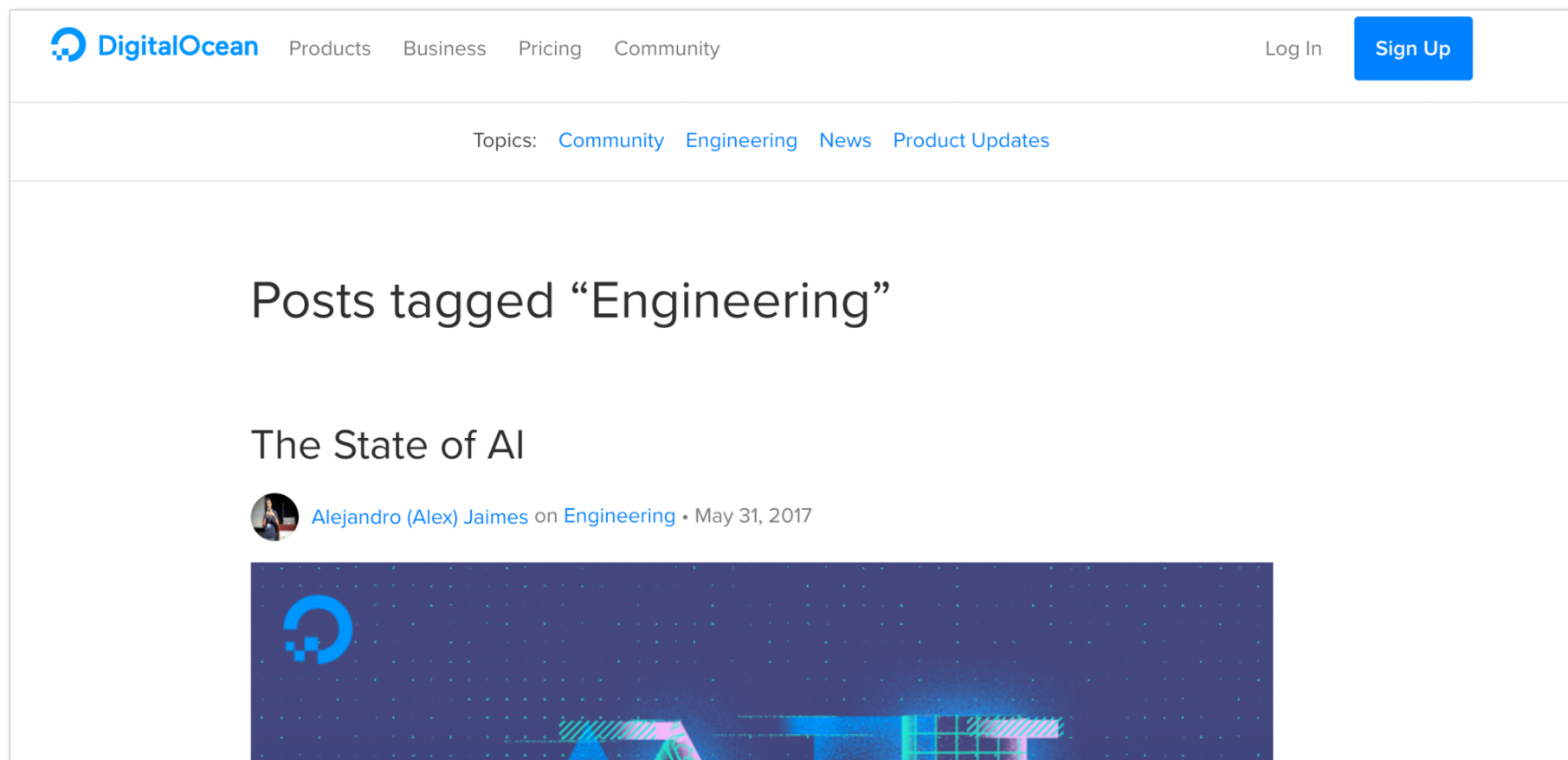
- Dedicate articles to news related to API uptime and release notes.
- Illustrate your services that help developers maintain their API integration with examples, use cases or guidelines.

The screenshot shows the Apigee website's developer section. The main article is titled "Generating Release Notes with APIs" by Floyd Jones, dated October 23, 2014. The article discusses the challenges of maintaining a fast-paced, agile environment and describes a solution where an API proxy is used to query a Jira REST API, convert the JSON response to XML, and then transform it into HTML for display in Postman. A diagram illustrates this workflow: Jira REST API sends JSON to the API Proxy, which uses JSON to XML conversion, then XSL Transform to produce HTML, which is then sent to Postman. A sidebar on the right lists trending topics such as OAuth, BaaS, Analytics, API Monetization, API Security, I Love APIs, SOA, and API Management.

Post with explanations how to generate release notes (Apigee blog)

Labels and subcomponents

In our research sample, we found developer portal blogs (blogs that are directly accessible from the developer portal) and more general blogs that also listed developer topics.

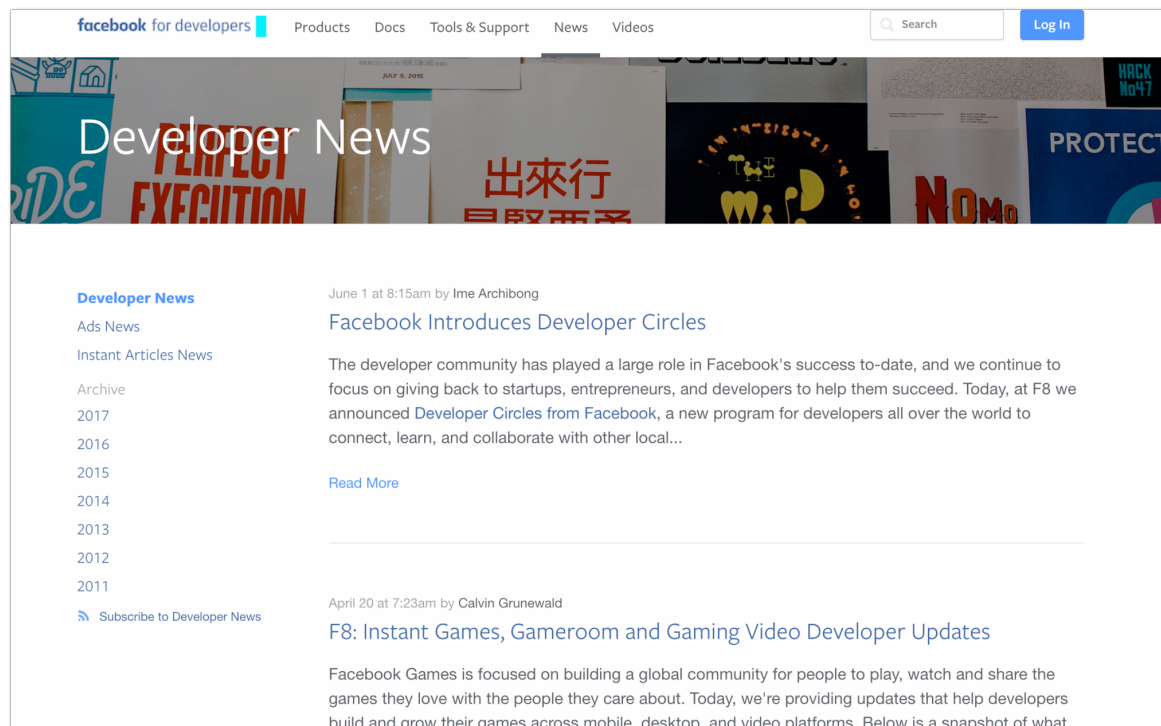


DigitalOcean's blog with audience focused topics

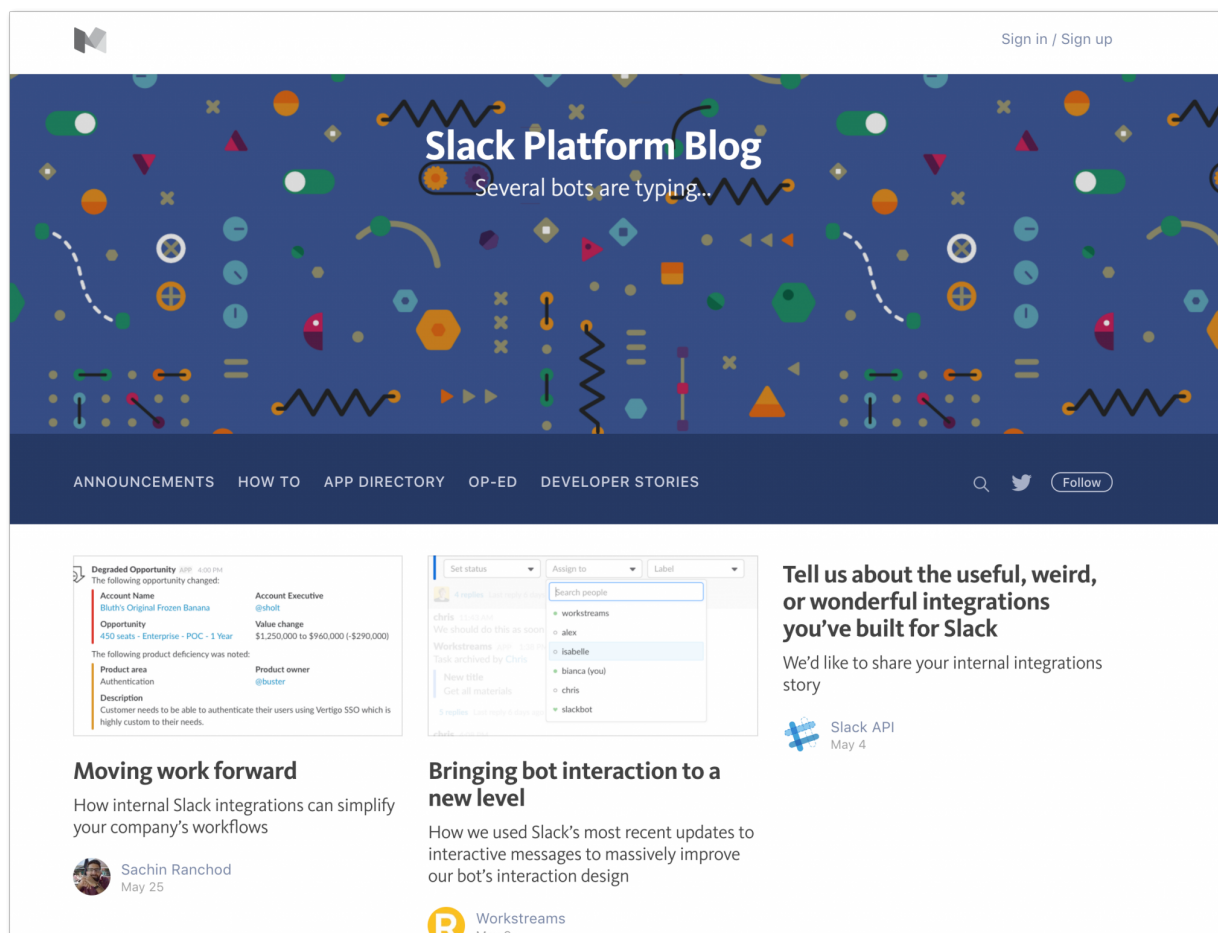
We found several labels:

- Companies put links to blogs in headers, footers, top menu bars, and sidebars reachable under category labels like blog, news, company, community, support, learn more, menu and products, API and docs.
- Blogs often received personalized names: besides "blog" or "developer blog", we found Developer News (Facebook), The Event Log (Keen IO), Spotify Labs (Spotify), Updates (Dwolla), Changes (GitHub).
- Apart from their ["What's new" page](#), Orange has a monthly newsletter, where they discuss topics that concern several aspects of the company.
- [Medium](#) as a platform for a company blog:

- Either with an overview page of blog posts on the company site, while the separate articles are published on Medium ([Keen IO](#)),
- Or also sometimes with the whole blog on Medium ([Slack](#)).



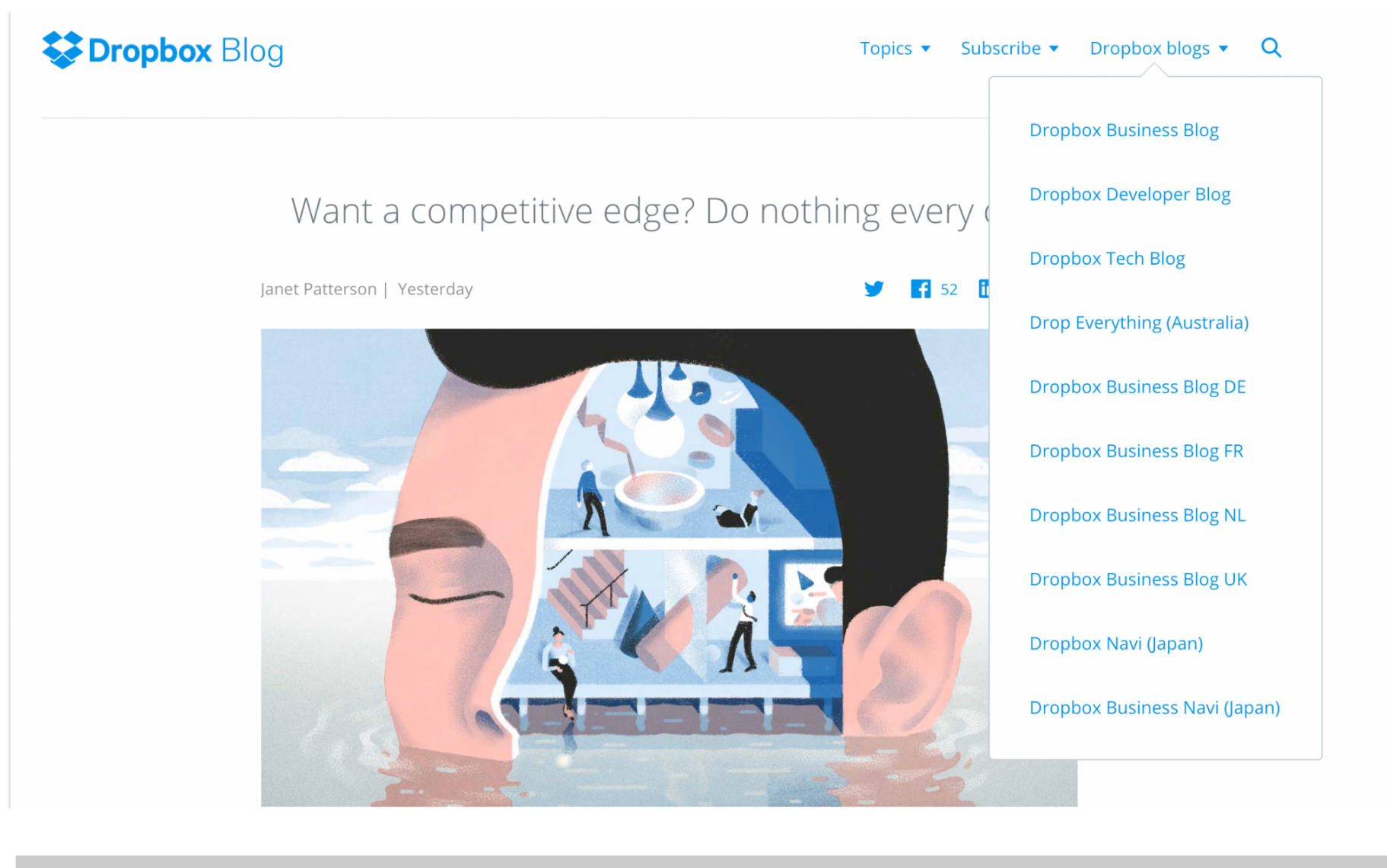
“Developer News” (Facebook blog)



Blog on Medium (Slack blog)

Subcomponents:

- Design elements, like gifs and images to hook users
- Categories, labels, topics, tags to make article selection easier
- Audience focused topics, categories or blogs to address different users
- Subscribe CTAs on the overview page and on blog post pages
- Links to social media, potentially with the total number of shares (twitter, facebook, linkedIn, google) for social proof
- List of contributors or authors to check writer IDs.



Audience focused blogs (Dropbox blogs)



Ready to Begin Your Career As A Freelance Developer? 3 Tips For Getting Started

[Twitter](#) [LinkedIn 75](#) [Facebook 25](#)

JUNE 06, 2017 • BY ALYSSA MAZZINA • IN CODE FOR A LIVING, FREELANCING, JOB HUNT

There has never been a more promising and profitable time to be a developer. The Bureau of Labor Statistics projects employment of software developers will grow an impressive 17% by 2024. Compare that with a 7% average growth rate across all occupations (and consider that the \$102,280 median wage among developers nearly triples the median for all occupations), and it's clear skilled coders are in increasingly high demand.

[Read more →](#)

[Code for a Living](#)

[Community](#)

[Culture](#)

[Diversity](#)

[Engineering](#)

[Code for a Living](#)

[Company](#)

CONTRIBUTORS



[Alyssa Mazzina](#)

Content Writer, Developer Marketing



[Juan M](#)

Community Manager, International Team Lead



[David Robinson](#)

Data Scientist



[Jess Pardue](#)

Operations Manager



[Anita Taylor](#)

List of contributors (StackOverflow blog)

Best practices and remarks

Along my research, I found a few tips and tricks that could help to attract and inspire users:

- Make the search function user-friendly:
 - Opt for topics and categories on top of the page (and not only at the bottom of the article).
 - Turn questions from your users into URLs via blog posts.
 - Choose maximum 15 topics or labels to define article categories.
 - Archive older posts, but make sure readers can still find them via tags or labels that indicate the article categories. No-one likes to struggle through
 - only - chronologically ordered articles.

Empowering a new generation of localization professionals

Wednesday, May 31, 2017

Posted by *The Google Localization Team*

When her grandmother turned 80, Christina Hayek – Arabic Language Manager at Google – and her sisters wanted to give their beloved sitto a gift that would bring her closer to them. Chadia lives in Lebanon but her children and grandchildren are spread across the world. To bridge this geographical gap, Christina and her siblings gave their grandmother an Android smartphone. Much to Chadia's surprise, she was able to use her phone in Arabic straight out of the box.

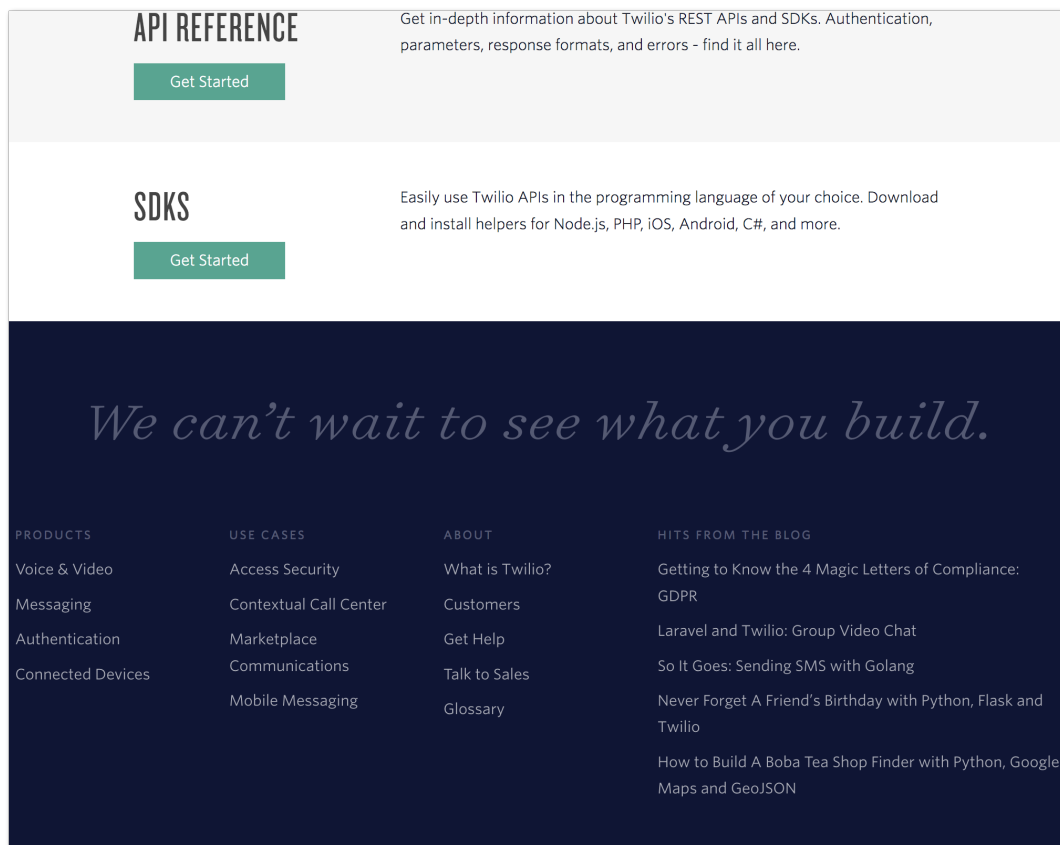
This isn't magic—it's the work of a dedicated localization team at Google. Spread over more than 30 countries, our team makes sure that all Google products are fun and easy to use in more than 70 languages. Localization goes beyond translation. While references to baseball and donuts work well in the US, these are not necessarily popular concepts in other cultures. Therefore we change these, for example, to football in Italy and croissant in France. Our mission is to create a diverse user experience that fits every language and every culture. We do this through a network of passionate translators and reviewers who localize Google products to make sure they sound

Labels

[#freeandopen](#)
[#GooglePlay](#) [#AndroidDevStory](#)
[#PlayStore](#) [#DeveloperConsole](#)
[#StoreListingExperiments](#)
[#io12](#)
[#io13](#)
[#io14](#)
[#io15](#)
[#io16](#)
[#io17](#)
[#io2012](#)
[#io2013](#)
[#io2014](#)
[+1](#)
[20% project](#)
[3d](#)

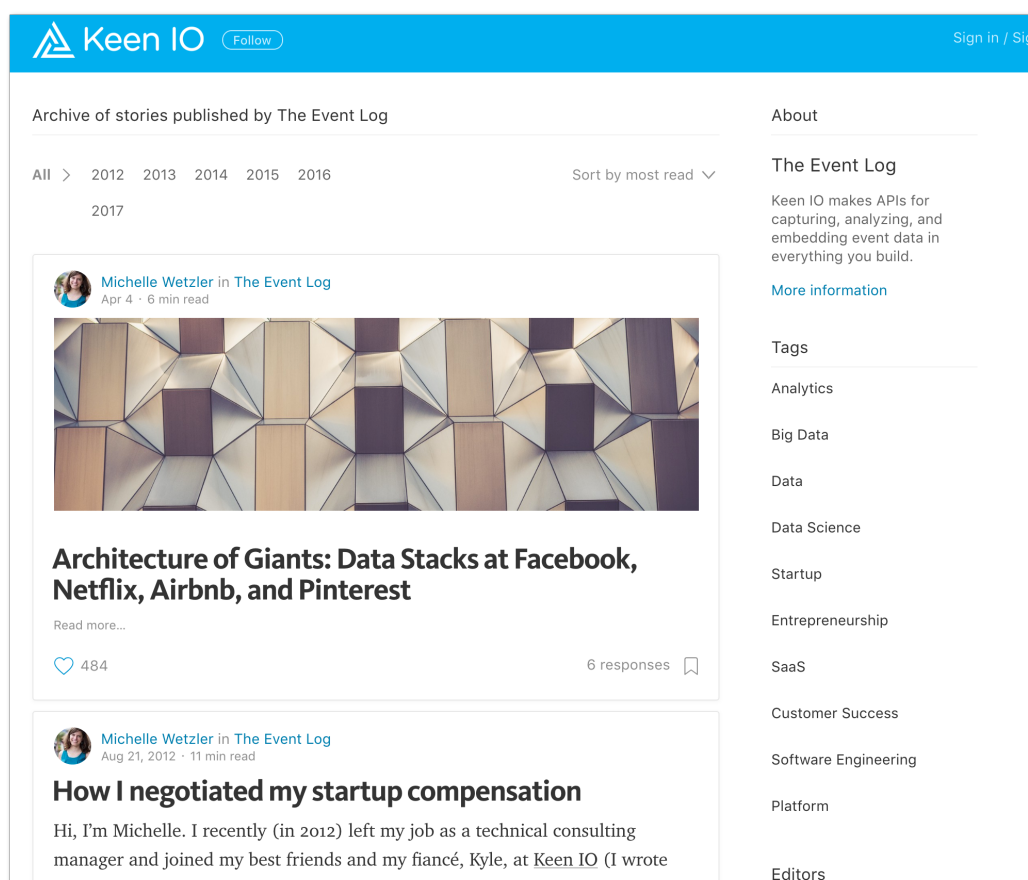
Google has too many labels to choose from: +50 for the letter “A” alone (Google blog)

- Provide CTAs at the end of each post to make sure your users can easily subscribe.
- Remove outdated blogs.
- List contributors and plan what you will include in the author biographies.
- Focus on different audiences through topics or via separate blogs.
- Provide links to other documentation types (like API references, support pages) on your blog to facilitate onboarding and to make it easier to replicate a demo. And vice-versa: give developer portal visitors the opportunity to check blog articles easily.



Developer portal footer: hits from the blog (Twilio)

- Indicate how much time the reader will have to spend on your article.



Indication of reading time (Keen IO blog)

Considering a blog as part of your communication strategy? Questions to answer

A blog is a great tool to help you develop new types of content on your developer portal. If you want to iteratively develop your content, and build out your content architecture as your community grows, it is a great place to experiment with delivery formats and documentation types. But before you start you need to ask if your company is ready to maintain one?

- Have you got a writer (team) or guest bloggers, a designer, a content strategist at your disposal?
- How regularly can you produce new content?
- Do you want to host your blog on your developer portal (and keep users on your site to find answers) or go for alternative platforms, like Medium, with custom design options, where you get an inbuilt audience and some distance from your brand to allow for experimentation?

We are working on a series of content services for developer portals, want to start a blog but need some help? - [Get in touch!](#)

The Function of API Use Cases & Case Studies on Developer Portals

<https://pronovix.com/blog/function-api-use-cases-case-studies-developer-portals>

Kathleen De Roo



Use cases and case studies are in the **grey area between developer and sales documentation**. Some would argue that they are not “real” documentation, and that they should not be included on a developer portal. However, as web APIs have become mainstream and strategic so did their documentation portals: the content must also address the less technical objectives of the non-developer stakeholders.

Use cases and case studies play **two crucial roles on a developer portal**:

- They act as **social proof** for your API product (sales function),
- They can be an introduction to **more specific, implementation scenario documentation types**.

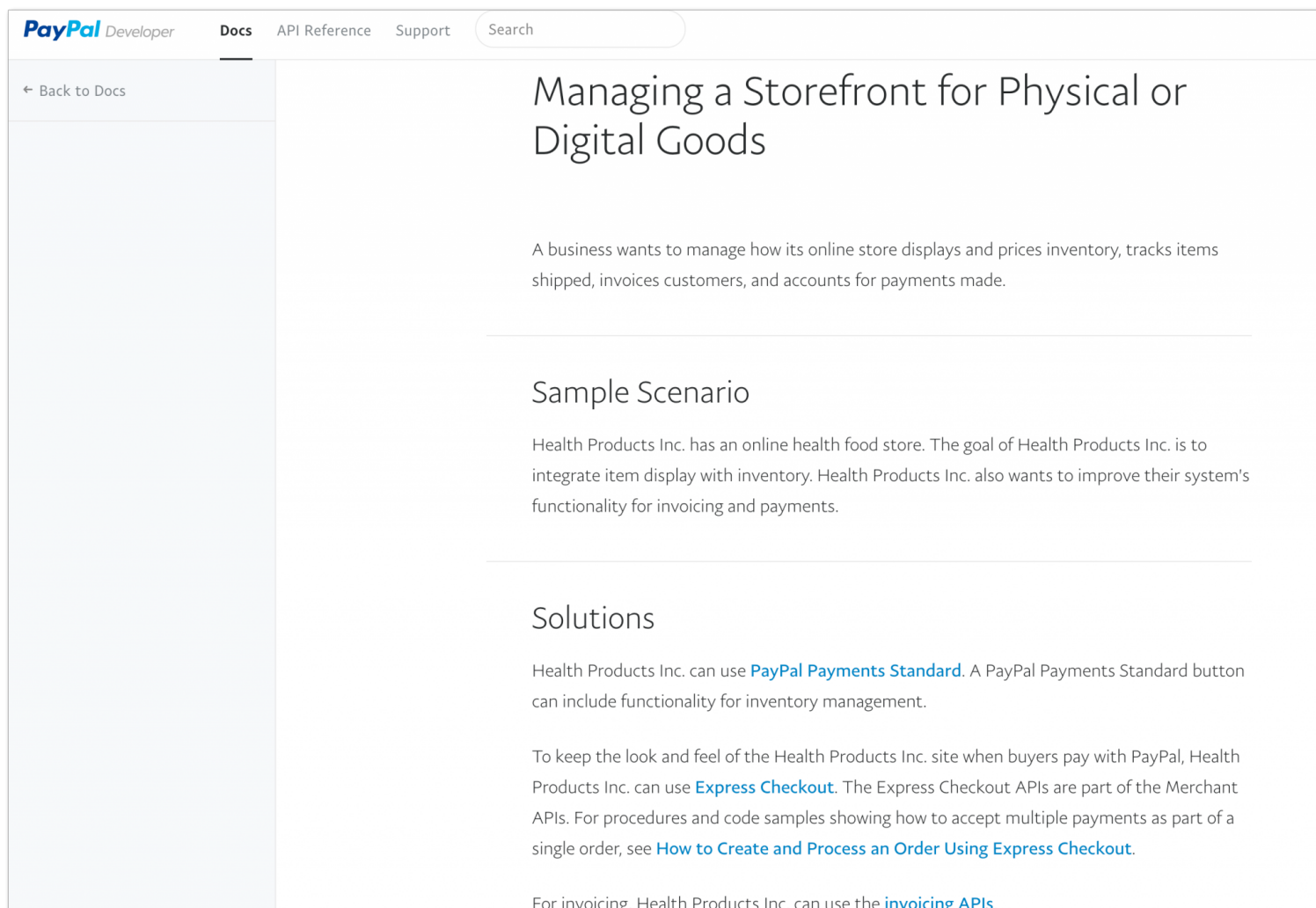
As with other types of documentation, there is not always a clear distinction between use cases and case studies: labels (taxonomy), structure (layout) and content depth and tone diverge depending on the team that is responsible for their implementation.

I selected 18 companies for this research; in this post I'll give an overview of how they displayed use cases and case studies: How do these documentation types help companies to tell their audiences a story? How do they guide users towards specific content? I'll talk about benefits and best practices and include some guidelines to set up a strategy to decide if use cases and case studies might be a good option for your own site.

What is a use case and what is a case study? And what about customer stories?

Use cases

[Use cases](#) describe actions or steps, defining interactions between the user (persona) and the system to achieve a specific goal or result. Use cases tell users how to obtain that end result and often address a technical audience that wants to evaluate and understand a specific problem or solution.



PayPal's use case subpages list goals, include a sample scenario and a section with solutions, links to the docs pages and a Call-To-Action

Case studies

Case studies present a problem-solving process from the perspective of the product/service provider and customer company. They are less about showing how something actually works, and are indispensable when you want to introduce a solution to a broad audience.

Dropbox Business Pricing Compare plans Features

THE CHALLENGE

Simplifying workflows during expansion

BNIM designs buildings that do more than just look great, winning national awards for sustainability and energy efficiency. This success has led the company to expand beyond its Kansas City headquarters, launching seven new offices across the US. But when it came time for employees in different locations to work together, existing processes clearly were hampering growth. Each office had a separate network drive, making it difficult for colleagues and consultants in different cities to collaborate, especially when handling complex files like 3-D Autodesk Revit models. Proposal deadlines snuck up earlier than expected, as architects needed to upload files onto the server the night before the due date, says Erin Gehle, Director of Communications. "We had to build in large margins just for communication," Gehle says. When outside the office, architects had to wrestle with VPN and project management software, a struggle that repeated itself every time anyone made an edit. "Our existing tools didn't perform when we shared large Revit models," Paul Waters, Director of IT says. "Some other mechanism had to be used."

To get around all these issues, employees started using Dropbox personal accounts under their BNIM email addresses. After evaluating a number of enterprise cloud providers, there was little surprise that Dropbox Business emerged as the clear winner. "From a device agnostic viewpoint, Dropbox Business was the easiest to operate," Waters says.

"We used to have so many steps for sharing and collaborating on files. Dropbox Business has eliminated those steps so our workflow is much more efficient."

— Paul Waters, Director of IT, BNIM

THE SOLUTION

A new blueprint for collaboration

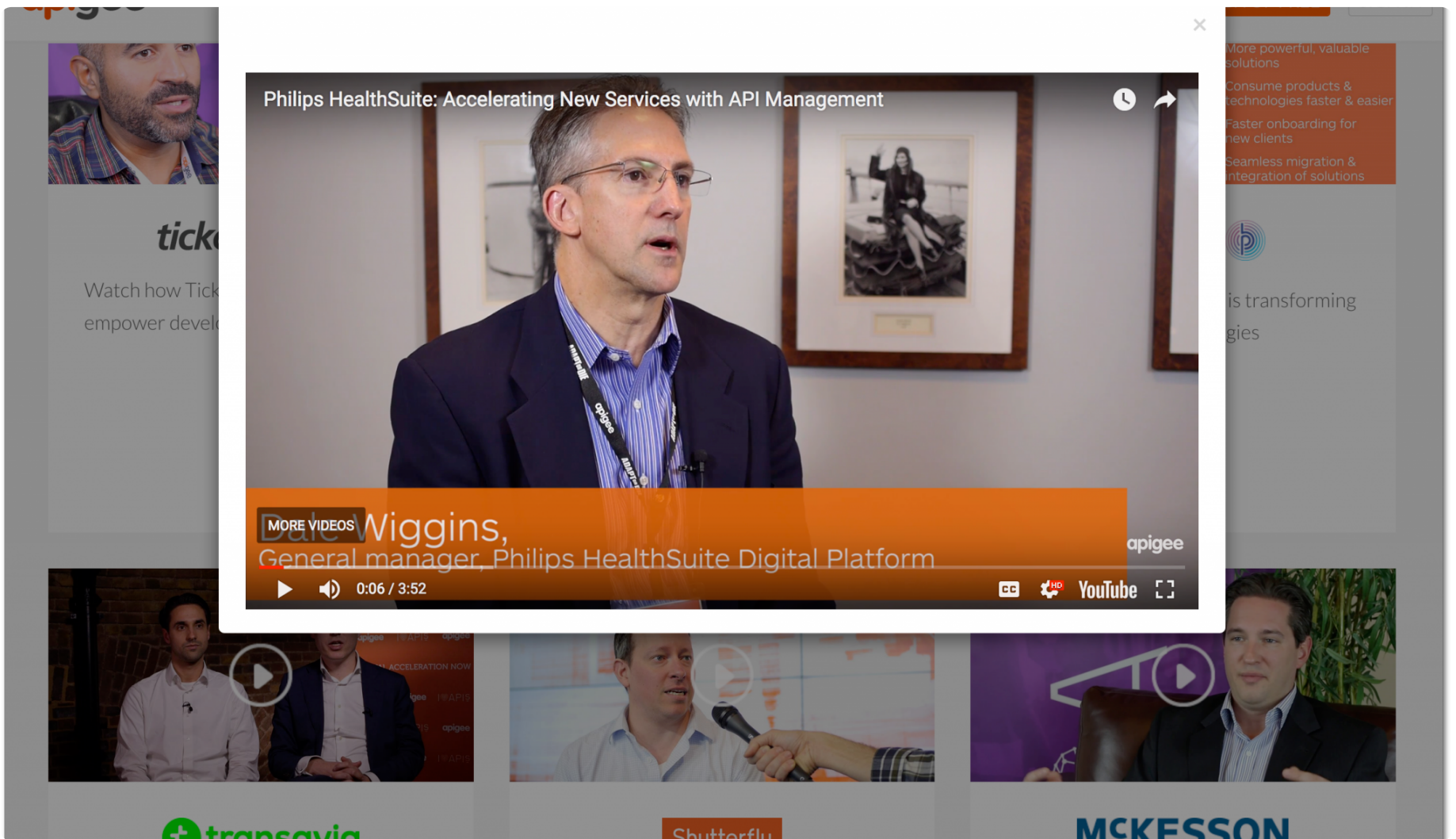
Example of a Dropbox case study that focusses on the experiences of the customer company. Customer quotes accompany a description of the project's challenge, solution

Customer stories

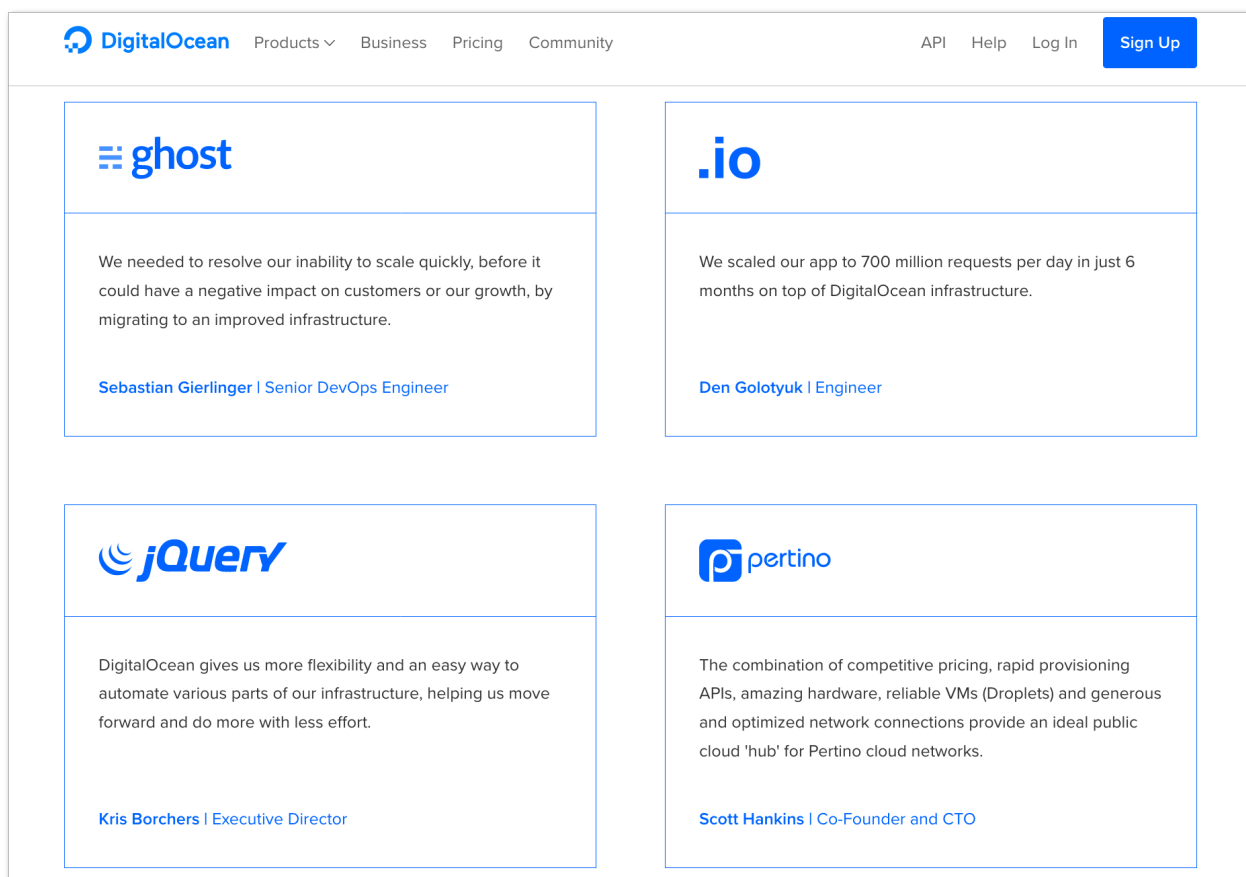
Customer stories advocate and/or review a product or service through the words of an actual user or representative of the customer company.

The companies often listed customer reviews into customer stories or customers. I found 3 possible formats. Customer stories can refer to:

- customer testimonials (text and/or video interviews),
- use cases/case studies that are written from the customer company's perspective (spiced up with customer quotes),
- articles written by authors from the client company.



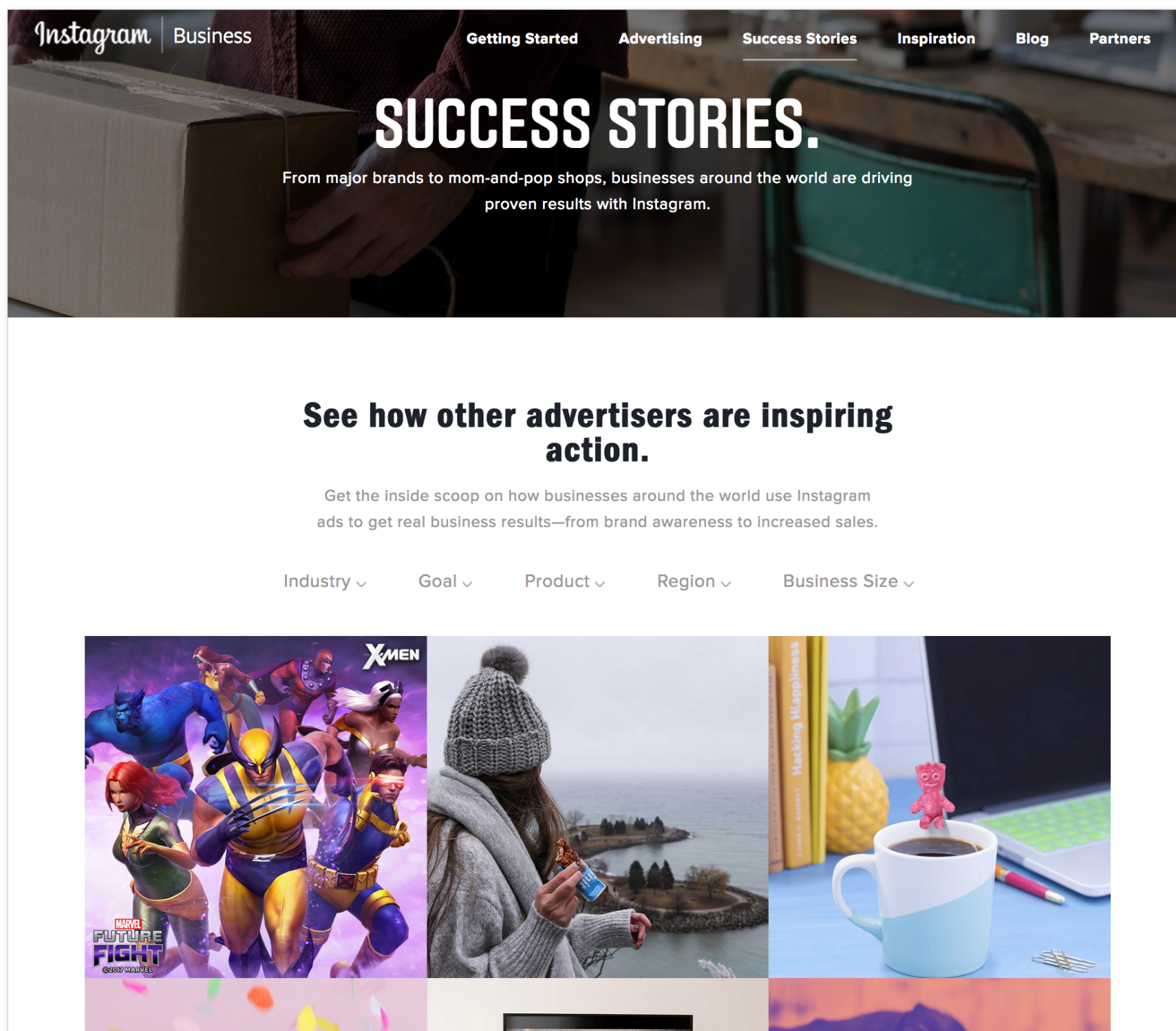
The label Customers leads to customer testimonials (e.g. via video interviews) and case studies (Apigee)



Customers: guest authors provide use cases and case studies. The content varies: depending on the position of the writer the reader gets more technical details (DigitalOcean)

Alternative labeling

Companies sometimes prefer their own custom labels to illustrate product and service analysis. The reviewed sites used *success stories*, *showcase* and *customer success* mostly for case studies (as per definition above).



The screenshot shows the Instagram Business 'Success Stories' page. At the top, there is a navigation bar with the Instagram logo, 'Business', and links for 'Getting Started', 'Advertising', 'Success Stories' (which is underlined), 'Inspiration', 'Blog', and 'Partners'. The main heading is 'SUCCESS STORIES.' in large white letters, with a sub-headline: 'From major brands to mom-and-pop shops, businesses around the world are driving proven results with Instagram.' Below this is a call to action: 'See how other advertisers are inspiring action.' followed by a descriptive sentence: 'Get the inside scoop on how businesses around the world use Instagram ads to get real business results—from brand awareness to increased sales.' There are five filter buttons: 'Industry', 'Goal', 'Product', 'Region', and 'Business Size', each with a dropdown arrow. Below the filters is a grid of three featured images: a Marvel Future Fight comic book cover, a person in a winter hat holding a small object, and a desk setup with a cup of coffee, a pineapple, and a laptop.

Instagram's Success stories list case studies of customer companies. The subpages have a story - quotes - goals - solution outline.

Display of use cases and case studies

Primary focus: customer, problem, product, solution?

Several articles give general guidelines on how to display use cases and case studies (e.g. [Wikipedia](#) has an extensive list of what you can include into use cases).

In practice, though, the companies in my sample lot often deviated from those theoretic blueprints and established their own set of rules.

The primary focus is usually similar. Most of the companies that listed “case studies” were customer oriented. “Use cases” mostly focused on product/solution (3), product/solution/customer (2) and solutions (2).

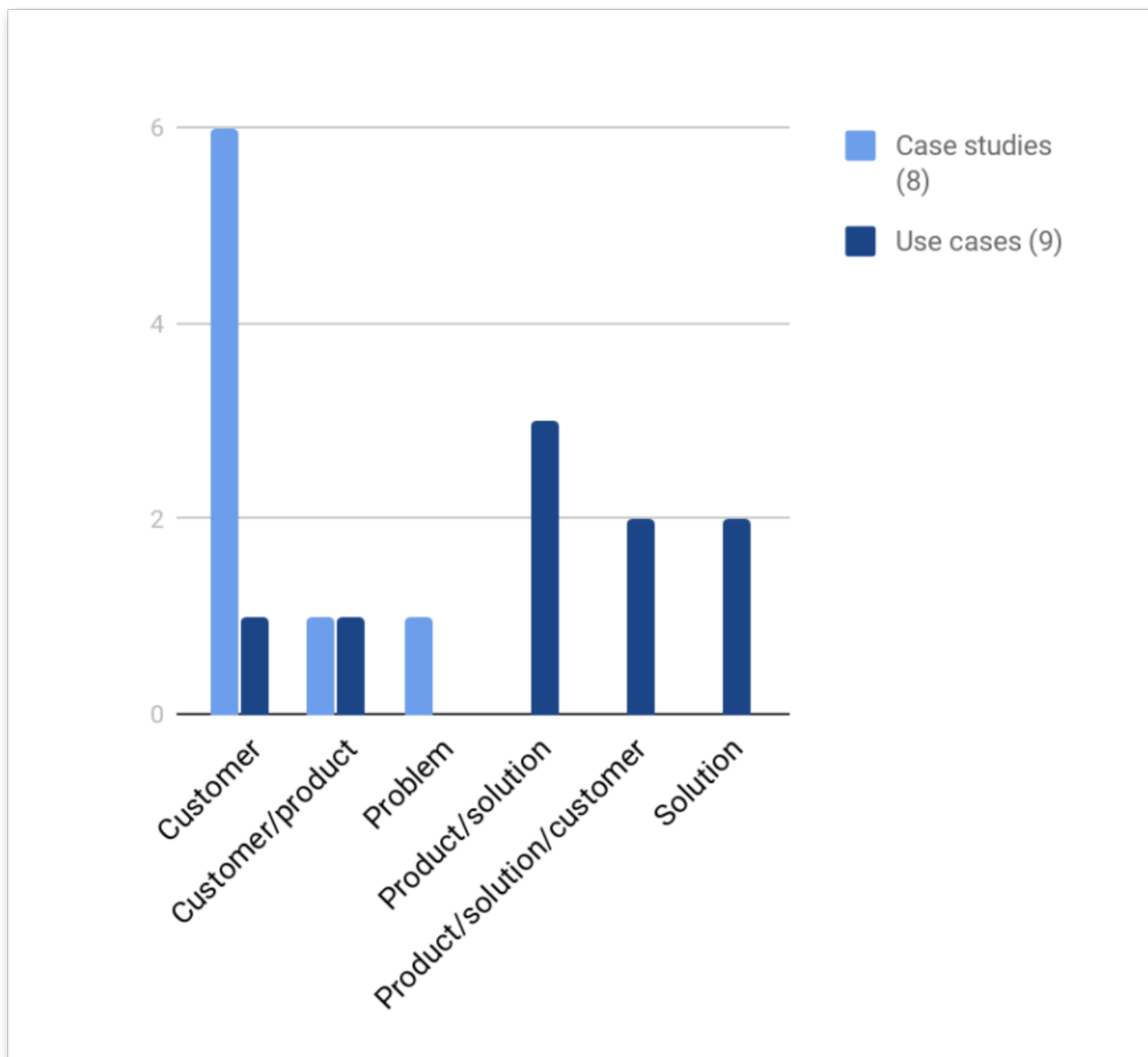
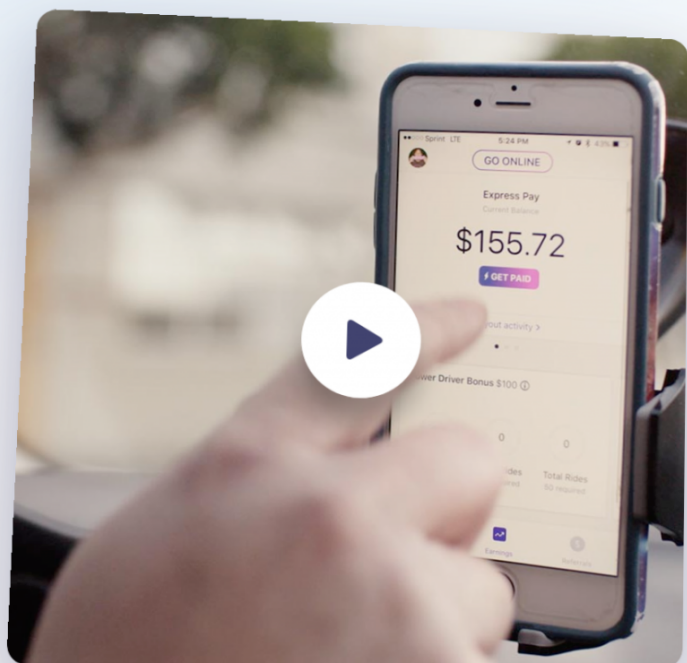


Chart: What do use cases and case studies primarily focus on? (based on the data provided by 18 companies)

Platforms across all industries are building on Connect

Whether you're building a platform to connect parents with babysitters or travel agents with tourists, Connect works for your business.



ON DEMAND SERVICES

RIDER



DRIVER

- Manage payments to drivers, couriers, plumbers or other service providers
- Attract more service providers through fast, high-conversion onboarding and the best payment options
- Outsource compliance tasks to Stripe
- Easily generate and distribute 1099-Ks and 1099-MISCs for IRS tax-reporting requirements
- Set payout timing, branding, and reporting for your service providers



Stripe use cases focus on product, benefits, solutions and customers.

We can make a light distinction between:

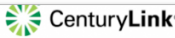
- Customer and problem oriented scenarios (How did we solve a problem or provide a service to our client?)
- Product and solution oriented scenarios (What can you achieve with our API product?)

Most companies use a variation of two structures to display information:


- 1 Customer and problem oriented scenarios could have the following layout:

Problem / situation	Customer business bio
Challenges / solutions	Guest author or interviewee, position
Results	Quote
Quotes	

Some examples:



Cloud Application Manager Customer Success Stories: DeNA



DeNA Speeds Provisioning 30x Faster and Gives Developers Freedom with Operational Control

Challenge

In each DeNA location like San Francisco or Santiago, game development is organized into multiple teams. Each one is independent, self-contained with game designers, back-end engineers, and front-end engineers. Each game needs a full-stack environment. For DeNA's operations team, the question is always how to get environments quickly to the game teams so they can chart their creative goals and change as they proceed.


Scaling game development

In the world of game development where things change quickly, launch schedules are still the priority. Projects have several checkpoints before release. Games run separately as different entities where teams want to choose their own hardware and environment. DeNA values game developer creative freedom. That means although the architectural team reviews all the production game environment stacks, the operations team doesn't force development teams to follow these configurations strictly. DeNA's IT operations team in San Francisco is small. Given the volume of game titles they push, providing game environment stacks custom fitted for every project checkpoint just doesn't scale.

Innovation without compromising governance

Also, the game teams include many junior developers who aren't operations-savvy. DeNA wanted a more efficient way of giving deployment freedom in the cloud without that going completely uncontrolled and unsupervised.

A better provisioning solution



Website
<http://dena.com>

Industry
Gaming/Entertainment

About DeNA
DeNA (pronounced "D-N-A") develops social games for iPhone and Android devices. Founded in 1999 by Tomoko Namba to delight and impact the world with new technologies, DeNA spans a global presence with offices from Tokyo to Santiago to San Francisco. Its latest title, Final Fantasy: Record Keeper, hit the 1 million download mark. DeNA also made news for its partnership with Nintendo to produce social games. To build dynamic, social games, DeNA relies on a vast team of talented engineers and

Case studies: challenge, solution, results on the left, the customer company info on the right (CenturyLink)



MAHMOUD ARRAM
CO-FOUNDER AND CTO

Bluecore Delivers Data-Driven Marketing Automation With Keen IO



Keen enables customer-facing analytics, dashboards, and visualizations in less than 24 hours.

INDUSTRY
MARKETING AUTOMATION

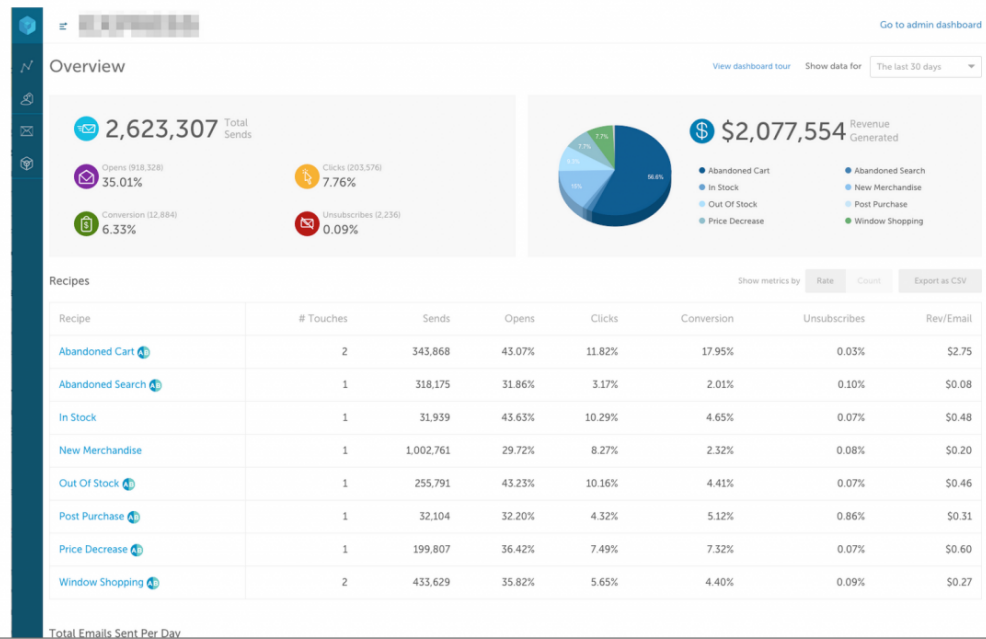
HEADQUARTERS
NEW YORK

REGIONS
GLOBAL

ENGINEERING TEAM
10

FUNDING
\$7.2 M

CUSTOMERS
120+, INCLUDING STAPLES, CABELAS, NEW EGG, AND EXPRESS



Case study on Keen IO's site: customer company info on the left; graph, mission and solutions on the right. CTAs at the bottom of the page.

2 Product and solution oriented scenarios could follow a layout like this:

Goals

Used features / benefits / solutions

Links to e.g. tutorials or API references

Timeline

CTAs to subscribe to newsletter or talk to sales

Icons of companies that used this product

Some examples:

DWOLLA | **Nomad** Case Study

A modern healthcare staffing platform leverages Dwolla's Access API to accept ACH payments from medical facilities and payout directly to clinicians' bank accounts.

Synopsis

Problem: A new staffing platform needed an ACH-optimized API that was both well-documented and flexible. The ability to easily onboard both enterprise clients and doctors within Nomad Health's own platform was critical to launch, as well as the capacity to send timely payouts.

Results: With only one developer available, the staffing platform integrated Dwolla's Access API to facilitate payments from the institutions and out to the clinicians within 2 weeks. Same Day ACH functionality allowed the platform to send payouts directly to doctors' bank accounts within the same day—cutting traditional industry payout times by 3 days.

According to [Staffing Industry Analysts](#), the temporary healthcare staffing industry is projected to reach \$15 billion in revenue in 2017. [Nomad Health](#) is the first known marketplace that enables clinicians and the institutions that hire them to find each other and transact directly, without a broker. Nomad believes it can reduce healthcare staffing costs by 40%.

The Situation

Doctors use Nomad Health to find freelance clinician work, while medical facilities use Nomad Health to find temporary clinicians—

“ We found Dwolla to have one of the best ACH-optimized APIs on the market. When it comes

Case study in PDF format with synopsis - situation - challenges - how Dwolla helped - features used - timeline structure (Dwolla)

Cloud IVR Overview How to build DOCS ? HELP LOG IN + SIGN UP MENU & PRODUCTS

Cloud IVR
Build your own Interactive Voice Response (IVR) system that precisely routes each caller.

[Talk to Sales](#)

CHANGE ON DEMAND
Business needs change all the time—your IVR should keep up. Building with APIs means you can quickly iterate using the web language of your choice.

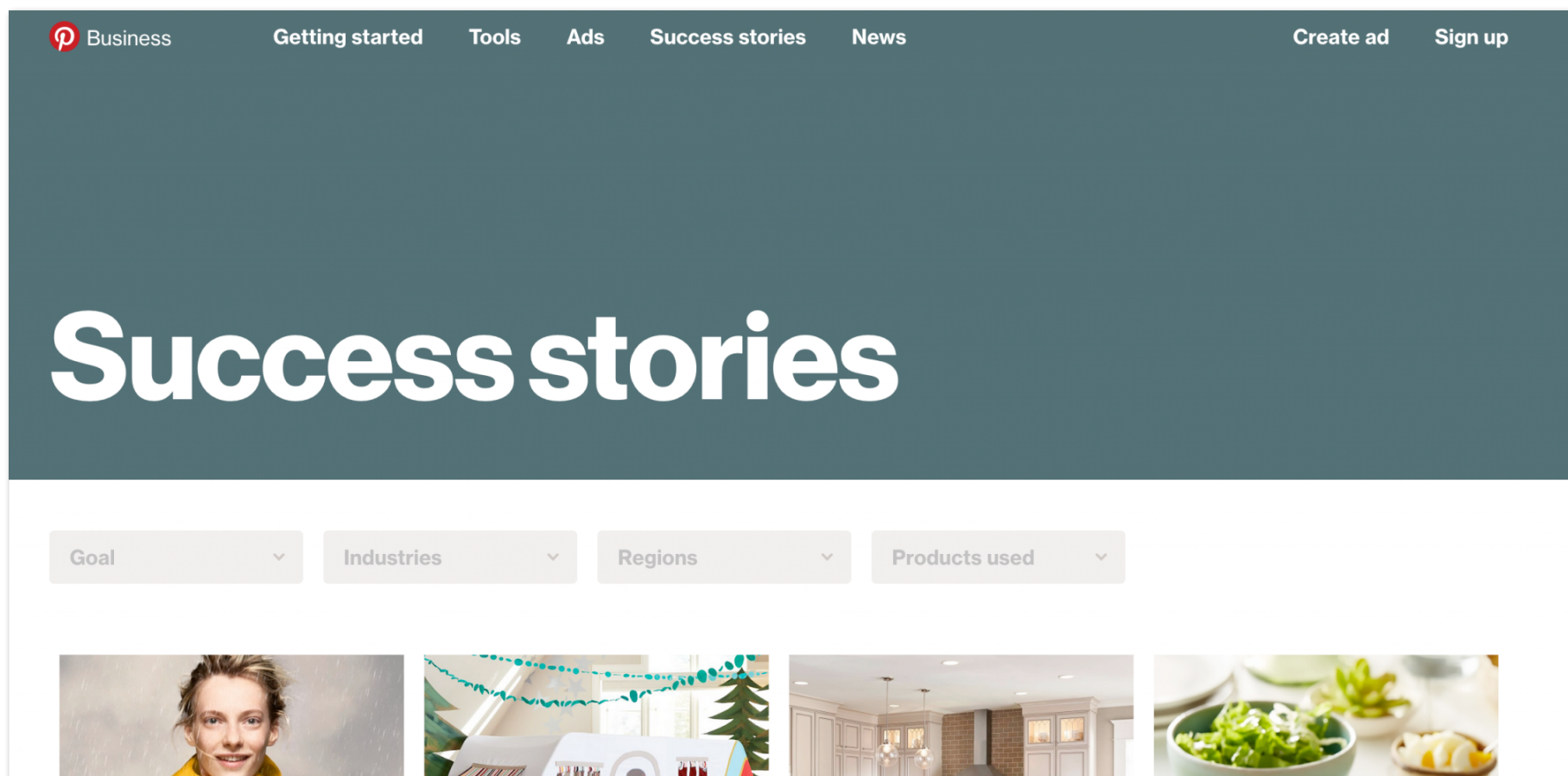
DATA-DRIVEN INTERACTIONS
Generate predictive IVR menus based on caller history and journey data from browsers or mobile apps.

PAY AS YOU GO
Pay for exactly what you use, and nothing more. Prices are based on call minutes—no upfront costs or licensing fees.

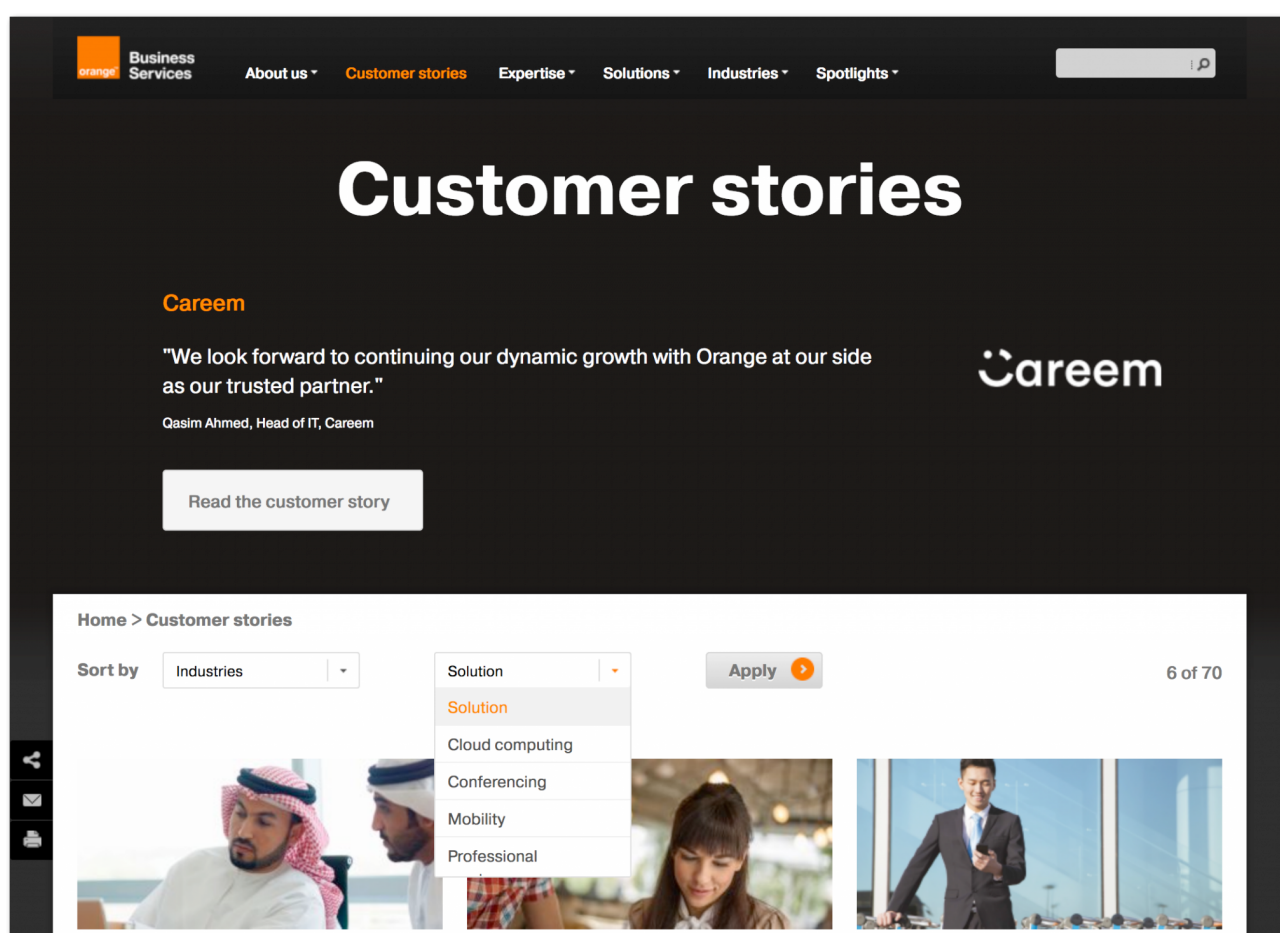
An example of a Twilio use case: Goal, benefits, CTA to contact the sales team, links to a “build it” tutorial, fact checks about companies that used this product.

Search easily: select from categories

Some sites included filtering options like companies, types of industry, topics, featured cases, regions, product, goals, solutions.



Filtering options: Goal, industry, region and products used all link to case studies of customer companies (Pinterest)

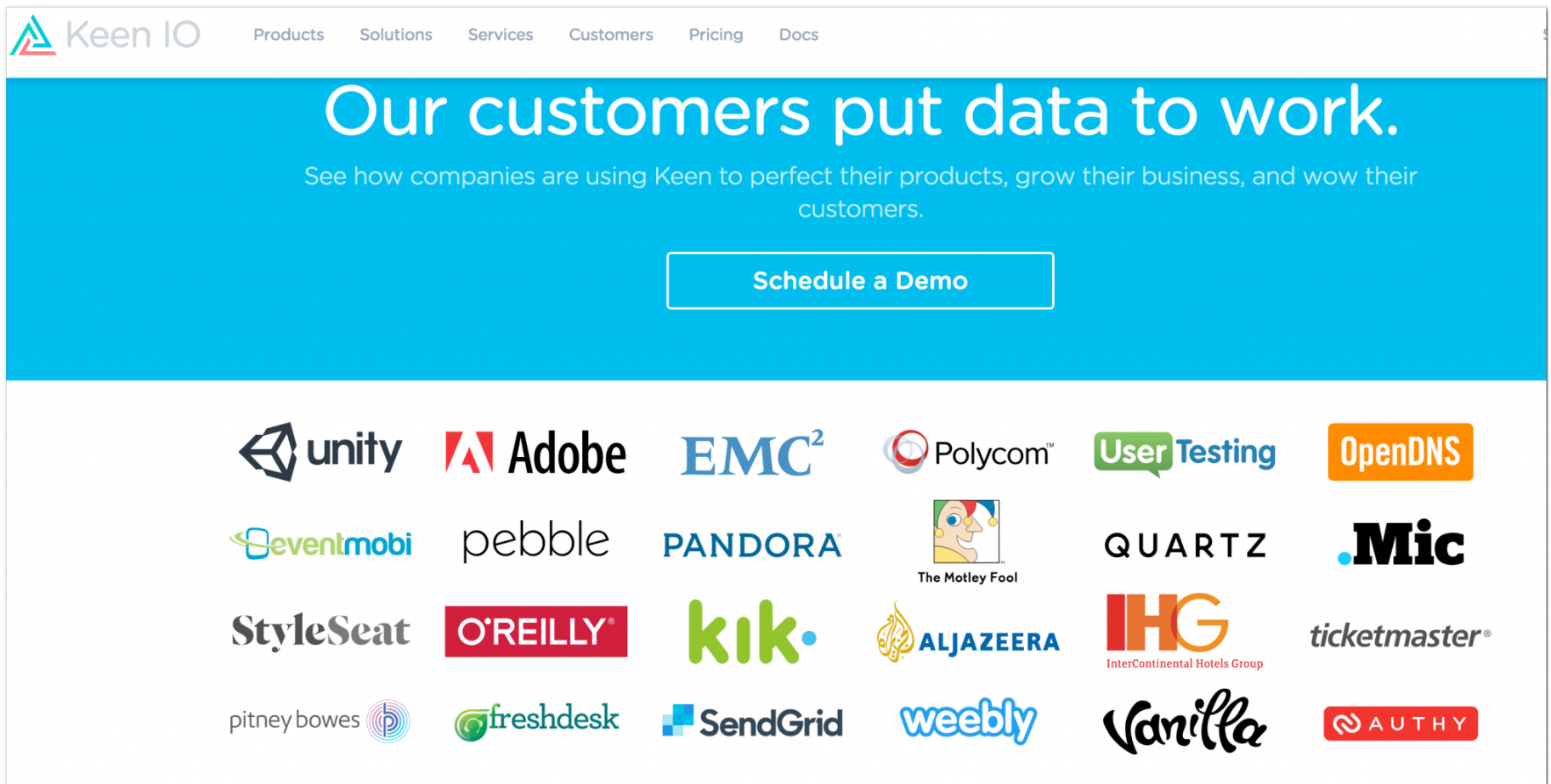


Filtering options:
Industries, solutions
link to case studies of
customer companies
(Orange)

Role in user journeys

Case studies and use cases are API documentation types that can play a role in the user journeys of both decision makers and developers.

First, they can help new site visitors evaluate what the company's API product is about by listing intriguing facts, showcasing features, sharing customer opinions, or celebrating interesting implementations.



Customers: interviews with customers, case studies with quotes, videos and articles by guest authors (Keen IO)

Second, they can help speed up the stakeholders' user journeys in general, e.g. provide specific product details (like code snippets) or audience-focused links that lead to a content niche on site. Where these case studies are placed in the site's architecture (business site, developer portal, blog) reveals which user-objectives they primarily target.

Web Fundamentals Tools Updates **Case Studies** Search ALL PRODUCTS

FEATURED BY VERTICAL BY REGION **ALL**

2017
 All Articles
 August
 Dance Tonite in WebVR
 May
 April
 March
 February
 2016
 2015
 Tags

Some devices can handle anything you throw at them and others are more constricted. We chose to implement a sliding scale. By continually measuring the amount of frames per second, we can adjust the distance of our fog accordingly. As long as our frame rate is running smoothly, we try taking on more render work by pushing out the fog. If the framerate isn't running smooth enough, we bring the fog closer allowing us to skip rendering performances in the darkness.

```

// this is called every frame
// the FPS calculation is based on stats.js by @mrdoob
tick: (interval = 3000) => {
  frames++;
  const time = (performance || Date).now();
  if (prevTime == null) prevTime = time;
  if (time > prevTime + interval) {
    fps = Math.round((frames * 1000) / (time - prevTime));
    frames = 0;
    prevTime = time;
    const lastCullDistance = settings.cullDistance;

    // if the fps is lower than 52 reduce the cull distance
    if (fps <= 52) {
      settings.cullDistance = Math.max(
        settings.minCullDistance,
        settings.cullDistance - settings.roomDepth
      );
    }
    // if the FPS is higher than 56, increase the cull distance
    else if (fps > 56) {
      settings.cullDistance = Math.min(
        settings.maxCullDistance,
        settings.cullDistance + settings.roomDepth
      );
    }
  }

  // gradually increase the cull distance to the new setting
  cullDistance = cullDistance * 0.95 + settings.cullDistance * 0.05;

  // mask the edge of the cull distance with fog
  viewer.fog.near = cullDistance - settings.roomDepth;
  viewer.fog.far = cullDistance;
}

```

Contents

The concept

1. DIY motion capture
2. Minimalism & costumes
3. Loop pedal for movement
4. Interconnected rooms

Optimizations for performance: don't drop frames

1. Instanced buffer geometry
2. Avoiding the garbage collector
3. Serializing motion & progressive playback
4. Interpolating movement
5. Syncing movements to music
6. Culling and fog

Something for everyone: building VR for the web

1. The yellow orb
2. Another point of view
3. Shadows: fake it 'till you make it
4. Being there
5. Sharing recordings
6. Solid ground: Google Cloud & Firebase
7. Service Workers


Conclusion

Code snippets and links to GitHub in a case study on the Google developer portal.

Twitter use cases on the developer portal focus on solutions first and then list benefits, products, (customer oriented) case studies, tutorials and links to more detailed developer documentation sections.

Developer Use cases Products Docs More

Display Tweets






Engage your users with live content from Twitter.

When something happens in the world, it happens on Twitter. Our display tools for web, iOS and Android allow you to bring Twitter's live content into your product, direct from the source. Use our tools to embed Tweets in your stories and articles. Configure timelines to automatically display live updates from trends, people, and places right in your app.

Relevant products

- Twitter for Websites [Explore tools >](#)
- Twitter Kit SDK for iOS and Android [Explore SDK >](#)
- oEmbed API [Explore endpoints >](#)

Relevant case studies

Relevant tutorials

- [Telling great stories with Tweets >](#)
- [High performance web widgets >](#)

NOT A DEVELOPER?

Find the use case solution best for your business.



What is Cloud Communications?

Embed communications directly into software applications.



Twilio Use Cases

See how to integrate communications into your business.



Buy from a Partner

Partners offer communications solutions powered by Twilio.



Find a Developer

Browse Twilio experts to help you build the solution you need.

Twilio provides a Not a Developer? page. The listed use cases try to lead the visitor towards more and more specific and thus personalized descriptions, with journeys that could either end on the “Talk to Sales” page or on the developer documentation pages.

“With so many snack brands out there, we needed to find a way to show how delicious our bar is and get people to try it. Facebook Creative Shop helped us create unique, creative to catch peoples’ attention in mobile News Feed that successfully lifted brand awareness.”

ELENA PARLATORE, DIRECTOR OF DIGITAL MARKETING, INIT SNACKS

THE GOAL

Launching on mobile first

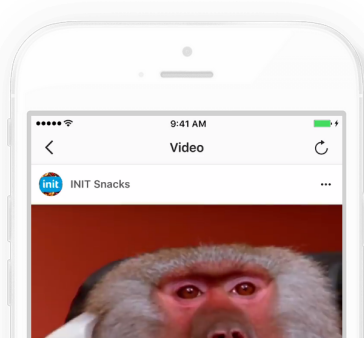
INIT Snacks wanted to raise awareness about its new line of snack bars by using videos with relevant creative that were optimized for viewing on mobile.

THE SOLUTION

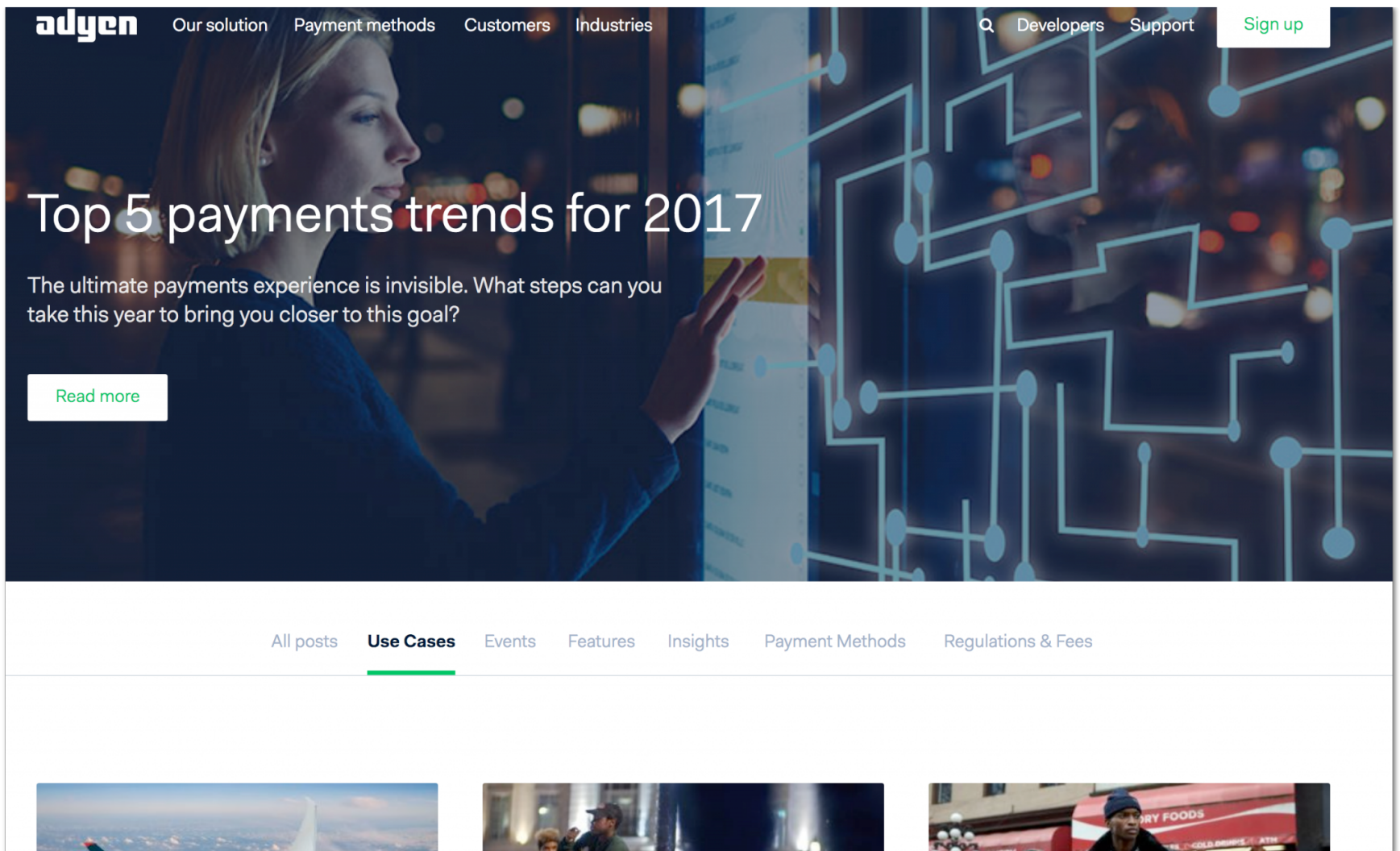
Showing what’s INIT

The INIT Snacks team partnered with Facebook Creative Shop to build a unique campaign that would raise awareness about its line of wholesome snack bars. The team also collaborated with communications agency OMD and digital marketing agency Resolution Media to plan and execute the campaign on both Instagram and Facebook.

The team developed 5 original videos to introduce each of the INIT Snacks bar flavors. The videos were designed specifically for a mobile audience and featured quick scene cuts, close-ups of the ingredients, text overlays and heavy branding throughout. The square-format videos also highlighted the bars’ delicious organic ingredients like nuts, dried fruit and dark chocolate to show “it’s



Primary focus on customer experience, secondary focus on “goal - solution” descriptions for decision makers (Instagram case studies on business site)



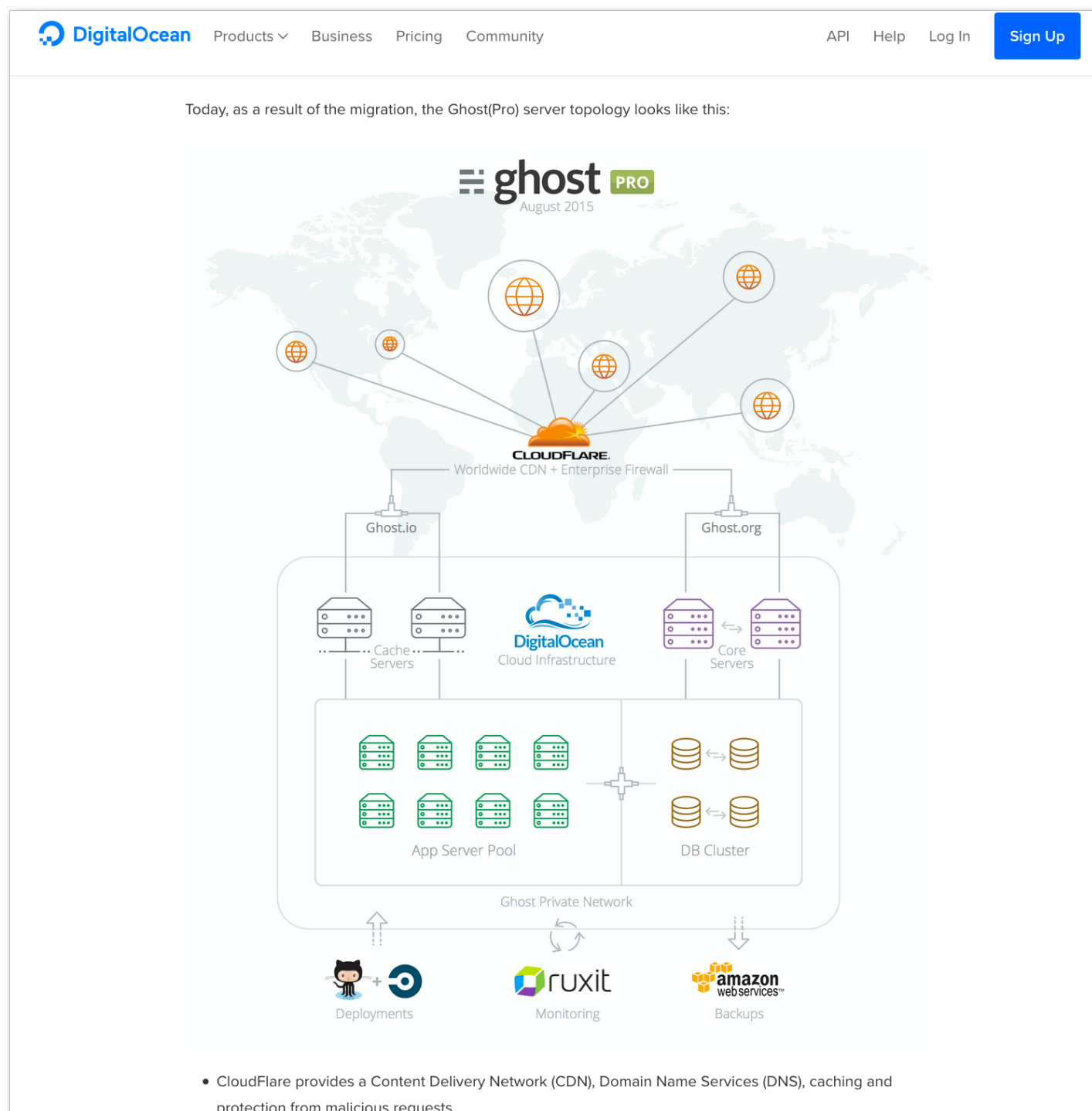
Blogs are informal, unstructured communication tools that can engage current and future customers: use cases as a content category on the Adyen blog.

Benefits

Use cases and case studies can help market your API:

- They offer a perfect opportunity to introduce product benefits and illustrate specific solutions. The balance between “[share knowledge, not features](#)” also defines to what extent you are likely to attract developers.
- They can feature the customers themselves and give, a unique insight into the whole process that goes with researching, evaluating, planning and implementing an API product.
- They can recommend and celebrate the usage of specific API products, this can attract new users.

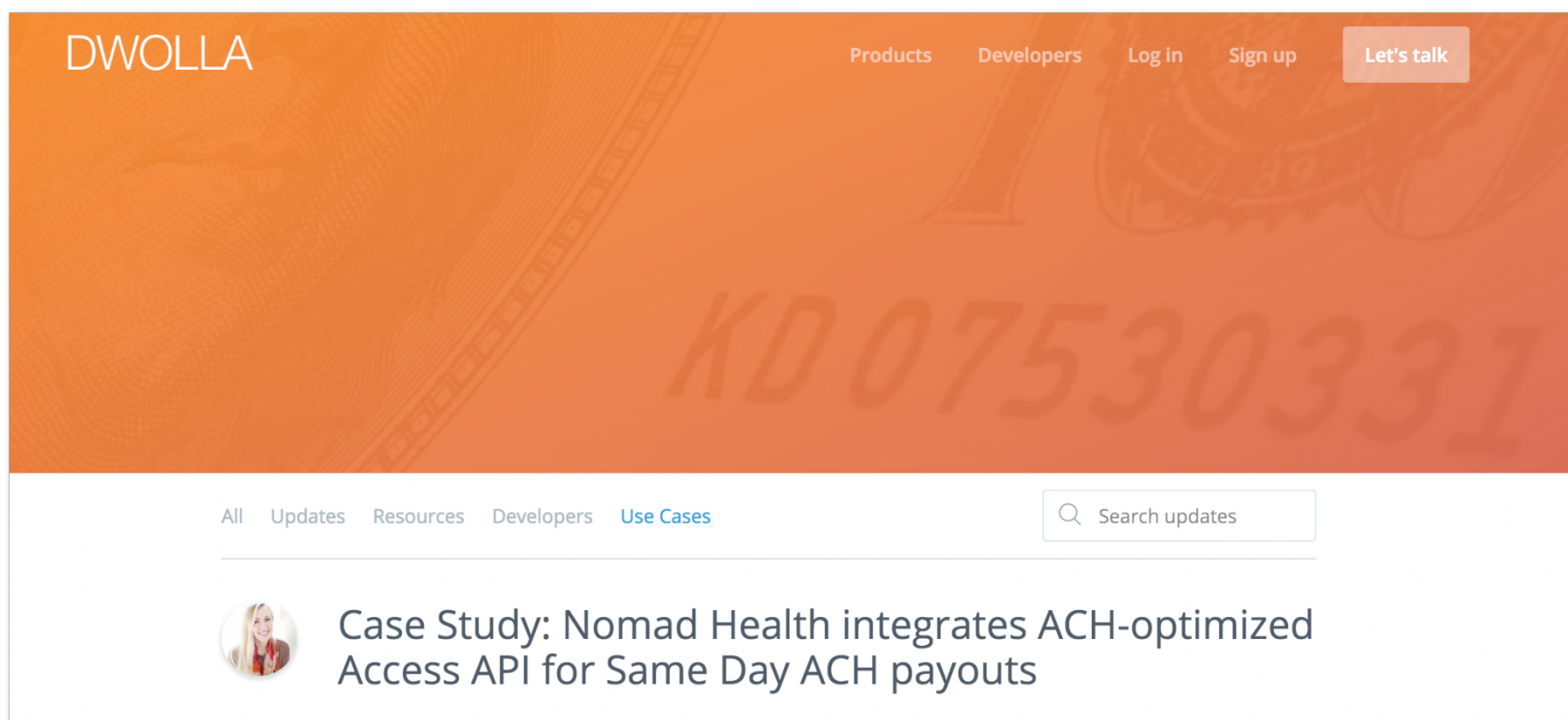
- They are easily accessible: case studies and use cases are usually written in everyday English. Their structure provides an ideal environment for quick scanning. Infographics and icons aid comprehension.
- They can shift the focus towards user specific content through subcomponents like step-by-step guidelines or descriptions, infographics, tables, code snippets, links to documentation pages, features.



Use case description with resources and guidelines (DigitalOcean)

Think ahead: combine your strategic decisions and industrial best practices

My research sample showed that use cases and case studies, but also customer stories are widely applied, but the companies that include them define their exact definitions, labels and use.



An example of overlap in word choice: use cases link to case studies on the Dwolla blog.

In order to be able to decide whether you would include use cases or case studies on your site to show expertise, I listed a few questions and added some best practices that can help you decide:

What role will the content play in the user journeys of your primary audience? How does that influence their place in the site architecture? What page structure, labels and subcomponents match your personas best?

Best practices: Focus on what your audience would expect to find: make the labels, their place in the site architecture and the specific subcomponents consistent, e.g.:

- Add use cases (label) with code snippets and descriptions with features (subcomponents) on the developer portal (place in the site architecture), among other API documentation types.
- Make customer stories (label) that advocate solutions (subcomponent: description with quotes) available on the business site (place in the site architecture).
- Add CTAs or in-links to related or more detailed information elsewhere on site. Include visual design elements, like infographics, videos and icons.

How can you make sure your users will be able to evaluate the provided content quickly?
How can you improve their user experience?

Best practices:

- Write plain English, provide a clear structure and an easily accessible format.
- Provide an overview page with filtering options.
- Add descriptions of how you define use cases and case studies at your company.

This research

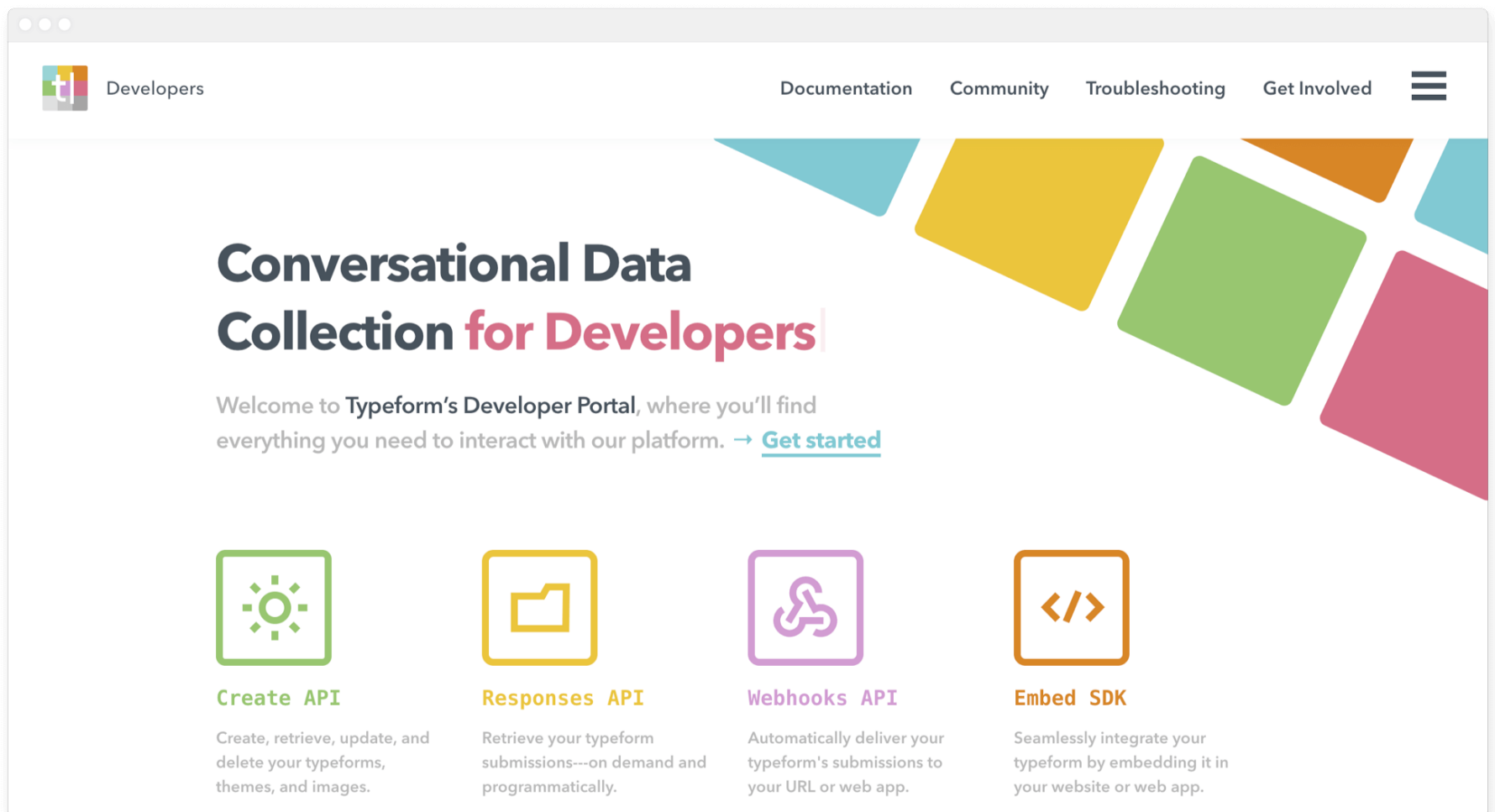
For this post, I derived data from the business and developer portal sites of 18 companies with different business profiles: Adyen ([Adyen case studies](#), [Adyen customers](#)), Amazon ([Amazon case studies](#), [Amazon use cases category](#)), Apigee ([Apigee API management use cases](#), [Apigee API management case studies](#), [Apigee customers](#)), CenturyLink ([CenturyLink case studies](#), [CenturyLink use cases category](#)), DigitalOcean ([DigitalOcean customers](#)), Dropbox ([Dropbox customers](#)), Dwolla ([Dwolla use cases](#)), Facebook ([Facebook success stories](#)), Google ([Google showcase](#)), Instagram ([Instagram success stories](#)), Keen IO ([Keen IO customers](#)), Mapbox ([Mapbox showcase](#)), Orange ([Orange customer stories](#)), PayPal (PayPal use cases), Pinterest ([Pinterest success stories](#)), Stripe ([Stripe use cases category](#), [Stripe customers](#)), Twilio ([Twilio use cases](#), [Twilio showcase use cases](#), [Twilio customers](#), [Twilio Not a Developer page](#)) and Twitter ([Twitter case studies](#), [Twitter use cases](#)).

Find out How Typeform is Building the Ultimate Developer Experience

<https://www.typeform.com/blog/inside-story/developer-platform/>

Jason Harmon





It's finally here. For months we've been working to release our new Developer Platform and Developer Portal. And we've learned a lot.

We'd love to share our experience along this journey, and tell you all about the technical goodies you should be excited about. But to spare you future headaches, let's first take a quick step back.

UX as a service

Here was our goal: build a developer community around our new Developer Platform. This includes the technology underlying our Application Programming Interfaces (APIs), Webhooks, and Embed Software Development Kit (SDK), along with our Developer Portal.

It started way back when we launched Typeform.io—a standalone beta, consisting of APIs to create and manage forms. People were building amazing apps with it, and developers were excited about “UX as a service.”

But after talking with lots of Typeform.io developers, we saw a clear problem: users wanted their forms to live in their Typeform.com account, and this just wasn't possible with the existing platform. (psst: if you're a Typeform.io developer, [please reach out](#) and we'll help you transition to our new APIs.)

And then there's [Typeform Version 2 \(V2\)](#), a brand new version of Typeform with a completely revamped UX that needed a total architecture overhaul. More on this soon.

So with .io and V2 in mind, we began a massive migration from one big PHP application to a series of Golang microservices. Translation: the architecture and APIs that now power our Dev Platform are the exact same as those under the hood of Typeform.com.

Basically, we're pretty confident you'll be stepping into a badass developer experience.

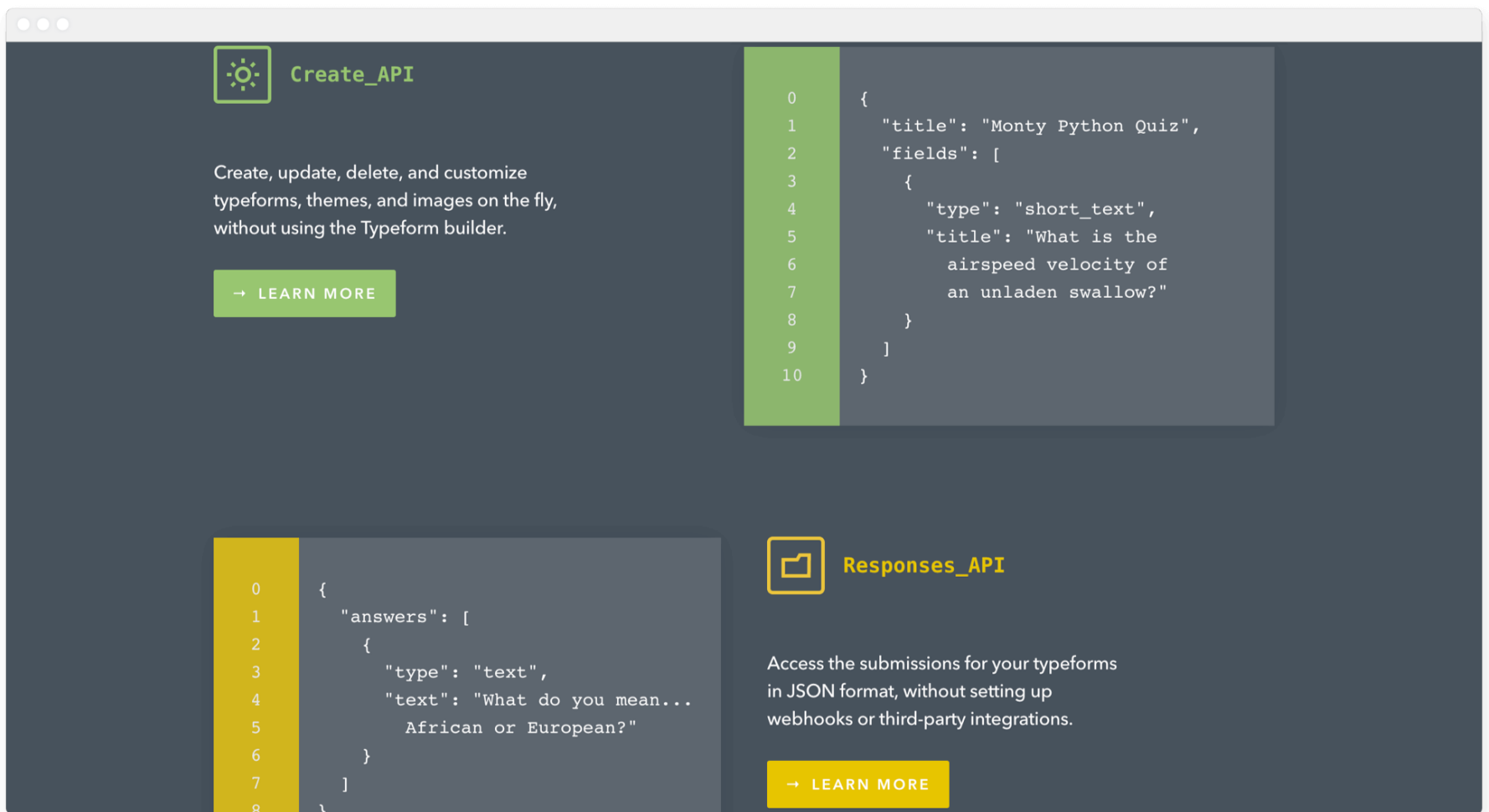
Building the ultimate developer experience

In the world of interface design, you often hear about UX—the user experience. In the developer world, we talk more about DX—you guessed it—the developer experience.

A lot of organizations think of DX as the UX of their developer portal. We went in with a different mindset. Here's why.

- 1 Technical users spend a lot of time using products like APIs with no obvious user interface. This makes it hard to improve the visual look of these products.

- 2 Interacting with technical products often happens without visiting a login page, exploring an account dashboard, or hitting an interface's button. This makes it hard to manage the user journey like you do with everyday apps.



It turns out that Typeform's Developer Platform has two major interfaces where design principles apply:

Technical documentation

We want developers to discover our new tools, imagine possibilities, and start using them right off the bat. But unlike everyday apps, reading documentation is a must when messing with technical products. So we made the structure, content, and readability of our documentation a key part of our strategy.

API usability

If people start tinkering with a shoddy product, they're not going to stick around. So we knew we had to nail the API experience. Even without reading the technical docs, a developer should have a good sense of what an API can do. This means that the domain language used in the URLs, request/response models, and any parameters had to be intuitive.

For this, two simple principles go a long way:

- Use your end-users' domain language
- Think more is less: simpler models are always easier to understand

We're pleased with how far we've come, but we know we still have work to do. That's why we're investigating things like SDKs in a variety of languages. Simple version: it'll make creating new projects using APIs or Webhooks a lot less intimidating.

Are you interested in SDKs? You can [get involved here](#).

1 → So, tell us... what do you have in mind?*

Please note that we are not fond of sales pitches

- A** I want to give useful feedback
- B** I built a project with the Dev Platform!
- C** I want to integrate you with my product
- D** I want to write an article for your blog
- E** I'm media and want to feature Typeform
- F** I organize events and want you as speaker
- G** I need help using the Developer Platform
- H** Something else

0% completed

Powered by Typeform

Don't build in a bubble

It's one of the easiest mistakes to make. You set out to build an amazing set of dev tools, and you start by creating a dedicated API team. Next thing you know, these select few are off building in a bubble.

We knew that to transform our vision into a platform people wanted, we had to draw on almost every part of our organization. This meant that virtually all our product development teams—with their 70+ team members—would be building microservices.

One initiative that makes this possible is [OpenAPI](#). It's the first stop for any team proposing API designs. We also use OpenAPI to share designs with collaborators. The result? Higher quality feedback sooner, from lots of non-technical stakeholders.

And then there's the other side of the office. By including our Marketing, Customer Success, and other non-technical teams, we were able to create a better experience for both developers and business-oriented users.

First lesson: collaboration. Now we've got the whole organization on board. So as we move forward into more platform-centric thinking, we'll have everyone up to speed and contributing in their areas of expertise.

Next lesson: understand your audience.

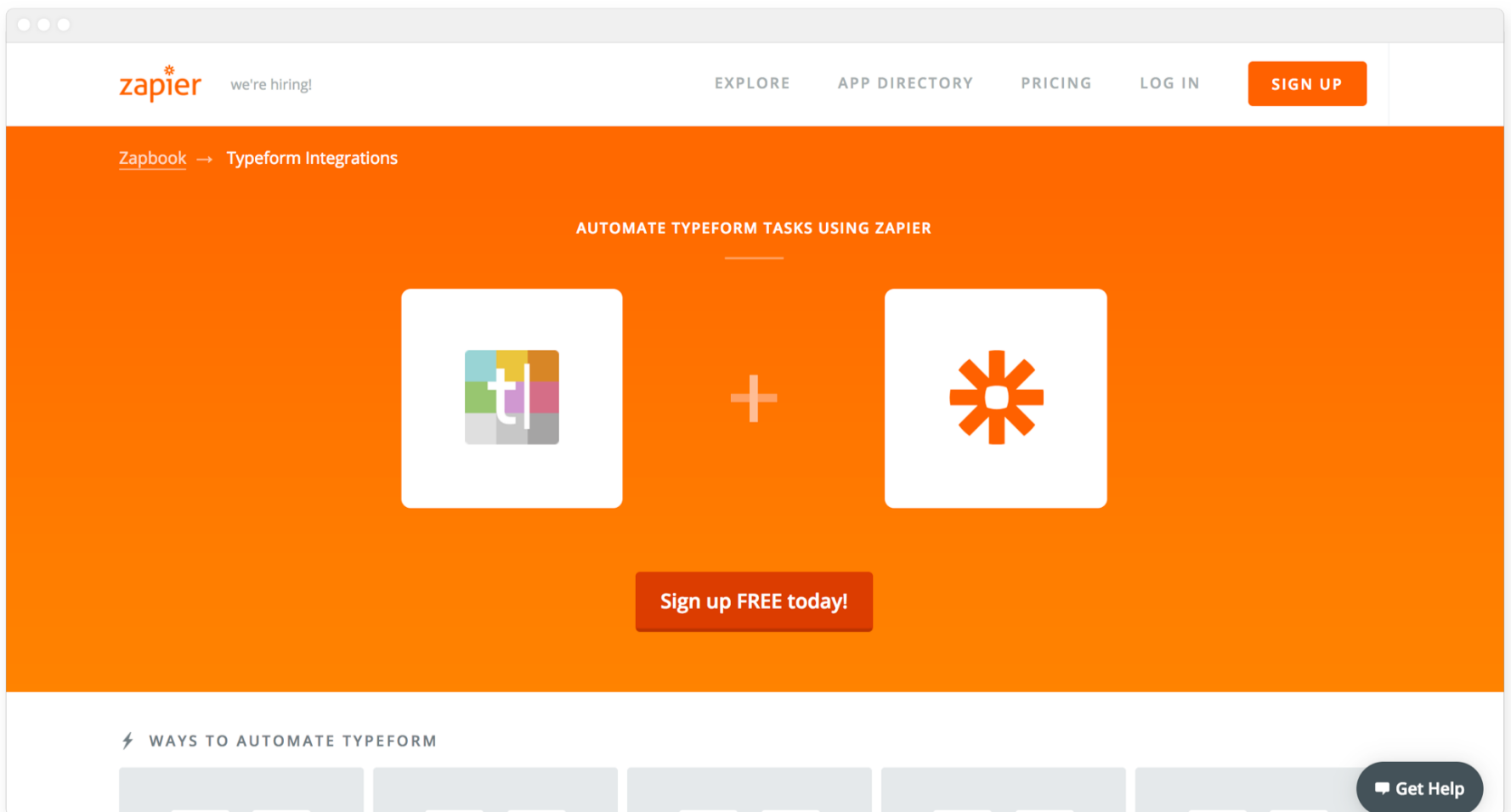
Developer tries, business buys

Creating a technical platform isn't just about communicating with a technical audience. This is what our [Developer Portal](#) is all about.

With our customers and partners, we often see that “developer tries, business buys.” Basically, developers are strong influencers, but they're not often the final decision makers.

Here's an example. Whenever a Typeform user exceeds around 100 responses, they start moving toward automation. This typically happens in two steps:

1. Use “point-and-click” integration options like Zapier to connect their typeform to their favorite apps or systems of record.



2. Build custom solutions for integrating with their proprietary backends when dedicated development resources become available.

To make this happen, business users have to validate whether a product has sufficient developer tooling/APIs to integrate into their product's infrastructure. And to do this, they need a quick way to review what's possible before bugging their engineering teams.

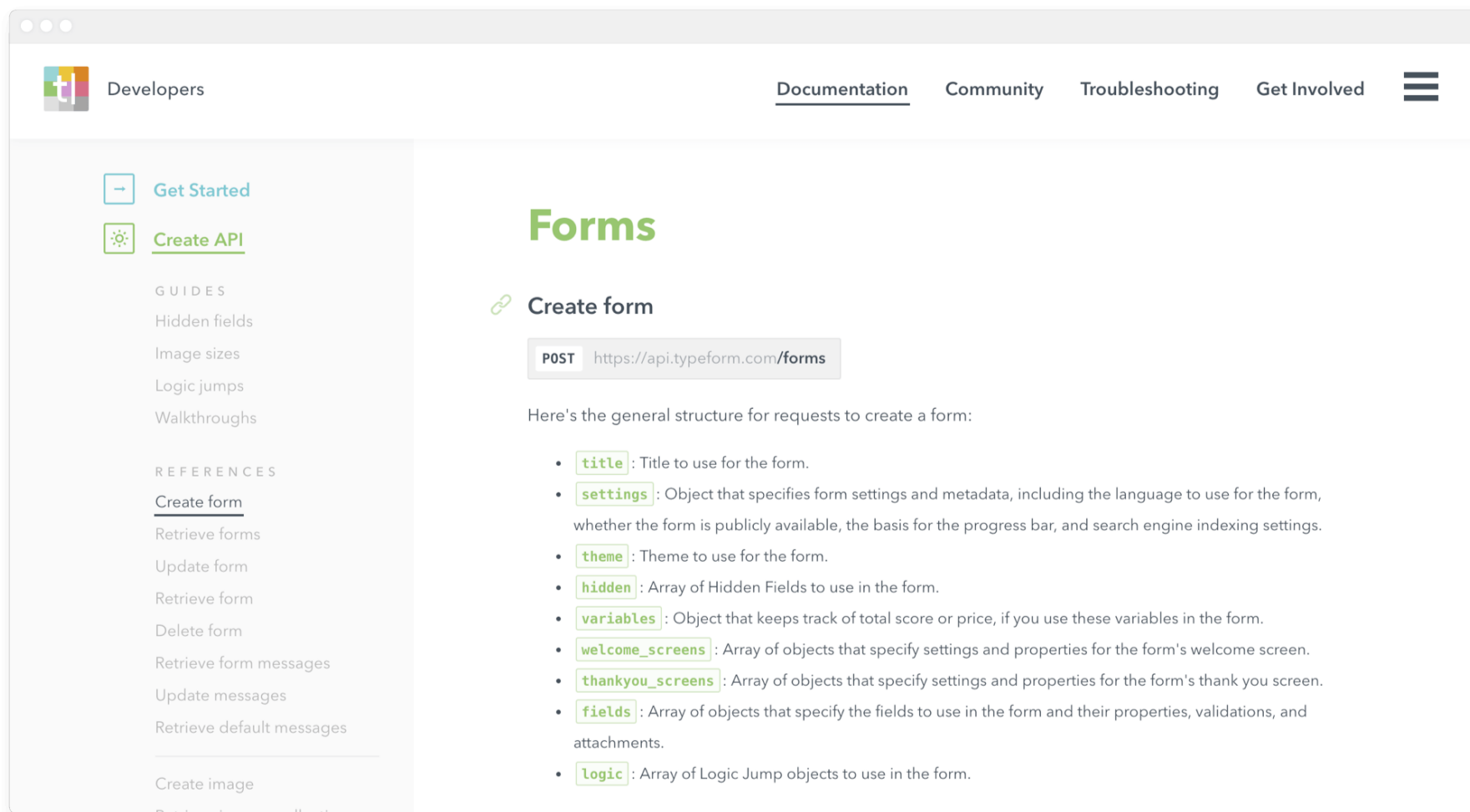
There's nothing worse than asking a developer to build something that isn't possible, especially when you're already paying for the product.

So what have we done for you? We've made a huge effort to create intro content in non-technical terms. We describe the functionality of the API without going into the 1s and 0s.

For example, for the [Create API](#) you'll find this:

“Create, update, delete, and customize typeforms, themes, and images on the fly, without using the Typeform builder.”

Even our Marketing Department understands this stuff. In fact, everyone can learn a lot about developer capabilities from our Developer Portal. We've seen it first hand. Numerous non-technical Typeformers have had "aha" moments after giving the content a cursory read.



Give it a spin

Now that you've heard a little about how we built our Developer Platform, here's some ways you can already use it:

- **OAuth 2:** set up integrations without copying and pasting an API key.
- **Create API:** create, update, delete, and customize typeforms, themes, and images on the fly, without using the Typeform builder.
- **Webhooks configuration API:** tell Typeform where your server is located, so we can send responses directly to your URL via Webhooks.

- **Responses API:** access the submissions for your typeforms in JSON format, without setting up webhooks or third-party integrations.
- **Embed SDK:** integrate your typeform straight into your website or web app—you get seamless integration, and people won't have to leave your site to respond.

Ready to give it a spin? Take a look at our [“Get Started”](#) page for some high-level ideas, and step-by-step guides for getting set up. We can't wait to see what you'll come up with!

By the way, our own developers have already started building new off-platform integrations using our APIs. We're also working with new partners to integrate Typeform into some of your favorite products using these same tools. Stay tuned.

You'll also be able to create typeforms and connect them directly to the data inside products you're already using. And very soon, you'll see Typeform connected with more platforms, including a Cloudflare app.

Follow-up posts on these new integrations are in production, so check back often.

In the meantime, let us know what you built, or are planning to build, and please pass the word around!

See you on the Interwebs!

What is the MVP for a Developer Portal?

<https://pronovix.com/blog/what-mvp-developer-portal>

Kathleen De Roo and Kristof Van Tomme



What information is absolutely essential on a developer portal? What kind of API documentation do you need? Is there a best practice that can be followed when launching a developer portal? In this post, we share the insights we've learned working on developer portals the last couple of years.

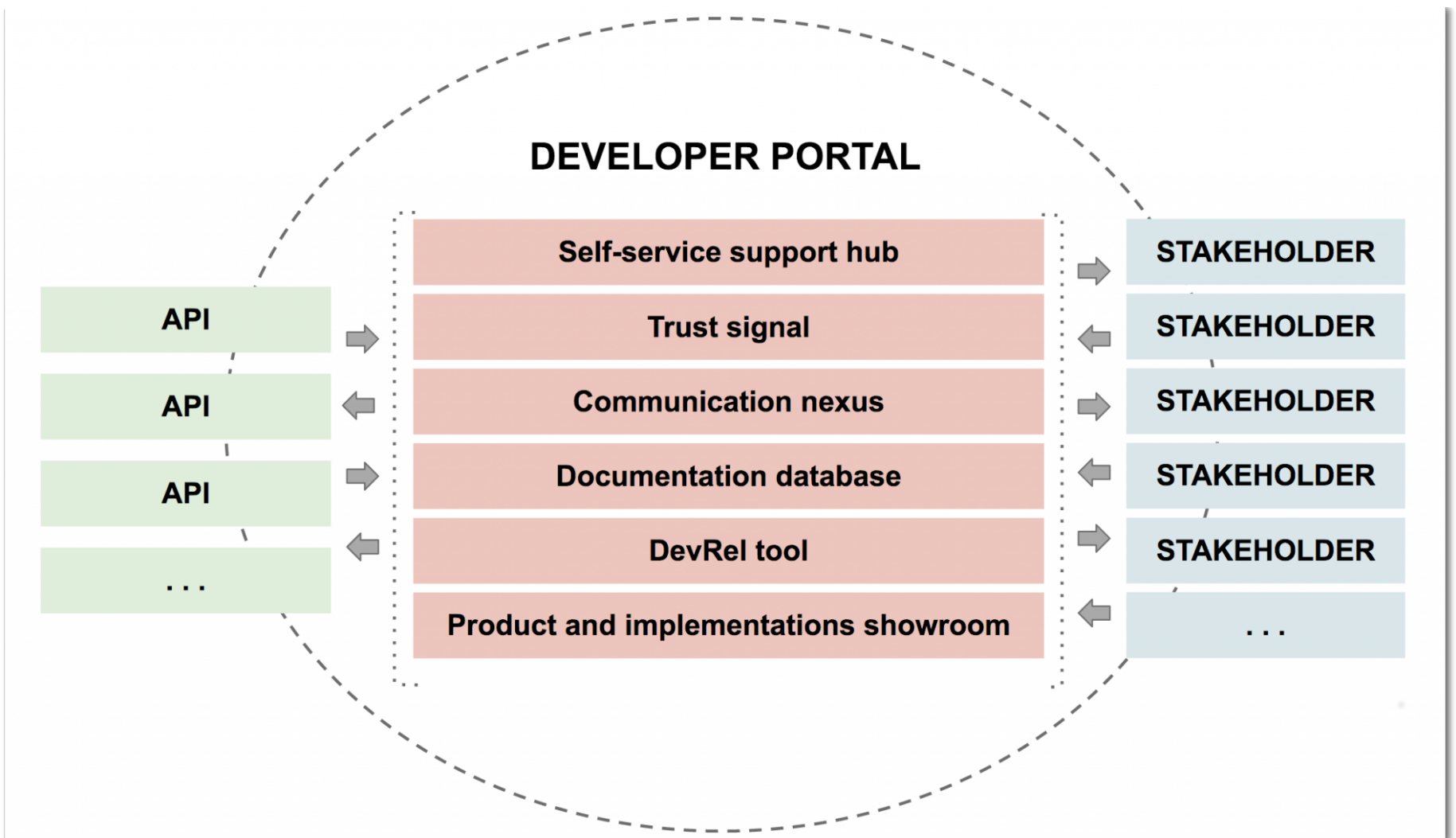
A quick look at some (API) developer portals will demonstrate that they can be very different in architectural structure and layout. That is remarkable, because most of the portals not only need to provide similar end results, they also address comparable audiences.

We formulated **11 questions** that can help you **define the content of your portal** to address the documentation needs of your stakeholders. To illustrate how it can be done in practice we built 3 mock sitemaps for developer portals and examine how they address the different stages of the developer's journey.

What is a developer portal?

A lot of API teams publish their "Swagger" documentation and call it a developer portal. That is wrong on two accounts: the documentation format formerly known as Swagger is now called the Open API spec, and more crucially, reference documentation is only one part of the minimum viable developer portal.

Yes, your developer portal needs to contain API reference **documentation** (no matter what specification format you use) but a developer portal should also be a sort of **self-service support hub**, a [trust signal](#), a **communication nexus for [API stakeholders](#)** and a **key DevRel tool** that helps an organization to provide the best possible developer experience for its APIs.



A developer portal is the interface between a set of APIs and their various stakeholders. The portal can play several roles to achieve the business goals of an organization.

11 questions your developer portal needs to answer

We compiled a first list of questions that provides users with the information they might need while working with your API product:

- 1 What is this API?
- 2 How do I get started with this API?
- 3 What do I need to understand about this API?
- 4 How do I get X done with this API?
- 5 Do I know all the details of this API?

- 6 How do I use your API in Y?
- 7 Is somebody still working on this API?
- 8 Where do I go when I have a problem with this API?
- 9 How do I get access to this API?
- 10 Can I afford this API?
- 11 Can I trust this API?

Most of these questions can have a dedicated section on a developer portal and detailed documentation types can be used to address them. In an MVP however, it is possible to answer these questions without having dedicated sections.

Throughout this post, we will focus on the (API consuming or also “downstream”) developer’s journey and apply the following colors to depict its 6 stages:

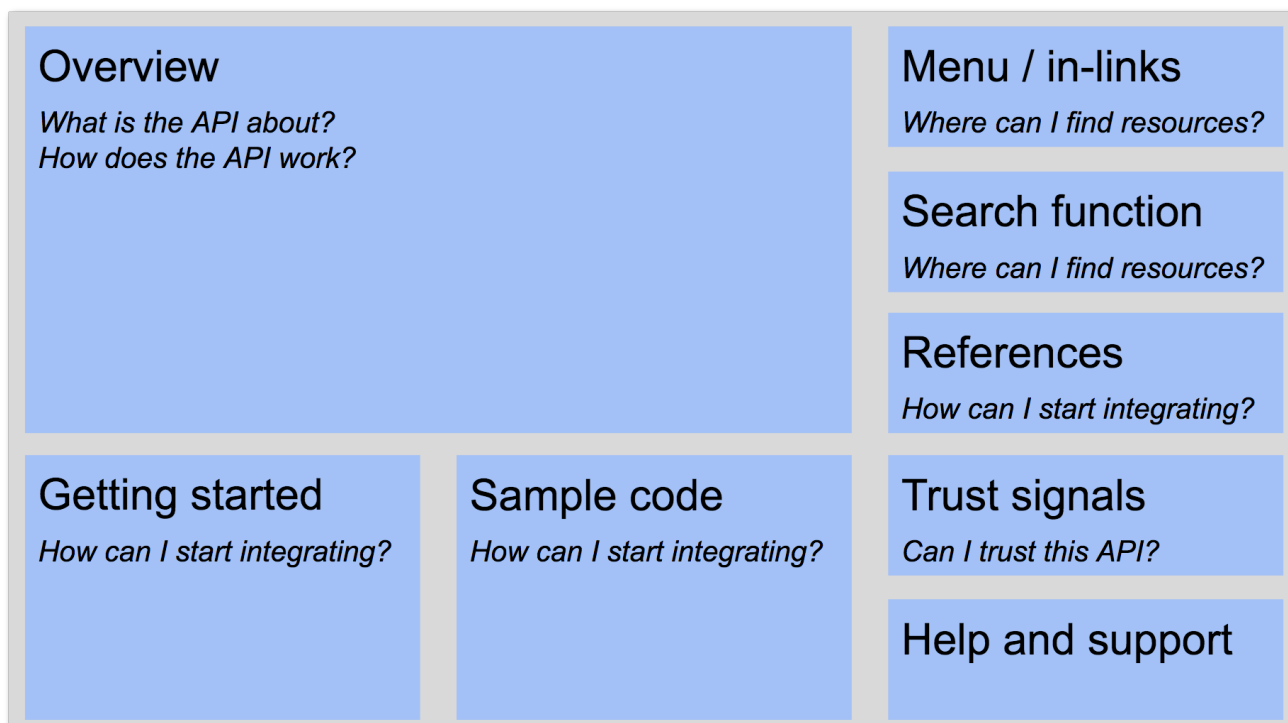


The 6 stages of the downstream developer journey are discover/ research, evaluate, get started, develop and troubleshoot,

1. What is this API?

Landing pages show the site architecture, help to find and navigate documentation, show what the API offers. Regardless their layout and design, landing pages (also called overview pages) best answer 5 questions immediately:

- **What is the API about?** (Purpose and main features of the API)
- **How does the API work?** (Technical architecture and programming workflow)
- **How can I start integrating?** (Implementing the API according to a developer's personal learning strategy)
- **Where can I find resources?** (Structure of documentation)
- **Can I trust this API?** (Trust signals in the broad sense, like pricing information, release notes, usage policies, API status)

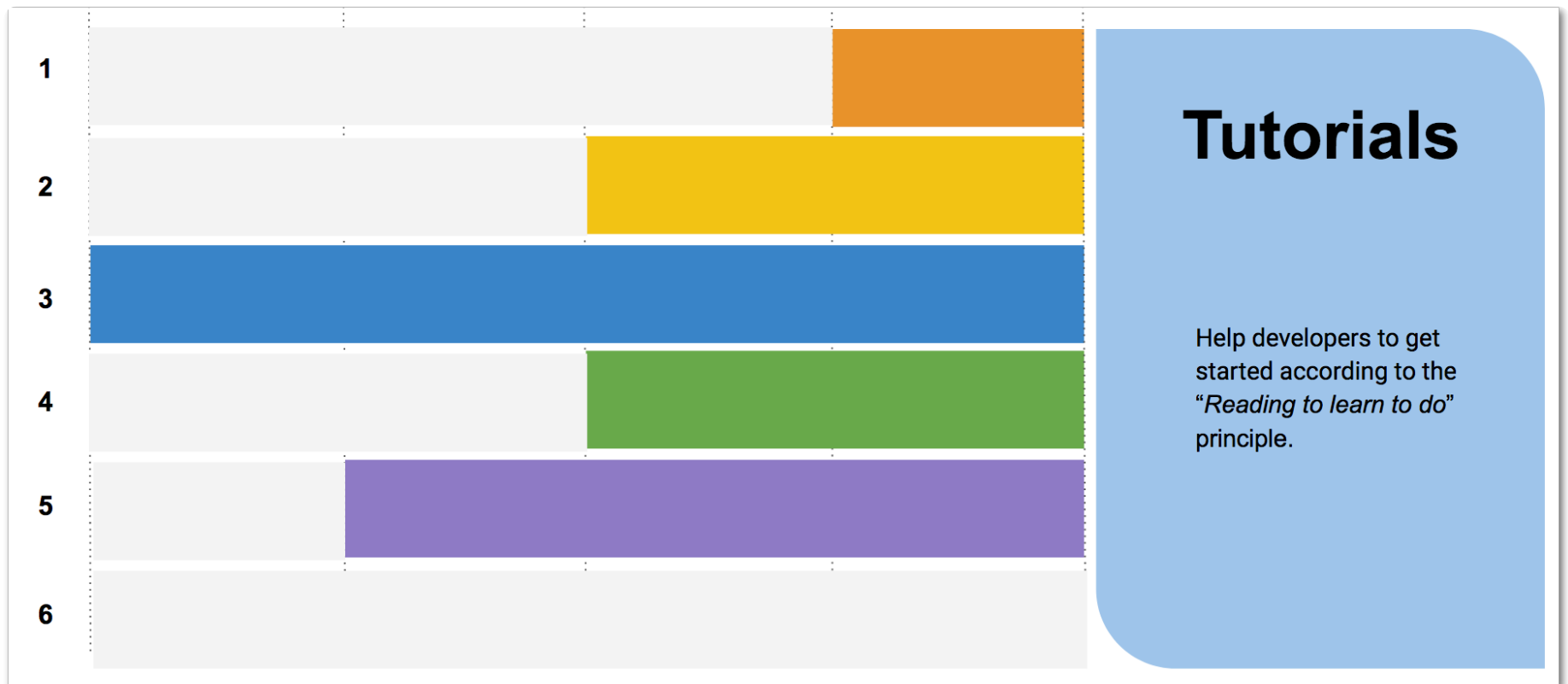


An example of landing page elements that address users with different learning strategies (concept-oriented vs code-oriented approach).

Note that the architecture and programming workflow explanations can also provide information about access restrictions. The help and support section ideally answers several questions at once. [This visual was inspired by the findings of [Peter Gruenbaum](#) (2010) and [Kata Nagygyörgy](#) (2015) and extended with the insights that [M. Meng et al.](#) (2017) and [Diána Lakatos](#) (2017) provided on the subject.]

2. How do I get started with this API?

Tutorials show how to do something step-by-step. Their primary role is to onboard users according to the [“Reading to learn to do”](#) principle. Include code examples to enable quick onboarding.



Tutorials are especially great in the Getting Started stage of the journey. [Asking your developers to write tutorials](#) can help them to celebrate their implementation work and create valuable resources for specific product use cases. To some extent tutorials can also help in the evaluation and discover/troubleshoot journey stages.

Other ways that help developers to get started are:

- Prototype building options (mock APIs, sandbox environments, test APIs),
- A glossary that explains concepts, and gives informative explanations for objects, methods, and parameters.
- Software Development Kits (SDKs) that focus on implementations in a specific programming language.

3. What do I need to understand about this API?

Conceptual docs explain portal specific concepts. Knowledge of portal and business specific words - like [“dunning”](#)- are not only important for developers that don’t know your industry, also more experienced developers less familiar with your product might benefit from a refresher about certain words in your domain language. There is a good chance that your organisation has developed a unique semantic meaning for at least a few words.

4. How do I get X done with this API?

Guides explain how to get something done. They explain how to solve problems via use cases, recipes or cookbooks. Guides take different formats:

- Topic guides can provide explanations and background information and help contextualize the topic,
- How-to guides and quickstart guides usually focus on the onboarding process.



Guides that provide code snippets or code examples can play an important role in the onboarding process. In the format of use cases they help [evaluate \(“Can I get my specific task done with this product?”\)](#) and [celebrate interesting implementations](#). Guides are also important in the research and develop phase (how can this be done theoretically and practically?).

5. Do I know all the details of this API?

Reference docs are crucial in the development and troubleshooting stage of the developer journey: they give detailed instructions on how to build the actual integration. API reference documentation is so important that API teams often mistakenly equate “API docs” with the reference documentation of an API. Welcomed as useful extras in reference documentation are:

- An error dictionary that describes error handling,
- Information on alternative classes, methods, parameters (where appropriate),
- Descriptions,
- Comments,
- Code language selectors,
- Conceptual information.

6. How do I use your API in Y?

SDKs ([Software Development Kits that are community driven, handcrafted or generated](#)) simplify development and help the developers that consume your API to implement best practices they might not be aware of. They have several functions throughout the developer’s journey:

- Evaluate (internally): by its nature an SDK needs to implement all or a large part of an API’s functionality. That is why it is a great opportunity to review and improve on an API’s design and implementation,
- Getting started: SDKs make it easy to start implementing in a specific programming language or framework, and help developers to overcome common problematic areas (like authentication),
- Develop and troubleshoot: Implement API calls in a popular programming language, so that developers can work in their favorite language or platform,
- Celebrate: Put developers in the spotlight that have created or contributed to an open source SDK,

- Maintain: SDKs can help keep applications in sync with API changes when used as a dependency in a project, and SDK metrics provide insight into API usage across the different developer communities.

7. Is somebody still working on this API?

API release notes (also called **changelog**) provide notifications about changes in the documentation. A regularly updated changelog is an important trust signal for your portal. **Blogs** can communicate solutions on a regular basis and can help prototype new content. If you have people to maintain and update your blog regularly, they can be a perfect content type to incubate new content, publish interesting usage scenarios, communicate changes and company strategies.



Your blog can play a role in the 6 stages of the developer journey, but our research showed that they are mostly used to engage users in the discover/research, evaluate and celebrate journey stages.

8. Where do I go when I have a problem with this API?

Support resources can offer solutions to niche problems and test documentation accuracy. We make a distinction between staffed support and peer-to-peer support:

- **Staffed support** can contain an audience focused FAQ page, a knowledge base, support pages where you can directly contact the company's support team,
- **Peer-to-peer support** is about facilitating communication between users via a community section, a developer forum or a third-party community page (e.g., on GitHub).

Support resources mostly address questions related to the getting started and develop/troubleshoot journey stages.

9. How do I get access to this API?

A fast and easy-to-use **API key generator** improves developer experience (DX): include a link in the code examples, sandbox environment, reference documentation and other pages where your users start implementing the API.

10. Can I afford this API?

Depending on the role and objectives of your visitors, unclear information about your **pricing** and business model might become a blocker.

11. Can I trust this API?

Policies (like security, cookie, partner policies) communicate principles that specify the relation between customer and API supplier. Make this data accessible, findable and easy to navigate. Sidebar [summaries](#) can help to orient your users.

MVP and beyond

Rather than including all the documentation or content types we listed above at once, it is more important to:

- Examine, but also align the information needs of your company's key audiences with your developer portal strategy,

- Adjust content to your users' expectations logically and efficiently, decide what roles you want the content types to play in the user journey.

An MVP should focus on the minimal content that your users need to do their job. New information can be added later to address issues you discover in the developer journey as you keep evaluating and iterating on your developer experience (DX).

While there are best practices, it is impossible to create a great DX without iterating. Too much content can sometimes be as much a problem as too little content. This iterative nature of the whole process is another reason why a [docs like code](#) approach has become so popular in the API community.

Bare minimum MVP

As a bare minimum an MVP should provide the following information and corresponding minimum content:

Questions	→	Content types
What is this API?	→	Landing page
How do I get started with this API?	→	Tutorials
Do I know all the details of this API?	→	References
Where do I go when I have a problem with this API?	→	Support contact details

First iteration

A first iteration could have a blog to experiment with new content, and could provide extended support options:

Questions	→	Content types (examples)
What is this API?	→	Landing page
How do I get started with this API?	→	Tutorials
Do I know all the details of this API?	→	References
Where do I go when I have a problem with this API?	→	Support contact details; FAQ page; forum
Is somebody still working on this API?	→	Blog

Second iteration

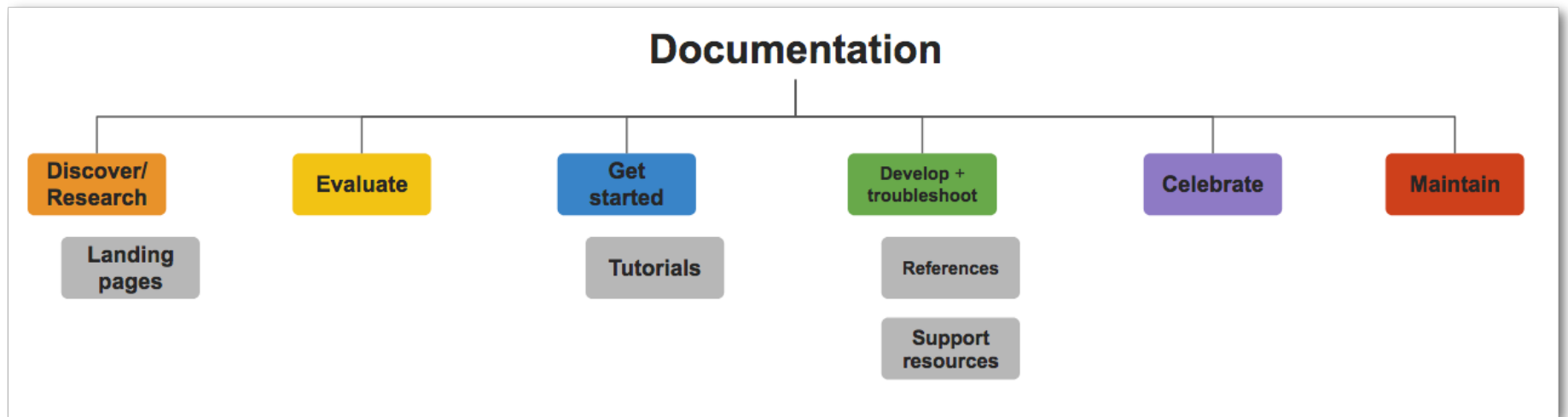
A second iteration could answer all 11 questions and provide exhaustive API documentation:

Questions	→	Content types (examples)
What is this API?	→	Landing page
How do I get started with this API?	→	Tutorials
Do I know all the details of this API?	→	References
Where do I go when I have a problem with this API?	→	Support contact details; FAQ page; forum
Is somebody still working on this API?	→	Blog; API release notes
What do I need to understand about this API?	→	Concepts
How do I get X done with this API?	→	Guides
How do I use your API in Y?	→	SDKs
How do I get access to this API?	→	API key generator
Can I afford this API?	→	Pricing
Can I trust this API?	→	API policies

MVPs and their role in the developer journey

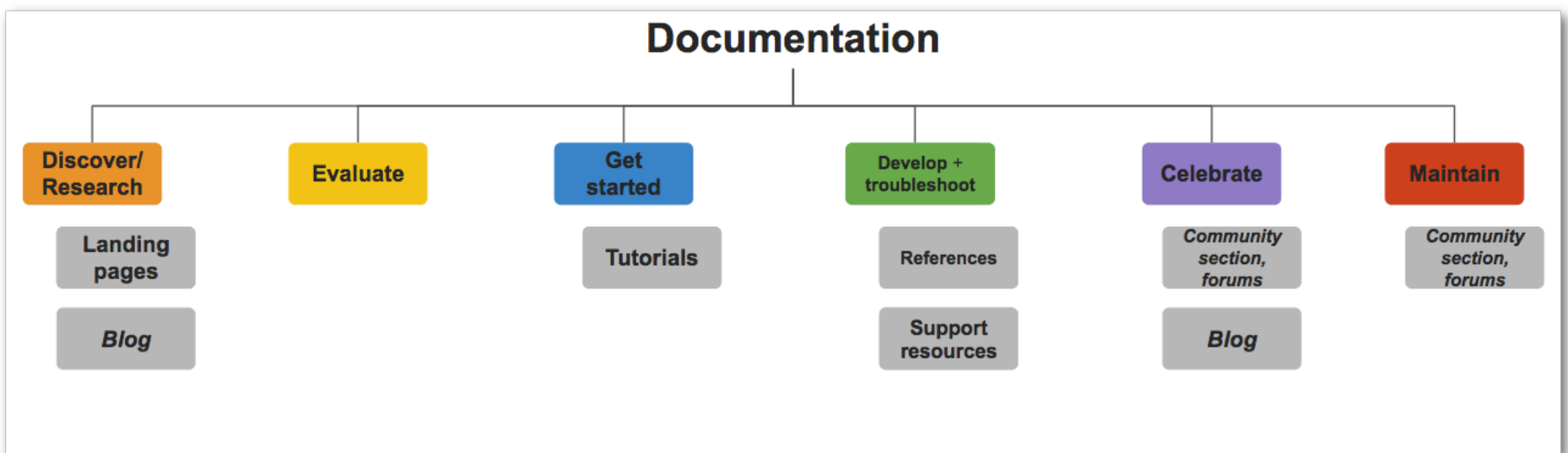
How do the listed MVP and the subsequent iterations address the 6 stages of the downstream developer journey?

Minimal MVP



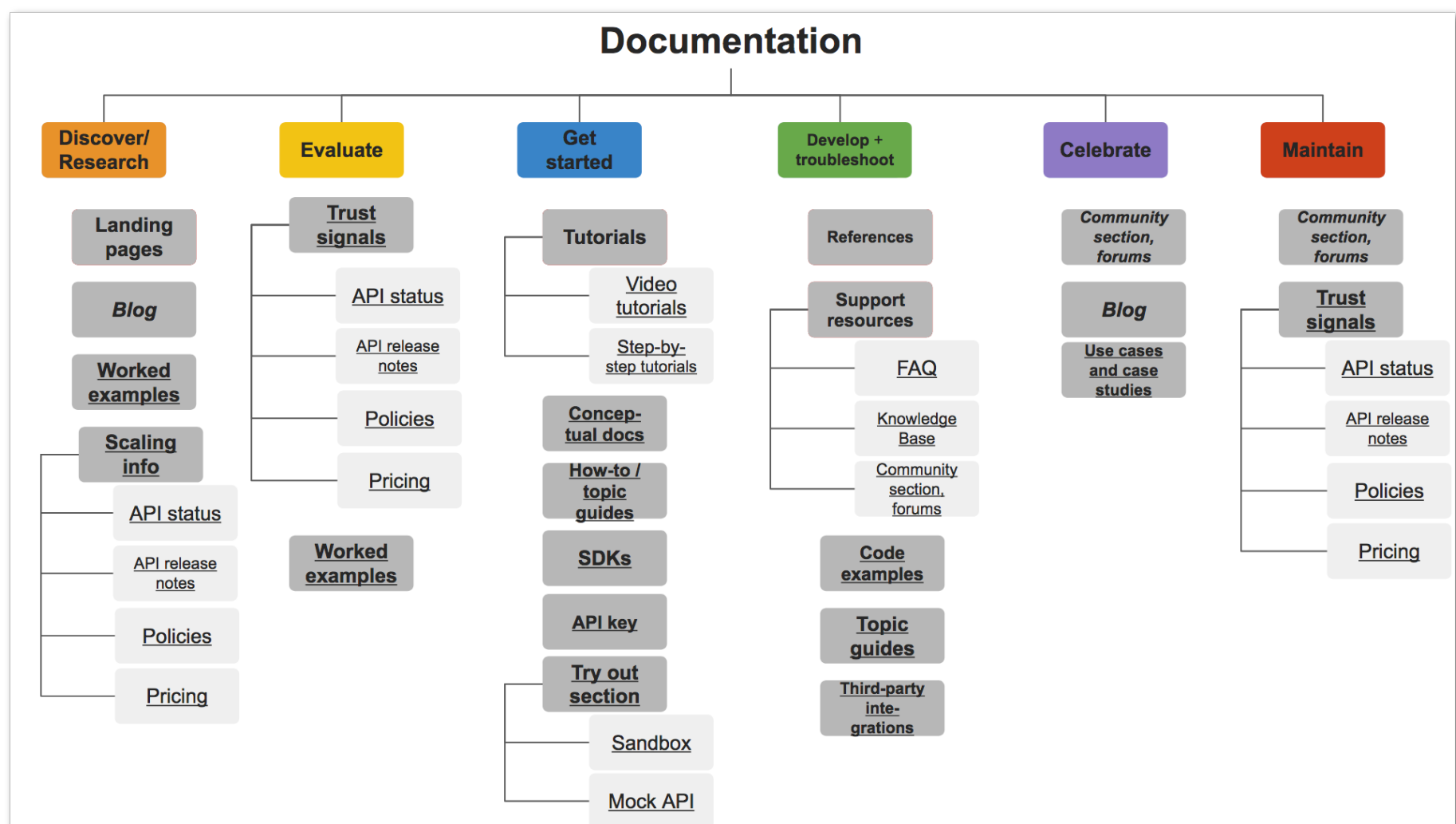
With this minimal MVP, developer portals only address the discover/research, get started and develop/troubleshoot stages: users find an answer to what the portal is about, how they can get started coding and where to find the code.

First iteration



This developer portal MVP already addresses 5 out of 6 stages in the developer journey, but with a limited number of documentation types. This kind of portal could provide a blog, but also support resources, like an FAQ page or a community forum. (Note that the content types that we added for this iteration are written in italic.)

Second iteration



An example of how all the questions above can be translated into documentation types that do not only address all the stages of the developer journey but also focus on the developers' different learning approaches more thoroughly. Some documentation types can appear in more than one stage of the journey. We also indicated some examples of subcomponents. (Note that the content types that we added for this iteration are underlined.)

Combine best practices with strategic decisions

Your developer portal is an interface for your API strategy, and, ideally, aligns the API communication with the documentation. That is why it is crucial to make a thorough study of your strategic objectives and the personas that will be interacting with your developer portal.

Would you like to get help developing your portal? [Get in touch](#) for a complementary Developer Portal Architecture Workshop or get a quote for a Content Architecture Workshop.

API the Docs

| <https://pronovix.com/api-docs-amsterdam-2017>



API the Docs Amsterdam: one-day conference about API documentation and developer portals.

December 4, 2017

- Cristiano Betta: The Seven Deadly Sins of Developer Onboarding
- Alaina Kafkes: Building Beginner-Friendly API Tutorials
- Koen Adolfs: Open banking - Let's Go Beyond Banking
- Anthony T. Sansone: Writing for Scale: Streamlining API Documentation Maintenance
- Emile Bremmer: Viewing a Developer's API Journey through Logs
- Jessica Ulyate: From dreadful to Dreddfull: Automated Testing for your API
- Roman Hotsiy: Self-Documented APIs: Myth or Reality?
- Karen Sawrey: Not all Rocket Scientists want to be Brain Surgeons: Lessons Learned Documenting Cryptography APIs
- Laurent Doguin: Continuous swagness for your APIs
- Nathalie Oostvogels: When the Specification Fails: Documenting Inter-Parameter Constraints
- Aleksei Akimov: Beyond the Basic Swagger UI: Adyen API Explorer
- Adam Butler: Engineering Great Documentation

Below you find the recordings, slide decks and Laura's notes from API the Docs Amsterdam.

The next API the Docs event takes place in Paris on April 24, 2018.

The Seven Deadly Sins of Developer Onboarding

[Cristiano Betta](#)

Developer Experience designer, [betta.io](#)

What is DX?

Developer Experience is the journey between a first site visit and a successful API integration and is driven by tooling (like SDKs) and information (documentation around the tools).

7 DX sins and 7 tricks to tackle them

The 7 sins

- Too much information
- Information that comes too soon
- Too little, too late information
- Unstructured information
- Unsupportive information
- Incomplete information
- Out of control tooling

The 7 takeaways

- Prevent cognitive overload
- Only ask questions when needed
- Present information with structure
- Present information on time
- Tell the best stories
- Tell the whole story
- Own the whole story

Throughout his whole presentation, Cristiano provides us with **real-life examples** and lists solutions that companies like [Braintree](#), [GitHub](#) and [Amazon](#) apply to **tackle DX problems**.

Find a [summary of Cristiano's talk on his website](#).

Cristiano's [slides](#) and [presentation](#).

Building Beginner-Friendly API Tutorials

[Alaina Kafkes](#)

Software engineer at [Medium](#)

Think like a reader: proofread your tutorials

Write to retain new users. Optimize your docs for novice users via beginner-friendly tutorials. But how can writers look at content they already know anew? In her talk, Alaina sums up **7 guidelines** with **loads of tips** to follow and includes **practical examples** taken from companies like [Clarifai](#), [Twilio](#) and [GitHub](#).

- 1 Offer a quick start
- 2 Perform a tech audit
- 3 Remember your environment
- 4 Share next steps
- 5 Anticipate errors
- 6 Stick to the standard library
- 7 Use inclusive language

Quality tutorials result in API user growth

Proofreading improves tutorial quality → quality tutorials lead to beginner retention → retention fosters a healthy community → community augments API user growth.

Alaina's [slides](#) and [presentation](#).

Open Banking: Let's Go Beyond Banking

[Koen Adolfs](#)

Product Owner of Open Banking at [ABN Amro](#)

ABN AMRO recently launched their API [developer portal](#) Pronovix also worked on. We are all super proud of it!

Koen showed us what it takes in a large financial organization to get such an initiative supported and realized into a live platform, and how they see their future path as an aggregator of many sectors. Albeit his presentation is not to be published in its entirety, we got approval for this one slide.

Writing for Scale: Streamlining API Documentation Maintenance

[Anthony Sansone](#)

Senior technical writer at [MongoDB](#)

How MongoDB improved its documentation

Unclear documentation can lead to more adoption friction and often means that organizations with larger deployments face challenges with using your application at scale. At [MongoDB](#), the API documentation was difficult to use. When trying to update that documentation, Anthony and colleagues discovered that how they managed those docs was unsustainable. In trying to help their users scale, they saw that they needed to help themselves scale the documentation:

- Refactor the docs: what do we have and what do we need to change?
- Refactor the process: do research before revising the process,
- Plan more, execute less,
- Establish own best practices to make the API specification consistent and complete,
- Schedule the work: prioritize what to convert,

- Automate the API specification and document how to use the API.

7 takeaways from MongoDB's docs process

- 1 The API specification is not the API docs, only part of it,
- 2 Automate the API specification,
- 3 Work with your engineers on revising the API specification,
- 4 Set and stick to standards,
- 5 Test your API: use it to be able to explain it on multiple platforms,
- 6 Plan for incremental rollout,
- 7 Invest in tutorials: focus on tasks and activities.

Anthony's [slides](#) and [presentation](#).

Viewing a Developer's API Journey through Logs

Emile Bremmer

Developer at [ABN Amro](#)

How to process feedback for [ABN Amro's developer portal](#)

ABN Amro uses the [Splunk](#) tool to turn raw data into searchable and visualizable information.

This enables the team to investigate:

- how developers work with the portal and reach success,
- what the repetitive errors are,
- what can be done to improve efficiency via the documentation or error descriptions,
- example API calls, developer profiling, common mistakes.

Challenges remain. Next steps on the agenda include log data, before/after comparisons, machine learning opportunities.

Contact Emile directly (emile.bremmer@nl.abnamro.com) with your own feedback.

Emile's [slides](#) and [presentation](#).

From dreadful to Dreddfull: Automated Testing for your API

[Jessica Ulyate](#)

Developer and Product Manager

How to test automated API documentation

APIs need good documentation, and to keep your reference docs up-to-date you need to test them. [Dredd](#) is a program that tests API references in specification languages [Swagger/OpenAPI Initiative](#) and [API Blueprint](#).

Jessica talks about how to use Dredd in practice and some of its perks, like:

- using hooks to modify request data,
- testing your docs continuously with CI/CD.

Test docs to make life simpler

- Automated testing the API reference docs is something that - with the right tooling - even people in not so large organizations, with little financial support can do,
- API reference docs are the basis of your documentation and have the highest potential for generating errors.

Check out: [Keeping documentation honest](#) - an article recommended by Jessica.

Jessica's [slides](#) and [presentation](#).

Self-documented APIs: myth or reality?

[Roman Hotsiy](#)

Software Engineer, [APIs.guru](#), [ReDoc](#)

Self-documented APIs: available solutions

Programmers often struggle writing API documentation that is free-form text based. There are several products on the market - like [Hypermedia/Hateous](#), [OpenAPI Initiative](#), and [GraphQL](#) - that allow for self-documented API references, but what exactly do they offer us? In his presentation, Roman provides us with several demos and examples, but also lists perks and insufficiencies.

Self-documented API documentation as a whole?

The references are only part of the API documentation. As a whole, API documentation could be categorized into:

- Technical details (How to send a request and extract data from the response?) that can be self-documented,
- Conceptual documentation (Why do we need this API?), which cannot be self-documented.

Conclusion: Establish API description formats as a middle ground (write concepts alongside code) and enable developers and tech writers to collaborate according to [docs like code](#).

In-between self-documented and written documentation: takeaways

- Reference docs are only part of the API documentation,
- Developers can auto-generate some parts of the API documentation,
- Technical writers are absolutely required to write the other parts,
- Programmers and tech writers should cooperate (check [Docs like Code](#)).

Roman's [slides](#) and [presentation](#).

Not all Rocket Scientists want to be Brain Surgeons: Lessons Learned

Documenting Cryptography APIs

[Karen Sawrey](#)

Technical writer at [Cossack Labs](#)

Cryptography is hard, explaining it is even harder

In her presentation, Karen talks about her job as a technical writer documenting cryptography APIs and the experience she gained, e.g.,

- Keep the balance between exact-scientific and usable,
- Verify continuously,
- Plan: what's the least number of actions that gives the best results.

Karen describes what the technical team's workflow looks like, what tools they use, which documentation types they provide, what challenges they face.

Master documenting cryptography APIs: tips and tricks

- Nothing is too complicated to explain,
- Choose simple words,
- Think about your documentation users,
- Boldly increase the number of communication channels (get to know your audiences),
- Don't be afraid to ask questions.

Karen's [presentation](#).

Continuous Swagness for your APIs

[Laurent Doguin](#)

Head of Developer Relations at [Clever Cloud](#)

Continuous integration and delivery

Laurent demo-ed us how they use [Swagger](#) to generate a website for their APIs' documentation and specific clients for each language they support. Each time they update the spec they test it, update the website, build, test and deploy the new Clients libraries automatically. He showed us (live demo) how to set everything up from scratch thanks to [Jenkins](#), [Artifactory](#) and [Clever Cloud](#).

Laurent's tips:

- The production has to be perfect,
- There is one process: build, then register, and live without data,
- Keep the more updated version,
- Statelessness is the key: create a factory of instances.

Laurent's [presentation](#).

When the Specification Fails: Documenting Inter-Parameter Constraints

[Nathalie Oostvogels](#)

PhD student at the [Vrije Universiteit Brussel](#)

Inter-parameter constraints are common in web API specifications

In web API specifications, constraints come along with parameters. You need to satisfy every constraint for a request to succeed. API specification languages ([API Blueprint](#), [RAML](#), [Swagger](#)) and their tools help to ease that process, but do not yet provide a solution to express constraints across parameters (inter-parameter constraints).

Nathalie researched popular web APIs (like [Twitter](#), [Google Maps](#), [YouTube](#), [Facebook](#)) and found three types of inter-parameter constraints:

- Exclusive constraints,
- Dependent constraints,
- Group constraints.

A new specification language to describe inter-parameter constraints

The speaker shows us how [JSON Schema](#) (Swagger/OpenAPI Initiative) features do not allow for an efficient description of inter-parameter constraints.

Nathalie started working on **an extension of the OpenAPI Initiative specification language that support inter-parameter constraints**. She also recommends the usage of **a truth table**.

Consequently, documentation tools should be extended:

- render the inter-parameter constraints as a separate block in the docs
- indicate that field is part of inter-parameter constraint

To end, Nathalie provides us with a **guide to recognize inter-parameter constraints in API documentation**.

Natalie's [slides](#) and [presentations](#).

Aleksei Akimov - Beyond the Basic Swagger UI: Adyen API Explorer

[Aleksei Akimov](#)

Technical writer at [Adyen](#)

From API to documentation

Adyen's rapid growth provides challenges concerning documentation. For their API references, they chose the [Swagger tool](#) (to implement the OpenAPI specification format - as Aleksei explains the semantic differences). The basic Swagger UI however, the speaker

indicates, lacked an effective layout and provides problems with nested structures. How could Adyen go beyond?

Reimagining the Swagger UI: from specification to documentation

- One user persona can wear different hats, and act e.g. as the **learning developer** or the **coding developer**,
- **Work in streams**: have team members with different professional profiles,
- Results: Among [Adyen's API Explorer](#)'s **first wins** are support for multiple specs, adaptive layout, clean structure for nested objects, the multiple code examples.

Tips and tricks after building Adyen's API Explorer

- Let technical writers and developers collaborate on the same content,
- Use the latest version of the OpenAPI,
- Use markdown,
- Integrate swagger in CI,
- OpenAPI supports custom annotations,
- Launch fast and iterate (be agile).

Aleksei's [slides](#) and [presentation](#).

DocOps - Engineering Great Documentation

[Adam Butler](#)

Technical Lead of [Nexmo Developer](#)

DocOps: Engineering to help maintain the documentation

Adam talked about the open-source platform and tools they built when developing [Nexmo Developer](#), for writing rich documentation collaboratively with ease. He talked about how they tackled problems concerning documentation requirements and met their goals.

- How can people contribute on docs while keeping the quality high?

- **Think like an engineer:** Build tools and processes that give the docs back to the contributing people (engineers, product owners, tech writers, customers, support, ...).

[Nexmo Developer](#) features

- Open source
- Tooling: [Ruby](#) + [Rails](#) + Nexmo flavoured markdown
- [Contribution guides](#)
- Automation
- [Docs like Code](#)

Read Adam's article on [Extending Markdown with middleware](#).

Adam's [slides](#) and [presentation](#).

Original recording of the conference by [Kristof Van Tomme](#), Creative Commons Attribution Share-Alike License v3.0

TOOLCHAINS

DOCS AS CODE



Case Study: Switching Tools to Docs-as-Code

http://idratherbewriting.com/learnapidoc/pubapis_switching_to_docs_as_code.html

Tom Johnson



For an overview of the docs-as-code approach, see [Docs-as-code tools](#). In this article, I describe the challenges we faced in implementing a docs-as-code approach within a tech writing group at a large company.

Changing any documentation tooling at a company can be a huge undertaking. Depending on the amount of legacy content to convert, the number of writers to train, the restrictions and processes you have to work against in your corporate environment and more, it can require an immense amount of time and effort to switch tools from the status quo to docs-as-code.

Additionally, you will likely need to make this change outside your normal documentation work, and you'll probably need to develop the new system while still updating and publishing content in the old system. Essentially, this means you'll be laying down a new highway while simultaneously driving down it.

Previous processes

Previously, our team published content through a content management system called [Hippo](#) (by Bloomreach). Hippo is similar to WordPress or Drupal but is Java-based rather than PHP-based (which made it attractive to a Java-centric enterprise that restricted PHP but still needed a CMS solution for publishing).

To publish a page of documentation, tech writers had to create a new page in the Hippo CMS and then paste in the HTML for the page (or try to use the WYSIWYG editor in the Hippo CMS). If you had 50 pages of documentation to publish, you would need to paste the HTML into each CMS page one by one. Originally, many writers would use tools such as [Pandoc](#) to convert their content to HTML and then paste it into the Hippo CMS. This copy-and-paste approach was tedious, prone to error, and primitive.

When I started, I championed using Jekyll to generate and manage the HTML, and I started storing the Jekyll projects in internal Git repositories. I also created a layout in Jekyll that was designed specifically for Hippo publishing. The layout included a documentation-specific sidebar (previously absent in Hippo on a granular level) to

navigate all the content in a particular set of documentation. This Jekyll layout included a number of styles and scripts to override settings in the CMS.

Despite this innovation, our publishing process still involved pasting the generated HTML (after building Jekyll) page by page into the CMS. Thus, we were halfway with our docs-as-code approach and still had room to go. One of the tenets of docs-as-code is to build your output directly from the server (called “continuous deployment”). In other words, you incorporate the publishing logic on the server rather than running the publishing process from your local computer.

This last step, publishing directly from the server, was difficult because another engineering group was responsible for the website and server, and we couldn’t just rip Hippo out and start uploading the Jekyll-generated files onto a web server ourselves. It would take another year or more before the engineering team had the bandwidth for the project. Once it started, the project was a wild ride of mismatched expectations and assumptions. But in the end, we succeeded.

Most of the lessons learned here are about this process, specifically how we transitioned to building Jekyll directly from an internal Git repo, the decisions we made and the reasoning behind those decisions, the compromises and other changes of direction, and so on. My purpose here is to share lessons learned so that other writers embarking on similar endeavors can benefit from understanding what might be on the road ahead.

Advantages of integrating into a larger system

Why did we want to move to docs as code in the first place? At most large companies, there are plenty of robust, internally developed tools that tech writers can take advantage of. The docs-as-code approach would allow us to integrate into this robust enterprise infrastructure that developers had already created.

Documentation tools are often independent, standalone tools that offer complete functionality (such as version control, search, and deployment) within their own system. But these systems are often a black box, meaning, you can’t really open them up and integrate them into another process or system. With the docs-as-code approach, we had the flexibility to adapt our process to fully integrate within the company’s infrastructure and

website deployment process. Some of this infrastructure we wanted to hook into included the following:

- Internal test environments (a gamma environment separate from production)
- Authentication for specific pages based on account profiles
- Search and indexing
- Website templating (primarily a complex header and footer)
- Robust analytics
- Secure servers in order to satisfy Information Security policies with the corporate domain
- Media CDN for distributing images
- Git repositories and GUI for managing code
- Build pipelines and a build management system

All we really needed to do was to generate out the body HTML along with the sidebar and make it available for the existing infrastructure to consume. The engineering team that supported the website already had a process in place for managing and deploying content on the site. We wanted to use similar processes rather than coming up with an entirely different approach.

End solution

In the end, here's the solution we implemented. We stored our Jekyll project in an internal Git repository — the same farm of Git repositories other engineers used for nearly every software project, and which connected into a build management system. After we pushed our Jekyll doc content to the master branch of the Git repository, a build pipeline would kick off and build the Jekyll project directly from the server (similar to [GitHub Pages](#)).

Our Jekyll layout omitted any header or footer in the theme. The built HTML pages were then pulled into an S3 bucket in AWS through an ingestion tool (which would check for titles, descriptions, and unique permalinks in the HTML). This bucket acted as a flat-file database for storing content. Our website would make calls to the content in S3 based on

permalink values in the HTML to pull the content into a larger website template that included the header and footer.

The build process from the Git repo to the deployed website took about 10 minutes, but tech writers didn't need to do anything during that time. After you typed a few commands in your terminal (merging with the `gamma` or `production` branch locally and then pushing out the update to origin), the deployment process kicked off and ran all by itself.

The first day in launching our new system, a team had to publish 40 new pages of documentation. Had we still been in Hippo, this would have taken several hours. Even more painful, their release timeframe was an early morning, pre-dawn hour, so the team would have had to publish 40 pages in Hippo CMS at around 4 am to 6 am, copying and pasting the HTML frantically to meet the release push and hoping they didn't screw anything up.

Instead, with the new process, the writer just merged her development branch into the `production` branch and pushed the update to the repo. Ten minutes later, all 40 pages were live on the site. She was floored! We knew this was the beginning of a new chapter in our team's processes. We felt like a huge burden had been lifted off our shoulders, and the tech writers loved the new system.

Challenges we faced

I've summarized the success and overall approach, but there were a lot of questions and hurdles in developing the process. I'll detail these main challenges in the following sections.

Inability to do it ourselves

The biggest challenge, ironically, was probably with myself — dealing with my own perfectionist, controlling tendencies to do everything on my own, just how I wanted. (This is probably both my biggest weakness and strength as a technical writer.) It's hard for me to relinquish control and have another team do the work. We had to wait about a year for the overworked engineering team's schedule to clear up so they would have the bandwidth to do the project.

During this wait time, we refined our Jekyll theme and process, ramped up on our Git skills, and migrated all of the content out of the old CMS into [kramdown Markdown](#). Even so, as project timelines kept getting delayed and pushed out, we weren't sure if the engineering team's bandwidth would ever lighten up. I wanted to jump ship and just deploy everything myself through the [S3 website plugin](#) on [AWS S3](#).

But as I researched domain policies, server requirements, and other corporate standards and workflows, I realized that a do-it-myself approach wouldn't work (unless I possessed a lot more engineering knowledge than I currently did). Given our corporate domain, security policies required us to host the content on an internal tier 1 server, which had to pass security requirements and other standards. It became clear that this would involve a lot more engineering knowledge and time than I had, as well as maintenance time if I managed the server post-release, so we had to wait.

We wanted to get this right because we probably wouldn't get bandwidth from the engineering team again for a few years. In the end, waiting turned out to be the right approach.

Understanding each other

When we did finally begin the project and started working with the engineering team, another challenge was in understanding each other. The engineering team (the ones implementing the server build pipeline and workflow) didn't understand our Jekyll authoring process and needs.

Conversely, we didn't understand the engineer's world well either. To me, it seemed all they needed to do was upload HTML files to a web server, which seemed a simple task. I felt they were overcomplicating the process with unnecessary workflows and layouts. And what was the deal with storing content in S3 and doing dynamic lookups based on matching permalinks? But whereas I had in mind a doghouse, they had in mind a skyscraper. So their processes were probably more or less scaled and scoped to the business needs and requirements.

Still, we lived in different worlds, and we had to constantly communicate about what each other needed. It didn't help that we were located in different states and had to interact virtually, often through chat and email.

Figuring out repo size

Probably the main challenge was to figure out the correct size for the documentation repos. Across our teams, we had 30 different products, each with their doc navigation and content. Was it better to store each product in its own repo, or to store all products in one giant repo? I flipped my thinking on this several times.

Storing content in multiple repos led to quick build times, reduced visual clutter, resulted in fewer merge conflicts, didn't introduce warnings about repo sizes, and had other benefits with autonomy.

On the other hand, storing all content in one repo simplified content re-use, made link management and validation easier, reduced maintenance efforts, and more. Most of all, it made it easier to update the theme in a single place rather than duplicating theme file updates across multiple repos.

Originally, our team started out storing content in separate repos. When I had updates to the Jekyll theme, I thought I could simply explain what files needed to be modified, and each tech writer would make the update to their theme's files. This turned out not to really work — tech writers didn't like making updates to theme files. The Jekyll projects became out of date, and then when someone experienced an issue, I had no idea what version of the theme they were on.

I then championed consolidating all content in the same repo. We migrated all of these separate, autonomous repos into one master repo. This worked well for making theme updates. But soon the long build times (1-2 minutes for each build) became painful. We also ran into size warnings in our repo (images and other binary files such as Word docs were included in the repos). Sometimes merge conflicts happened.

The long build times were so annoying, we decided to switch back to individual repos. There's nothing worse than waiting 2 minutes for your project to build, and I didn't want

the other tech writers to hate Jekyll like they did Hippo. The lightning-fast auto-regenerating build time with Jekyll is part of its magic.

Creative solutions for theme distribution across repos

I came up with several creative ways to push the theme files out to multiple small repos in a semi-automated way. My first solution was to distribute the theme through [RubyGems](#), which is Jekyll's official [solution for theming](#). I created a theme gem, open-sourced it and the theme (see [Jekyll Doc Project](#)), and practiced the workflow to push out updates to the theme gem and pull them into each repo.

It worked well (just as designed). However, it turns out our build management system (an engineering tool used to build outputs or other artifacts from code repositories) couldn't build Jekyll from the server using [Bundler](#), which is what RubyGems required. (Bundler is a tool that automatically gets the right gems for your Jekyll project based on the Jekyll version you are using. Without Bundler, each writer just installs the [jekyll gem](#) locally and builds the Jekyll project based on that gem version.

My understanding of the build management system was limited, so I had to rely on engineers for their assessment. Ultimately, we had to scrap using Bundler and just build using `jekyll serve`. I still had the problem of distributing the same theme across multiple repos.

My second attempt was to distribute the theme through [Git submodules](#). This involved storing the theme in its own Git repo that other Git repos would pull in. However, our build management system couldn't support Git submodules either, it turned out.

I then came up with a way to distribute the theme through [Git subtrees](#). Git subtrees worked in our build system (although the commands were strange), and it preserved the short build times. However, when the engineering team started counting up all the separate build pipelines they'd have to create and maintain for each of these separate repos (around 30), they said this wasn't a good idea from a maintenance point of view.

Not understanding all the work involved around building publishing pipelines for each Git repo, there was quite a bit of frustration here. It seemed like I was going out of my way to accommodate engineering limitations, and I wasn't sure if they were modifying any of their processes to accommodate us. But eventually, we settled on two Git repos and two pipelines. We had to reconsolidate all of our separate repos back into two repos. You can probably guess that moving around all of this content, splitting it out into separate repos and then re-integrating it back into consolidated repos, etc., wasn't a task that the writers welcomed.

There was a lot of content and repo adjustment, but in the end, two large repos was the right decision. In fact, in retrospect, I wouldn't have minded just having one repo for everything.

Each repo had its own Jekyll project. If I had an update to any theme files (e.g., layouts or includes), I copied the update manually into both repos. This was easier than trying to devise an automated method. It also allowed me to test updates in one repo before rolling them out to the other repo. To reduce the slow build times, I created project-specific config files that would cascade with the default configuration file and build only one directory rather than all of them. This reduced the build time to the normal lightning-fast times of less than 5 seconds.

More specifically, to reduce the build times, we created a project-specific configuration file (e.g., `acme-config.yml`) that sets, through the `defaults`, all the directories to `publish: false` but lists one particular directory (the one with content you're working on) as `publish: true`. Then to build Jekyll, you cascade the config files like this:

```
jekyll serve --config _config.yml,acme-config.yml
```

The config files on the right overwrite the config files on the left. It works quite well.

Also, although at the time I grumbled about having to consolidate all content into two repos, I realized it was the right decision. Recognizing this, my respect and trust in the engineering team's judgment grew considerably. In the future, I started to treat the

engineers' recommendations and advice about various processes with much more respect. I didn't assume they misunderstood our authoring needs and requirements so much, and instead followed their direction more readily. Ensuring everyone builds with the same version of Jekyll

Another challenge was ensuring everyone built the project using the same version of Jekyll. Normally, you include a Gemfile in your Jekyll project that specifies the version of Jekyll you're using, and then everyone who builds the project with this Gemfile runs Bundler to make sure the project executes with this version of Jekyll. However, since our build pipeline had trouble running Bundler, we couldn't ensure that everyone was running the same version of Jekyll.

Ideally, you want everyone on the team using the same version of Jekyll to build their projects, and you want this version to match the version of Jekyll used on the server. Otherwise, Jekyll might not build the same way. You don't want to later discover that some lists don't render correctly or that some code samples don't highlight correctly because of a mismatch of gems. Without Bundler, everyone's version of Jekyll probably differed. Additionally, the latest supported version of Jekyll in the build management system was an older version of Jekyll (at the time, it was 3.4.3, which had a dependency on an earlier version of Liquid that was considerably slower in building out the Jekyll site).

The engineers finally upgraded to Jekyll 3.5.2, which allowed us to leverage Liquid 4.0. This reduced the build time from about 5 minutes to 1.5 minutes. Still, Jekyll 3.5.2 had a dependency on an older version of the [rouge gem](#), which was giving us issues with some code syntax highlighting for JSON. The process of updating the gem within the build management system was foreign territory to me, and it was also a new process for the engineers.

To keep everyone in sync, we asked that each writer check their version of Jekyll and manually upgrade to the latest version. This turned out not to be much of an issue since there wasn't much of a difference from one Jekyll gem version to the next (at least for the features we were using).

Ultimately, I learned that it's one thing to update all the Jekyll gems and other dependencies on your own machine, but it's an entirely different effort to update these gems within a build management server in an engineering environment you don't own.

Figuring out translation workflows

Figuring out the right process for translation was also difficult. We started out translating the Markdown source. Our translation vendor affirmed they could handle Markdown as a source format, and we did tests to confirm it. However, after a few translation projects, it turned out that they couldn't handle content that mixed Markdown with HTML, such as a Markdown document with an HTML table (and we almost always used HTML tables in Markdown). The vendors would count each HTML element as a Markdown entity, which would balloon the cost estimates.

Further, the number of translation vendors that could handle Markdown was limited, which created risks around the vendors that could even be used. For example, our localization managers often wanted to work with translation agencies in their own time zones. But if we were reliant on a particular vendor for their ability to process Markdown, we restricted our flexibility with vendors. If we wanted to scale across engineering, we couldn't force every team to use the same translation vendors, which might not be available in the right time zones. Eventually, we decided to revert to sending only HTML to vendors.

However, if we sent only the HTML output from Jekyll to vendors, it made it difficult to apply updates. With Jekyll (and most static site generators), your sidebar and layout are packaged into each of your individual doc pages. Assuming that you're just working with the HTML output (not the Markdown source), if you have to add a new page to your sidebar, or update any aspect of your layout, you would need to edit each individual HTML file instance to make those updates across the documentation. That wasn't something we wanted to do.

In the end, the process we developed for handling translation content involved manually inserting the translated HTML into pages in the Jekyll project and then having these pages build into the output like the other Markdown pages. We later evolved the process to create container files that provided the needed frontmatter metadata but which used

includes to pull the body content from the returned HTML file supplied by the translation vendors. It was a bit of manual labor, but acceptable given that we didn't route content through translation all that often.

The URLs for translated content also needed to have a different `baseurl`. Rather than outputting content in the `/docs/` folder, translated content needed to be output into `/ja/docs/` (for Japanese) or `/de/docs/` (for German). However, a single Jekyll project can have only one `baseurl` value as defined in the default `_config.yml` file. I had this `baseurl` value automated in a number of places in the theme.

To account for the new `baseurl`, I had to incorporate a number of hacks to prepend language prefixes into this path and adjust the permalink settings in each translated sidebar to build the file into the right `ja` or `de` directory in the output. It was confusing and if something breaks in the future, it will take me a while to unravel the logic I implemented.

Overall, translation remains one of the trickier aspects to handle with static site generators, as these tools are rarely designed with translation in mind. But we made it work. (Another challenge with translation was how to handle partially translated doc sets — I won't even get into this here.)

Overall, given the extreme flexibility and open nature of static site generators, we were able to adapt to the translation requirements and needs on the site.

Other challenges

There were a handful of other challenges worth mentioning (but not worth full development as in the previous sections). I'll briefly list them here so you know what you might be getting into when adopting a docs-as-code approach.

Moving content out of the legacy CMS

We probably had about 1,500 pages of documentation between our 10 writers. Moving all of this content out of the old CMS was challenging. Additionally, we decided to leave some deprecated content in the CMS, as it wasn't worth migrating. Creating redirect scripts that would correctly re-route all the content to the new URLs (especially with

changed file names) while not routing away from the deprecated CMS pages was challenging. Engineers wanted to handle these redirects at the server level, but they needed a list of old URLs and new URLs.

To programmatically create redirect entries for all the pages, I created a script that iterated throughout each doc sidebar and generated out a list of old and new URLs in a JSON format that the engineering team could incorporate into their redirect tool. It worked pretty well, but migrating the URLs through comprehensive redirects required more analysis and work.

Implementing new processes while still supporting the old

While our new process was in development (and not yet rolled out), we had to continue supporting the ability for writers to generate outputs for the old system (pasting content page by page into the legacy Hippo CMS). Any change we made had to also include the older logic and layouts to support the older system. This was particularly difficult with translation content since it required such a different workflow. Being able to migrate our content into a new system while continuing to publish in the older system, without making updates in both places, was a testament to the flexibility of Jekyll. We created separate layouts and configuration files in Jekyll to facilitate these needs.

One of the biggest hacks was with links. Hippo CMS required links to be absolute links if pasting HTML directly into the code view rather than using the WYSIWYG editor (insane as this sounds, it's true). We created a script in our Jekyll project to populate links with either absolute or relative URLs based on the publishing targets. It was a non-standard way of doing links (essentially we treated them as variables whose value was defined through properties in the config file). It worked. Again, Jekyll's flexibility allowed us to engineer the needed solution.

Constantly changing the processes for documentation

We had to constantly change the processes for documentation to fit what did or did not work with the engineering processes and environment. For example, git submodules, subtrees, small repos, large repos, frontmatter, file names, translation processes, etc., all fluctuated as we finalized the process and worked around issues or incompatibilities.

Each change created some frustration and stress for the tech writers, who felt that processes were changing too much and didn't like to hear about updates they would need to make or learn. And yet, it was hard to know the end from the beginning, especially when working with unknowns around engineering constraints and requirements. Knowing that the processes we were laying down now would likely be cemented into the pipeline build and workflow for long into the distant future was stressful.

I wanted to make sure we got things right, which might mean adjusting our process, but I didn't want to do that too much adjustment because each time there was a change, it weakened the confidence among the other tech writers about our direction and expertise about what we were doing.

During one meeting, I somewhat whimsically mentioned that updating our permalink path wouldn't be a bad idea (to have hierarchy in the URLs). One of the tech writers noted that she was already under the gun to meet deadlines for four separate projects and wasn't inclined to update all the permalinks for each page in these projects. After that, I was cautious about introducing any change without having an extremely compelling reason for it.

The experience made me realize that the majority of tech writers don't like to tinker around with tools or experiment with new authoring approaches. They've learned a way to write and publish content, and they resent it when you modify that process. It creates an extreme amount of stress in their lives. And yet, I kind of liked to try new approaches and techniques.

In the the engineering camp, I also took some flak for changing directions too frequently. I had to change directions to try to match the obscure engineering requirements. In retrospect, it would have helped if I had visited the engineers for a week to learn their workflow and infrastructure in depth.

Styling the tech docs within a larger site

Another challenge was with tech doc styles. The engineering team didn't have resources to handle our tech doc styling, so I ended up creating a stylesheet (3,000 lines long) with

all CSS namespaced to a class of `docs` (for example, `.docs p`, `.docs ul`, etc). I implemented namespacing to ensure the styles wouldn't alter other components of the site. Much of this CSS I simply copied from [Bootstrap](#). The engineers pretty much incorporated this stylesheet into their other styles for the website.

With JavaScript, however, we ran into namespace collisions and had to wrap our jQuery functions in a special name to avoid conflicts (the conflicts would end up breaking the initialization of some jQuery scripts). These namespace collisions with the scripts weren't apparent locally and were only visible after deploying on the server, so the test environment constantly flipped between breaking or not breaking the sidebar (which used jQuery). As a result, seeing broken components created a sense of panic from the engineers and dread among the tech writers.

The engineers weren't happy that we had the ability to break the display of content with our layout code in Jekyll. At the same time, we wanted the ability to push out content that relied on jQuery or other scripts. In the end, we got it to work, and the returned stability calmed down the writers.

Transitioning to a git-based workflow

While it may seem like Jekyll was the authoring tool to learn, actually the greater challenge was becoming familiar with Git-based workflows for doc content. This required some learning and familiarity with the command line and version control workflows.

Some writers already had a background with Git, while others had to learn it. Although we all ended up learning the Git commands, I'm not sure everyone actually used the same processes for pulling, pushing, and merging content (there's a lot of ways to do similar tasks).

There were plenty of times where someone accidentally merged a development branch into the master or found that two branches wouldn't merge, or they had to remove content from the master and put it back into development, etc. Figuring out the right process in Git is not a trivial undertaking. Even now, I'll occasionally find a formatting error because Git's conflict markers `>>>>>>` and `<<<<<<` find their way into the content, presumably from a merge gone wrong. We don't have any validation scripts (yet) that look for marker stubs

like this, so it's a bit disheartening to suddenly come across them.

Striking a balance between simplicity and robustness in doc tooling.

Overall, we had to support a nearly impossible requirement in accommodating less technical contributors (such as project managers or administrators outside our team). The requirement was to keep doc processes simple enough for non-technical people to make updates (similar to how they did in the old CMS), while also providing enough robustness in the doc tooling to satisfy the needs of tech writers, who often need to single-source content, implement variables, re-use snippets, output to PDF, and more.

In the end, given that our main audience and contributors were developers, we favored tools and workflows that developers would be familiar with. To contribute substantially in the docs, we decided that you would have to understand, to some extent, Git, Markdown, and Jekyll. For non-technical users, we directed them to a GUI (similar to GitHub's GUI) they could interact with to make edits in the repository. Then we would merge in and deploy their changes.

However, even the less technical users eventually learned to clone the project and push their updates into a development branch using the command line. It seems that editing via the GUI is rarely workable as a long-term solution.

Building a system that scales

Although we were using open source tools, our solution had to be able to scale in an enterprise way. Because the content used Markdown as the format, anyone could easily learn it. And because we used standard Git processes and tooling, engineers can more easily plug into the system.

We already had some engineering teams interacting in the repo. Our goal was to empower lots of engineering teams with the ability to plug into this system and begin authoring. Ideally, we could have dozens of different engineering groups owning and contributing content, with the tech writers acting more like facilitators and editors.

Also significant is that no licenses or seats were required to scale out the authoring. A writer just uses Atom editor (or another IDE). The writer would open up the project and work with the text, treating docs like code.

Within the first few weeks of launching our system, we found that engineers liked to contribute updates using the same code review tools they used with software projects. This simplified the editing workflow. But it also created more learning on our part, because it meant we would need to learn these code review tools, how to push to the code review system, how to merge updates from the reviews, and so forth.

Additionally, empowering these other groups to author required us to create extensive instructions, which was an entire documentation project in itself. I created around 30+ topics in our guide that explained everything from setting up a new project to publishing from the command line using Git to creating PDFs, navtabs, inserting tooltips and more. Given that this documentation was used internally only and wasn't documentation consumed externally, there wasn't a huge value or time allotment for creating it. Yet it consumed a lot of time. Making good documentation is hard, and given the questions and onboarding challenges, I realized just how much the content needed to be simplified and easier to follow.

Conclusion

Almost everyone on the team was happy about the way our doc solution turned out. Of course, there are always areas for improvement, but the existing solution was head and shoulders above the previous processes. Perhaps most importantly, Jekyll gave us an incredible degree of flexibility to create and adapt to our needs. It was a solution we could build on and make fit our infrastructure and requirements.

I outlined the challenges here to reinforce the fact that implementing docs-as-code is no small undertaking. It doesn't have to be an endeavor that takes months, but at a large company, if you're integrating with engineering infrastructure and building out a process that will scale and grow, it can require a decent amount of engineering expertise and effort.

If you're implementing docs-as-code at a small company, you can simplify processes and use a system that meets your needs. For example, you could simply use [GitHub Pages](#), or use the [S3 website plugin](#) to publish on AWS S3, or better yet, use a continuous deployment platform like [CloudCannon](#) or [Netlify](#). (I explore these tools in more depth here: [Publishing tool options for developer docs](#).) I might have opted for either of these approaches if allowed and if we didn't have an engineering support team to implement the workflow I described.

Building a Developer Portal? Here are Four Key Questions to Answer First

<https://apigee.com/about/blog/api-technology/building-developer-portal>

István Zoltán Szabó & Kristof Van Tomme



In collaboration with Apigee, for about a year now we at Pronovix have been custom-fitting the default Drupal-based Apigee developer portal to Apigee customers' individual requirements.

We've noticed some patterns emerging regarding the types of developer portals that organizations need and how they can address those needs using Apigee's developer portal.

We have been experimenting with a series of targeting questions to help our customers think through what kind of portal they actually need, and what kind of features it should incorporate.

Thanks to that effort, we've identified the four most important questions—the ones that show the common set of requirements our customers usually request. The answers to these questions help identify the purpose of the developer portal and help to facilitate the decision-making process.

Which audience does your developer portal target?

The answer to this question defines whether your portal will be open to the wider public or only to a certain group of people (your partners or your internal developers, for example). Is your site fully accessible to whomever visits, or are there barriers hiding any of the content?

If it isn't open for everyone, your portal must have a reliable and flexible access control system. Often companies have partnerships that only grant selective access to certain APIs; not all their APIs are available to all their partners. With an access control system, it's possible to manage API visibility without exposing the existence of anything else but what is accessible.

There are companies that use developer portals only internally. In these cases, the portals are not available for anyone outside of the company but may still need an access control system to manage the API availability for different developer teams within different business units.

Developer portals open to the public or to partners often have custom visual elements such as logos and brand colors to make it easier to recognize them. In the case of portals for internal use, branding is usually less important.

How many APIs does your developer portal handle?

The number of APIs you offer might be an important factor in the decision-making process about developer portals. If the portal provides access to hundreds of APIs throughout an organization wherein many teams are building their own APIs, then that portal will also act as a catalog of those APIs.

In these cases, it's crucial to make it possible to find (and use) the APIs across the various teams. The whole purpose of APIs is to interconnect and to make it possible to use each other's data (read our blog post "[What is an API?](#)" to learn more). Portals with lots of APIs have custom search functions to make the site-wide search easier and to provide better search results.

If a portal manages only a handful of APIs, then a sophisticated search solution is not that important. A small handful of APIs could be well presented in groups based on their purpose.

What's your API strategy and how can your developer portal support it?

Do you want to charge a price for API usage? If so, you need a dashboard where administrators can track the usage of APIs (calls and responses, for example). This analytical data is also useful if your business model is based on fixed prices—or if you don't charge for API usage at all, because you can keep an eye on your API traffic.

With the Apigee Edge Monetization extension, you can track API usage and even bill your customers for using your services.

What content do you provide along with your APIs?

Companies want to expose not only APIs but other related content on their developer portals. Writing, publishing, and improving documentation, blog posts, and onboarding

materials on a portal is a complex workflow and might involve a lot of people (technical writers, copywriters, marketing people, editors).

Even if you have only a couple of technical writers working on API documentation, you need the infrastructure to create and publish the content. The more varied content the content on your portal, the more complex the infrastructure.

The answers to these questions help lay out a clear path you can follow during the planning phase. Coming up next, we'll discuss the most common requirements that arise when implementing developer portals.

8 Common Customizations for Drupal-based Developer Portals

<https://apigee.com/about/blog/api-technology/8-common-customizations-drupal-based-developer-portals>

István Zoltán Szabó & Kristof Van Tomme



In the previous article, we covered four questions that help identify the purpose of a developer portal and help to facilitate the decision-making process. Answering these questions has helped us, in collaboration with the Apigee team, custom fit the Apigee Drupal-based developer portal to a variety of Apigee's customers' individual requirements (when this post refers to developer portals, it is referring to the Drupal-based portal, not the [new, lightweight portals](#)).

Here, we'll discuss the most commonly requested custom implementations. You'll see it's a diverse set, both in scope and in function.

SSO implementations

Single sign-on (SSO) is an authentication process that enables users to employ one set of login credentials to access multiple services or applications. It simplifies the authentication process, because if a system or a service (for example Gigya, Okta, or Google) already authenticated a user, then the users don't have to login again at every visit.

Pronovix customers often have a large group of websites. With an SSO implementation, it's possible to log in only once on one website and enable a user to access the other sites of the particular group without having to log in again.

Role-based access control

Role-based access control (RBAC) provides a scrupulously customizable access system implemented on the Drupal developer portal. RBAC is able to control the accessibility of the API products and the corresponding API documentation based on the groups created and managed within the system.

With this system, developer portal administrators can create groups, assign content to groups, add members (users) to them, and manage group visibility or the visibility of specific group content individually.

There are more technical ways to apply RBAC to a developer portal. The method always depends on the customer's exact requirements.

Landing pages

A landing page plays an important role on the business side of an API strategy. By definition, it is the first thing users see and interact with, so these pages are the first (and arguably most important) marketing, sales, and onboarding tool of a portal.

This is the best place to involve your creative team in framing the kind of experience that you will support elsewhere in your portal. It involves deep thinking about who your API customers are and understanding what they will need to be successful. Furthermore, because it is the most impactful branding opportunity for your API program, it will determine much of the look and feel of the entire site.

At the same time, expect that you will evolve your landing page over time, both as you learn what's working through user testing as well as just evolving with your API program over time. Some great examples of landing pages include [Kaiser Permanente](#) and [AccuWeather](#).

Custom theming

When a developer portal is open to an external audience (a common use case), the site owners will want to use their branding on the portals. They'll want to use the company logo, the company colors, their own design, and custom menu architecture on the portal. This way, it harmonizes with the other members of the company website family.

With Drupal, we have plenty of tools to provide non-trivial front-end solutions for the customers. The process ideally also involves a UX review to make sure that the visual design and information architecture will perform well in practice. Companies like Hiscox, Hermes, and Digital Insight have their own developer portal theming applied.

Data and user import

Data and user import comes into play when a company already has a developer portal but wants to change the platform. In this case, it's important that the company can access all of its data on the new platform without any loss, and convert the portal data to fit with the new content architecture.

Security and module updates

Upgrading a developer portal from an old version to a new one usually requires security and module updates. During the process, developers perform code reviews and update where necessary to ensure better performance and compliance with the latest security standards for the portal.

Content creation workflow

Developer portals are all about APIs and connection. To make the user's work easier, developer portals have tutorials, onboarding pages, documentation pages, blog posts, and many other materials that provide useful knowledge for developers about the portal's APIs. Creating all of this content is a big challenge; it involves a lot of people, including documentarians, developers, marketing staff, and content administrators.

A mature, well-structured, and detailed content creation workflow with the necessary technical tools is a basic requirement to deploy various content and to synchronize the work of the content teams. Content creation workflow is always based on specific roles. These roles have different kinds of permission to access, create, edit, or delete the various content types on the developer portal.

Custom search

Basic Drupal search might not fulfill the expectations for developer portals with large numbers of APIs. In these cases, we build custom search functions that perform better. Faceted search and [Solr](#) provide fast and relevant search results, for example.

The modified search makes it possible to search in API documentation or other special content types, where basic search does not work, to create more interactive user interfaces, and to filter the results on various ways.

This list isn't exhaustive, but it helps prove the flexibility of the Drupal content management system. Judging from the results of our customer engagements, we can say that Drupal does a great job handling the wide variety of custom requirements on a developer portal.

Tool the Docs

<https://pronovix.com/tool-docs-fosdem-2018>

A developer track for documentarians at FOSDEM 2018, dedicated to free and open source tools for the writing,



Free & OS tools for the writing, managing, testing and rendering of documentation

This year, a long time dream came true when we finally got a tech writers' DevRoom accepted at FOSDEM, co-organized by Chris Ward and Kristof Van Tomme. Yay! A big shout-out to the organizers of the conference and thank you for the presenters! With great pleasure we share with you the recordings, slide decks and Laura's notes from Tool The Docs.

A developer track for documentarians at FOSDEM 2018, dedicated to free and open source tools for the writing, managing, testing and rendering of documentation.

- Ferit Topcu: Automating style guide documentation
- Stefan Knorr: DocBook Documentation at SUSE
- Honza Javorek: Test your API docs!
- Kitti Radovics: Docs like code in Drupal
- Stephen Finucane: A lion, a head, and a dash of YAML
- Shaun McCance: Mallard, Pintail, and other duck topics
- Jessica Parsons: Finding a home for docs

Automating style guide documentation

How we at Zalando Retail Dept automated our Styleguide & source code and reduced the gap for contribution

[Ferit Topcu](#)

Front-end Software Engineer, [Zalando SE](#)

Enable developers to create docs to a new feature asap, without switching context.

As soon as a new feature is developed, instant testing against styleguide via inhouse styleguide app. [Stylelint](#) (scss, css): ensure code consistency accross many developers and UX. For each pull request they test build again and also lint style.

Development in components - document in components

[useJSDocs documentation](#) is an OS library, generate Markdown from your JS docs (part of CI) automated html output. Document all properties of a component, give an @example for how to use that feature see [demo project](#).

Docs part of Definition Of Done

Pull request only accepted with documentation had to be centrally decided and enforced policy pro: easy onboarding

Low entry level for documentation

Markdown and GitHub knowledge is sufficient for a start.

[JAMstack approach](#)

[see demo](#)

[Ferit's talk proposal outline](#)

Ferit's [presentation](#) and [slides](#).

DocBook Documentation at SUSE

Automatically Ensuring Quality of SUSE Documentation

[Stefan Knorr](#)

Technical Writer at [OpenSUSE Documentation Team](#)

Workflow

Almost only docbook output. Accept input in many formats. Single-sourcing, multiple docs, multiple formats. Git+GitHub+GitFlow model+PRs+reviews. The gitflow branching model helps them keep track of the branches. OBS builds documentation RPM packages (OS toolchain necessary)

[DAPS](#)

Solved the toolchain gaps they found with upstream DocBook (publishing gaps too). Editor agnostic, they use many different editors. 1. profiling stylesheets, then 2. output stylesheets. Does not handle translation, outsourced.

They use their own RelaxNG schema (DocBook 5.1) DAPS validate

Styleguide:

language & structure rules (avoid confusing readers and translation costs):

- synonyms, eg. use keycap to highlight keys, and change hit to press
- "soft" syntax rules (not to crash validation) eg. avoid wordy phrases, avoid lonely sections

Their style-checker produces quite some output, with some false positives, this is not ideal.

Future plans:

- Improve spell check, source lines (hard problem)
- need both html and plain txt output but the xml format does not translate well.

Travis CI:

Publish docs to GitHub pages. ToDo: Integrate a style-checker without inundating people with error messages.

Stylesheet checks

[DAPScompare](#), their own DocBook validation sheet. Future plan: need more example docs.

PS: xslt is only well documented in German.

[Stefan's talk proposal outline](#)

Stefan's [presentation](#) and [slides](#).

Test your API docs!

It's tested or it's broken

[Honza Javorek](#)

Maintainer of the API testing tool [DREDD](#) at @apiaryio

Every interface is Human User Interface

In the end humans need to leverage the interface, so consider and develop every interface as UI. How to design good UI? Eat you own dogfood. How to position yourself as your future user? Use [Test Drive Development](#).

[Behaviour Driven Development](#)

Cucumber/Gherkin Makes what you create testable 1. How should it work?

- 1 How would I test that?
- 2 Write and run test as if it already existed. Test fails.
- 3 Implement until it fits the original design. You fulfilled your original promise.

[REQUEST library](#), see Kenneth Reiz's essay "[How I develop things and why](#)"

Readme Driven Development: Responsive API design

"...instead of engineering something that will only get the job done, you start to interact with the problem itself and build an interface that reacts to it."

"With Readme Driven Development, by the time you are done, the docs are already there."

Doctest: parse and run the examples, they have to run and succeed as in the readme. This ensures sync between code (implementation) and readme (essential contract with your users).

DREDD

Human and machine readable API.md.

DREDD works as doctest does for code.

Extend testing with hooks, most are contributed, you are welcome to add your own language into the open library.

Honza's talk proposal outline

Honza's [presentation](#) and [slides](#).

Docs like code in Drupal

Introducing Open DevPortal, an open source CMS based documentation tool

Kitti Radovics

Front-end Developer at Pronovix

What is docs like code

In Docs like code, a team uses version control systems (like GitHub) to collaborate on documentation inside a code repository.

Changes to the docs are then automatically deployed using CI/CD and static site generators.

Advantages of docs like code

- Keep pace with code changes
- Collaborate with the contributors of the documentation
- Anyone can contribute (Technical Writers, Developers)

When to use a CMS for docs like code

- When regenerating the whole site for each edit is too slow (e.g. lots of content, or lots of updates and writers)
- When you need other content than just docs (landing page, marketing, blog)
- When you want built-in search (keywords, tags, categories)
- When you need Role Based Access Control (internal & partner APIs)
- When you want interactive documentation that talks with the API
- When you need rich interactions for gamification

After the introduction Docs like code in Drupal, Kitti explained how she is working with her team on a Drupal based solution that can import Markdown and Open API specifications from GitHub.

Upstream docs

How a CMS could be used to integrate documentation from different repositories, while maintaining a consistent UI for readers and for casual contributors.

[Kitti's talk proposal outline](#)

Kitti's [presentation](#) and [slides](#).

A lion, a head, and a dash of YAML

Extending Sphinx to automate your documentation

[Stephen Finucane](#)

Software Engineer at Redhat

Intro

reStructuredText - syntax (write) Docutils - parsing (write, individual files) Sphinx - build, multiple cross-referenced files

Sphinx extensions

Roles & Directives in Docutils

Example for roles: recurring format defined in docutils as a role, eg. GitHub issue number becomes a hyperlink in http output.

Create directive in docutils, eg. connecting to the GitHub API, import information on that GitHub issue when it appears in the docs.

Events in Sphinx

(not in Docutils)

Sphinx contrib, hundreds of extensions, you can use those the contrib ones mixed with your own.

[Stephen's talk proposal outline](#)

Stephen's [presentation](#) and [slides](#).

Mallard, Pintail, and other duck topics

topic-oriented help at the GNOME project

[Shaun McCance](#)

GNOME Developer, Writer, and Community Advocate. Runs the Open Help Conference & Sprints. Upstream documentation projects at RedHat Open Source and Standards department.

Mallard

Think about your docs differently:

- 1 topics people want to read about, then
- 2 build the docs bottom up.

Create a linking structure, where the links work both ways.

Can embed existing standards, vocabularies. Eg. its standard for xml translations.

Gives you a data model of your content.

The only problem with xml is that it's xml...

Need a lightweight syntax that has the possibilities of xml: ducktype.

Ducktype

Similar to Markdown or reStructuredText.

All the mallard-critical metadata are possible.

Allows nested notes, works completely with indentation, no limit to indentations.

Conditionals are still complicated but possible and easier to read than xml.

Anything xml does without its complicated syntax.

Can make use of all the xml tools e.g. validation.

[Pintail](#)

Supports Markdown and AsciiDoc but it's Mallard first.

A documentation publishing tool in the Mallard ecosystem.

Yelp.io

Config file: how to build the output.

Can mix documentation formats.

Search

Important to have context. Good first start to restrain search domain to the project where you are.

Allow to change search domain for user (e.g. dropdown).

[Shaun's talk proposal outline](#)

Shaun's [presentation](#).

Finding a home for docs

How to choose the right "path" for documentation in open source projects

[Jessica Parsons](#)

Documentation Engineer at Netlify

Finding a home for the [NetlifyCMS docs](#)

Started with a readme on GitHub, like most projects...

Needed a website for all the docs: copied docs from GitHub to website (with Hugo), this resulted in two sources of truth.

Markdown magic: comments inside your files, pull them into .md files. Use it for taking from code repo to website repo, they took out the docs from GitHub. Problem with CI/CD, it broke the deploy previews.

Issue 750, 2017/okt/27: "Publishing docs in two places is no fun."

Want the ease of changing docs just as code changes go. Back to GitHub. Docs edits in the same pull request.

Tag code releases and tag doc releases: you see the previous code with the matching docs (if you did publish your docs with the code).

[Jessica's talk proposal outline](#)

Jessica's [presentation](#) and [slides](#).

Original recordings at [Fosdem](#), licensed under the Creative Commons Attribution 2.0 Belgium Licence.

AUTHORS





Kathleen De Roo

Technical copywriter at
Pronovix

Kathleen De Roo

As a technical writer and member of the Pronovix content team, Kathleen is responsible for writing, reviewing and editing website copy and blog posts, mainly on developer portal documentation aspects. She is currently also diving into the world of information architecture.

She holds master's degrees in history and in archival science & records management.



Bill Doerrfeld

Editor in Chief at
Nordic APIs

Bill Doerrfeld

Bill Doerrfeld is an API specialist, focusing on API economy research. He is the Editor in Chief for Nordic APIs. He leads content direction and oversees the publishing schedule for the Nordic APIs blog. Bill personally reviews all submissions for the blog and is always on the hunt for API stories. Follow him on Twitter, or visit his personal website." Twitter - @DoerrfeldBill



Jason Harmon
CPO at Typeform

Jason Harmon

Jason has a diverse background in backend engineering at companies like AT&T, uShip and PayPal. He led an engineering-wide initiative at PayPal focusing on microservice definition using REST APIs on one of the largest and most storied API programs in the industry. He also has a wide range of startup experience as an early software engineer at Coremetrics.com (now a division of IBM), a Software Architect at Wayport (acquired by AT&T), and ran product for the API program at uShip, a successful shipping marketplace. He is a recognized industry expert on the topic of API and microservice governance. He served on the founding OpenAPI Initiative committees, and is a regular speaker at API conferences around the world.



Erik Hogan
Director of product
management at
PayPal

Erik Hogan

Erik Hogan has over 20 years of product management experience across a variety of software and hardware products in both small, venture-funded startups and large organizations. He has typically focused on developing strong, reusable platforms that enable a variety of customer solutions and accelerate the business. More recently, he has driven the API portfolio management and standardization program at PayPal and is now working on elevating the internal PayPal developer experience into a world class, strategic asset for the organization. After many years in the Bay Area, he recently relocated to Austin where he is enjoying a more humane existence with his family.



Tom Johnson
Technical writer at
Amazon Lab126

Tom Johnson

Tom Johnson is a technical writer for Amazon in Sunnyvale, California. He writes a popular blog on technical writing called Idratherbewriting.com, where he explores topics such as API documentation, trends, information design, and more. He also has an extensive online course on API documentation that includes extensive tutorials and other exercises you can follow to build your expertise with APIs, including the OpenAPI specification, Swagger, and more.



Diána Lakatos
Senior Technical
Writer at Pronovix

Diána Lakatos

Diána is a Senior Technical Writer at Pronovix. She is specialized in API documentation, topic-based authoring, and contextual help solutions. She writes, edits and reviews software documentation, website copy, user documents, and publications. She also enjoys working as a Program Monitor for NHK World TV and Arirang TV.



Maxime Locqueville
Lead Developer at
Algolia

Maxime Locqueville

Maxime Locqueville worked on creating search plugins for php frameworks and cms, then transitioned to work on the Algolia documentation. He is now the lead developer in charge of the docs, the DocSearch project, and customer support tools & workflows



István Zoltán Szabó
Technical Writer at
Pronovix

István Zoltán Szabó

Steve is a Technical Writer at Pronovix. He specializes in developer and user documentation for developer portals, API reference docs, technical articles and web copy.

Besides this, he's translating books from English to Hungarian for a publishing company. Steve has a journalist/writer background, his works are frequently published in various online and printed journals.



Kristof Van Tomme
CEO and co-founder
of Pronovix

Kristof Van Tomme

Kristof Van Tomme is an open source strategist and architect. He is the CEO and co-founder of Pronovix. He's got a degree in bioengineering and is a regular speaker at conferences in the API, DevRel, and technical writing communities. For a few years now he's been building bridges between the documentation and Drupal community. He is co-organiser of the London Write the Docs meetup, and cheerleader for the Amsterdam and Brussels Write the Docs meetup groups. This year he is working on a number of new initiatives to help API product owners learn from their peers (API the Docs and the API product owner masterminds).



Jenny Wanger
Product Manager at
Arity

Jenny Wanger

I'm a product manager who believes that a user-centered mindset can be applied to any process, and love to bring agile and user-centered practices to unusual spaces like service design. My experience designing digital products and tools, structuring new business models, and facilitating design thinking workshops at some of the world's largest companies has provided me the skill set that can help a company sustainably grow and scale.



Bob Watson
Assistant Professor at
Mercer University

Bob Watson

Bob earned his Ph.D. in Human Centered Design and Engineering from the University of Washington in 2015. He is currently an Assistant Professor at Mercer University researching and teaching technical communication. Prior to his current position in academia, he worked as a programmer-writer for 15 years and documented countless APIs. Bob was also a software developer for about 15 years before he discovered how much he enjoyed technical communication.

